Dynamic Priority Scheduling for Periodic Systems using ROS 2

Lukas Dust, Saad Mubeen

Mälardalen University, Västerås, Sweden
(first.last)@mdu.se

Abstract. In this paper, a novel dynamic priority scheduling algorithm for ROS 2 systems is proposed. The algorithm is based on determining deadlines of callbacks by taking the buffer size and update rates of channels into account. The efficacy of the scheduling algorithm is demonstrated on an illustrative example, where the needed buffer size is reduced in comparison to the conventional single-threaded executor in ROS 2.

Keywords: Robot Operating System 2 · Scheduling · Executor

1 Introduction and Background

Robot Operating System (ROS) 2 is a middleware introducing real-time capabilities to its predecessor ROS [1]. With the end of support for ROS in 2025, researchers and practitioners are forced to transition their systems to ROS 2. Hence, increased research activities have been seen in the past few years. ROS 2 systems consist of so-called Nodes as main components distributed in a network, communicating via designated channels in the Data Distribution Service (DDS). As depicted in Fig. 1, each node consists of a scheduler, called an executor, that schedules the schedulable entities, called callbacks. Two types of trigger events release callbacks. Data-triggered callbacks are connected to a specified channel in the DDS. Time-triggered callbacks are connected to a system timer. Generally, there are four types of callbacks, namely, Timer, Subscription, Service, and Client callbacks. An input buffer with a configurable size for each callback collects the trigger instances, such as messages and timestamps. In order to explore the real-time capabilities of ROS 2, the inbuilt scheduling algorithm has been analyzed [2]. Alternative priority-based scheduling algorithm has been proposed [4], assigning static priorities to callbacks. In the default executor, only one instance of each callback, released before the scheduler interaction with the middleware (polling-point), is considered for scheduling. Polling is performed when the set containing one instance of every released callback has been emptied. Blocking of callbacks [3], [6], and missing configuration options have been

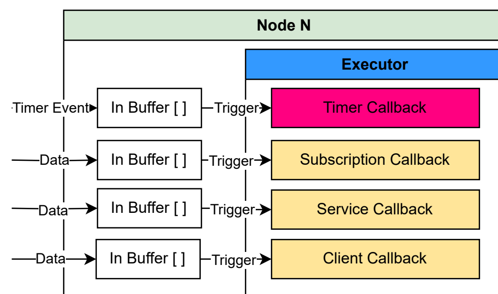


Fig. 1. Schematic example of a ROS 2 Node and its essential components for scheduling.

exposed as a weakness by [2], [5]. In this paper, supported by the increased demand for enhanced scheduling options, we propose a new dynamic priority scheduling algorithm developed for periodic nodes. For the sake of simplicity, a periodic node is defined, where all n callbacks contained in a node execute periodically where the period P of each callback $P_{cb} > 0$. Each callback is released by the trigger events contained in the Buffer B_{cb}, where r^k is the kth trigger instance in the buffer and $t(r^k_{cb})$ is the stored arrival time of the kth trigger instance of callback *cb*. This paper shows that the algorithm can reduce the needed buffer size compared to the native ROS 2 scheduling algorithm, while potentially reducing the number of resource-demanding interactions with the ROS Middle Ware (RMW) compared to the fixed-priority scheduling.

2 Algorithm

Algorithm 1: Proposed Scheduling Algorithm

```
1  foreach callback cb where nextrelease(cb) ≤ systime do
2  |   collect entity(cb) from RMW
3  |   if New Data available then
4  |   |   add cb to readyset;
5  |   |   calculate T_cb^min and D_cb;
6  |   end
7  end
8  if readyset ≠ Null then
9  |   cb = callback with shortest deadline
10 |   pop data from input buffer
11 |   if buffer(cb) empty then
12 |   |   remove cb from readyset
13 |   end
14 |   else
15 |   |   calculate new deadline
16 |   end
17 |   execute cb
18 end
```

The proposed algorithm, shown in Alg. 1, assigns a deadline to every released callback collected in the so-called `readyset`. A scheduling decision is performed following the earliest-deadline-first metric. The deadline for a callback is determined by the predicted time the input buffer to overflow, following (2). Initially, the deadline is set to infinite. When new data arrives, the minimum time difference T^{min}_{cb} between two consecutive arrivals is determined using (1). Now, the deadline as the predicted time of a buffer overflow can be calculated by knowing T^{min}_{cb} and the buffer utilization $U(B_{cb})$. When running the algorithm, the scheduler updates the `readyset` by scanning the input buffers in the RMW. When newly arrived data is detected, the callback is added to the `readyset`. In order to reduce the needed interactions with the middleware, based on T^{min}_{cb}, the time for the next trigger instance arrival is predicted and stored as the `nextrelease` for every callback. An interaction with the middleware is performed only when the system time exceeds or equals the `nextrelease`. Initially, the `nextrelease` is set to zero, forcing a scan in the middleware until the first trigger instance has arrived. In the second step, the callback with the earliest deadline is selected for execution. In case of shared deadlines, the callback with the highest buffer utilization is given the highest priority, followed by the registration order in case of further shared priority. The data for the selected callback is removed from the buffer. A new deadline is calculated, or the callback is removed from the `readyset` when no trigger instance is left in the buffer.

$$T^{min}_{cb} = \begin{cases} t(r^k_{cb}), & if\ T^{min}_{cb} = 0 \\ min(T^{min}_{cb}, t(r^k_{cb}) - t(r^{k-1}_{cb})), & if\ T^{min}_{cb} > 0 \end{cases} \quad (1)$$

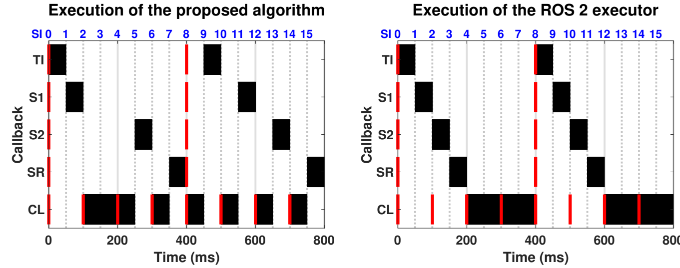
T_{cb}^{min} is the minimum time difference between two consecutive trigger events, and $t(r_{cb}^{k})$ is the arrival time of the last trigger instance k.

$$D_{cb} = \begin{cases} inf, & if\ T_{cb}^{min} = 0 \\ T_{cb}^{min} * (S(B_{cb}) * (1 - U(B_{cb})) + t(r_{cb}^{k}), & if\ T_{cb}^{min} > 0 \end{cases} \quad (2)$$

D_{cb} is the deadline of callback cb, T_{cb}^{B} is the minimum time difference between two consecutive trigger instances, $S(B_{cb}$ is the input buffer size, $U(B_{cb})$ is the utilization of the buffer and $t(r_{cb}^{k})$ is the arrival time of the latest (kth) message.

3 Illustrative Example and Comparison to ROS 2

In this section, the scheduling of the proposed algorithm is shown in an example and compared to the executor of ROS 2. Execution traces are carefully created by hand based on the algorithms. The following scenario is taken: A ROS 2 node consists of five callbacks that are triggered periodically. The callbacks are one timer callback `TI`, two subscriber callback `S1, S2`, and one service callback `SR`, all triggered every 400 ms and one client callback `CL` triggered every 100 ms. For the sake of simplicity, the input buffers have a size of five, and the execution time for each callback is 50 ms. Fig. 2 shows the execution of the scenario by the proposed algorithm on the left and ROS 2 on the right plot. At the start, all callbacks have been triggered once at 0 ms. Therefore, each



(a) Execution example of the proposed algorithm

(b) Execution example of the ROS 2 executor

Fig. 2. Execution traces using the proposed (left) and the ROS 2 executor (right). Red lines are trigger events and the blue numbers the scheduling iterations (SI).

task's deadline is set to infinite. As all tasks have the same deadline and buffer utilization, execution is conducted after the registration order, leading to the execution of `TI` and `S1`. At *SI2*, `CL` is triggered a second time. The scheduler now determines T_{CL}^{min} as 100 ms and a Deadline of $D_{CL} = 400ms$. Furthermore, the predicted next release time is 200 ms. As all other callbacks still do not have two consecutive releases, their deadline is still infinite. Therefore, the client callback is executed, and the deadline is calculated until the buffer is empty, as no other callback gains a lower deadline. At *SI8*,for `TI, S1, S2` and `SR`, T_{cb}^{min} is be determined as 400 ms and $D_{cb} = 2000$ ms. For `CL`, $D_{cb} = 800$ ms. Hence, `CL` is executed first. In comparison to the ROS 2 execution, `CL` is in all cases executed closer to the trigger event. The buffer utilization never exceeds 40%, while the maximum utilization in the ROS 2 system is 60%. If now `CL` has the

smallest buffer size of all callbacks, even in the first 400 ms, there will never be more than one element in the buffer as `CL` would gain the highest priority. The number of needed interactions with the RMW is presented in Tab. 1. At the initial 400 ms, except for `CL`, the predicted next arrival time is 0. Therefore an interaction is performed during every scheduling iteration. After S2, the period of `CL` is known, and interactions with the middleware are only needed every second scheduling iteration. After 400 ms, the period for all callbacks is known, reducing the number of interactions with the RMW significantly. Therefore, fewer interactions with the RMW are needed than in the ROS 2 executor and static priority algorithms, that need to interact with the RMW at each scheduling iteration for each channel.

Tab. 1. Amount of RMW interactions

	Proposed Alg.	ROS 2	Static priority Alg.
First 400 ms SI 0 - SI 7	37	20	40
Second 400 ms SI 8 - SI 15	8	20	40

4 Discussion and Ongoing Work

The presented algorithm is created to give developers further configuration options while preventing buffer overflow. This work is in its infancy, and the proposed algorithm is at a conceptual level. Nevertheless, the given scheduling example showed the algorithm to have the potential to decrease the needed space of buffer size and give the developer more configuration options regarding priorities compared to the single-threaded executor in ROS 2. The needed interactions with the RMW are only increased at the first iterations. In the long run, the number of interactions can be decreased compared to the actual executor in ROS 2. Nevertheless, further analysis is needed to determine possible weaknesses of the proposed scheduling mechanism. Furthermore, the algorithm will be implemented in the ROS 2 stack and tested to be compared with the other existing algorithms on a real system. Adaptions of the algorithm might be needed to mitigate errors caused by offsets, changes in publishing rates, and message arrival jitter and make the algorithm usable in non-periodic systems.

Bibliography

[1] OpenRobotics ROS 2: Docs, 2023, https://docs.ros.org/en/humble.

[2] Blaß, T., Casini, D., Bozhko, S., Brandenburg, B.B.: A ros 2 response-time analysis exploiting starvation freedom and execution-time variance. In: IEEE Real-Time Systems Symposium. pp. 41–53. IEEE (2021)

[3] Casini, D., Blaß, T., Lütkebohle, I., Brandenburg, B.: Response-time analysis of ros 2 processing chains under reservation-based scheduling. In: 31st Euromicro Conference on Real-Time Systems. pp. 1–23 (2019)

[4] Choi, H., Xiang, Y., Kim, H.: Picas: New design of priority-driven chain-aware scheduling for ros2. In: IEEE 27th Real-Time and Embedded Technology and Applications Symposium. pp. 251–263. IEEE (2021)

[5] Dust, L., Persson, E., Ekström, M., Mubeen, S., Seceleanu, C., Gu, R.: Experimental evaluation of callback behavior in ros 2 executors. In: 28th International Conf. on Emerging Technologies and Factory Automation (2023)

[6] Tang, Y., Feng, Z., Guan, N., Jiang, X., Lv, M., Deng, Q., Yi, W.: Response time analysis and priority assignment of processing chains on ros2 executors. In: IEEE Real-Time Systems Symposium. pp. 231–243 (2020)