Response Time Analysis for Dynamic Priority Scheduling in ROS2

Abdullah Al Arafat  
Sudharsan Vaidhun  
Kurt M. Wilson  
Jinghao Sun  
Zhishan Guo
University of Central Florida, USA
Dalian University of Technology, China
University of Central Florida, USA

ABSTRACT

Robot Operating System (ROS) is the most popular framework for developing robotics software. Typically, robotics software is safety-critical and employed in real-time systems requiring timing guarantees. Since the first generation of ROS provides no timing guarantee, the recent release of its second generation, ROS2, is necessary and timely, and has since received immense attention from practitioners and researchers. Unfortunately, the existing analysis of ROS2 showed the peculiar scheduling strategy of ROS2 executor, which severely affects the response time of ROS2 applications. This paper proposes a deadline-based scheduling strategy for the ROS2 executor. It further presents an analysis for an end-to-end response time of ROS2 workload (processing chain) and an evaluation of the proposed scheduling strategy for real workloads.

ACM Reference Format:

1 INTRODUCTION

Over a decade, Robot Operating System (ROS) has been the standard and most popular framework for developing robotics software, mainly for its modularity and composability. However, the first version of ROS was fundamentally limited in terms of real-time capabilities, which eventually necessitated the emergence of ROS2 in 2017. ROS2 got immediate attention from both the autonomous systems industry and academia, due to its capability of providing real-time guarantees through enabling the Distributed Data Service (DDS) communication interface.

As a foundational cornerstone for autonomous and robotic systems, it is essential that the response time of ROS2 workload can be bounded. However, there was no such formal analytical model for ROS2 before the pioneering work of Casini et al. [3]. It presented a model for the default ROS2 scheduler to enumerate the response-time of workloads, the architectural hierarchy of the ROS2 framework, and working principle of the default ROS2 scheduler.

We thank Nan Guan from City University of Hong Kong for fruitful discussions on the paper. This work is supported by NSF grants CNS 1850851, and PPoSS 2028481.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC ’22, July 10–14, 2022, San Francisco, CA, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9142-9/22/07. $15.00
https://doi.org/10.1145/3489517.3530447

Overview of ROS2. Fig. 1 presents the architecture of ROS2. ROS2 applications are typically composed of series of individual nodes distributed in the application layer. Nodes preserve the molecularity of application programs, and communicate with each other through a publish/subscribe mechanism: nodes first publish messages on a topic, and then the topic broadcasts messages to nodes subscribed to the topic. When receiving a messages, nodes invoke callbacks (fundamental programming blocks) to process the message. To implement the publish/subscribe system, ROS2 leverages the underlying data distribution service (DDS), which is an anonymous and asynchronous message passing framework. To deploy a ROS2 application, individual nodes are mapped onto operating system processes. ROS2 uses executors (in the client library) to coordinate the execution of the callbacks of the nodes assigned in a process.

An executor maintains a readySet that stores the ready callbacks assigned to the executor. At each time, the executor selects one callback from the readySet to be executed non-preemptively on the process. The updating strategies of readySet as follows—it is only updated when it becomes empty, and it cannot contain two same callbacks from different instances of a chain simultaneously. In contrast, a timer callback (first callback of a processing chain) can enter the readySet instantly, and it starts execution either immediately or after the completion of a non-preemptively executing callback.

Limitations of default ROS2. Following the properties of readySet, there are two critical issues raised that affect the response time of processing chains. (i) There is no notion of priority among different processing chains in an executor, as any processing chain can enter the readySet directly through activating the timer callback. Therefore, in default ROS2 executor, it is not possible to provide higher priority to any critical chain for better response time. (ii) Any processing chain instance can receive interference from both past and future instances of the same processing chain (self-interference) due to the updating strategy of readySet and the privilege entrance of the timer callback to the readySet.

The limitations of the readySet-based scheduling policy of ROS2 executor lead to longer response time of workloads in ROS2. To eliminate such limitations of readySet-based scheduling, we aim to design a new priority-based scheduling scheme for ROS2 executor. Intuitively, priority-driven scheduling policies (e.g., fixed- and
dynamic-priority-based scheduling) dominate over default readySet-based scheduling, as the former schedulers can mitigate the self-interference of processing chains by providing a unique priority-order to each instance.

**Related Works.** Casini et al. [3] presented the first formal analysis and modeling of ROS2 for bounding the end-to-end latency of ROS2 application. This paper pointed out the peculiar scheduling strategy of the default ROS2 executor. It developed the response time-bound of any ROS2 processing chain that may span multiple executors leveraging the compositional performance analysis tool. Later, Tang et al. [9] published a follow-up paper improving the starvation response time response analysis for ROS2 exploiting the starvation bound. A concurrent work from Blaß et al. [2] developed a better tool. Later, Tang et al. [9] published a follow-up paper improving the critical observations of the default ROS2 executor scheduler and proposed a deadline-driven dynamic-priority-based scheduling for ROS2 executor. It developed the response time-bound of any ROS2 processing chain that may span multiple executors leveraging the compositional performance analysis tool. The system consists of a set of \( N_e \) executors, \( E = \{ E_k \mid 1 \leq k \leq N_e \} \). Each callback is assigned to an executor and the mapping is fixed throughout the runtime of the system. An executor can be either single-threaded or multi-threaded. The same as in [3, 4, 9], this paper considers the **single-threaded executor**.

**Executor.** We consider a set of \( N_e \) executors, \( E = \{ E_k \mid 1 \leq k \leq N_e \} \). Each callback is assigned to an executor and the mapping is fixed throughout the runtime of the system. An executor can be either single-threaded or multi-threaded. The same as in [3, 4, 9], this paper considers the **single-threaded executor**.

**Chain Model.** We represent the workload generated by all the activations of the callbacks as a set of \( N_T \) processing chains, \( \Gamma = \{ \Gamma_i \mid 1 \leq i \leq N_T \} \). Each chain \( \Gamma_i \) consists of a sequence of callbacks,

\[
\Gamma_i = \langle \tau_{j_1}, \tau_{j_2}, \ldots, \tau_{j_{|\Gamma_i|}} \rangle
\]

where, \( \tau_{j_i} \in \tau \) and \( |\Gamma_i| \) is the length of the chain. The starting callback, \( \tau_{j_1} \), of the chain is usually a timer callback, and remaining callbacks, \( \{ \tau_{j_i} | i \in \{2, \ldots, |\Gamma_i|\} \} \) are the regular callbacks.

Each chain \( \Gamma_i \) is represented by the following 3-tuple:

\[
\Gamma_i = (C_{\Gamma_i}, D_{\Gamma_i}, T_{\Gamma_i})
\]

where \( C_{\Gamma_i} \) is WCET of the chain, derived from the sum of the WCET of all callbacks in the chain,

\[
C_{\Gamma_i} = \sum_{\forall \tau_{j_i} \in \Gamma_i} C_{\tau_{j_i}}
\]

\( D_{\Gamma_i} \) and \( T_{\Gamma_i} \) are the relative deadline and the minimum inter-arrival time of the chain, \( \Gamma_j \), respectively. Note that, all callbacks \( \tau_i \in \Gamma_j \) have same absolute deadline instant as the associated chain \( \Gamma_j \).

Now, we define the workload of an executor \( E_k \) as a set of processing (sub)chains\(^1\) in the **executor**.

\[
E_k(\Gamma) = \{ \Gamma_i^k | \Gamma_i^k \in E_k \}
\]

Here, \( \Gamma_i^k \) is either the chain \( \Gamma_j \) or a segment of \( \Gamma_j \) that belongs to the executor \( E_k \) and \( \Gamma_j = \bigcup_{i=1}^{N_T} \Gamma_i^k \). We further denote the relative deadline of \( \Gamma_i^k \) as \( D_{\Gamma_i^k} \) which either equal to \( D_j \) or derived from \( D_j \) using the fact that all callbacks in a chain have same absolute deadline.

**Resource Model.** The threads utilized by the **executors** are scheduled using the Linux scheduler. Linux scheduler works by scheduling the thread with the highest priority, and if there are multiple threads with the same priority, then a specific scheduling policy is followed to determine the thread to schedule. In our model, all ROS2 threads will be preemptively scheduled with the same (highest) static priority and follow the sched_fifo policy. The budget guarantees are provided to the threads by implementing them as constant bandwidth server.

**Communication Overhead.** The communication among the callbacks adds delay (overhead) in addition to the execution time of callbacks. However, we neglect the communication overhead if the callbacks in a communication link are in the same executor. In case

\(^1\) (sub)chain can be either a chain or a segment of a chain
two communicating nodes are in two executors, we add a fixed communication delay between $k^{th}$ ($\tau_k \in E_k$) and $\ell^{th}$ ($\tau_\ell \in E_\ell$) callbacks as follows:

$$d(\tau_k, \tau_\ell) = \begin{cases} 0, & \text{if } E_k = E_\ell \\ y, & \forall \ell : E_k \neq E_\ell \end{cases}$$

**Overload Handling Mechanism.** ROS2 contains an overload handling mechanism to drop a timer callback (and so the associated chain) in case the timer callback missed one or more of its period to start execution. The overload handling mechanism is activated by running `rcl_timer_call` function at the release instant of timer callback. First, the `next_call_time` variable is incremented by the timer’s period. Then `next_call_time` is compared with current time instant to see whether `next_call_time` is in the past. If so, `next_call_time` is incremented to the number of periods timer already behind, ensuring that the timer skips missed periods.

3 PROBLEM AND METHOD

3.1 Problem Statement

As mentioned earlier, the default scheduler of the ROS2 executor introduces two critical issues that adversely affect the response time of processing chains. The existing chain-level fixed priority-based scheduling scheme of ROS executor [4] addressed those critical problems. However, the chain-level priority proposed by PiCAS [4] disproportionately affects lower priority chains, which is further worsened in an overloaded scenario where the lowest priority chains may receive no service. In this paper, we particularly attempt to resolve two issues:

1) We address the limitations of default `readySet`-based scheduling scheme of ROS2 executor replacing the default scheduler with a dynamic-priority based scheduler.

2) We address the limitations of chain-level priority-based scheduling when the workload has chains with equal (semantic) priority.

3.2 Proposed Scheduler

To replace the default scheduler of ROS2 executor, we proposed a deadline-based (Earliest Deadline First (EDF)) scheduling policy redesigning the `readySet` as a `readyQueue`.

**Definition 1.** `readyQueue` $\Omega$ is maintained in `executor` similar to `readySet` in default ROS2. Unlike `readySet`, `readyQueue` is updated after the completion of each non-preemptively executing callback. The priority of the `readyQueue` is set based on the deadline of each callback, where earliest deadline callback has higher priority than the later one.

As the `readyQueue` prioritizes any callback only based on the deadline parameter, there is no privilege priority of timer callback in the `readyQueue`. This property of the `readyQueue` mitigates both of the limitations of the default scheduler, as any released chain has to wait to execute until the completion of high priority chains.

**Scheduling Strategy.** Our deadline-based scheduling policies for (sub)chain scheduling in the `executor` are as follows:

1) Whenever any chain instance releases, it enters to the waitset. The `readyQueue` $\Omega$ of the `executor` is updated after the completion of the execution of the current active callback due to the non-preemptive execution of the callbacks.

<table>
<thead>
<tr>
<th>Chains</th>
<th>Specifications [sec]</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) $\tau_i &lt; \tau_j, \tau_\ell \tau_k &gt;$</td>
<td>$C_1 = 0.109; C_{(2,3)} = 0.131; \tau_1 = 1$</td>
</tr>
<tr>
<td>(2) $\tau_{(4,\ldots,n)} &gt;$</td>
<td>$C_4 = 0.109; C_{(5,\ldots,10)} = 0.131; \tau_2 = 1$</td>
</tr>
</tbody>
</table>

Table 1: Chain set for illustrative experiments.

<table>
<thead>
<tr>
<th>Chain ID</th>
<th>Mean</th>
<th>Max</th>
<th>Min</th>
<th>STD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>1.478</td>
<td>1.879</td>
<td>0.632</td>
<td>0.355</td>
</tr>
<tr>
<td></td>
<td>2.108</td>
<td>4.913</td>
<td>2.415</td>
<td>0.445</td>
</tr>
<tr>
<td>PiCAS</td>
<td>0.422</td>
<td>1.373</td>
<td>0.373</td>
<td>0.565</td>
</tr>
<tr>
<td></td>
<td>2.055</td>
<td>2.654</td>
<td>1.655</td>
<td>0.314</td>
</tr>
<tr>
<td>OURs</td>
<td>1.424</td>
<td>1.922</td>
<td>0.903</td>
<td>0.310</td>
</tr>
</tbody>
</table>

Table 2: End-to-end latency results [sec] of two chains (Table 1) running in default ROS2 scheduler, PiCAS [4], and our deadline-based scheduler of ROS2 executor.

(2) The `readyQueue` $\Omega$ is prioritized based on the absolute deadline of each callback, and the callback with the earliest absolute deadline is scheduled to execute non-preemptively. The higher priority chain’s callback can only preempt any active chain (executing) after the completion of the currently executing callback.

(3) Suppose a chain instant is released, but the previous instant of the chain has not started executing yet. In that case, the previous instant of the chain is dropped by enabling the (original) overload handling mechanism of ROS2.

**Remark.** Tardiness is allowed as ROS2 is unaware of deadlines. Guarantees of meeting deadlines can only be achieved by verifying that the worst-case latency of the chains do not exceed the deadlines. In this work, we use the deadlines only to dynamically prioritize chains (and callbacks) during runtime, but not to drop workloads. This is acceptable since ROS2 inherently provides its own overload handling mechanism to prevent unbounded latencies.

3.3 Illustrative Experiment

We present an experimental study on a simple workload (Table 1) consisting of two processing chains. The experimental setup on the ROS2 environment is presented in the evaluation section. We perform the experiments for default ROS2 scheduler, PiCAS [4], and our deadline-based scheduler of ROS2 executor. The execution patterns for these three algorithms are illustrated in the Gantt charts in Fig. 2. From the Gantt charts, Fig. 2 (b, c), in priority-based scheduling, all chain instances complete the execution before starting later instances, and only the chains with higher priorities are active, reducing the interference in an active window. The end-to-end latency of the chains for all three schedulers is shown in Table 2. Our schedulers outperform the default scheduler for both of the chains. Compared with PiCAS, our scheduler performs better than PiCAS on average, and specifically, low-priority chains consistently outperformed PiCAS.

4 RESPONSE TIME ANALYSIS

4.1 Overview

We first analyze the worst-case response time (WCRT) of a (sub)chain (analyzed chain) in an `executor`, then extend the analysis for end-to-end latency of the chain span over the multiple `executors` using CPA tool [6]. Fig. 3 illustrates a possible scenarios of ROS2 processing.
(b) PiCAS Scheduler [4]

Figure 2: Gantt-charts of ROS2 scheduling example for the workload of Table 1. The first three lines, in blue, represent the executions of Chain 1. The remaining lines, in blue, represent the executions of Chain 2. In PiCAS scheduler (Fig. 2(b)), no chain instances from chain 1 (orange) is dropped, but three instances (at 3, 6, and 9) from second chain (blue) is dropped. In contrast, our scheduler dropped chain instances from both chain 1 (at 4 and 9) and chain 2 (at 3 and 7).

chains assigned in the executors. We are particularly interested in analyzing the response time of the sub-chain \( \Gamma^B_j \) in executor \( B \), then extend the analysis to calculate the end-to-end response time of \( \Gamma = \Gamma^A_j \cup \Gamma^B_j \) that span over executors \( A \) & \( B \).

**Arrival Curve.** The release patterns of chains/sub-chains in an executor \( E_k \) is characterized by \( \alpha(\Delta) \). Chains start with a timer callback have a staircase arrival curve, \( \alpha(\Delta) = \left( \frac{\Delta}{T} \right) \) where at each period one instance of the chain is released. However, the arrival curve for a chain starts with regular callback (sub-chain) depends on the arrival curve of the starting sub-chain of the parent chain and the response time of all precedent sub-chains (see Fig. 3). The request bound function (rbf) of a chain \( \Gamma^k_j \) in a time range \( \Delta \) is upper bounded by [3],

\[
\text{rbf}(\Gamma^k_j, \Delta) = \alpha(\Delta) \cdot C^k_{\Gamma^k_j}
\]

therefore, the demand bound function (dbf) of the chain is upper-bounded by [5],

\[
\text{dbf}(\Gamma, \Delta) = \text{rbf}(\Gamma^k_j, \Delta - D^k_j)
\]

\( \text{dbf}(\Gamma, \Delta) \) implies that the maximum resource demand generated by the chain instances of \( \Gamma \) that have both release and deadline instances in the range of \( \Delta \) times.

**Resource Model.** We consider a general case where resource supply to each server is lower bounded by the supply-bound function \( \text{sbf}() \). \( \text{sbf}(\Delta) \) implies the minimum guaranteed supply to the server in each \( \Delta \) time interval [7, 8]. In our analysis, we assume \( \text{sbf}(\Delta) \) is known (provided by the system designer). We define the pseudo-inverse of \( \text{sbf}(\Delta) \) as follows,

\[
\text{sbf}^{-1}(x) = \min(\Delta | \text{sbf}(\Delta) = x)
\]

\( \text{sbf}^{-1}(x) \) provides the minimum time in which the server gets \( x \) unit of processing time.

Intuitively, to bound the WCRT, we first calculate the slack time\(^5\), instead of directly calculating the WCRT. For analyzed (sub)chain instant similar to [5]. So, the WCRT, \( R_j \) of the analyzed chain, \( \Gamma_j \), is as follows:

\[
R_j = D_j - S_j^\star
\]

The minimum slack time, \( S_j^\star \), for all (sub)chain instances of the analyzed (sub)chain \( \Gamma_j \) is estimated from the resource demand of all (sub)chains in the executor and the available resource supply to the executor.

Finally, we develop the end-to-end WCRT of the analyzed chain, adding the individual sub-chain WCRT and communication delays between the sub-chains of different executors.

4.2 Properties

To compute the WCRT of a (sub)chain, \( \Gamma^k_j \) inside an executor \( E_k \), we consider a ‘busy period’ of the executor. Any busy interval, \([t_0, t_d]\), in the executor is an interval where at least one chain instance is ready to execute at any instant in the interval with a release time no earlier than the \( t_0 \) and deadline no later than \( t_d \). Here, \( t_0 \) is the last idle instant of the busy interval. To develop the resource demand of all (sub)chain instances in such a busy period, let consider a minimal set \( \sigma(\Gamma) \) that includes all (sub)chain instances with a release time no earlier than the \( t_0 \) and deadline no later than the \( t_d \). Following are the two facts of the minimal set \( \sigma(\Gamma) \).

**Fact 1.** All (sub)chain instances in \( \sigma(\Gamma) \) must have an execution in \([t_0, t_d]\) but the one with a deadline equal to \( t_d \).

Proof. If any (sub)chain instance does not execute in the busy interval, the instance is removed from the minimal set \( \sigma(\Gamma) \). So, if any instance in \( \sigma(\Gamma) \) does not execute in \([t_0, t_d]\), it contradicts the minimality of \( \sigma(\Gamma) \). However, a chain instance \( \sigma(\Gamma) \) with deadline \( t_d \) cannot execute at all, implying that it will miss the deadline. \( \square \)

**Fact 2.** In any busy interval, \([t_0, t_d]\), at most one (sub)chain instance with deadline > \( t_d \) can execute non-preemptively for one callback.

\(^5\)Slack time is the difference between the deadline and completion time of a chain instance. Note that slack time can be negative, which usually occurred for ‘tardy’ instance, and we allow tardiness in our model.
Proof. Following the deadline-based strategy, any chain instance can be preempted by a higher priority chain at the completion instants of callbacks of the executing chain. Therefore, the chain instance with deadline > \( t_d \) can at most execute non-preemptively for one callback with an execution starting time instant < \( t_o \). Further, once the chains in \( \sigma(\Gamma) \) with deadline \( \leq t_d \) starts executing, any chain instance with deadline > \( t_d \) cannot execute in \((t_o, t_d]\) following the minimality constraint of \( \sigma(\Gamma) \) and the fact 1.

Following facts 1 and 2, we can bound the blocking term for a busy interval by low-priority chain instances.

**Lemma 1.** The maximum blocking time of analyzed chain, \( \Gamma^k_j \) in a ‘busy interval’ of executor \( E_k \) is upper-bounded by:

\[
B^k_j = \max_{\tau_j \in E_k(\Gamma)} \left\{ \frac{C_i}{T^k_j} \right\}
\]

(7)

and \( T^k_j \) is the minimum inter-arrival time of the analyzed chain, \( T^k_j \).

Proof. Following fact 2, the blocking time by a lower priority chain can be at most the execution time of ‘one callback’. However, the blocking time for the analyzed chain, \( \Gamma^k_j \) cannot be from the lower-priority instances of \( \Gamma^k_j \) as the release time of lower-priority instances always later than the current instance. Therefore, lower-priority blocking (sub)chain instances are from \( \forall \tau_j \in E_k(\Gamma) \setminus \Gamma^k_j \).

Now, from the overload handling mechanism, if any chain instance cannot start execution for one or more periods of the chain instance, then the chain instance is dropped. Therefore, \( \frac{C_i}{T^k_j} \cdot T^k_j \) of \( C_i \) is not included in blocking time.

So, the maximum possible blocking time in a ‘busy interval’ of the executor \( E_k \) is,

\[
B_k = \max_{\forall j, \Gamma^k_j \in E_k(\Gamma)} \{ B^k_j \}
\]

(8)

**Lemma 2.** The ‘Resource Demand’ in a busy interval \((t_o, t_d]\), \( RD(t_o - t_o) \) of the executor \( E_k \) is upper-bounded by:

\[
RD(\ell) = B_k + \sum_{\forall j, \Gamma^k_j \in E_k(\Gamma)} \text{dbf}(\Gamma^k_j, \ell)
\]

(9)

where, \( \ell = t_d - t_o \) and \( \text{dbf}(\Gamma^k_j, t_o - t_o) \) is the demand-bound function of the chain \( \Gamma^k_j \) in the time interval of \((t_o, t_d]\).

Proof. The demand-bound function of any chain in the executor can be estimated using Eqn. (4). Therefore, \( \sum_{\forall j, \Gamma^k_j \in E_k(\Gamma)} \text{dbf}(\Gamma^k_j, \ell) \) provides the maximum possible resource demand by the workload \( E_k(\Gamma) \) in \((t_o, t_d]\) interval and \( B_k \) is the maximum blocking term (Eqn. 8) for the workload. Hence, the lemma follows.

Without loss of generality, we consider the idle instant as \( t_o = 0 \), so the resource demand for any ‘busy interval’ \( \ell \):

\[
RD(\ell) = B_k + \sum_{\forall j, \Gamma^k_j \in E_k(\Gamma)} \text{dbf}(\Gamma^k_j, \ell)
\]

**4.3 Response-Time inside an Executor**

So far, we have both the maximum resource demand \( RD(\ell) \) and minimum resource supply by the supply-bound function, \( \text{sbf}(\ell) \). Now, we compute the minimum slack time for the analyzed chain instances \( \Gamma^k_j \) in the executor \( E_k \) using following lemma.

**Lemma 3.** The slack time of the analyzed (sub)chain \( \Gamma^k_j \) in executor \( E_k \) is lower-bounded by:

\[
S^k_j = \min_{\forall \delta \in DBF, \ell = \delta} \left\{ \delta - \text{sbf}(\delta) \right\}
\]

(10)

where \( L^k_j \) is an upper-bound of range \([1]\).

Proof. The lemma directly follows Theorem III.2 of [5].

Lemma 3 provides the exact slack for the tardy chains but gives pessimistic slack time for hard sporadic chain scheduling [5]. However, we allow the tardiness of the processing chains. So, we do not develop an exact test for hard sporadic chains.

**Theorem 1.** The response time of the analyzed chain \( \Gamma^k_j \) in the executor \( E_k \) is

\[
R_j = D_j - S^k_j
\]

(11)

Proof. It directly follows from response time defined in (6).

The response time of the analyzed chain \( \Gamma^k_j \) can further be bound from above for the overloaded system using the overload handling mechanism of ROS2 as follows,

\[
R_j^k = \begin{cases} R_j^k, & \text{if } R_j^k \leq T_j + C_{\gamma}, \\ T_j + C_{\gamma}, & \text{otherwise} \end{cases}
\]

(12)

this works following the fact that the overloading handling mechanism allows at max \( T_j \) idle time for any chain instance to start execution and the EDF makes the chain instance highest priority among all active instances.

**4.4 End-to-end Response Time**

Finally, we analyze the end-to-end response time of the analyzed chain, \( \Gamma_j = \bigcup \Gamma^k_j \), which can span over the multiple executors in the system.

**Theorem 2.** The end-to-end response time of the analyzed chain \( \Gamma_j \) is:

\[
R_j = \sum_{\forall k} R_j^k + (K_j - 1) \cdot \gamma
\]

(13)

where \( K_j \) is the number of executors that the chain \( \Gamma_j \) spanned.

Proof. It follows from the principle of the CPA tool: the end-to-end response of the chain is the sum of individual response time in each executor and the total communication delays.

**5 EVALUATION**

In this section, we first explain our implementation of the proposed chain instance-level priority scheduler in ROS2. Next, we present experimental results comparing the different schedulers.

\[ L^k_j = \frac{r^k_j}{c^k_j} \cdot \min\left\{ D_j \right\} \text{ where, } \sum_{\forall \ell \in E_k(\Gamma)} \left(C_{\ell_j}/T_{\ell_j}\right) \leq c[1]. \]
The ROS2 schedulers are implemented on an Linux scheduling class. When testing the default ROS executor, we added chain membership and priority fields and followed the allocation scheme presented in the paper.

**Results.** We compare our deadline-based executor against the default ROS SingleThreadedExecutor and the existing chain-aware priority-based execution strategy developed by Choi et al., PiCAS [4]. We observe that for any chain, our proposed scheduler has a lower or equal latency compared to PiCAS as well as the default scheduler. Within each category (RT, BE), we observe that our proposed scheduler has better average latency compared to others for higher chain indices (Fig. 4). As a general trend, the default scheduler has a lower average irrespective of the chain, but higher worst-case latencies. In contrast, our proposed approach maintains a lower average while having lower worst-case latencies as well. All algorithms appear to favor lower index over higher index chains and we believe this due to the tie-breaking behavior.

**Note.** Our latency times are not directly comparable to those found in [4], due to the difference in the definition of latency. In our work, we define latency as the time difference between release time of the chain and the completion time of the last callback in the chain. Whereas, [4] measure the latency starting from the point the timer callback begins execution. The presented latency is in line with the definition of latency in the real-time scheduling theory.

### 6 CONCLUSION AND FUTURE WORKS

In this paper, we proposed a deadline-based scheduling scheme for ROS2 executor to overcome the limitations of the default readySet-based scheduling technique. We used deadline as a tool to realize dynamic priority setting, and presented an end-to-end response time analysis for each processing chain spanning over multiple executors. We evaluated the proposed scheduler using a case study of the ROS2 application. Our future goal is to develop techniques to optimally assign dynamic priorities (deadlines) for the workload, such that the user-specified response time and latency requirements can be met. While no such results exists yet, this work serves as a first and important step, where the latency for a given workload can be upper bounded, while deadlines are used to set dynamic priorities.

### REFERENCES


---

**Table 3**: Specifications for the case study tests, where first six chains are real-time chains and last row specifies the best-effort (six) chains with same parameters.

<table>
<thead>
<tr>
<th>Chains</th>
<th>Specifications (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt; t₁, t₂ &gt;</td>
<td>C₁ = 2.3, C₂ = 16.1, T₁ = 80</td>
</tr>
<tr>
<td>&lt; t₁, t₃, t₄, t₅ &gt;</td>
<td>C₃ = 2.2, C₄ = 18.4, C₅ = 9.1, T₁ = 80</td>
</tr>
<tr>
<td>&lt; t₋₉₋₉₋₉ &gt;</td>
<td>C₋₉₋₉₋₉ = [25.7, 12.4, 21.7, 17.9], T₀ = 100</td>
</tr>
<tr>
<td>&lt; t₁₋₁₀, t₁₋₁₁, t₁₋₁₂ &gt;</td>
<td>C₁₋₁₀, C₁₋₁₁, C₁₋₁₂ = 6.6, T₁₋₁₀ = 100</td>
</tr>
<tr>
<td>&lt; t₁₋₉₋₋₉ &gt;</td>
<td>C₁₋₉₋₋₉ = [11.7, 11.6, 6.7, 9], T₁₋₋₉ = 160</td>
</tr>
<tr>
<td>(BE₁₋₋₋₋₆)</td>
<td>C₁₋₋₋₋₆ = 33.2, C₆ = 6.6, T = 120</td>
</tr>
</tbody>
</table>

**Figure 4**: End-to-end latency - tested with the default ROS2, PiCAS [4], our modified executor scheduler for the case study workloads presented in Table 3.

**5.1 Implementation**

Our implementation features an alternate executor based on the default SingleThreadedExecutor provided by rclcpp. Our modified executor maintains a priority queue, named readyQueue, and populates it from the wait set after every callback execution. We created a unified structure to represent any type of callback along with its chain membership, period, and deadline parameters, and replaced the default executor’s unordered callback lists.

When a chain completes execution, the executor checks the time Until trigger field of the chain’s associated timer and sets the deadline field of the chain’s callbacks to time Until trigger + period. Before a chain completes execution, the timer callback may be triggered resulting in multiple chain instances ready for execution. To prevent multiple chains from running at once, each callback in a chain has a counter that keeps track of where in the chain execution is happening. The counter can be used to ensure that the newly released chain instance is not scheduled until the counter reaches the end of the current chain. The callbacks in the queue is prioritized by chain instance first and then by callback deadlines.

**5.2 Case Study**

The ROS2 schedulers are implemented on an Nvidia Jetson Xavier AGX in the 30W mode, with the clock frequencies fixed at 1.2Ghz.

**Workload.** To evaluate the deadline executor implementation, we perform tests with a node layout inspired by case study II from [4]. The workload parameters are presented in Table 3.

Each test uses 4 CPU cores, and each thread is set to the sched_fifo Linux scheduling class. When testing the default ROS executor and our modified executor, we use 8 executors. We distribute the real-time chains across the first four executors and the best-effort chains across the next four executors. Each executor runs in its own thread. Threads containing real-time (RT) chains have their priorities set to 99 (the maximum priority available in Linux), and the best-effort (BE) chains are assigned priority 98. We then distribute the executors across 4 CPU cores for load balancing. For testing the PiCAS executor, we added chain membership and priority fields and followed the allocation scheme presented in the paper.