# Dynamic Priority Scheduling of Multi-Threaded ROS 2 Executor with Shared Resources

Abdullah Al Arafat*, Kurt Wilson*, Kecheng Yang†, Zhishan Guo*
*North Carolina State University, †Texas State University
{aalaraf, kwilso24, zguo32}@ncsu.edu, yangk@txstate.edu

***Abstract*—The second generation of Robot Operating System (ROS 2) received significant attention from the real-time system research community, mostly aiming at providing formal modeling and timing analysis. However, most of the current efforts are limited to the default scheduling design schemes of ROS 2. The unique scheduling policies maintained by default ROS 2 significantly affect the response time and acceptance rate of workload schedulability. It also invalidates the adaptation of the rich existing results related to non-preemptive (and limited-preemptive) scheduling problems in the real-time systems community to ROS 2 schedulability analysis. This paper aims to design, implement, and analyze a standard dynamic priority-based real-time scheduler for ROS 2 while handling shared resources. Specifically, we propose to replace the readySet with a readyQueue, which is much more efficient and comes with improvements for *callback selection, queue updating, and a skipping scheme* to avoid priority inversion from resource sharing. Such a novel ROS 2 executor design can also be used for efficient implementations of fixed priority policies and mixed-policy schedulers. Our modified executor maintains the compatibility with default ROS 2 architecture. We further identified and built a link between the scheduling of limited-preemption points tasks via the global earliest deadline first (GEDF) algorithm and ROS 2 processing chain scheduling without shared resources. Based on this, we formally capture the worst-case blocking time and thereby develop a response time analysis for ROS 2 processing chains with shared resources. We evaluate our scheduler by implementing our modified scheduler that accepts scheduling parameters from the system designer in ROS 2. We ran two case studies–one using real ROS 2 nodes to drive a small ground vehicle, and one using synthetic tasks. The second case study identifies a case where the modified executor prevents priority inversion. We also test our analysis with randomly generated workloads. In our tests, our modified scheduler performed better than the ROS 2 default.**

***Index Terms*—ROS 2, ready queue, Non-Preemptive EDF, Processing Chains**

## I. Introduction

Robot Operating System (ROS), an open-source framework, has been extensively utilized in designing robotics applications and autonomous systems over the past decade, primarily due to their modularity and composability. Most applications involving autonomous systems and robotics software are associated with safety-critical systems, where ensuring 'timing correctness' is a prerequisite prior to deployment. However, despite the heavy use of ROS in these applications, ROS has inherent limitations concerning real-time capabilities.

Consequently, ROS was completely refactored in the second generation, denoted as ROS 2 [1], to add real-time capabilities. Casini *et al.* [8] first provided a formal scheduling model of ROS 2 executor and developed a response time bound for the ROS 2 workload (*i.e.*, processing chains), revealing a significant difference between standard real-time scheduling model and default ROS 2 executor scheduling model. The key source of difference is that ROS 2 executor maintains a set to record callbacks (executable units), denoted as `readySet`, with unique properties of set update and callback selection policies. Since then, several works [27], [26], [7], [28], [9], [2] improved the analysis of response time bound modeling the ROS 2 workloads as either processing chains or a directed-acyclic-graph (DAG) for the ROS 2 executor scheduling model. However, most of these methods are developed for a *single-threaded* executor and are limited to analyzing the default `readySet`-based executor scheduling scheme. Recently, Jiang *et al.* [16] and Sobhani *et al.* [24] presented a scheduling model and analysis for default *multi-threaded* executor. Moreover, [16] observed that if all callbacks in the system shared a common resource, then the multi-thread ROS 2 performs inconsistently (*i.e.*, there exists a concurrency bug); however, no solution was provided to resolve the issue.

As the scheduling model of default ROS 2 executor significantly differs from the standard real-time scheduling model, one can hardly adapt existing results for the ROS 2 scheduling problem. Therefore, one natural question arises: is it possible to modify the ROS 2 executor to adapt standard scheduling analysis techniques without breaking the fundamental properties of ROS 2? Arafat *et al.* [2] first attempted to modify a single-threaded ROS 2 executor to apply a dynamic-priority-based scheduler. This paper focuses on designing, implementing, and analyzing a *multi-threaded* ROS 2 executor for dynamic priority-based scheduling.

One of the key obstacles to the shift toward a multi-threaded executor is *resource sharing* between callbacks. ROS 2 allows resource sharing among callbacks by putting them in a *mutually exclusive callback group*, which the user can use to protect critical sections and prevent deadlock. This, in addition to redesigning the `readySet` to make it priority-based sorting, makes designing a multi-threaded executor for priority-based scheduling very challenging and significantly different than designing one for a single-threaded executor.

**Contribution.** Our contributions are three-fold:

- We design a (flexible) multi-threaded ROS 2 executor that can be used for fixed-priority, dynamic-priority, and

First two authors contributed equally to this work. This work was supported in part by the National Science Foundation under Grant CMMI 224667.

mixed-priority-based scheduling where the user can select a preferred scheduling policy through user input. We propose to have the executor maintain a queue, denoted as `readyQueue`, which replaces the `readySet` in ROS 2 to record the ready callbacks. To cope with `readyQueue` maintain compatibility with default ROS 2 architecture, we design callback selection, queue updating, and a skipping scheme to avoid priority inversion from resource sharing (ref. Sec IV). Such a design significantly reduces the complexities related to the queue (or set) update and callback selection policies of the executor compared to its default design. Notably, the designed executor can successfully overcome the concurrency bug related to resource-shared callbacks that exist in the default ROS 2 multi-threaded executor (please refer to Case Study 2 in Sec. VI-A for more details).

- We focus on analyzing the response time for (callback-level) non-preemptive earliest deadline first (EDF) for the multi-threaded ROS 2 executor, even though our modified executor can be used for other schedulers . We identified and built a link between the scheduling of limited-preemption points tasks via GEDF and ROS 2 processing chain scheduling without shared resources. Based on this, we formally capture the worst-case blocking time and thereby develop a response time analysis for ROS 2 processing chains with shared resources (ref. Sec V).
- We evaluate our scheduler using two real-world case studies, and show that it improves upon the default executors. We identify issues with the default ROS 2 executors and discuss how our modifications work around them (ref. Sec VI-A). We then evaluate the overheads of the proposed executor and compare them with existing executors (ref. Sec. VI-B). We further test our response time analysis with synthetic workloads and show that it can successfully schedule more workloads than the default ROS 2 executors (ref. Sec VI-C).

## II. Background: Multi-Threaded ROS 2

ROS 2 is a collection of libraries that provide a middleware between the operating system and application layers for robotics applications (Fig. 1). Specifically, ROS 2 provides a client library *rcl* with language-specific libraries (*e.g.*, *rclcpp, rclpy*) containing the executors, and middleware library (*rmw*) containing the publisher-subscriber mechanism for inter-process communication to the Data Distribution Service (DDS). ROS 2 integrates with open source and commercially available DDS systems [11], [10], [13].

The minimum executable unit of the ROS 2 application layer is called *callback*. There are four types of callbacks in ROS 2 such as *timer, subscriber, service,* and *client* with a semantic priority order: timer ≻ subscriber ≻ service ≻ client. For ease of presentation, throughout this paper, we refer to non-timer callbacks as *regular callbacks*. Callbacks can be run in response to messages, service calls, or timers in the ROS 2 system. Callbacks are organized into nodes, which separate related callbacks into logical groups. In ROS 2, applications are typically composed of a series of individual nodes distributed in the application layer. Nodes use DDS for real-time message exchange through a publish-subscribe mechanism. Nodes can listen for messages from other nodes (including itself) using subscribers. Service calls are an extension of messages, where a service provider responds to all incoming messages with a response message. Nodes use timers to run callbacks at specific periods.

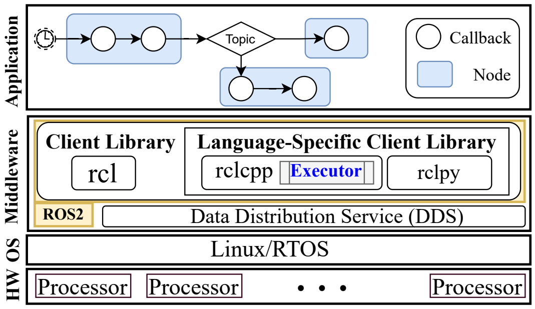


Fig. 1: *Simplified ROS 2 Architecture*

Callbacks are usually arranged into chains, where each chain starts with a timer, and each callback in the chain sends a message that starts another callback until the last callback, which produces a result or controls an actuator.

Multiple nodes can be launched within a single process, where the callbacks are managed and run by an *executor*. The executor maintains a set, denoted as `readySet`, for ready callbacks. The executor continuously polls the `readySet` for an eligible callback to run. By default, the executor searches the `readySet` in order of callback type [8], [27]. `readySet` maintains the default priority order of the callbacks in the set. Callbacks of the same type are ordered by registration order.

ROS 2 offers two default executors: a *single-threaded* executor and a *multi-threaded* executor. Fig. 2 shows the callback selection flow of a multi-threaded executor. The multi-threaded executor spins on multiple cores. ROS 2 offers the concept of callback groups such as: *mutually exclusive callback groups*, where an executor will only run one callback from each mutually exclusive group at a time, and *reentrant callback group*, where an executor is allowed to run multiple instances of a callback at any given time. Mutually exclusive callback groups affect how the `readySet` is managed. If a callback from a mutually exclusive group is currently running, callbacks in the same group are considered not eligible, even if one of them is in the `readySet`.

There is a drawback to the default ROS 2 multi-threaded executor: the callbacks in the `readySet` are only refreshed in two cases—when the `readySet` is empty, or when all callbacks in the `readySet` are not eligible. We show this point in Fig. 2. In previous works, this refresh is known as a polling point. To refresh the `readySet`, the default executor clears all the lists and attempts to retrieve one message (or timer release) for each callback. Since a polling point does not happen every callback execution, there can be cases where response times are increased [16]. Additionally, if the multi-threaded executor cannot find a callback to run due to mutually exclusive callback groups, the executor clears the `readySet` and adds only callbacks that can be run at that instant.

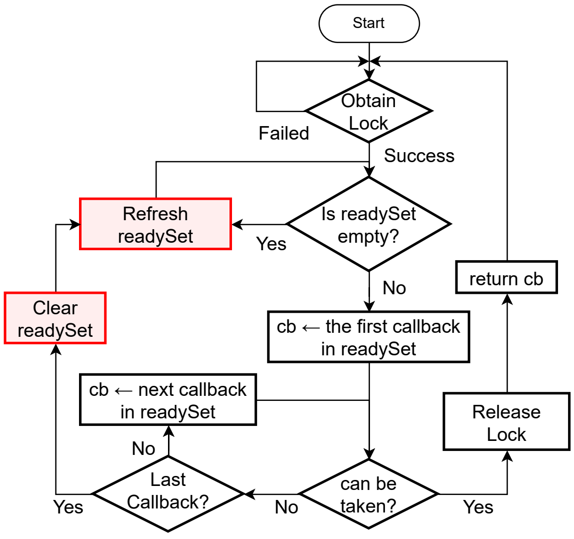


Fig. 2: *Thread workflow inside the default ROS 2 executor*

Callbacks that were removed from the `readySet` will only be added back to the `readySet` at the next polling point.

## III. System Model

This section presents the formal analytical model for ROS 2 workload and default executor scheduler. We consider a set of $n$ processing chains[1] $\Gamma = \{\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_n\}$ as the workload of ROS 2. Each processing chain (in short, *chain*) consists of a sequence of callbacks. Executors select and dispatch the callbacks in threads to execute following scheduling policies. Our focus in this paper is limited to scheduling ROS 2 workloads inside a single 'multi-threaded' executor. Without loss of generality, we consider integer time instances only aligned with the granularity of the processor clock tick. All the notations used in the paper are listed in Table I.

**Callbacks.** Each callback belongs to a processing chain. Let us denote the $j^{th}$ callback of $i^{th}$ processing chain as $c_{i,j}$. The worst-case execution time (WCET) of $c_{i,j}$ is denoted as $e_{i,j}$. Callbacks are scheduled to execute non-preemptively. The priority of a callback is determined by its semantic priority and registration order. Each callback can potentially release infinitely many instances where the timer callback is periodically released, and regular callbacks are event-triggered.

A ROS 2 callback system has a single *reentrant callback group* and may have multiple *mutually exclusive callback groups*. Each callback either belongs to the reentrant callback group or belongs to one of the mutually exclusive callback groups. For notational simplicity, we index the callback groups by integers where index 0 denotes the reentrant callback group, and each of the positive integers denotes a mutually exclusive callback group. Then, we define $\mathcal{G}(c_{i,j})$ as a function that takes a callback $c_{i,j}$ as an argument and returns the index of the callback group the callback $c_{i,j}$ belongs to. Then, $\theta_i = \cup_{1 \le j \le |\mathcal{C}_i| \wedge \mathcal{G}(c_{i,j}) \neq 0} \{\mathcal{G}(c_{i,j})\}$ is the set of indices of all mutually exclusive callback groups to which a callback in chain $\mathcal{C}_i$ belongs.

TABLE I: Notation Summary

| Symbol | Description |
|---|---|
| $n$ | Number of processing chains |
| $\Gamma$ | Set of processing chains |
| $\mathcal{C}_i$ | i$^{th}$ processing chain |
| $\lvert\mathcal{C}_i\rvert$ | Number of callbacks in $\mathcal{C}_i$ |
| $c_{i,j}$ | $j^{th}$ callback of chain $\mathcal{C}_i$ |
| $c^k_{i,j}$ | $j^{th}$ callback of $k^{th}$ instance of chain $\mathcal{C}_i$ |
| $e_{i,j}$ | WCET of callback $c_{i,j}$ |
| $E_i$ | WCET of chain $\mathcal{C}_i$ |
| $D_i$ | Relative deadline of chain $\mathcal{C}_i$ |
| $T_i$ | Period of chain $\mathcal{C}_i$ |
| $\mathcal{C}^k_i$ | $k^{th}$ instance of chain $\mathcal{C}_i$ |
| $a^k_i$ | Arrival time of $k^{th}$ instance of chain $\mathcal{C}_i$ |
| $d^k_i$ | Absolute deadline of $k^{th}$ instance of chain $\mathcal{C}_i$ |
| $\mathcal{G}(c_{i,j})$ | Index of the callback group where $c_{i,j}$ belongs to |
| $\theta_i$ | Union of $\mathcal{G}(c_{i,j}) \neq 0$ for all $j$'s |
| $\mathcal{E}$ | Executor |
| $m$ | Number of threads in a executor $\mathcal{E}$ |
| $\pi_i$ | $i^{th}$ thread |
| $R(\mathcal{C}^k_i)$ | Response time of $k^{th}$ instance of chain $\mathcal{C}_i$ |
| $R_i$ | WCRT of chain $\mathcal{C}_i$ |
| $\Omega$ | readyQueue |
| $S^{A_k}_t$ | Problem window of length $t$ for an instance of $\mathcal{C}_k$ |

**Chains.** A chain $\mathcal{C}_i = \{c_{i,1}, c_{i,2}, \cdots, c_{i,|\mathcal{C}_i|}\}$ is a sequence of $|\mathcal{C}_i|$ callbacks, where $c_{i,1}$ is the first callback and $c_{i,|\mathcal{C}_i|}$ is the last callback of the chain. Depending on the type of first callback, a chain can be classified as time-triggered (*i.e.*, $c_{i,1}$ is *timer* callback) or event-triggered (*i.e.*, $c_{i,1}$ is a *regular* callback) chain. Except for the first callback, any $c_{i,j}$ can only become ready to execute once $c_{i,j-1}$ finished its execution since each callback is released by the previous callback in the chain publishing its results (*i.e.*, intermediate callbacks in the chain cannot be time-triggered callback). A chain $\mathcal{C}_i$ is characterized via tuple $(E_i, D_i, T_i)$, where

- $E_i = \sum_{\forall j} e_{i,j}$ is the WCET of the chain $\mathcal{C}_i$, which is the sum of its callbacks' WCET.
- $T_i$ is the minimum inter-arrival time (period) between two chain instances. A time-triggered chain $\mathcal{C}_i$ will be periodically released every $T_i$ time instants. A chain can potentially release infinite instances, and $k^{th}$ instance of chain $\mathcal{C}_i$ is denoted as $\mathcal{C}^k_i$.
- $D_i$ is the relative deadline of the chain and $D_i \le T_i$.

The **response time** of $\mathcal{C}^k_i$, $R(\mathcal{C}^k_i)$, is the time difference between the release instant of its first callback $c^k_{i,1}$ and the completion time instant of the last callback $c^k_{i,|\mathcal{C}_i|}$. The worst-case response time (WCRT) is the maximum response among all possible release instances of the chain, $R_i = \max_{\forall k} R(\mathcal{C}^k_i)$. A chain is considered schedulable if all its instances meet the deadline, *i.e.*, $R_i \le D_i$. A ROS 2 workload $\Gamma$ will be *schedulable* if all chains are schedulable, *i.e.*, $\forall i,\ R_i \le D_i$.

**Executor.** We consider a multi-threaded executor $\mathcal{E}$ consisting of $m$ working threads $\mathcal{E} = \{\pi_1, \pi_2, \cdots, \pi_m\}$. Aligning with previous works in multi-threaded executor for ROS 2 [16], [24], we consider the one-to-one assignment of each thread $\pi_i$

[1] In ROS 2 workload graph, a callback can be shared by multiple chains. However, due to decomposing the workload graph as independent processing chains, each will contain an independent replica of a shared callback [8].

to a processor core for maximizing the concurrent executions of callbacks. We assume processing cores are homogeneous. We further assume a dedicated resource supply to each thread from the corresponding processing core and, without loss of generality, all processing cores as unit-speed cores. Therefore, the total resource supply for $m$ threads is $m$.

**Default Scheduling Model for Executor.** Any callback $c_{i,j}$ in a chain $\mathcal{C}_i$ can only be *ready* once $c_{i,j-1}$ completes in execution. The default ROS 2 executor maintains a readySet to record ready callback instances that can be selected for execution. However, a ready callback instance cannot directly enter the readySet. Instead, it can only enter the readySet once the readySet becomes empty or any thread in the executor is idle. A callback instance is ready but waiting to enter the readySet is denoted as "*pending*." The set of pending callbacks is known as *wait_set*. The readySet update instances are known as *polling points*, and the duration between the two consecutive polling points is known as *polling window*. Once a callback instance is selected from the readySet, it begins executing non-preemptively.

A pending callback instance can also be in the state of "*not eligible*" to be in the readySet depending on the membership of a mutually exclusive callback group. For instance, only one callback from each mutually exclusive group can enter the readySet at a time. A callback of a mutually exclusive group can receive two types of blocking from other members of the group. First, if a callback is *pending* but cannot enter to readySet due to the presence of another callback from the same mutual exclusive callback group, then the blocking is denoted as "pending and blocked" (*i.e.*, **P-blocked**). Second, if a callback is currently in the readySet but cannot be selected if another callback from the same mutual exclusive group is executing in any thread. This blocking is denoted as "ready and blocked" (*i.e.*, **R-blocked**).

## IV. Dynamic-Priority-Based Executor

This section presents the design and scheduling model of a dynamic-priority-based ROS 2 executor.

### A. Design of Dynamic-Priority based Executor

We extend the default multithreaded executor by replacing the readySet with a readyQueue, where the readyQueue is implemented as a PriorityQueue. Each callback instance is wrapped in a struct that contains the scheduling parameters of the callback, as well as its type. The readyQueue stores these structs and sorts them using a custom comparator. The comparator sorts the callback instances in order of their absolute deadline[2], placing earlier deadlines first. Callbacks without explicitly defined scheduling parameters[3] are placed last. Similar to ROS 2's default executor, ties are broken by the registration order. However, unlike the default ROS 2 scheduler, our comparator does not consider the callback type; *i.e.*, all callback types are considered equally. The executor also respects the overload handler in timers, which is a default ROS 2 feature that detects if a timer callback is blocked for more than one period, and moves the *next* release forward by one period. This prevents two successive timer callback executions, allowing in-progress chains to complete in an overloaded system. If this happens, the executor adjusts the chain's deadline to reflect the new timer release. The readyQueue is defined as follows:

**Definition 1. (readyQueue $\Omega$)** *is maintained in the* executor *to record the ready callbacks similar to* readySet *in default ROS 2. However,* readyQueue *is always updated before any executor thread selects a callback to run. The priority of the callbacks in* readyQueue *is set based on the deadline of each callback, where a callback with an earlier deadline has a higher priority than the one with a later deadline.*

To account for the fact that the first callback on the readyQueue may not be executable (due to mutually-exclusive callback groups), we use a custom queue implementation that allows iterating through its elements.

We now discuss three key components and principles related to the design of a dynamic-priority-driven *executor*.

**(i) Callback Selection.** Algorithm 1 presents the details related to the callback selection policies from readyQueue. At the very beginning, the executor starts some worker threads, where the number of threads is specified by the user. Each worker thread is pinned to a CPU core. Each worker thread polls for callbacks similarly to that of the single-threaded executor. A *mutex lock* protects the readyQueue so that only one worker thread can update it at a time. When a thread becomes idle, it attempts to take the lock, update the readyQueue, and select a callback. If another thread is holding the lock, the thread is blocked until the lock is available. To select a callback, it selects the highest priority callback that is currently eligible to execute. The executor removes the selected callback from the readyQueue, releases the lock, and begins to execute the selected callback non-preemptively (ref. line 21). Once the lock is released, other worker threads can access the readyQueue. Callbacks that are not selected for execution immediately are kept in the readyQueue and can be run later. To prevent race conditions caused by callback groups running in other threads, if a callback is running as part of a group at any point during the callback selection process, the group will always be skipped (ref. line 8), even if the offending callback stops execution during the selection process.

**(ii) readyQueue Updating.** To update the readyQueue, the executor checks all callbacks in the system for newly released instances and adds them to the readyQueue. The executor also updates the positions of callbacks that are already in the readyQueue, if any new callback instance is added to the queue. To maintain the assumptions and restrictions of ROS 2's DDS interface[4], the readyQueue is restricted to hold one and only one instance of each callback at a time. This does not affect the execution order – all

[2] The comparator can be replaced by the user to use different comparison metrics, such as fixed callback-level priorities, or mixed scheduling policies– where some callbacks have dynamic priorities, and some have fixed priorities.

[3] This may include automatically created callbacks by ROS 2, such as the one for the parameter system.

[4] Due to API design, the DDS interface only exposes whether it has at least one message available per topic.

**Algorithm 1:** Callback selection from `readyQueue`

```
Data: readyQueue, mutex
if lock(mutex) = success then
    refresh(readyQueue);
    skippedGroups ← [];
    foundExecutable ← false;
    iter ← readyQueue.iter();
    while !iter.empty() && !foundExecutable do
        executable ← next(iter);
        if executable.group in skippedGroups then
            continue;
        end
        if !executable.group.can_be_run() then
            // can_be_run() is false if the group is mutually
            exclusive, and another callback instance is
            running. It is always true for reentrant groups
            skippedGroups.append(executable.group);
            continue;
        end
        readyQueue.remove(executable);
        foundExecutable ← true;
        break;
    end
    unlock(mutex);
    if foundExecutable then
        executable.run();
    end
end
```

**Algorithm 2:** Updating the `readyQueue`

```
Data: readyQueue, callbacks
for callback ← callbacks do
    if not callback.ready then
        continue;
    end
    if callback instance in readyQueue then
        update position;
    else
        add the callback instance to the queue;
    end
end
```

instances of the same callback have the same scheduling parameters. Once an executor removes a callback instance from the `readyQueue`, another instance of the callback will re-enter the queue the next time an executor updates the queue (if another callback instance exists). Algorithm 2 presents the pseudo-code related to the `readyQueue` updating.

Depending on the DDS configuration, published messages may not immediately appear in the ready queue, even though they are refreshed during callback selection. By default, ROS 2 DDS runs in *asynchronous* mode, where message transport happens in a separate thread. If a message is published at the end of a callback, the DDS thread running in the background may not complete before the executor threads poll the `readyQueue`. To ensure that recent publications always appear on the `readyQueue`, the DDS must be set to *synchronous* mode, which causes calls to publish to block until the message is ready to be processed.

TABLE II: Thread Interleave: a race condition resulted in priority inversion (for ease of presentation in the table, we use $\{c_1, c_2, c_3\}$ as mutually exclusive callbacks without matching notion for callback defined earlier)

| Threads | | `readyQueue` |
|---|---|---|
| $\pi_1$ | $\pi_2$ | ($1^{st}$ $c_i$ is the head of queue) |
| - | - | $[\mathbf{c_1}, c_2, c_3]$ |
| $(\uparrow)c_1$ | - | $[\mathbf{c_2}, c_3]$ |
| $c_1$ | - | $[c_2, \mathbf{c_3}]$ |
| $c_1$ | - | $[\mathbf{c_2}, c_3]$ |
| $c_1(\downarrow)$ | - | $[c_2, \mathbf{c_3}]$ |
| – | $c_3$ | $[\mathbf{c_2}]$ (priority inversion!) |

**(iii) Preventing Priority Inversion from Race Conditions.** During callback selection, additional steps are required to avoid priority inversion (where a lower-priority task is incorrectly selected over a higher-priority task). We illustrate how race conditions can occur and how to prevent priority inversion via a toy example. Let us consider a thread-interleaving diagram for the race condition presented in Table II, where the status of the search of `readyQueue` is indicated by putting the callback in **bold**. Suppose on a two-thread $(\pi_1, \pi_2)$ system, there are three callbacks $(c_1, c_2, c_3)$ sharing the same resource and thus belong to the same mutually exclusive group; $c_1$ is executing on the thread $\pi_1$, and the other two are in the `readyQueue`. The thread $\pi_2$ searches the `readyQueue` for a callback to run. It reaches the first callback $(c_2)$ in the `readyQueue`, but skips it due to its membership in a currently-executing callback group. During the time instant between checking $c_2$ and $c_3$, the thread $\pi_1$ finishes its callback and sets the callback group to eligible. The thread $\pi_2$ then checks the callback $c_3$ in the `readyQueue`, finds it eligible, and selects it for execution, *even though $c_2$ (who has higher priority) in the `readyQueue` is now also eligible*, preventing the $c_2$ on the `readyQueue` from running. To prevent this racing scenario during callback selection from the `readyQueue`, as a **design principle**, *the executor should skip any callbacks that are part of a callback group that was running at any point during the `readyQueue` search.* Once the executor encounters a blocked callback group, it adds it to a set and skips any callbacks that are part of a group in the set, even if those callbacks are eligible later in the search. This is done using the `skippedGroups` set in Algorithm 1. From this point, the executor can either (i) pick a ready callback that is not part of the callback group, or, (ii) if none exists, restart the `readyQueue` selection process, and pick the highest priority task from that callback group. Note that choosing the first option does not cause priority inversion by selecting a lower-priority task—remember that the thread $\pi_1$ will also be in task selection, and will not skip the callback group.

**Remark 1.** *The callback eligibility defined in our proposed method differs from the one defined for the default multi-threaded executor in [16]. In our proposed `readyQueue`, blocking for a callback due to* a mutually exclusive group membership *is checked only once before dispatching to a thread. Once a callback becomes pending, it will always enter the `readyQueue` in the following update instant. However, in the default `readySet`-based scheduling scheme, there are two ways of blocking a callback from a mutually exclusive callback group. A callback can receive blocking before entering the `readySet` (i.e., P-blocked) as well as after entering* *the `readySet` (i.e., R-blocked).*

**Remark 2.** *Once a callback enters the `readyQueue`, it will remain in the queue until being dispatched to a thread, which implies that the `readyQueue` is built only once. Then, in updating instances, the `readyQueue` needs to update the priority of newly entrant callbacks. However, in the case of `readySet`, it needs to be empty before updating with new callback instances by either dispatching all exiting callbacks to threads or returning them to the wait_set again. Therefore, the maintenance cost of `readyQueue` (e.g., $O(\log n)$) is significantly less than the `readySet` (e.g., $O(n \log n)$); where $n$ is the number of callbacks.*

### B. Dynamic Scheduling Model for Executor

Our proposed executor maintains a `readyQueue` $\Omega$ during runtime to record the dynamic priority of all eligible callbacks. The dynamic priority of a callback $c_{i,j}$ is determined using the absolute deadline of chain $\mathcal{C}_i$; *i.e.*, all callbacks within a chain share the same deadline. For instance, if the arrival time of chain instance $\mathcal{C}_i^k$ is $a_i^k$, then the absolute deadline of the chain instance is $d_i^k = a_i^k + D_i$. Now, any callback $c_{i,j}^k$ (for $1 \le j \le |\mathcal{C}_i|$) will have an absolute deadline of $d_i^k$. A callback with an earlier deadline has a higher priority than the one with a later deadline. In other words, the callback scheduling decisions are determined following the EDF algorithm.

An executor thread is either 'busy' if a callback instance is executing on it, or 'idle' if no callback instance is executing on the thread. A dispatch point occurs whenever a thread becomes idle. At the dispatch point, the $\Omega$ is updated with all pending callbacks. Among the callbacks in $\Omega$, callbacks are checked one by one, following the priority order (*i.e.*, the highest priority one is selected first). The idle thread selects the highest priority callback that is eligible to run. A callback runs non-preemptively as soon as it is selected. A thread *sleeps* if it fails to find a callback, while it can be waked by the release of the next callback, which leads to a repetition of the process.

To update $\Omega$, the executor checks all callback types in the system for eligible callbacks. Any new releases will be placed in $\Omega$ according to the priority provided by the scheduling parameters. Callbacks in the $\Omega$ persist between updates so that the queue does not need to be entirely rebuilt during updates.

Note that not all callbacks in $\Omega$ are eligible to run. Depending on the membership of callback groups, a callback instance $c_{i,j}$ in $\Omega$ is either 'eligible' or 'ready and blocked' (*R-blocked*):

- If the callback $c_{i,j}$ is a member of the reentrant callback group, as soon as $c_{i,j}$ enters $\Omega$, it is *eligible* to run.
- If the callback $c_{i,j}$ is a member of a mutually exclusive callback group, there can be two cases. **Case A:** if there are no other callbacks (including an instance of $c_{i,j}$ itself) from the same mutually exclusive group in $\Omega$ or currently executing in a thread, then the callback becomes *eligible* as soon as it enters $\Omega$. **Case B:** otherwise, the callback $c_{i,j}$ is *R-blocked* and skipped during task selection.

## V. Response Time Analysis

This section presents the response time analysis (RTA) for the ROS 2 processing chains under deadline-based scheduling. We first present the RTA for processing chains without callback groups and then the RTA with callback groups.

### A. RTA without Callback Groups

To avoid deriving the RTA for ROS 2 workloads without callback groups from the first principles, we will directly utilize the existing state-of-the-art (SOTA) analysis for GEDF [31] with fixed preemption points in homogeneous multi-processors. Notably, such usage of existing results was the motivation for our novel executor design of ROS 2. First, we will state the scheduling model, denoted as FPP-GEDF, for a workload with fixed preemption points for each task scheduled on homogeneous multi-processors following the GEDF algorithm. Then, we will prove the equivalence of our proposed ROS 2 scheduling model and FPP-GEDF. We then state the SOTA RTA presented by Zhou *et al.* [31] for FPP-GEDF. Then, we will expand the RTA for ROS 2 workloads with callback groups, which is the focus of this paper.

**FPP-GEDF scheduling model.** A set of $n$ tasks $\mathcal{T} = \{\tau_1, \ldots, \tau_n\}$ with constrained deadlines, where each task has a fixed number preemption point, are scheduled on $m$ homogeneous processors following the GEDF algorithm. If $i^{th}$ task $\tau_i$ has $k$ preemption point, then there are $k+1$ non-preemptive regions in $\tau_i$ which higher-priority tasks cannot preempt once they start executing. In addition, priority is dynamically assigned to each instance of a task, not to each non-preemption region of a task instance.

**Proposition 1.** *FPP-GEDF scheduling model and the proposed ROS 2 scheduling model without considering the callback groups are equivalent.*

*Proof.* We will establish a bijection by mapping the FPP-GEDF scheduling model to the ROS 2 scheduling model and vice versa to prove the equivalence of the scheduling models. **FPP-GEDF to ROS 2.** Each task $\tau_i$ can be mapped as a ROS 2 chain $\mathcal{C}_i$, where each non-preemptive region of $\tau_i$ would work as a callback in $\mathcal{C}_i$. Therefore, if a task $\tau_i$ in FPP-GEDF has $k$ preemption points, then corresponding chain $\mathcal{C}_i$ in ROS 2 has $k+1$ callbacks. Now, $m$ homogenous processors can be mapped to $m$ threads in a ROS 2 executor as each thread is assigned to an individual core. Therefore, the scheduling problem of the workload $\mathcal{T}$ in $m$ processors following global-EDF can directly reduce to the scheduling problem of a set processing chains $\Gamma$ on $m$ threads using GEDF in ROS 2.
**ROS 2 to FPP-GEDF.** Using a similar argument, we can show that the scheduling problem of a set of processing chains $\Gamma$ on $m$ threads using GEDF directly reduces to the problem of a task set $\mathcal{T}$ on $m$ processors using GEDF.

Hence, the scheduling model of FPP-GEDF and ROS 2 processing chains without callback groups are equivalent. □

We will leverage SOTA RTA for FPP-GEDF proposed by Zhou *et al.* [31] for RTA of ROS 2 processing chain without callback groups. First, we report the supporting results in Lemma 1, 2, 3 to use the RTA from [31].

Let us consider the $j^{th}$ instance of chain $\mathcal{C}_k$, $\mathcal{C}_k^j$, as the chain instance under consideration for RTA. As soon as $\mathcal{C}_k^j$ is released at $a_k^j$, the first callback $c_{k,1}^j$ is also released and

becomes eligible. The subsequent callbacks of $\mathcal{C}_k^j$ will become ready once the preceding callbacks complete their execution. Let us define the problem window for $\mathcal{C}_k^j$ for RTA as follows:
**Problem Window.** Given a chain instance $\mathcal{C}_k^j$, denote $t'$ as the start time of the last callback with priority lower than $c_{k,l}$ (for $1 \leq l \leq |\mathcal{C}_k|$) that starts its execution before $a_k^j$, and denote $t''$ as the earliest time instant satisfying that all processors are busy in $[t'', a_j^k)$. Then, a problem window of $\mathcal{C}_k^j$ is $[t_0, t_1)$, where $t_0 = \max\{t', t''\}$ and $t_1 \in [a_k^j + E_k - e_{k,|\mathcal{C}_k|} + 1, d_k^j]$.

Let us denote the problem window for $\mathcal{C}_k^j$ as $S_t^{A_k}$, where $t = t_1 - t_0$ and $A_k = a_k^j - t_0$. We denote a chain as *carry-in* if it releases an instance before $t_0$ and has a deadline after $t_0$; others are *non-carry-in* chains.

Next, we will bound the work done by the *carry-in* and *non-carry-in* chains in the problem window of $S_t^{A_k}$.

**Lemma 1.** [31] *Given a chain instance $\mathcal{C}_k^j$ with a problem window $S_t^{A_k}$, the interference on $\mathcal{C}_k^j$ by any chain $\mathcal{C}_i$ as non-carry-in chain and $i \neq k$ in $S_t^{A_k}$ is upper bounded by $\mathcal{I}_{i,k}^{NC}(t, A_k)$, satisfying following equation,*

$$\mathcal{I}_{i,k}^{NC}(t, A_k) = \begin{cases} \left\lfloor \frac{t}{T_i} \right\rfloor E_i + \min\{t \bmod T_i, E_i\}, \\ \qquad \textit{if } \alpha \leq L \\ \left\lfloor \frac{t}{T_i} \right\rfloor E_i + \min\{\gamma, t - \beta\}, \\ \qquad \textit{if } \alpha > L \textit{ and } \beta < A_k \\ \left\lfloor \frac{t}{T_i} \right\rfloor E_i + \min\{\lambda, t - \beta\}, \\ \qquad \textit{if } \alpha > L \textit{ and } \beta \geq A_k \end{cases} \tag{1}$$

*where $L = A_k + D_k$, $\alpha = \left\lfloor \frac{t}{T_i} \right\rfloor T_i + D_i$, $\beta = \left\lfloor \frac{t}{T_i} \right\rfloor T_i$, $\gamma = \sum_{l=1}^{\min\{|\mathcal{C}_i|, |\mathcal{C}_k|\}} e_{i,l} - \min\{|\mathcal{C}_i|, |\mathcal{C}_k|\} + 1$, and $\lambda = \sum_{l=1}^{\min\{|\mathcal{C}_i|, |\mathcal{C}_k|-1\}} e_{i,l} - \min\{|\mathcal{C}_i|, |\mathcal{C}_k| - 1\}$.*

**Lemma 2.** [31] *Given a chain instance $\mathcal{C}_k^j$ with a problem window $S_t^{A_k}$, the interference on $\mathcal{C}_k^j$ by any chain $\mathcal{C}_i$ as a carry-in chain and $i \neq k$ in $S_t^{A_k}$ upper bounded by $\mathcal{I}_{i,k}^{CI}(t, A_k)$, satisfying following equation,*

$$\mathcal{I}_{i,k}^{CI}(t, A_k) = \begin{cases} \mathbb{A} + \mathbb{B}; & \textit{if } \alpha \geq 0 \textit{ and } \beta \leq L \\ \max\{\mathbb{C}, \mathbb{D}\}; & \textit{if } \alpha \geq 0 \textit{ and } \beta > L \\ \min\{t, E_i\}; & \textit{if } \alpha < 0 \textit{ and } \gamma \leq L \\ \max\{\mathbb{E}, \mathbb{F}\}; & \textit{if } \alpha < 0 \textit{ and } \gamma > L \end{cases} \tag{2}$$

*here* $L = A_k + D_k$; $\alpha = t - E_i - T_i + R_i$;
$\beta = E_i + T_i - R_i + \left\lfloor \frac{\alpha}{T_i} \right\rfloor T_i + D_i$; $\gamma = E_i + D_i - R_i$;
$\mathbb{A} = (\lfloor \alpha / T_i \rfloor + 1) \cdot E_i$; $\mathbb{B} = \min\{\alpha \bmod T_i, E_i\}$;
$\mathbb{C} = \mathbb{A} + \min\{\sum_{l=1}^{\min\{|\mathcal{C}_i|, |\mathcal{C}_k|\}} e_{i,l} - \min\{|\mathcal{C}_i|, |\mathcal{C}_k|\} + 1, \alpha \bmod T_i\}$;
$\mathbb{D} = \lfloor (L - D_i)/T_i \rfloor E_i + \min\{T_i - L + t, E_i\} + \max\{(L - D_i) \bmod T_i - T_i + R_i, 0\}$;
$\mathbb{E} = \max\{\min\{L - D_i + R_i, t\}, 0\}$; *and*
$\mathbb{F} = \min\{\sum_{l=1}^{\min\{|\mathcal{C}_i|, |\mathcal{C}_k|-1\}} e_{i,l} - \min\{|\mathcal{C}_i|, |\mathcal{C}_k| - 1\}, t\}$.

So, by Lemma 1 and Lemma 2, we get the non-carry-in and carry-in interference from any chain $\mathcal{C}_i$ ($i \neq k$) on $\mathcal{C}_k^j$ in $S_t^{A_k}$. Now, the following lemma will bound non-carry-in and carry-in interferences from the instances of $\mathcal{C}_k$.

**Lemma 3.** [31] *Given a chain instance $\mathcal{C}_k^j$ with a problem window $S_t^{A_k}$, the non-carry-in interference and carry-in interference on $\mathcal{C}_k^j$ by $\mathcal{C}_k$ upper bounded by $\mathcal{I}_{k,k}^{NC}(t, A_k)$ and $\mathcal{I}_{k,k}^{CI}(t, A_k)$, respectively,*

$$\begin{aligned} \mathcal{I}_{k,k}^{NC}(t, A_k) &= \mathcal{I}_{k,k}^{CI}(t, A_k) \\ &= \max\{\min\{A_k - T_k + R_k, e_{k,|\mathcal{C}_k|}\}, 0\} \end{aligned} \tag{3}$$

**Lemma 4.** [31] *Given a ROS 2 workload $\Gamma$ scheduled on $m$-threads in an executor using deadline-based* `readyQueue` *and a chain instance $\mathcal{C}_k^j$ with a problem window $S_t^{A_k}$, the non-carry-in and carry-in interference on $\mathcal{C}_k^j$ by any chain $\mathcal{C}_i$ in $S_t^{A_k}$ are upper bounded by $FI_{i,k}^{NC}(t, A_k)$ and $FI_{i,k}^{CI}(t, A_k)$, respectively,*

$$FI_{i,k}^{NC}(t, A_k) = \min\{\mathcal{I}_{i,k}^{NC}(t, A_k), t - E_k + e_{k,|\mathcal{C}_k|}\} \tag{4}$$

$$FI_{i,k}^{CI}(t, A_k) = \min\{\mathcal{I}_{i,k}^{CI}(t, A_k), t - E_k + e_{k,|\mathcal{C}_k|}\} \tag{5}$$

Now, we can calculate the total inference from all carry-in and non-carry-in chains on $\mathcal{C}_k^j$ in $S_t^{A_k}$. Let $FI_{i,k}^{\mathsf{diff}}(t, A_k) = \max(FI_{i,k}^{CI}(t, A_k) - FI_{i,k}^{NC}(t, A_k), 0)$ and $F(t, A_k, x)$ as the sum of the first $x$ items of non-increasing order of $FI_{i,k}^{\mathsf{diff}}(t, A_k)$ for all $\mathcal{C}_i$. Then following are two upper bound of the interferences, $\Psi_1(t)$ and $\Psi_2(t)$, on $\mathcal{C}_k^j$ by all chains in $\Gamma$,

$$\Psi_1(t) = \sum_{\forall \mathcal{C}_i \in \Gamma} FI_{i,k}^{NC}(t, A_k) + F(t, A_k, m - 1) \tag{6}$$

$$\begin{aligned} \Psi_2(t) = & m \cdot A_k + \\ & \sum_{i \neq k} \max\{FI_{i,k}^{CI}(t - A_k, 0), FI_{i,k}^{NC}(t - A_k, 0)\} \end{aligned} \tag{7}$$

Now, the response time of chain $\mathcal{C}_k$ can be determined using the following theorem:

**Theorem 1.** [31] *Given a ROS 2 workload $\Gamma$ to be scheduled on a $m$-threaded executor following EDF (without considering the callback groups among callbacks), the last callback of any chain instance $\mathcal{C}_k^j$ with a problem window $S_t^{A_k}$ must be executed before $a_k^j + t'$, where $t'$ is the minimum solution of,*

$$E_k - e_{k,|\mathcal{C}_k|} + 1 + \left\lfloor \frac{\min\{\Psi_1(x + A_k), \Psi_2(x + A_k)\}}{m} \right\rfloor \leq x + A_k \tag{8}$$

*Then, the WCRT of $\mathcal{C}_k$ is,*

$$R_k = t' + e_{k,|\mathcal{C}_k|} - 1 \tag{9}$$

### *B. RTA with Callback Groups*

Due to the presence of callback groups and the prevention of concurrent execution of callbacks from a mutually exclusive group, an additional blocking (for a mutually exclusive callback group) must be considered in the RTA.

Let us first derive the maximum blocking received by a callback solely for the membership in a mutually exclusive callback group,

**Lemma 5.** *A callback $c_{k,j} \in \mathcal{C}_k$ of a mutually exclusive callback group with index $\mathcal{G}(c_{k,j}) \neq 0$ can receive a maximum blocking of $\max_{\forall \mathcal{G}(c_{i,l}) = \mathcal{G}(c_{k,j})}\{e_{i,l}\}$, where callback $c_{i,l}$ from any chain $\mathcal{C}_i \in \Gamma \setminus \mathcal{C}_k$.*

*Proof.* Following the `readyQueue` design, a callback only experiences the blocking from other members of a mutually exclusive callback group by the 'R-blocked' state. As an 'R-blocked' callback can be selected to execute as soon as the currently executing callback (that is also a member of the same mutually exclusive callback group), the maximum blocking due to the member of a mutually exclusive callback is equal to the maximum execution time of a callback in that group. Note that if there exist callbacks in a mutually exclusive group with publisher-subscriber relation (i.e., from the same callback chain), then the additional blocking due to precedence constraint for those callbacks is not required to take in the account as these callbacks cannot be ready at the same time. However, the Theorem 1 already includes blocking for precedence constraints. Therefore, the maximum blocking of a $c_{i,j}$ callback from a mutually exclusive group callback is by the one that is not in the same callback chain $\mathcal{C}_i$. □

Let $\mathcal{I}_k^X$ be the total blocking received by the callbacks of chain instance $\mathcal{C}_k^j$. Using Lemma 5,

$$\mathcal{I}_k^X = \sum_{1\leq j\leq |\mathcal{C}_k| \wedge \mathcal{G}(c_{k,j})\in\theta_k} \max_{\forall \mathcal{G}(c_{i,l})=\mathcal{G}(c_{k,j})} \{e_{i,l}\} \tag{10}$$

where, callback $c_{i,l}$ can be from any chain $\mathcal{C}_i$.

Finally, we state the following theorem for ROS 2 processing chains scheduling on a $m$-threaded executor with mutually exclusive callback groups.

**Theorem 2.** *Given a ROS 2 workload $\Gamma$ to be scheduled on a $m$-threaded executor following EDF, the last callback of any chain instance $\mathcal{C}_k^j$ with a problem window $S_t^{A_k}$ must be executed before $a_k^j + t'$, where $t'$ is the minimum solution of,*

$$E_k - e_{k,|\mathcal{C}_k|} + 1 + \mathcal{I}_k^X + \left\lfloor \frac{\min\{\Psi_1(x+A_k), \Psi_2(x+A_k)\}}{m} \right\rfloor \leq x + A_k \tag{11}$$

*Then, the WCRT of $\mathcal{C}_k$ is given by*

$$R_k = t' + e_{k,|\mathcal{C}_k|} - 1 \tag{12}$$

*Proof.* The proof of the theorem follows a similar approach for Theorem 1 except for the inclusion of blocking due to the mutually exclusive callback groups. Note that the effective blocking received by chain $\mathcal{C}_k$ for $m$-threads is $m \cdot \mathcal{I}_k^X$, as in the worst case, even if $m-1$ threads are idled, and one thread is executing one callback from the group, others cannot execute. So total blocking added in the L.H.S. of Equation (12) is $m \cdot \mathcal{I}_k^X / m = \mathcal{I}_k^X$. □

It is obvious that for a schedulability check of the workload $\Gamma$, one must verify the WCRT of each processing chain is on greater than the deadline. I.e., a ROS 2 workload $\Gamma$ is schedulable on an $m$-threaded executor following EDF if the following inequality holds for any chain $\mathcal{C}_i$: $R_i \leq D_i$; $\forall i$, where $R_i$ is given by Equation (12).

**RTA for chains span over multiple executors.** To compute the response time of processing chains spanning over multiple executors, one potential way could be a similar approach used in the existing single-threaded works [8], [9], [2], where the RTA for each executor is computed independently. Then, the end-to-end response time is computed using the Compositional Performance Analysis tool [15] and adding communication latencies for every two consecutive executors where the chain segments are executed. However, further studies are needed on whether these approaches can directly be extended to multithreaded executors, which we have left for future work.

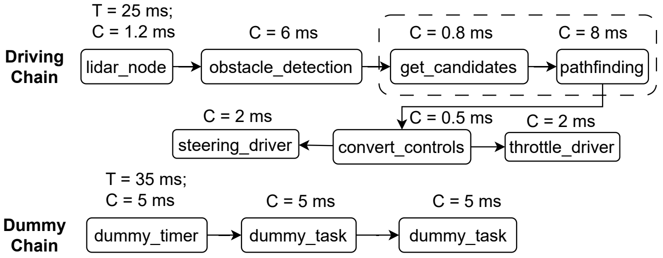


Fig. 3: *Layout of the workloads used in the experiment with F1Tenth car. Each box is a callback. In the driving chain, each callback is in its own node, except `get_candidates` and `pathfinding`, which share a node. `convert_controls` splits the control output from `pathfinding` into two messages, a steering and acceleration message, which is sent to the appropriate hardware driver nodes. The chain is considered complete once both `steering_driver` and `throttle_driver` have completed. Besides the driving chain, we used two dummy chains with similar configurations.*

## VI. Evaluation

In this section, we present the evaluation of our proposed executor using on-board case studies, overhead analysis, and schedulability test of response time analysis for the dynamic scheduler and compare it against two existing analyses for default ROS 2 executor.

### *A. On-Board Case Studies*

We run our case studies on an Nvidia Jetson Xavier AGX in MAXN mode, where the main frequency of all CPU cores is fixed at 2.2 GHz. Executor threads are set to run using the SCHED_FIFO class at the highest priority (99). For multithreaded executors, each thread is pinned to a unique CPU core. Other implementation details can be found in Section IV . The workloads are controlled to run no longer than their specified WCETs.

*1) Case Study 1:* To show a real-world use case, we use ROS 2 executor to schedule tasks that drive an F1Tenth car. **Experimental Setup.** We use our modified ROS 2 executor implementation to schedule a taskset that drives the F1Tenth car around a track. Nodes in the system poll a LIDAR sensor, process the incoming LIDAR data, make driving decisions, and pass actions to motor controllers. Together, the callbacks in these nodes form a chain, which we refer to as the driving chain, as shown in Fig. 3. Each callback is in its own mutually-exclusive callback group. The driving chain and dummy chains represent most of the load on the system, but some auxiliary tasks exist as well, which produce odometry output and other system statistics. These auxiliary tasks have a collective utilization of 0.07. The auxiliary tasks are configured as fixed priority tasks, where deadline tasks always take precedence.

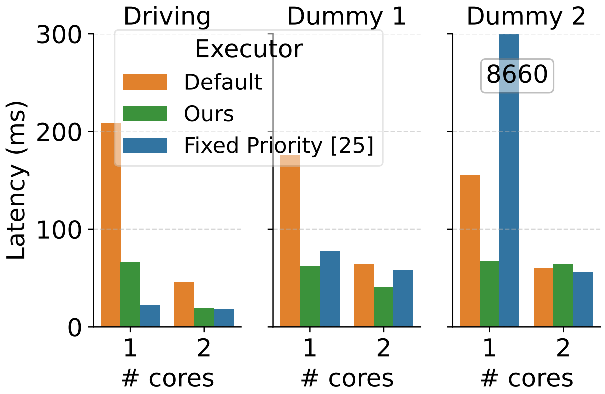


Fig. 4: *Average and maximum latencies for each chain in the case study system. We tested the default executor and our executor with 1 and 2 threads. The driving chain performed better under our executor compared to the default, especially in single-core mode, where the system is overloaded. In the overloaded single thread case under the fixed priority executor, the maximum latency of the second dummy chain was* 8660 $ms$*, due to the second dummy chain having the lowest priority in the system.*

We ran this test with two dummy chains  to increase the utilization of the system. Each chain uses implicit deadlines, so the driving chain has a deadline of 25 $ms$, and the dummy chains have a deadline of 35 $ms$. Running the system with the modified executor decreases the average and maximum latency of the main driving chain, and improves the latency of the dummy chains in an overload scenario.

**Observations.** Using the modified executor, we observed improved response time of the driving chain in both the one and two-core tests and all three chains in the single-core tests (Fig. 4). In the two-core test, the driving chain had a maximum latency of 46.16 $ms$ on the default executor and 19.23 $ms$ on our executor. With fixed priorities, the driving chain had a worst-case latency of 17.25 $ms$.

In the single-core case, where the system is overloaded, the driving chain had a maximum latency of 208.19 $ms$ on the default executor and 66.48 $ms$ on ours. When running under the fixed-priority executor, the second dummy chain was frequently blocked by the driving chain and first dummy chain, and had a maximum response time of 8.66 seconds.

We also ran a single core test with just the driving chain and the auxiliary system tasks. In this case, the driving chain had a maximum latency of 21.43 $ms$ on the default executor, and 20.00 $ms$ on ours. This improvement comes from the fact that our executor will not preempt the driving chain to service callbacks from auxiliary tasks.

*2) Case Study 2:* We use the same workload defined in [16], which inspired our earlier discussion on callback group concurrency bugs. The workload is presented again in Table III. All chains are placed in a single mutually exclusive callback group, ensuring that only one callback, and therefore one thread, can execute at any time.

**Experimental Setup.** We ran two tests: one with the ROS 2 multithreaded executor (using two threads), and another with the single-threaded executor (using one thread), both modified with our task selection process. The Fixed-Priority and EDF schedulers always refresh the `readyQueue` before selecting a callback to run and, therefore, behave the same within both the single-threaded and multithreaded executors, meeting deadlines in both situations.

TABLE III: Case Study 2

| $C_i$ | $T_i = D_i$ | $c_{i,j}$ | Callback Group | WCET | Priority |
|---|---|---|---|---|---|
| $C_1$ | 100 | $c_{1,1}$ | M1 | 50 | 1 |
| $C_2$ | 150 | $c_{2,1}$ | M1 | 60 | 2 |
| $C_3$ | 900 | $c_{3,1}$ | M1 | 50 | 3 |

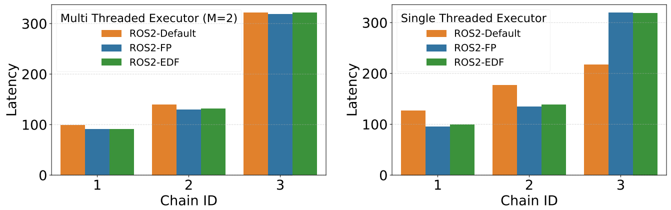


Fig. 5: *Demonstration of a weakness of ROS 2's default multithreaded executor. The callback group assignments only allow one thread to perform work at any given time. The same workload performs worse in the default multithreaded executor than in the default single-threaded executor (although intuitively and theoretically, they should perform the same, as all callbacks are in the same group). The Fixed Priority and Deadline based schedulers, which refresh the* `readyQueue` *before every callback execution, behave similarly in single-threaded and multithreaded mode.*

**Observations.** The results are shown in Fig. 5. The default scheduler behaves differently due to the fact that the default multithreaded executor will clear the `readySet` if none of the callbacks are eligible to run due to membership in callback groups.

To understand how this affects execution, assume all three callbacks have been released. The default multithreaded executor runs $c_{1,1}$ on Thread 1. During this time, Thread 2 attempts to find an eligible callback to run, but cannot as $c_{2,1}$ and $c_{3,1}$ are both in the same mutually exclusive callback group as $c_{1,1}$. Thread 2 clears the `readySet`, and since no callbacks are eligible to run, the `readySet` remains empty. This continues until $c_{1,1}$ completes, making $c_{2,1}$ and $c_{3,1}$ eligible to run. Since the `readySet` is now empty, a thread (whichever takes the mutex lock first) refreshes the `readySet` and places $c_{2,1}$ and $c_{3,1}$ back. This cycle repeats with $c_{2,1}$ instead of $c_{1,1}$. During $c_{2,1}$'s execution, $c_{1,1}$ is released again. Since the idle thread clears the `readySet`, the `readySet` gets rebuilt with one of $c_{1,1}$ and $c_{2,1}$ taking priority over $c_{3,1}$.

In the single-threaded executor, ineligible callbacks are not removed from the `readySet`, and there is no second thread to refresh the `readySet`, so after $c_{1,1}$ and $c_{2,1}$ starts to run, $c_{3,1}$ is the only item in the `readySet`, even though $c_{1,1}$ may have been released again during $c_{2,1}$. Only after $c_{3,1}$ runs, does the executor refresh the `readySet`.

The Fixed Priority and Deadline-based executors avoid this problem by 1) storing ready callbacks in a queue, and 2) keeping callbacks in the queue, even if they are not immediately runnable due to membership in a mutually exclusive callback group. In this case, like the default multithreaded

TABLE IV: Asymptotic overhead for Queue/Set refresh and callback selection for different executors

| Executor | Refresh | Select Best Case | Select Worst Case |
|---|---|---|---|
| Default | $O(n)$ | $O(1)$ | $O(n)$ |
| Fixed Priority [25] | $O(n)$ | $O(n)$ | $O(n)$ |
| Ours | $O(n \log n)$ | $O(1)$ | $O(n)$ |

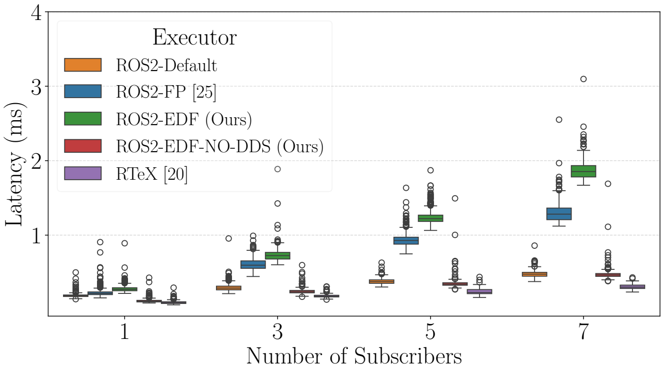


Fig. 6: *End-to-end latency of multiple subscribers on a single topic. Each subscriber has an execution time of* $0$ $ms$*, so the effects of executor are evident in the end-to-end times.*

executors, only one thread can be running a callback at any given time, but the idle thread does not manipulate the `readyQueue`, except for when the timers release, where it simply adds the released callback to the queue. When the working thread finishes executing the callback, either thread (whichever takes the lock first) will perform another check for newly-released callbacks, and select the callback with the highest priority or earliest absolute deadline. By not clearing the `readyQueue`/`readySet`, our modified executor behaves more consistently than the default executors when running in single-threaded and multi-threaded modes, preventing deadline misses, which can occur when using the default executor.

### *B. Overhead Analysis*

Table IV reports the asymptotic overhead of executor's queue/set refresh, and callback selection for ours, default ROS 2, and existing fixed-priority [24] executors.

To empirically measure the overhead of our modified executor, we compare it to existing works by Sobhani *et al.* [24] (denoted as 'Fixed Priority') and RTeX [19], and default ROS 2 executor. We measure the end-to-end latency of a system with a timer callback publishing to multiple subscriber callbacks. To accurately represent the effects of the different executors, we use the publicly-available implementations of [24], [19].

The Fixed Priority executor selects callbacks by searching the `readySet` for the highest-priority eligible callback. The RTeX executor removes the locks held during the queue refresh step, and replaces the `readySet` with a concurrent linked-list. The RTeX executor is unique in that immediately after a callback is run, the executor adds the subsequent callback in the chain to the queue directly, avoiding the need to refresh the queue and poll the DDS layer. This significantly decreases the overhead of the RTeX executor, but at the expense of DDS compatibility. To support receiving messages from other processes or over the network, users of the RTeX executor need to use an additional thread to listen for incoming messages and add them to the queue. For a fair comparison with RTeX, we also test a variant of our executor (EDF-NO-DDS), which updates the queue similarly to RTeX, where published messages are placed directly into the `readyQueue`, removing the need for queue refreshes.

*Workloads.* We take the test parameters from [19]. Each callback has an execution time of $0$ $ms$, and the end-to-end latency is the time between timer releases and the completion of the last subscriber callback. Our test uses 2 threads. Because the callbacks themselves do not perform any work and only publish to the next callback in the chain, the end-to-end latency reflects the time taken to receive, sort, and select the callbacks.

*Observations.* We show the results of this test in Fig. 6. Since no callback groups exist, the default executor and our executor always exhibit the best-case callback selection performance. Due to the extra overhead in queue refreshes, and a refresh is always performed before each callback selection, our executor's response time increases quickly as more callbacks are added to the system. The NO-DDS version of our executor is competitive with the default executor and RTeX.

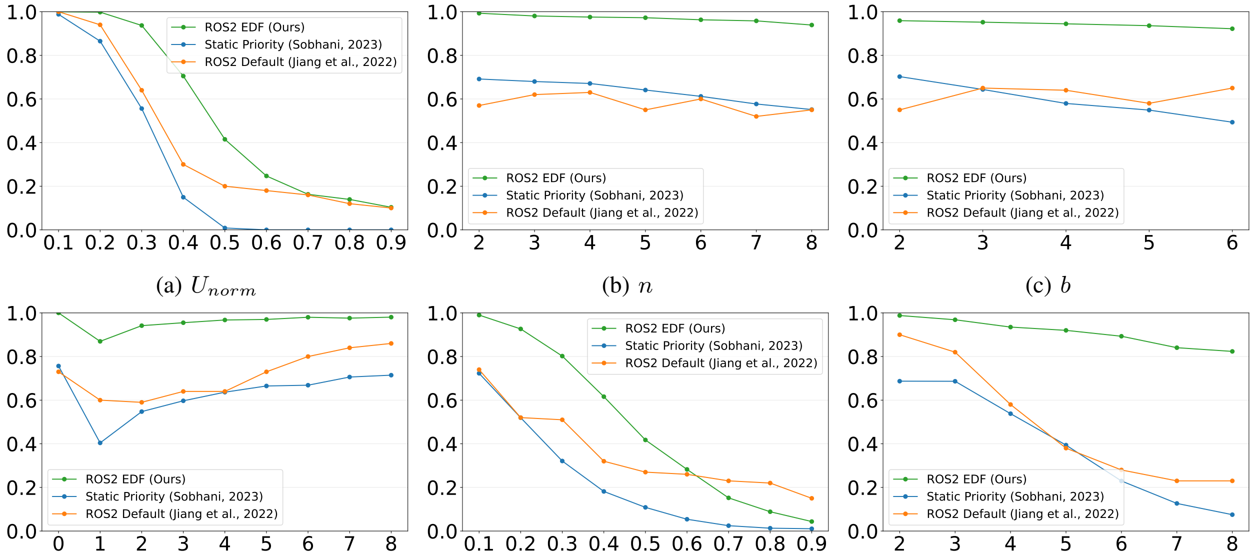
The additional work required during the queue refreshes means that our modified executor has a larger overhead, especially as the number of callbacks in the system increases, but the case study demonstrates that using the `readyQueue` and dynamic priorities allows the executor to make decisions that reduce the overall system latency.

**Compatibility of executor with default ROS 2 architecture.** Our modified executor is implemented as a ROS 2 package, and does not outright replace the default ROS 2 multithreaded executor. Instead, it uses subclasses of existing data structures, so it does not interfere with packages that rely on the default ROS 2 data structures and classes. The package can be placed in any ROS 2 workspace and called from user code when required. It does not change any of the existing data structures in `rclcpp` or `rmw`, and does not require any modification of the DDS layer, allowing the use of both open-source and proprietary DDS systems.

Not all callbacks need to have explicitly declared deadlines, but callbacks without deadlines are always given a lower priority than callbacks with deadlines.

### *C. Schedulability Evaluation via Synthetic Workload*

**Experimental Setup.** We use the workload parameters from [16]. Workloads are randomly generated from parameters: $m$: the number of threads the workload will be run on, $n$: the maximum number of chains in the workload, $b$: the maximum number of callbacks in any chain, $U_{norm}$: the utilization of the workload, $g$: the maximum number of mutually exclusive callback groups, and $\alpha$: the ratio of callbacks that will be members of a mutually exclusive callback group. The total utilization of the workload is $m \cdot U_{norm}$. The utilization of each chain is found with UUnifast-discard. Chain utilizations above 1 are set to 1. For each chain, generate the utilization of each callback with UUnifast-discard. Each chain's period is randomly selected from $[50, 200]$. The chain's period is also its deadline. Each callback's WCET is the chain's period multiplied by the callback's utilization. Callback WCETs are rounded to the nearest integer. Chains not in a mutually



(a) $U_{norm}$ (b) $n$ (c) $b$

(d) $g$ (e) $\alpha$ (f) $m$

Fig. 7: *Schedulability ratio (percentage of schedulable tasksets) comparisons by varying one parameter at a time.*

exclusive callback group are assigned to their own reentrant callback group. The number of groups in the workload is randomly chosen from $[0, g]$, and the number of callbacks in any group is $|C| \cdot \alpha$. We randomly select $|C| \cdot \alpha$ callbacks, and distribute them to the callback groups.

We compare our schedulability test with the test given in [16] and [24]. The workload parameters are $m = 4$, $n = 8$, $b = 5$, $U_{norm} = 0.3$, $g = 2$, and $\alpha = 0.2$. We ran 2000 task sets per data point.

**Observations.** Fig. 7 shows how varying each parameter affects the percentage of schedulable tasksets for the deadline-based, static priority, and default ROS 2 executors. For most situations, the deadline-based analysis schedules more workloads than the default ROS 2 executor by a significant margin. Varying $U_{norm}$ (Fig. 7a) results in expected behavior—workloads with higher utilization are less likely to be schedulable. Increasing $n$ (Fig. 7b), the maximum number of callbacks in a workload, caused a slight decrease in schedulability for each executor. The default executor was largely invariant to changes in $b$ (Fig. 7c), the maximum number of callbacks in any chain. This is likely due to the fact that the default ROS 2 executor tries to make progress along all running chains. In contrast, the priority-based executors will run a higher priority chain to completion at the cost of blocking others. The results in Fig. 7d are best understood when remembering that the ratio of callbacks that are within some group compared to those not in any group is constant ($\alpha = 0.2$). The exception is that when $g$ is 0, there are no mutually exclusive callback groups, and no callback group blocking can occur. When $g$ is 1, 20% of the callbacks are in one mutually exclusive group, so the chances of callbacks blocking each other are high. As the number of mutually exclusive groups increases, there is a smaller chance that any two callbacks will block each other. Cases where more than 60% of callbacks are in a mutually exclusive group are an exception—the analysis of the default ROS 2 multi-threaded executor by [16] handles these cases especially well, as shown in Fig. 7e.

## VII. Related Works

Earlier works related to the ROS mostly focused on improving the real-time performance [21], [30], [20]. Satio *et al.* [20] developed a priority-based message transmission algorithm for publishers to send data to multiple subscribers; [14] performed an empirical study and measured WCRT between nodes for ROS 2; [30] proposed RT-ROS to run two OS—one for non-real-time tasks and another for real-time tasks.

Several works have been done analyzing and improving the performance of ROS 2's executor system following the pioneering work of Casini *et al.* [8]. [8] first formally modeled the ROS 2 executor scheduling policies and figured out the unique scheduling strategy of ROS 2. [8] also developed the first response time analysis of ROS 2 processing chains. Later, Tang *et al.* [27] improved the previous analysis by observing the properties of polling points and processing windows of default ROS 2 executor. Blaß *et al.* [7] further improved the response time, exploiting the execution time uncertainties and starvation properties of ROS 2 callbacks. Teper *et al.* [28] developed end-to-end response-time analysis for ROS 2 considering the data age and reaction time between sensor outputs and actuation. Tang *et al.* [26] presented the analysis modeling ROS 2 workload as the directed-acyclic-graph (DAG) workload model. All these works model the ROS 2 workload using default priority orders and types of callbacks. Choi *et al.* [9] added unique priorities to each processing chain and the callbacks instead of using the default priority order among callbacks. They also designed a static callback-thread assignments policy. [9] demonstrated that designing fixed-priority orders among callbacks reduces the self-blocking of a processing chain by its past and future instances and improves the processing chains' response time.

Recent works [16], [24] presented the scheduling model and analysis frameworks for multi-threaded ROS 2. Their works demonstrated significant differences between the single- and multi-threaded scheduling policies, mainly for adding complexities for multiple threads and introducing callback groups. Sobhani *et al.* [24] further enhanced the callbacks with a fixed-priority order similar to PiCAS [9] to further improve the timing performance. Compared with existing works, our work falls under the customized multi-threaded ROS 2 executor. We present a modified executor to support a priority-based scheduler without breaking the key properties of ROS 2. However, earlier, Arafat *et al.* [2] presented the modified single-threaded executor for dynamic-priority-based scheduling. Compared with this work, designing a multi-threaded executor involves more challenges than a single-threaded one, such as issues related to the callback groups necessitating careful 'update policy design', concurrency and/or racing bugs that only exist for a multi-threaded one.

In real-time scheduling, many works analyzed processing chains such as [4], [22], [23] and non-preemptive scheduling such as [3], [12]. However, our scheduling model falls under the limited preemptive scheduling, and existing works [29], [31], [5] analyzed schedulability for limited preemptive scheduling problems. Our analysis is based on [31].

Besides the scheduling analysis of ROS 2 executor, Blaß *et al.* [6] discussed the benefits, challenges, and opportunities related to ROS 2; Li *et al.* [17] analyzed timing disparity between messages. Moreover, Suzuki *et al.* [25] developed ROS extension on CPU/GPU mechanism, and Li *et al.* [18] developed a real-time ROS 2 GPU management framework.

## VIII. Conclusion

This paper presented the design, implementation, and analysis of a dynamic-priority-driven scheduler for a multi-threaded ROS 2 executor. Our proposed executor has the flexibility to support user-defined scheduling schemes. With such freedom, one can easily develop a formal timing verification method to verify the timing correctness of the to-be-implemented scheduler by leveraging the rich existing schedulability results. Specifically, we developed an efficient queue updating policy for ready callbacks and callback selection policies for dispatching to threads without priority inversion. Finally, we developed a response time analysis for non-preemptive callback scheduling using the EDF algorithm and implemented it via both case studies and synthetic workload. We compared our response time analysis with the default ROS 2 executor and another priority-enhanced executor, finding that ours allows for schedulable workloads. We believe our modified executor design opens the door to designing more efficient middleware, allowing ROS 2 to adapt standard real-time scheduling models, enabling existing results to be used ROS 2 systems.

**Limitations and Challenges.** By checking for new callback releases before all selections, our modified executor adds additional overhead compared to the default executor. Users of our modified executor must carefully select deadline values in order to ensure safe behavior of the system. It is the user's responsibility to declare callback chains, and determine appropriate deadlines for each.

Since ROS 2 already supports changing the executor behavior by using a subclass of `rclcpp::Executor`, the modified executor could be added as a component of `rclcpp`, or added as a separate optional package. Our executor adds additional complexity to the executor implementation, so inclusion in the default ROS 2 distribution could add work to documentation, testing, and maintenance tasks. Including our executor in the default ROS 2 distribution is made easier by the fact that our executor does not require changes to the existing data structures in `rclcpp`.

## References

[1] ROS 2 Documentation. https://docs.ros.org/en/foxy/index.html.

[2] A. A. Arafat, S. Vaidhun, K. M. Wilson, J. Sun, and Z. Guo. Response time analysis for dynamic priority scheduling in ros2. In *DAC*, 2022.

[3] S. K. Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 2006.

[4] M. Becker et al. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 2017.

[5] M. Bertogna and S. Baruah. Limited preemption edf scheduling of sporadic task systems. *TII*, 2010.

[6] T. Blaß et al. Automatic latency management for ros 2: Benefits, challenges, and open problems. In *RTAS*. IEEE, 2021.

[7] T. Blaß et al. A ros 2 response-time analysis exploiting starvation freedom and execution-time variance. In *RTSS*. IEEE, 2021.

[8] D. Casini et al. Response-time analysis of ros 2 processing chains under reservation-based scheduling. In *ECRTS*, 2019.

[9] H. Choi, Y. Xiang, and H. Kim. Picas: New design of priority-driven chain-aware scheduling for ros2. In *RTAS*. IEEE, 2021.

[10] Eclipse Foundation. Eclipse cyclone DDS™. https://projects.eclipse.org/projects/iot.cyclonedds.

[11] eProsima. eProsima fast DDS. https://github.com/eProsima/Fast-DDS.

[12] N. Guan et al. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *RTSS*. IEEE, 2008.

[13] GurumNetworks. GurumDDS. https://gurum.cc/gurumdds_rmw_eng.

[14] C. S. V. Gutiérrez et al. Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications. *arXiv preprint arXiv:1809.02595*, 2018.

[15] R. Henia et al. System level performance analysis–the symta/s approach. *IEE Proceedings-Computers and Digital Techniques*, 2005.

[16] X. Jiang et al. Real-time scheduling and analysis of processing chains on multi-threaded executor in ros 2. In *RTSS*. IEEE, 2022.

[17] R. Li et al. Worst-case time disparity analysis of message synchronization in ros. In *RTSS*. IEEE, 2022.

[18] R. Li et al. Rosgm: A real-time gpu management framework with plug-in policies for ros 2. In *RTAS*. IEEE, 2023.

[19] S. Liu et al. Rtex: an efficient and timing-predictable multi-threaded executor for ros 2. *TCAD*, 2024.

[20] Y. Saito et al. Priority and synchronization support for ros. In *CPSNA*. IEEE, 2016.

[21] Y. Saito et al. Rosch: real-time scheduling framework for ros. In *RTCSA*. IEEE, 2018.

[22] J. Schlatow and R. Ernst. Response-time analysis for task chains in communicating threads. In *RTAS*. IEEE, 2016.

[23] S. Schliecker and R. Ernst. A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In *CODES+ISSS*, 2009.

[24] H. Sobhani, H. Choi, and H. Kim. Timing analysis and priority-driven enhancements of ros 2 multi-threaded executors. In *RTAS*, 2023.

[25] Y. Suzuki, T. Azumi, S. Kato, and N. Nishio. Real-time ros extension on transparent cpu/gpu coordination mechanism. In *ISORC*. IEEE, 2018.

[26] Y. Tang et al. Real-time performance analysis of processing systems on ros 2 executors. In *RTAS*. IEEE, 2023.

[27] Y. Tang, Z. Feng, et al. Response time analysis and priority assignment of processing chains on ros2 executors. In *RTSS*. IEEE, 2020.

[28] H. Teper et al. End-to-end timing analysis in ros2. In *RTSS*. IEEE, 2022.

[29] A. Thekkilakattil et al. Multiprocessor fixed priority scheduling with limited preemptions. In *RTNS*, 2015.

[30] H. Wei et al. Rt-ros: A real-time ros architecture on multi-core processors. *Future Generation Computer Systems*, 2016.

[31] Q. Zhou et al. Response time analysis for tasks with fixed preemption points under global scheduling. *TECS*, 2019.