Delft University of Technology
Master of Science Thesis in Embedded Systems

# A Priority-Based Real-Time Scheduling Framework for ROS2

**Grzegorz Krukiewicz-Gacek**
Supervisor: Dr. Mitra Nasri


TU Delft Delft University of Technology

# A Priority-Based Real-Time Scheduling Framework for ROS2

Master of Science Thesis in Embedded Systems

Embedded and Networked Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands

Grzegorz Krukiewicz-Gacek
g.m.krukiewicz-gacek@student.tudelft.nl
gkgacek@gmail.com

31.07.2021

**Author**
Grzegorz Krukiewicz-Gacek (g.m.krukiewicz-gacek@student.tudelft.nl)
(gkgacek@gmail.com)
**Title**
A Priority-Based Real-Time Scheduling Framework for ROS2
**MSc Presentation Date**
05.08.2021

**Graduation Committee**

| | |
|---|---|
| Koen Langendoen | Delft University of Technology |
| Arjan van Genderen | Delft University of Technology |
| Mitra Nasri | Eindhoven University of Technology |

**Abstract**

Since its introduction in 2007, ROS (*Robot Operating System*) has become one of the most popular framework for developing automated solutions in a variety of applications, ranging from automotive to manufacturing.

Meeting safety-critical timing requirements is a crucial element in many fields where ROS is present. The ROS community has introduced several improvements to allow developers to create more time-predictable implementations, with the introduction of ROS2 being the major step forward.

In the following thesis, we examine structure of ROS2 to discover task scheduling and data distribution related challenges that developers will face when trying to implement latency-sensitive ROS2 applications based on periodic tasks.

We propose a set of improvements that allow to mitigate presented challenges. A prototype of a new scheduling mechanisms embedded within ROS2 logic is introduced to combine both callback scheduling and efficient intra-process data exchange.

Furthermore, by exploring possible new ROS2 extensions, we address scenarios where applications have to dynamically adjust to temporary changes in timing-constraints. We extend the existing ROS2 APIs with an option to easily create new scheduling policies to handle specifc requirements for each application without the need for any kernel level scheduler modifications.

*"If we knew what it was we were doing,*
*it would not be called research, would it?"* – ALBERT EINSTEIN

# Preface

With the expanding market for advanced robotics applications, ROS (and ROS2) has become one of the go-to platforms for education, research and industry level prototyping. Its friendly API interface, a growing developer community and ease of use has encouraged developers to make it their middleware of choice. Plenty of robotics applications, such as manufacturing, drones, automotive, etc., require software that will be able to operate according to real-time software principles. The community is continously trying to answer the question if ROS (ran on top of Linux) can be considered a path forward for developing safety critical solutions. The aim of this work is to further expand our understanding and provide prototype solutions for a set of challenges related to implementing real-time systems on top of one of the most popular robotic development frameworks in the world - ROS.

The work presented in this thesis is a direct result of inspiring enthusiasm, patience and trust of my supervisor Dr. Mitra Nasri. She has been a source of amazing ideas and provided invaluable guidance for navigating the world of academic research. I believe I have been trully lucky to be able to work with her on this project. I would like to also thank my friends from the Embedded Systems programme who were always supportive throughout the time of writing this thesis.

Grzegorz Krukiewicz-Gacek

Delft, The Netherlands
31st July 2021

# Contents

# Chapter 1

# Introduction

## 1.1 ROS framework

*Robot operating system* (ROS) has become one of the most widely spread open-source software frameworks in the robotics industry [2]. Contrary to what it's name might suggest, it is not an *operating system* in a classical meaning of a bridge element between computer hardware and the software running on top of it. It runs on top of a regular operating system (such as Ubuntu) and does not manage the available hardware resources or the execution of all software in the system. In fact, ROS is an open-source middleware suite containing a set of software libraries and tools that help build applications for robot control.

Thanks to a large community, frequent updates (at the time of writing this document, latest release was published on May 23rd 2021, just a year after the previous release), and a wide selection of open-source packages, ROS accelerates the initial prototyping phase of development, thus driving down the research and development cost. Currently, the number of available open-source ROS packages reaches around 2 thousand and the number of active users is estimated to be near 20 thousand. Big players, such as Canonical, the company behind the Ubuntu Linux distribution, officially backs the development of ROS by collaborating by Open Robotics (the company steering the development of ROS). All these factors have increased interest in ROS by industries such as automotive, aerospace, and manufacturing.

## 1.2 ROS application model

In ROS, applications are constructed of *nodes*. By definition, each node should encapsulate a single functionality of the system (e.g parsing sensor data, telemetry logging, controlling motors, etc.). Nodes consist of *callbacks*, which are functions that implement the actions performed by a node. In this work we will refer to them either as callbacks or *tasks*. One node can consist of one or more callbacks. The callbacks can be executed based on timer (periodically) or as a way to handle several types of events available in the ROS framework.

This brings us to the main important aspect of ROS which is the *publisher-subscriber* mechanism that allows for communication between nodes. Each node can publish data on a *topic*, which later can be received by another node that

subscribes to that specific topic. A topic (e.g. radar distance measurements topic), can be understood as an asynchronous message bus. Data is encapsulated in a *messages* which can consist of one or several instances primitive data types (such as integers, floats, strings, etc.). Once data is published on a topic, callbacks assigned specifically to that topic are called in the receiving nodes.

Consequently, ROS can be considered a framework that supports a modular application architecture approach, with clear separation between different elements of the application. Understanding this aspect of ROS provides as good insight into why it encouraged so many developers to create third-party libraries that can be easily integrated into existing applications.

## 1.3 ROS vs. ROS2

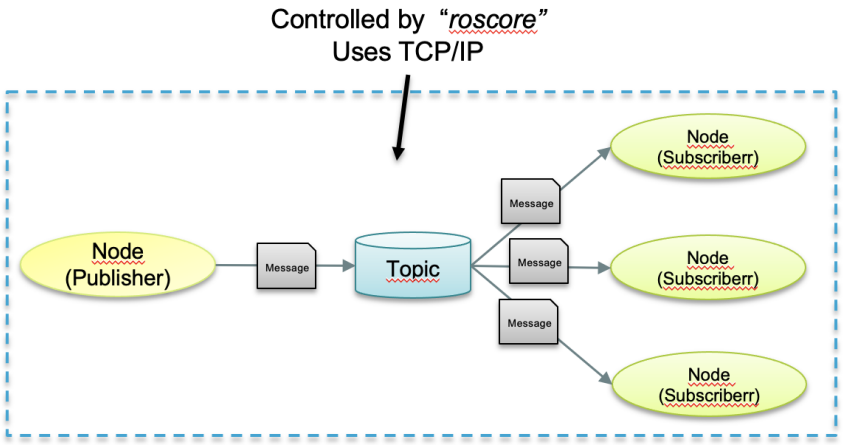



Figure 1.1: **Simplified representation of a ROS application structure**

In 2017, a refactored version of ROS, called ROS2 [11] was introduced to meet growing and varied demands from the developers community. One of the main driving factors for the creation of ROS2 was enhanced support for time-sensitive applications.

Similar to its ancestor, ROS2 provides a publish-subscribe mechanism for communication between nodes. To reduce message transmission overhead and ensure more predictability ROS2 uses third-party implementations of *data distribution service* (DDS) that replaces the TCP based message transport. In addition to that, the message transport can be fine-tuned using Quality of Service (QoS) settings specific for each DDS. Furthermore, ROS2 adds an intra-process message exchange system API that utilises a shared memory space for fast data sharing between nodes.

Finally, ROS2 replaces the centralized coordination and scheduling approach of ROS (*roscore*) with a distributed, multi-layer concept called *executor*. A visual representation of a ROS and ROS2 application structures are presented on Fig. 1.1 and Fig. 1.2.

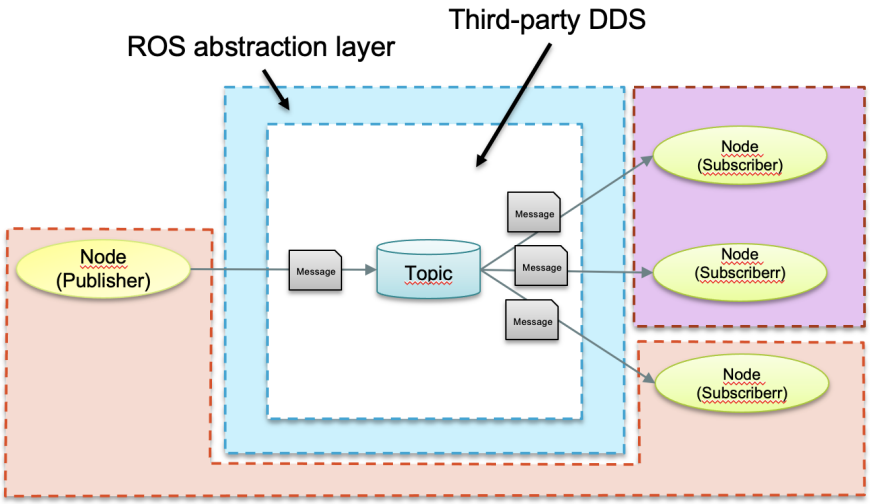



Figure 1.2: **Simplified representation of a ROS2 application structure**

## 1.4 Real-time systems and ROS

Meeting safety-critical timing requirements is a crucial element in many fields where ROS is present [3]. The manufacturing, autonomous cars or unmanned aircrafts (e.g. recently popular drones) are just one of a lot of examples.

ROS Industrial, a separate distribution of ROS was created to better match the needs of the manufacturing sector. Companies such as Apex.AI or Autoware Foundation are using ROS based stack for their autonomous driving solutions. Recently, Toyota announced it is going to use a system build on top of ROS for the development of their self-driving systems.

The common factor for all those safety-critical systems is ensuring that the software stack that operates them is capable of guaranteeing a time predictable behavior. By guaranteeing time predictable behavior, such system can be used for hard real-time applications where missing a deadline for a task execution can impact the safety of the system. An example of what a deadline miss means in context of tasks in a real-time system is presented on Fig. 1.3.

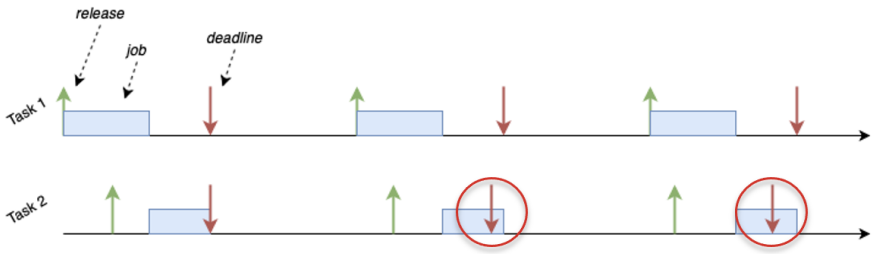



Figure 1.3: **A simple example of a deadline miss (marked with red circles) in a system with two tasks**

## 1.5 Organization and contributions of this work

The arguments brought up in Sec. 1.4 prove a high need for providing the developer community with detailed analysis of the strengths and weaknesses of one of the most established robot development platforms such as ROS. In Sec. 2 of this work, we perform a deep dive into the challenges for implementing real-time applications, posed by limited options of controlling callback execution priority and how this is further impacted by the message transport overhead. Next, in Sec. 3, we explore the related challenges and possible solutions already brought up by the academic community. Based on the challenges and opportunities discovered in Sec. 2, in Sec. 4, we try to propose a set of improvements to the existing ROS2 implementation that would have a chance of mitigating the discovered ROS2 shortcomings. This is followed by a detailed evaluation described in Sec. 5, where the proposed changes are compared against the baseline ROS implementation. Lastly, we conclude the thesis in Sec. 6 by presenting a set of final remarks.

## 1.6 Research goals

In this work we will focus on applications based on periodic tasks and the related challenges that ROS2 developers will face when trying to implement periodic task sets on top of the ROS2 middleware. We also find potential approaches to enhanced the ROS2 framework with support for application-aware scheduling. The summarised goals of this work are as follows:

- Analyze the ROS2 framework in the context of real-time applications consisting of periodic tasks
- Explore possible approaches on how to make execution of such task sets on ROS2 more predictable
- Attempt to provide means for implementing an application-aware (dynamic) preemptive scheduling policy for ROS2

# Chapter 2

# Motivation and Problem Definition

As described in Sec. 1, ROS2 is rapidly making it's way to systems with high requirements in terms of time-predictability. In order to ensure applications based on ROS2 can be commissioned on safety-critical systems, a good understanding of ROS2 implementation characteristics is needed with context of real-time systems in mind.

## 2.1 Assumptions

In this work, we focus on analyzing ROS2 from a perspective of specific subset of ROS2 application architectures. One common approach to building ROS2 based applications is based on *callback chains*. In such approach, majority of callbacks are run as a result of new data being available on a topic. Such approach is presented on Fig. 2.1. When looking at ROS2 callback chains, one of the most important metrics is the end to end latency of a callback chain [6]. Therefore, the analysis for callback-chain based applications would focus on aspects that can impact how much time passes from the moment the first callback in the chain starts to the moment the last callback in the chain finishes it's execution.

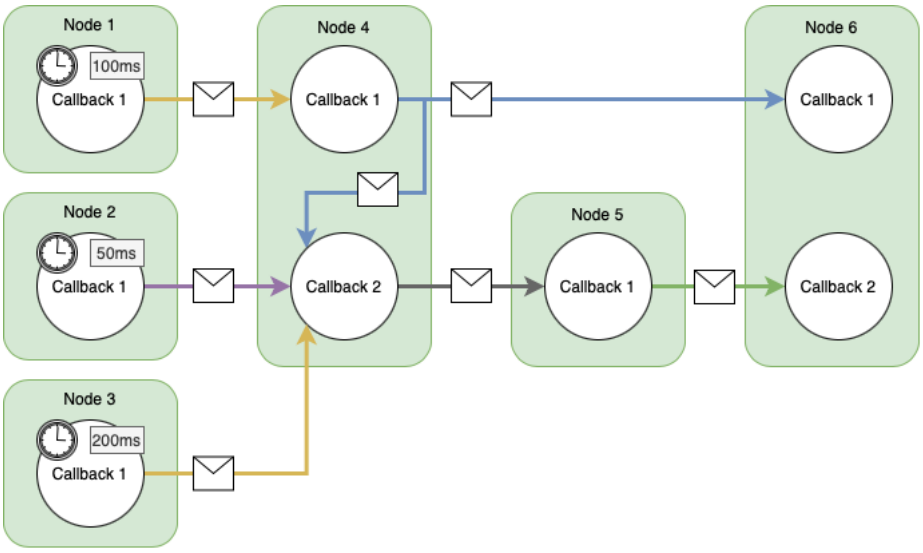



Figure 2.1: **Schematic representation of an application based on callback chains. Different arrow colors represent different topics.**

Another approach to the ROS2 application design would be solely timer based, where all callbacks are periodic. In such approach, presented on Fig. 2.2, all computation happens only in *timer based (periodic) callbacks*. However, nodes can still share data between each-other, but the *subscription callbacks* (that are called upon data retrieval) do not perform any computation data immediately but only store the data locally within a node for later use by the periodic computation tasks.

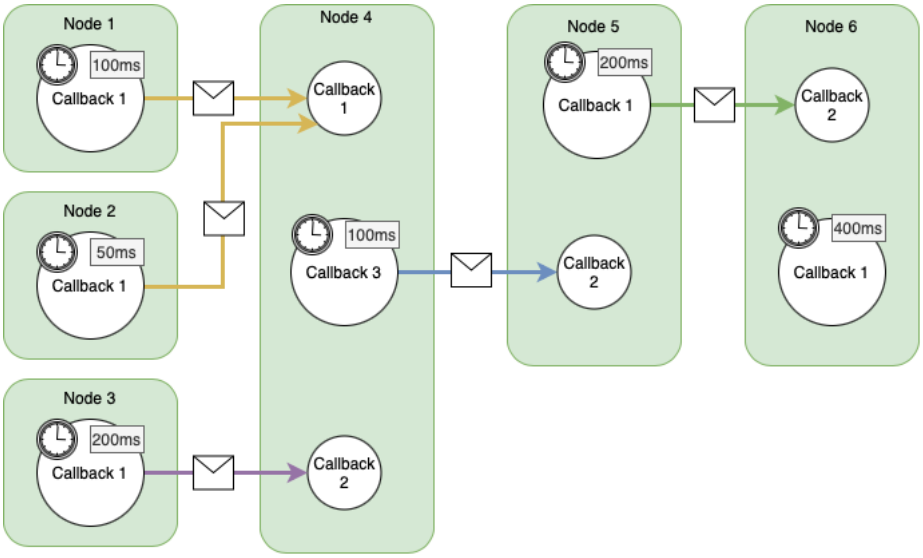



Figure 2.2: **Schematic representation of an application based on periodic callbacks. Different arrow colors represent different topics.**

In our work, we focus solely on applictions based on periodic callbacks On top of that, to make the analysis more specific we specify a few more constraints. First, as mentioned before, computation can only happen in time-triggered callbacks. Nodes can share data between each other but no processing can happen upon data retrieval. Secondly, we allow nodes to contain several periodic callbacks. Lastly, periodic tasks have specific priorities assigned.

With those assumptions in mind we analyze the current capabilities of ROS2 for ensuring such applications can run in a time-predictable manner and can be analyzed as periodic task based real-time systems.

## 2.2 Task scheduling in ROS applications

In order to understand the way ROS2 manages the execution and scheduling of a task the inner workings for the *executor* mechanism have to be explored. Executor is the main entity in ROS2 that is responsible for controlling the execution of callbacks. Fig. 2.3, shows a simplified visualization of a executor logic flow. In a ROS2 application, nodes are assigned to executors that control the execution of the callbacks contained in them. A ROS2 application can consist of a single executor controlling all the nodes or many separate ones controlling the execution of one or more nodes callbacks.

Once a ROS2 application is started, the executors start *spinning*, (top element on Fig. 2.3) which means they start infinitely (until the user or system stops the application) executing their main logic in a looped manner.

First an executor looks if there is a timer based callback ready to be executed. If this is true, the callbacks gets executed and the loop starts again.

If there is no pending timer based callback, the executor checks if any of the subscription based callbacks are ready for execution, if this is the case, the callback is executed and the executor loop starts again.

If neither periodic nor subscription based callbacks are ready for execution, other types of callbacks, such as *actions* or *services*, are checked. In this work we focus only on the timer based and subscription based callbacks.

ROS2 offers two main types of executors, namely the *single threaded* executor and *multi threaded* executor. In the single threaded executor, the callback (timer or subscription based) is ran on the same thread as the executor, thus halting further execution of the executor logic until the work of the callback is finished. This also prevents multiple callbacks to be executed at the same time,however it is the most lightweight implementation.

In contrast to the single threaded executor, the multi threaded executor, delegates the callback execution to a separate thread. This allows the executor to continue running after dispatching the callback execution to a separate thread. Furthermore, this enables simultaneous execution of multiple callbacks within the same executor. It is important to notice that the threads run under the same umbrella executor process.

### 2.2.1 Prioritizing different types of tasks

By understanding the logic behind the executor mechanism we can begin to realise the implications it might have on the way callbacks in ROS2 applications are managed and the constraints it poses. At this point we would like to put

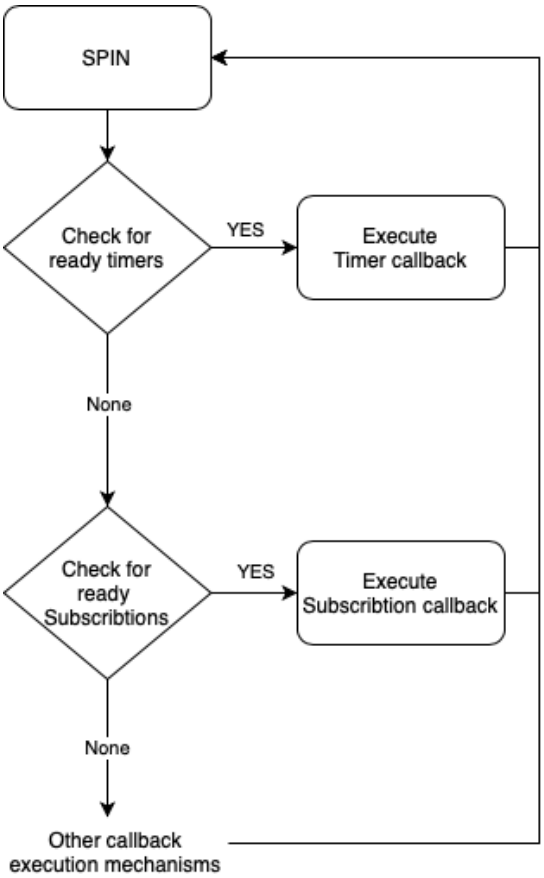



Figure 2.3: **Simplified visualization of the ROS2 executor logic**

the previously described order of executions of different types of callbacks in a spotlight. We will use a simple set of three tasks to present a problem that will appear due to the fact that the executor always prioritizes timer based callbacks over subscription based ones.

Fig. 2.4 presents execution flow of three tasks that would be observed if they were controlled by a single threaded executor. `Task 1` - produces data (e.g. sensor readings), `Task 2` - is an intermediate task (e.g. telemetry), `Task 3` - subscribes to the data produced by `Task 1` and utilizes it for it's periodic computation (e.g. sensor data parsing).

First, we can see that even though `Task 2` has lower priority that `Task 3` it is going to run first (assuming that they were registered in the executor in a `Task 1`, `Task 2`, `Task 3` order, as stated in Sec. 2.3). This is due to the fact that a ROS2 executor has no notion of callback priority; but this will be discussed in detail later in Sec. 2.3.

The second issue that can be noticed is that the subscription callback, responsible for delivering data for Task 3 is executed at the very end. As a result, even though the data were produced by `Task 1` prior to the execution of `Task 3`, `Task 3` still had to run on old data. This is the effect of the executor logic, that will always prioritize an awaiting timer callback over a subscription call-

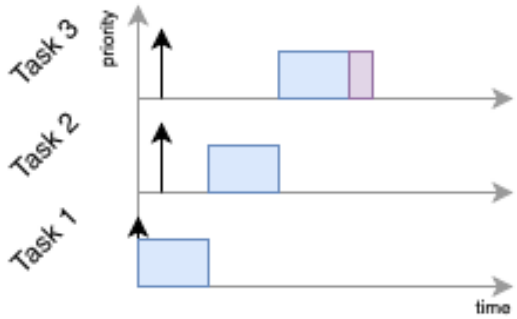



Figure 2.4: **Representation of ROS2 based execution of a three task application described in Sec. 2.2.1. Blue represents periodic callback execution, purple represents subscription based callback execution.**

back. Fig. 2.5 presents the expected execution flow, that is not achieved using a single threaded executor.

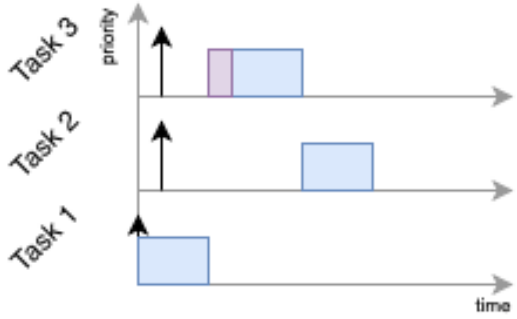



Figure 2.5: **Representation of a desired execution flow for the example presented on Fig. 2.2.1. Blue represents periodic callback execution, purple represents subscription based callback execution.**

## 2.3 Possible approaches to task organization

**The baseline0 approach.** In Sec. 2.2.1 we mentioned that the executor in ROS2 has no notion of callback priority. This poses a challenge for real-time system designers where ensuring task prioritization is a crucial element of the system. One approach to organizing nodes and their callbacks is assigning them all to one executor. Throughout will this work, we will refer to this approach as *baseline0*. In such scenario, the system designer does not have the capacity to assign different priorities to different nodes or specific callbacks since everything is ran under one process in the system and the executor does not have internal scheduling capabilities. In this approach, in case multiple timer based callbacks are pending for execution, they will be executed in the order the nodes they belong to were added to the executor.

**The baseline1 approach.** Another approach is presented on Fig. 2.6. In this approach, referred to throughout this thesis as *baseline1*, we assign each nodes to separate executors. Those executors can then be ran in separate processes. This allows us to leverage the system level scheduling capabilities to enable prioritising callbacks from one node over another. The Linux scheduling APIs are well documented but the developers are limited to what the underlying system offers in terms of scheduling policies.

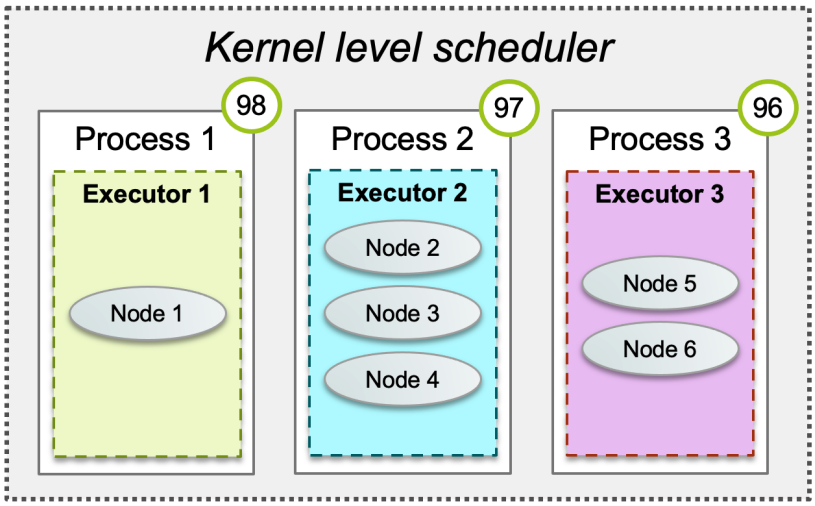



Figure 2.6: **A visualization of the *baseline1* approach with nodes distributed between multiple executors**

## 2.4 Message transport overhead

One of the biggest changes between ROS1 and ROS2 is the way messages are transported between nodes [8]. ROS1 utilized a TCP/IP based methodology governed by centralised `roscore`. This way of transporting messages was proven to be one of the elements blocking ROS from becoming a backbone for real-time systems. This challenge was addressed with the release of ROS2 where *DDS* (Data Distribution Service) based message exchange was introduced. With ROS2, developers can now utilize a set of different DDS's from different vendors such as `FastRTPS`, `OpenSplice` or `Connext`. DDS based message exchange offers a more predictable and controllable way of exchanging data then the ROS1 TCP/IP based system. On top of that, ROS2 added an fast *intra-process* data sharing approach which allows for a very low latency message exchange.

We were particularly interested in the difference between the message transport overhead for DDS and intra-process based data exchange. The reason for that is the promise that intra-process based data exchange should further reduce the message transport overhead and provide a much more predictable performance. Those aspects are key to ensuring a predictable and efficient behavior of a system overall. We performed additional evaluation of differences between transport times for different message sizes using FastRTPS DDS and

intra-process data exchange.

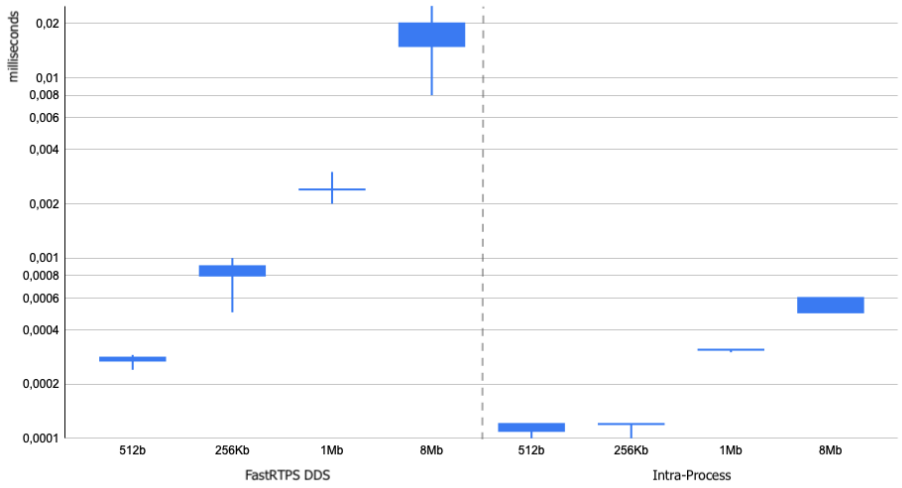



Figure 2.7: **Comparison of message transport times (seconds) between the FastRTPS DDS and inter-process communication**

On Fig. 2.7, which presents the result of our message transport overhead experiment, it can be observed, that for all tested message sizes (512b, 256Kb, 1Mb, and 8Mb) the intra-process data exchange is significantly faster and more time-predictable. For example, for medium sized messages of 256Kb, intra-process was more then 6 times faster. For large message sizes such as 1Mb, the difference was even more significant with an over 10 times faster message transport using the intra-process approach.

As presented on Fig. 2.8, data exchange overhead can become a more significant part of the application execution time, thus increasing the utilization. Eventually, it may lead to utilization exceeding *100%* and leading to missing task deadlines and starved subscription callbacks. If the time it took to execute the subscription callbacks in the example presented on Fig. 2.8, was lower (smaller purple blocks), eventually deadline miss could have been avoided.

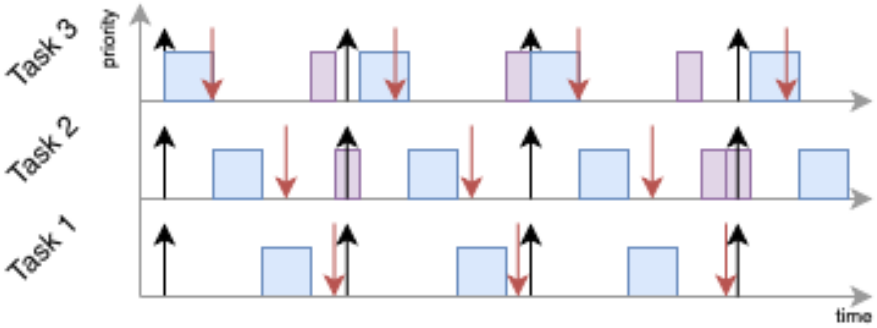



Figure 2.8: **Message transport time overhead resulting in a deadline miss and starved subscription callback execution. Blue represents periodic callback execution, purple represents subscription based callback execution.**

## 2.5 Discovered challenges

At first glance, it might seem that the previously presented `baseline1` approach is a good solution for many challenges for real-time applications in ROS2. With the possibility to distribute the nodes between multiple executors and having better methods than TCP/IP to share data between nodes we should be able to cover the needs of applications described in Sec. 2.1. However, `baseline1` approach introduces a limitation in terms of the possible approaches to message transport between nodes. When nodes are separated into different processes, the only way to exchange data is using a *DDS* (Data Distribution Service) based message exchange while the intra-process method is possible only for nodes that sit within the same executor (`baseline0` approach).

Eventually we are left with a choice to either have a way to efficiently exchange data between nodes with a single executor but not be able to schedule them according to desired priorities (`baseline0` approach), or have a way to assign priorities to certain callbacks by distributing nodes between executors in different processes (`baseline1` approach) but have to use DDS which might potentially turn out not good enough for certain types of applications (as presented earlier on Fig. 2.8).

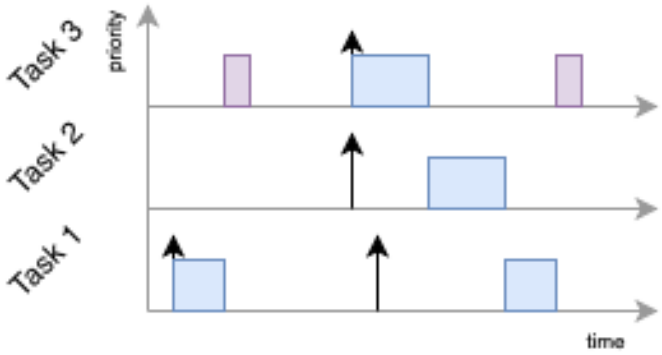
On top of that, the possible delay of executing subscription based callbacks, described in Sec. 2.3 might cause further complications when it comes to analyzing and predicting the behavior of ROS2 applications in the context of real-time systems.

## 2.6 Application aware scheduling

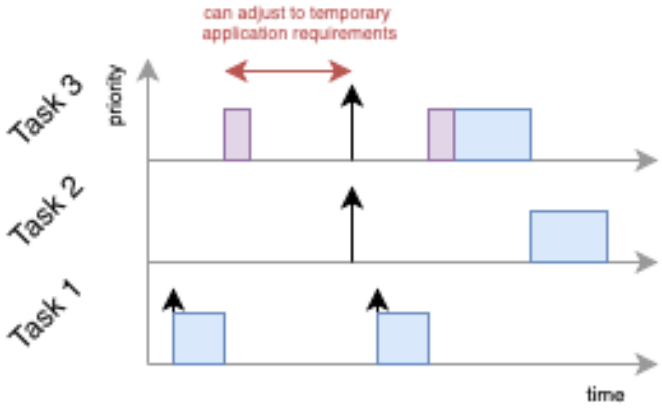
By distributing nodes into separate processes to allow us to use the system level scheduling APIs, we make the scheduling taking place in the underlying OS layer rather than executor level, thus not allowing the executor and the nodes to adjust the parameters used for scheduling if needed. However, ROS2 applications are inherently exposed to context changes, when they have to adapt

to new temporary missions [9].

Let's imagine that at some point the system has to be much more cautious about the action it takes (e.g. if a drone detects an increased collision probability or a rover has to dock into a charging station). In a normal operation scenario, some tasks might be considered more important, however, in case of a temporary context change, producing fresh proximity data and processing them should be given higher priority while the scheduling parameters for the other nodes in the application have to adjust [7]. At the time of writing, ROS2 does not provide a clear directive on how to handle such scenarios in an application wide manner. One possible approach is to create a shared topic where nodes would inform each other about critical changes in the application requirements. This way, each node can react and, for example, change its priority or suspend execution. For such approach, the reaction time can be insufficient and hard to predict, when each node has to receive a message and then perform appropriate action.




(a) **Execution schedule using `baseline0` ROS2 approach**




(b) **Execution schedule with potential application-aware scheduling**

Figure 2.9: **Comparison between simplified execution schedule representations for `baseline0` ROS2 approach and what could be achieved with application-aware scheduling**

As an example, we can again imagine a three task application such as the one presented in 2.2.1. In a regular scenario, `Task 2` has a higher priority over

`Task 1`. However, in a special temporary scenario, `Task 3` could request that the age of the data produced by `Task 1` cannot exceed a certain threshold. Fig. 2.9 presents a comparison between how an execution of such application looks like with a `baseline0` ROS2 approach and with a potential application-aware scheduling policy.

# Chapter 3

# Related Work

In this chapter we perform a deep dive into the most relevant recent work related to real-time support for ROS applications. This section is divided into two sections, in Sec. 3.1 we look at work that focuses on analytical approach to the challenge. In Sec. 3.2 we bring up contributions that present practical propositions that try to improve the state of ROS.

## 3.1 Analysis of ROS behavior

In **Exploring the performance of ROS** [8] Shinpei et al. conducts a detailed analysis of node to node message transport overhead. Not only various possible DDS configurations are compared, but also the intra-process approach and the ROS1 TCP/IP based message exchange. It is observed that the overheads vary significantly between the available DDS configurations and system designers should carefully consider the selected Quality of Service for their application if DDS is chosed to handle the message transport in the application.

**Response-Time Analysis of ROS 2 Processing Chains Under Reservation - Based Scheduling** [15] by Casini et al. provides insight into the end-to-end latencies of processing chains when implemented using ROS2. The paper focused on analyzing the single threaded executor and flagged potential concurrency issues that might arrise if multiple executors are ran simultaneously.

The work **Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications** [6] by Gutiérrez et al., similarly to [8], looks into the possible worst case latencies of message transport in ROS2 in the context of implementing real-time applications. It is observed that for a system under load, the latency and jitter of DDS based message transport time noticeably increases.

In **Automatic Latency Management for ROS 2: Benefits, Challenges, and Open Problems** [1] Blass et al. introduce a `ROS-Llama` framework for managing end-to-end latencies for ROS2 callback chains. Furthermore, they provide detailed feedback on Linux and ROS itself and present a set of challenges still stading in the way for real-time system designers (such as high latency I/O operations).

## 3.2 Extending ROS capabilities

**ROSCH: Real-Time Scheduling Framework for ROS** [13] by Yukihiro et al. introduces a scheduling framework which guaranties specific end-to-end latencies by guaranteeing a specific frequency of execution for the last callback in the chain. Intrestingly, `ROSCH` also adds a fail-safe system that signals early if the system is underperforming allowing for earlier reaction (e.g. slowing down an autonomous robot). Unfortunately, this work focuses solely on ROS1 (the originall implementation of ROS) which is not the focus of our work.

In **mROS: A Lightweight Runtime Environment for Robot Software Components onto Embedded Devices** [14], Takase et al. provide a novel approach to ROS nodes execution on light-weight embedded systems. It utilizes a real-time OS instead of the classical approach of running ROS on top of a Linux based system. In addition, the authors develop a custom TCP/IP based communication library to allow efficient node to node communication. The work is however based on the original distribution of ROS (ROS1). Moreover, differently to our work, it focuses on ommiting the need to use Linux to allow ROS to be available on even more resource constrained devices.

The work described in **Real-time control architecture based on Xenomai using ROS packages for a service robot** [5] by Delgado et al. takes an approach of utilizing the Xenomai real-time framework to support priority-based scheduling. The drawback of this approach is the additional complexity related with the usage of Xenomai which could discourage developers due to the need of non-trivial additional system configuration.

**PiCAS: New Design of Priority-Driven Chain-Aware Scheduling for ROS2** [4] presented by Choi et al. introduces a new scheduling framework for ROS2 that again enables to ensure end-to-end latency for processing chains in ROS2. The concept of `critical chains` is introduced to better select which chains should be prioritized and have their latency limited to a predictable bound.

Work that was recently presented as a Master Thesis at TUDelft by Charles Randolph, titled **Improving the Predictability of Event Chains in ROS 2** [12] tries to solve the issue of callback chain prioritization by automatically assigning priorities to all callbacks within a task based on the callback chain they belong to.

## 3.3 Conclusions

While analyzing available work, we noticed that majority of researchers so far focus solely on a callback chain based approach to application construction. Up to a certain extent, some of this work could be applied as well to prioritise execution of callback chains that consit only of one periodic node, however none focuses on reducing the message transport latency to enable more time for the computation thus allowing for higher utilizations.

Moreover, up to our knowledge, none of the published efforts focused on enabling application state aware scheduling.

Several publications aim to ensure time-predictability by taking the limitations of Linux out of the equation [5], [14]. Due to the fact that majority of ROS2 documentation, tutorials, courses, etc., focus on a setup based on reg-

ular Ubuntu distribution, we decided to focus on applications built with this most common approach. That is why we aim to not compare our findings with work that either runs ROS nodes outside of the Linux environment or requires modifications such as Xenomai.

# Chapter 4

# Proposed Solution

## 4.1 High-level idea

As described in Sec. 2, ROS2 as a framework does not provide developers with any mechanism that would help ensure a priority-based order of periodic node callback execution. As discussed in Sec. 2, ROS2 developers can use the `baseline1` approach, but it requires not only the knowledge about the available ROS2 and operating system APIs but also a good understanding of the inner workings of the middleware (such as the executor and DDS) in order to be sure all the application callbacks will be executed in a desired manner.

**Introducing the proposed approach.** In this work we propose a framework (referred to throughout the rest of this document as *proposed approach*) that extends the current ROS2 APIs for periodic callback execution to mitigate the challenges summarised in Sec. 2.5. With our framework we aim to resolve the dilemma between fast message exchange (grouping nodes in one executor) and enabling node prioritization (by separating nodes between multiple executors). We knew that we could approach the problem from two angles.

1. One option would be to try adding a faster data exchange option for nodes separated in multiple processes. It would have to be comparable to what is possible with intra-process data exchange, as discussed in 2.4.

2. Another way would be to keep nodes within one executor and add periodic callback scheduling capabilities to the executor logic.

In Sec. 2.5, we also highlighted the potential issues that can arise due to the fact that subscription based callbacks are always executed after timer based callbacks. This was the deciding factor that directed our work towards enhancing the executor logic (approach 2) rather than focusing solely on improving the message transport times between multiple processes.

First, in order to allow the executor to execute the periodic callbacks according to their priority, we had to ensure that information about the priority of each node can be accessible by the executor. It should be also possible to add more properties such as deadline (to enable Earliest Deadline First policies) can be added to the timer. In order to utilize those new information available for the timer based callbacks, execution of a new scheduling function has to be incorporated within the executor.

It is important to note at this point that the priorities etc., are assigned on *callback*, not *node* level. This means that a node will be able to contain several callbacks with different parameters potentially simplifying the application structure, as separating callbacks into different nodes just to achieve different execution priority will not be needed any more. A *priority property* needs to be added to the ROS2 timer description. To make things resemble Linux scheduling APIs, the available priorities for periodic callbacks should range from 1 to 98 (just as the available real-time priorities in many Linux systems). The extended information about timer based callbacks is not only priority, deadline, etc., but also the topics that are used to deliver data that are later used in the periodic computation.

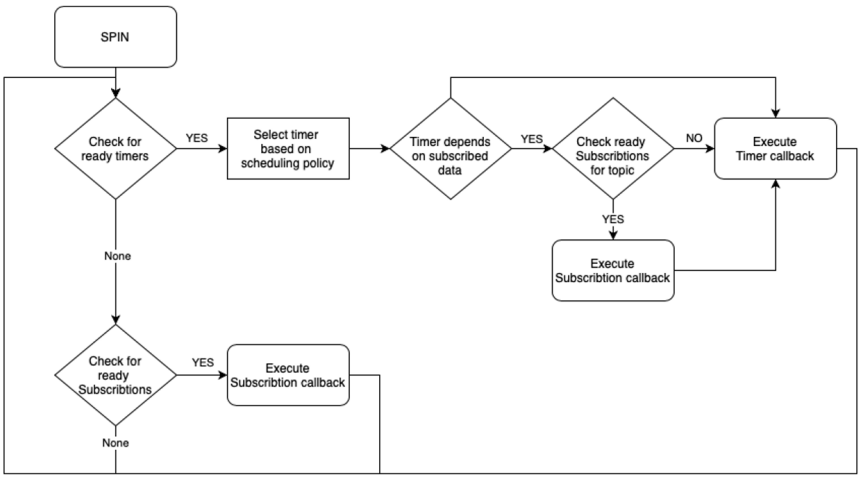



Figure 4.1: **Simplified representation of the proposed updated ROS2 executor flow**

Fig. 4.1 presents the main executor logic flow for the proposed approach. It can be compared with the baseline logic presented on Fig. 2.3. Except of the additional scheduling logic (that implements the scheduling policy of choice), the logic also ensures that the effect, described in Sec. 2.2.1, caused by deprioritizing subscription callbacks is mitigated. When a timer based callback is chosen for execution, first we utilize the extended periodic callback parameters to do a check on all assigned topic dependencies. By doing this, we ensure that in case that for any of assigned topics a related subscription callback is pending for execution we first execute the subscription callback to allow data to be delivered to the node prior to periodic callback execution.

## 4.2 Extending ROS logic

In order to be able to extend the ROS2 executor logic, a detailed analysis of it's implementation structure was needed. Fig. 4.2 presents the analysis of the inner workings of the `executor` and how this relates to the high level

commands taken on the application layer. We can see that eventually it's the `get_next_executable()` and `get_next_timer()` functions (surprisingly located in a file called `alocator_memory_strategy.hpp`) are the places where selection of the next callback to execute happens. Therefore, we decided to target those functions with the improvements described in Sec. 4.1.

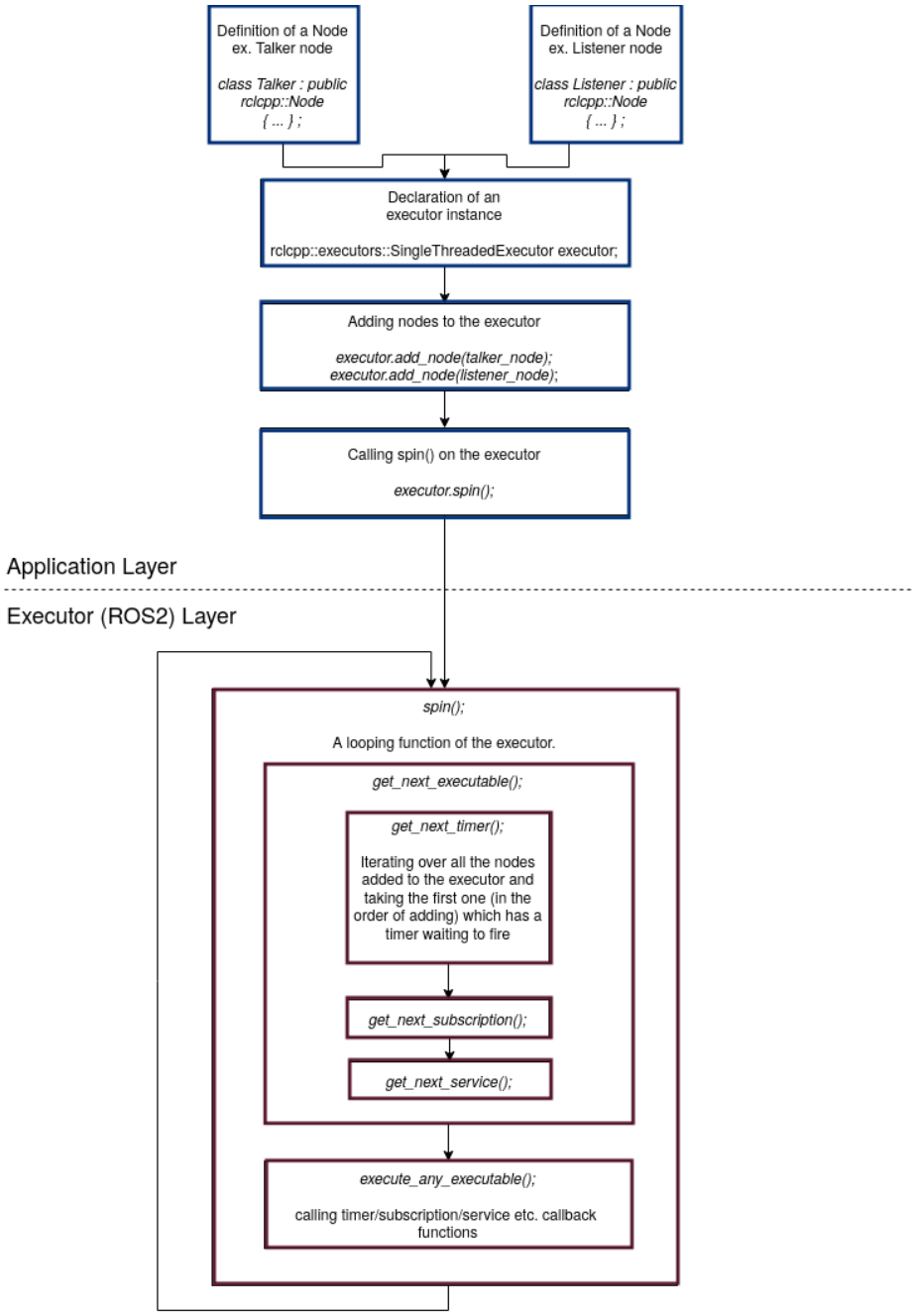



Figure 4.2: **Detailed analysis of the baseline executor implementation**

### 4.2.1 Supporting different callback priorities

As noted on Fig. 4.2, the `get_next_timer()` function, in baseline executor is simply a dummy iterator over timers which selects first one that is ready for execution. In our proposed approach, we enhance the dummy selection of executed callback with sorting the pending timers using a `comparator function`. The comparator function execution is represented on Fig. 4.1 as "*Selecting timer based on scheduling policy*". The comparator function takes two awaiting periodic callback parameters as input and outputs them in an order of desired execution priority. This allows the executor to sort the awaiting callbacks. As a part of this work we developed several basic comparator functions that allow to execute callbacks according to `Fixed Priority i.e FP` and `Earliest Deadline First i.e EDF` schedules.

### 4.2.2 Supporting preemption

By using the `baseline1` approach and distributing nodes in executors that reside in separate processes, we are capable of utilizing the operating system level preemption mechanisms, where processes with higher priority parameter will be able to pause the lower priority ones. In order to not deprive developers from that capability and be able to compete with the `baseline1` approach we had to ensure that our task execution logic supported preemption as well.

In order to do that we made the executor manage a pool of 98 threads and ensured that the executor itself is running on the highest available real-time system level priority of 99. In the executor logic, we simply dispatch callbacks onto threads with appropriate system level priorities. The subscription callbacks, that do not have an explicit priority assigned, are ran with the priority of the highest callback within a node that depends on data from related topic.

Additionaly, not only we support preemption, but also we can configure the executor to disable preemption and do not dispatch new work untill all previously started callbacks finish. Surprisingly, such option is not available in the Linux kernel without complex low-level modifications. With the option to disable preemption, we can explore non-preemptive scheduling policies if needed and explore the possibilities of non-preemptive work-conserving an non-work-conserving algorithms [10].

#### Required operating sytem configuration

It is important to notice, that for now, in order to support preemption, our solution, requires the application to be given two dedicated cores, one for the executor and the other one for the managed callbacks. This is achieved using the `isolcpus=` kernel configuration and later on, the `taskset -p` command to assign processes to a specific CPU. This is to ensure no other process will interrupt the execution of our application. Actually, the same approach can be used to isolate CPU's for applications ran using the `baseline0` and `baseline1` approach.

### 4.2.3 Supporting application-aware scheduling

In order to support the functionality of application-aware scheduling, brought up in Sec. 2.6, we needed a way of communicating the application state to the

executors scheduling function. During the execution of the application, it is a resposibility of a logic placed withing a callback to realise that the state of the application should change. For example a proximity data processing node can identify that a robot is approaching a docking station scenario. In order to enable each callback to request a change in the scheduling policy, there needs to be a way of communicating with the executor. This is achieved with a shared `scheduling strategy` object which is passed to each node during initialization. This object is also accessible for the executor and accessible to the scheduling function. Earlier we mentioned that in our proposed approach we implemented two basic scheduling policies of `FP` and `EDF`. Those implementations actually do not reside within the executor code but are rather passed as a function to the executor. Similarly any other custom scheduling policy function can be passed to the executor, thus allowing for implementation of logic that adjusts to the changes in `scheduling strategies` in a way designed specifically for a certain application.

## 4.3 Ensuring support from the Operating System

We can work very hard on ensuring that any part of the ROS2 stack ensures a latency-bounded execution, can be mathematically proven and become as close to a hard real-time software as possible, but until the underlying OS (Linux in most cases) will fall short to deliver those requirements all our efforts are in vain. In majority of cases, Linux will be just fine for real-time applications but the more strict the requirements become and we enter the realm of hard real-time system use cases from soft real-time use cases the bigger the need for ensuring that as many operations in the system overall (not only in the higher level middleware such as ROS2) are latency bounded. That is where the `PREEMPT_RT` kernel patch comes into play. We will not provide a detailed description on how `PREEMPT_RT` modifies the Linux kernel, however, we believe it was crucial to note that such modification the system is highly recommended to achieve higher confidence in the performance of the ROS2 applications that aim to achieve real-time performance.

# Chapter 5

# Evaluation

In this chapter, we evaluate the proposed framework to provide answers on the following questions: 1) what is the cost of utilizing a more complex executor logic compared to baseline approaches 2) what is the potential performance improvement between the proposed implementation and the baseline approaches? 3) what are the potential benefits of utilizing an application-aware scheduling policy compared to baseline approaches and proposed framework without such functionality enabled?

This chapter explains how we generate fair and representative ROS applications to evaluate the proposed framework under various parameters and setups. The aim is to select appropriate criteria to achieve best understanding of gains and losses related to our framework by comparing the proposed solution against existing baseline approaches. This section also includes a qualitative discussion about the benefits and the convenience of the APIs introduced in our framework.

## 5.1 Metrics

In this section we introduce the metrics used for evaluating the before-mentioned aspects.

**Executor overhead** is measured as the time it takes to execute the `get_next_executable()` function described in detail in Sec. 4. This data allows us to calculate detailed metrics such as mean or standard deviation of the executor overhead that is later used for comparison between all presented approaches.

The **Deadline miss** metric is obtained by counting the number of deadline miss events flagged by the callbacks. This required additional instrumentation in the callback code. The same logging functions were used for all approaches to ensure fair comparison. The number of deadline misses are then divided by the total count of execution for each callback respectively. This allows us to later see the correlation between the calculated deadline miss numbers and the utilization of a task set.

**Data freshness** is measured by counting the number of times that a periodic callback was executed while fresher data was already produced. This is calculated by checking if there is a pending subscription callback that would deliver more recently produced data for the computation used in the periodic callback.

To allow for comparison between the proposed and baseline approaches, additional logging was added to the callbacks rather than executor. The periodic callbacks were pritining out incremented identifiers for data that they use for the computation simmilarly to the nodes that produce the data. In case a producer callback has already generated data with identifier `2` while the periodic callback will print out information that data with identifier `1`, such event is calculated as an execution on old data.

**Data age** metric, used for evaluating the application-aware scheduling is calculated as a time it took between data was produced, by the callback that generates that data, and utilized by the computation in the periodic callback. Simmilarly as for the data freshness metric, the producer callbacks print out timestamps and identifiers for the data that they generate and the periodic nodes print out timestamps of when data with certain identifier was utilized. It allows us, in later log postprocessing, to calculate the data age. An example of log produced during test runs for capturing this metric is presented on List. 5.1.

Listing 5.1: **Example of logs used to calculate the data age metric**

```
[1626023628.982256552]: producing_data_topic_id sensor1 452
[1626023629.52480321]: producing_data_topic_id sensor3 321
[1626023629.883291035]: producing_data_topic_id sensor4 520
[1626023630.12144923]: using_data_topic_id sensor1 452
```

## 5.2 Synthetic application generation

A fair evaluation is only possible once the proposed approach is tested on a large and varying set of ROS applications. Simply because there are not enough accessible open-source ROS2 applications that could be used for the evaluation of our solution, we decided to create a small tool-set for generating synthetic ROS2 applications. The tool set allows to create ROS2 applications by taking a set of controllable parameters as an input. The presence and type of controllable parameters is directly linked to the type of metrics that need to be obtained. Under the constraints given by the input parameters, the other aspects of the application are randomised. For example, in order to assess how utilization impacts the occurrence of deadline misses, we need to be able to control the utilization of the task set in the generated application. On the other hand, the exact period for each task should be randomizable (or randomly selected from a given list of available ones). The full list of controllable parameters and their descriptions are show in Tab.5.1. and Tab.5.2. The utilization of a task set is calculated as 5.1 where $C_i$ is the worst-case computation time for task and $T_i$ is the period for a task.

$$U = \sum_{i}^{n} \frac{C_i}{T_i} \tag{5.1}$$

| parameter name | values / units |
|---|---|
| nodes | int (min. 1) |
| utilization | float (min. 0.01, max. 1.0) |
| periods | milliseconds |
| pubsub | tuple of floats (min. 0, max.1) |
| deadlines | "period" or "wcet" |
| deadlines_wcet | float (min. 1) |
| priorities | "rate-monotonic" or "random" |
| message_size | "256Kb", "512Kb", "1Mb", "2Mb", "8Mb" |

Table 5.1: **Automatic application generation values and units**

| parameter name | description |
|---|---|
| nodes | Number of nodes the application should consist of |
| utilization | The CPU utilization of the task set |
| periods | Period values that should be spread out equally across the task set |
| pubsub | Percentage of publishers and subscribers in the application |
| deadlines | The deadline for each task can be equal to it's period or based on it's worst case execution time |
| deadlines_wcet | The multiplier for calculating deadline based on wcet (only applies if "wcet" is selected as deadlines) |
| priorities | he method for assigning priorities in the application |
| message_size | Size of the messages produced by published |

Table 5.2: **Automatic application generation parameters description**

Those parameters together form an input for the application generation script. Since performance was not an important factor for the application generation framework, for the purpose of quick prototyping and ease of debugging, the script was implemented in Python3.

The generation script can be started using the Linux command line by specifying the python version, the main script file name and the before-mentioned parameters (an example is presented on List. 5.2). The script takes a JSON file as an input parameter. An optional *repeat* parameter can be passed to run the generation algorithm a specified number of times to achieve a set of different synthetic applications that meet the passed criteria. A sample JSON file containing all necessary parameters can look as presented on List. 5.3. The output are 3 ROS2 applications using 3 different approaches to control callback execution priority control described in this document (baseline 0, baseline 1, proposed approach respectively). Those approaches were described in detail in Sec.2 and Sec.4.

Listing 5.2: **Running the application generation script**

```
$ python3 eval_gen.py --params parameters.json --repeat 10
```

Listing 5.3: **Sample JSON file with selected synthetic application parameters**

```
{
```

```
    "parameters": {
        "nodes":3,
        "utilization":0.7,
        "periods":[10,20],
        "pubsub"[0.5, 0.5],
        "deadlines":"period",
        "deadlines_wcet":[1.0, 2.0],
        "priorities":"rate-monotonic",
        "message-size":"512kB"
    }
}
```

In order to obtain representative data, a large set of varying applications had to be generated. With the application generation tool at hand, the missing part was to create the JSONs containing input parameters. One approach here would be to manually create a lot of JSON files with definitions of the applications that we want to obtain. However, to speed up the process even further, a second smaller script was developed to automatically generate the input parameter JSON files. The format of the input file is the same as for the application generation script with the difference that each parameter is not a single value but an array of values that can be selected from.

Of course, if each combination was to be generated, the number of output files could easily reach tens of thousands without ensuring that all necessary parameter variations were generated. Hence, an additional compulsory `force_all_combinations` parameter was added. It that defines, which parameters should have all available options exhausted. For example if we define `nodes`, let's assume 2 and 4, and `message-size`, let's assume 512Kb and 2Mb, in the `force_all_combinations`, it forces the JSON generation script to exhaust all combinations of those two parameters (at least 4 applications will be created). This way we ensure that the generated applications will cover the most important range of combinations for certain tests. An example input to the parameter-generation script is presented on List. 5.4.

Listing 5.4: **Sample JSON file with range of available synthetic application parameters**

```
{
    "parameters": {
        "force_all_combinations":["nodes", "utilization"]
        "nodes":[3,5,10,50],
        "utilization":[0.5, 0.6, 0.7, 0.8, 0.9, 1.0],
        "periods":[[5,10], [10,20], [10,20,50], [10,20,50,100]],
        "pubsub":[[0.1, 0.1], [0.2, 0.8], [0.6, 0.4]],
        "deadlines":["period", "wcet"],
        "deadlines_wcet":[[1.1], [1.0, 2.0], [1.1, 1.5]],
        "priorities":["rate-monotonic", "rand"],
        "message-size":["512kB", "2Mb"]
    }
}
```

**Assumptions of the synthetic application-generation tool.** It is important to note that the worst-case execution time for each task is generated

based on the utilization selected for the task set. Furthermore, there are certain limitations to the available configurations. They also apply to the generation algorithm, as it is not allowed to create a task set that violates the restrictions. They restrictions are as follows:

1. Number of nodes has to be greater than 1
2. Utilization cannot be greater than 1
3. Number of publishers cannot exceed the number of nodes
4. Number of subscribers cannot exceed the number of nodes
5. A task can both be a publisher and a subscriber but not on the same topic
6. deadline cannot be greater than period
7. worst-case execution time cannot be greater than period

During each test, each application was ran 5 times for a strict period of 30 seconds. Same application configurations were not ran one after another, instead the execution order was chosen randomly. This was done to reduce the potential impact of system level behavior related to throttling down CPU, memory and other components after a prolonged period of running the tests.

Given the number of test runs that have to be performed to obtain measurements for so many sample generated applications, running them all manually would consume an unacceptable amount of time. An example way to start looks as on List.5.5. Running one test takes about 30 seconds. For some tests, it was planned to run around 50 different applications. This would mean that every minute for almost 3 hours a new application would have to be started manually. To avoid such burden for each test, a simple automated batch runner was developed.

Listing 5.5: **Running a ROS2 application**

```
$ ros2 run sample_package sample_app
```

**Generation of a single application.** Based on the input passed in the parameters JSON file the implemented tool generates an application in the following manner. First, a new empty ros package is created. Later, based on the `nodes` parameter, appropriate amount of new node classes is created (named simply `node1`, `node2`, etc.) with every node containing one timer based callbacks. Later, the periods are randomly assigned and added as timer periods for the before-mentioned callbacks. Each period value should be used for the same amount of nodes (if possible). Based on the `pubsub` parameter, the callbacks are modified accordingly. Once periods are assigned, priorities can be assigned as well (the detailed differences between generated code depend on the approach the application is generated for). `Deadlines` is simply a timing parameter, checked and loggd. the end of each callback execution. Last but not least, utilization is calculated for each node based on Fig. . 5.1. At the end, messages of correct size are assigned to specific publisher and subscriber callbacks.

## 5.3 Tracing method

In order to extract necessary timing parameters, correct logging and log-parsing system had to be selected or developed. To stay close to native ROS implementations, we performed all logging using an *RCUTILS* logging functionality. A sample log instrumentation of the unmodified ROS2 *executor* looked as presented on List.5.6.

Listing 5.6: **Example of code instrumentation using RCUTILS**

```
RCUTILS_LOG_INFO("get_next_timer_start");
memory_strategy_->get_next_timer(any_executable, weak_nodes_);
RCUTILS_LOG_INFO("get_next_timer_finish");
```

With this approach it was possible to either output the combination of timestamp and event name to the console or directly to a file (by adding a `> log.txt` to the launch command from List.5.5). Another benefit of using *RCUTILS* was that the timestamp is recorded at the time *RCUTILS* is ran not at the time of saving to the file (ensures precise measurements).

Because *RCUTILS_LOG_INFO* outputs the event name together with a UNIX timestamp, it was fairly straightforward later on to combine and parse all output from a single 30 second application run.

A separate *log-parsing tool* was developed in order to transform raw logs into understandable and ready to plot results. For example, for the measurement presented on List. 5.6, we have a measurement of *get_next_timer* duration instrumented with two logs, one starting with *start*, the other one with *finish*. The log parser can extract the duration of each *get_next_timer* by comparing the *start* and *finish* times, which later on can be averaged out or plotted as a histogram if needed.

Lastly, to ensure that *RCUTILS_LOG_INFO* is the right logging solution, an overhead measurement was taken by measuring the execution time of 100 executions. The measurement was repeated a 100 times. As presented on Fig.**??**, it clearly showed that the overhead is within acceptable range and should not impact the quality of the evaluation.

## 5.4 Performance comparison

### 5.4.1 Scheduling overhead

#### Hypothesis

In this thesis, we have expanded the executor implementation with an additional scheduling logic. As described in detail in Sec. 4, the scheduler extension adds looped iterations over awaiting timers and subscriptions. Our hypothesis is that for a small number of callbacks in the system, it is not expected to observe any significant difference in the executor's overhead. However, when there are a larger number of callbacks in the system, the difference might become more apparent (the more callbacks in the system, the bigger the difference, especially if the application is laid out in a way that will cause multiple timers to be ready for execution at the same time). Also, for more callbacks in the system, there might be more data exchange on several topics. We also expect the executor

overhead to increase with the number of subscriptions in the system (as they have to be iterated over prior to periodic callback execution).

### Setup

In order to compare the scheduling overhead, we need to compare the differences in the *executor* execution times between the baseline ROS2 implementation and the proposed solution. In order to do so, we have to ensure that the synthetic ROS2 applications that are used for this test will stress the executor. In other words, the new scheduling logic has to be forced to process larger sets of callbacks and related subscriptions. This way, it should be possible to understand to what extent can the overhead change. The parameters for creating applications for this tests are presented in Tab. 5.3.

Secondly, we measured the impact of the number of subscriptions in the system on the overhead. To achieve that, the aforementioned large callback count application has been prepared in two flavors, one with zero subscriptions and another one where eighty percent of the nodes are subscribed to at least one topic. The parameters for creating applications for this tests are presented in Tab .5.4.

A set of smaller applications is also used for the evaluation to see the average impact of the changes in the scheduling overhead (not only for edge-cases). The parameters for creating applications for this test are presented in Tab. 5.5.

It is important to notice that in this test, only *baseline0* is compared with the new proposed approach. The reason for that is the fact, that in the *baseline1* approach, the executor takes care only of a single callback and the scheduling happens on the operating system level.

| **parameter name** | **selected values** |
|---|---|
| nodes | 50 |
| utilization | 0.5 |
| periods | 10, 20, 40 |
| pubsub | 0, 0 |
| deadlines | period |
| deadlines_wcet | - |
| priorities | rate-monotonic |
| message_size | - |

Table 5.3: **Synthetic application parameters - executor test with high number of nodes**

| | Min | Q1 | Median | Q3 | Max | Mean | SD |
|---|---|---|---|---|---|---|---|
| baseline0 | 21 | 22 | 24.5 | 27 | 58 | 26.77 | 8.04 |
| proposed | 23 | 26 | 28 | 31 | 78 | 30.8 | 10.53 |

Table 5.6: **Executor overhead for small set of callbacks**

| parameter name | selected values |
|---|---|
| nodes | 50 |
| utilization | 0.5 |
| periods | 10, 20, 40 |
| pubsub | 0.2, 0.8 |
| deadlines | period |
| deadlines_wcet | - |
| priorities | rate-monotonic |
| message_size | 16b |

Table 5.4: **Synthetic application parameters - executor test with high number of nodes and subscriptions**

| parameter name | selected values |
|---|---|
| nodes | 50 |
| utilization | 0.5 |
| periods | 10, 20 |
| pubsub | 0.4, 0.6 |
| deadlines | period |
| deadlines_wcet | - |
| priorities | rate-monotonic |
| message_size | 16b |

Table 5.5: **Synthetic application parameters - executor test for a small application**

| | Min | Q1 | Median | Q3 | Max | Mean | SD |
|---|---|---|---|---|---|---|---|
| baseline0 | 23 | 25 | 27 | 30 | 65 | 29.43 | 8.3 |
| proposed | 29 | 32.5 | 43.5 | 51 | 91 | 45.37 | 15.1 |

Table 5.7: **Executor overhead for large set of callbacks (no data exchange)**

| | Min | Q1 | Median | Q3 | Max | Mean | SD |
|---|---|---|---|---|---|---|---|
| baseline0 | 23 | 24.25 | 27 | 30.75 | 60 | 29.73 | 9.16 |
| proposed | 39 | 49.5 | 62.5 | 68 | 120 | 63.67 | 19.49 |

Table 5.8: **Executor overhead for large set of callbacks**

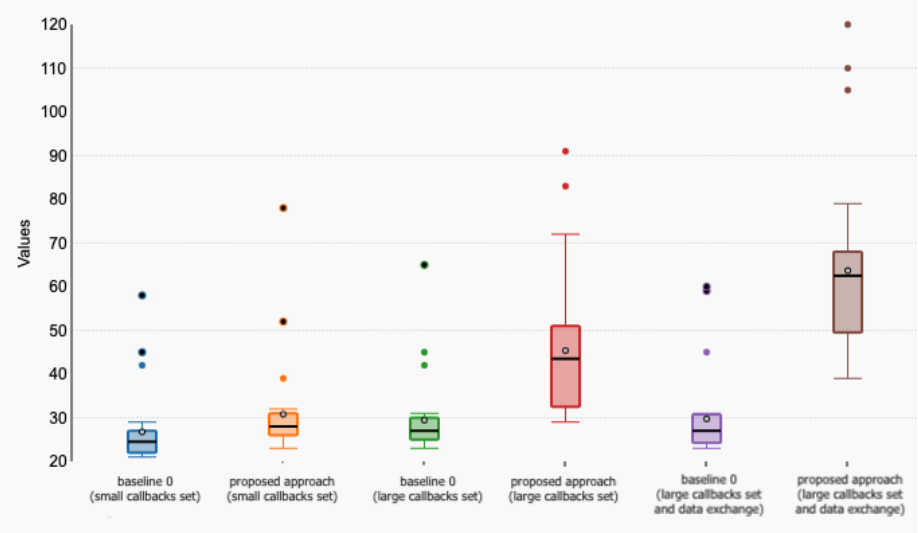



Figure 5.1: **Executor overhead for large set of callbacks**

**Discussion of results** Figure 5.1 represents the results obtained during the evaluation of the updated executor mechanism. Detailed information on the obtained measurements are presented in tables 5.6, 5.7 and 5.8.

As expected, the enhancements to the `executor` logic developed as a part of this thesis are resulting in a higher execution overhead. A quick look at the results shows that when the number of nodes increase, the overhead of our `executor` increases. For small applications (between 1 to 10 nodes), the proposed executor causes around 15% increase on average overhead as demonstrated in Tab. 5.6 and Fig. **??**.

For the new proposed executor logic, more nodes in an application means more complex scheduling computation (sorting). As a result, a bigger difference between the baseline and proposed executor overhead can be observed, as presented in Tab. 5.7 and Fig. **??**. For larger applications (consisting of around 50 nodes), it was noticed that the proposed executor causes around 54% increase in mean overhead time. Further more, the standard deviation increases significantly (from 8.1 microseconds to 15.1 microseconds). This can be appointed towards significant differences between executor logic execution times depending on the count of pending periodic executions.

The results also point towards another aspect that impacts the overhead, which is the number of nodes that also contain a subscription callback as presented in Tab. 5.8 and Fig. **??**. . For each of those, prior to execution, the `executor` needs to check for pending subscription data and allow it's execution if necessary. In such cases we see a 114% increase in the mean overhead and greater standard deviation (up to 19.49 microseconds).

### 5.4.2 Examining utilization and deadline miss correlation

#### Hypothesis

One of the main goals of our work is to ensure that all task sets can be ran without or with less deadline miss, no matter the utilization. By enhancing the

executor with preemptive callback scheduling capabilities, this work allowed to utilize the efficient intra-process data exchange while still being able to prioritize periodic callback execution in a desired manner.

The `baseline0` approach allows for intra-process data sharing as well, however with the `baseline0` approach, we are deprived of the callback scheduling capabilities hence we expect to see a significant number of deadline misses not directly dependent on the number of nodes, data sharing or utilization. When it comes to comparison between the `baseline1` and the `proposed approach` our hypothesis is that differences should not be visible for applications with zero or little data exchange between nodes. There are two factors that are expected to differentiate the results between the `baseline1` and the `proposed approach`. The more nodes share the data and the bigger the shared data, the more chances for the `baseline1` approach to show the drawbacks of using the inter-process data sharing. However, the more callbacks in the system, the higher the impact of the additional overhead induced by the enhanced executor implementation.

### Setup

In total, with the help of the parameter generation script, we created 48 parameter combinations for synthetic application generation from the parameters presented in Tab. 5.9. The most important variable for this test is utilization. Utilization of 0.5, 0.75 and 1.0 were selected for this test. We created, sixteen different configurations with varying message size, node count and subscribers count for all of the three selected utilization options, giving together 48 synthetic applications.

| **parameter name** | **selected values** |
|---|---|
| force all combinations | nodes, utilization, message-size, pubsub |
| nodes | 4 or 20 or 50 |
| utilization | 0.5 or 0.75 or 1.0 |
| periods | 10, 20 |
| pubsub | [0, 0] or [0.2, 0.8] |
| deadlines | period |
| deadlines_wcet | - |
| priorities | rate-monotonic |
| message_size | 256kB or 512kB or 2Mb or 8Mb |

Table 5.9: **Synthetic application parameters - utilization and deadline miss correlation**

To enable good understanding of possible gains from using the proposed implementation, the generated configurations contain applications with 4, 20 or 50 nodes. The configurations also contain varying message size which can be representative of cases where simple small data arrays are exchanged between nodes (256kB) up to big data chunks (e.g. images, video clips - 8Mb). These are obviously two extremes, however it should give a good view of how an increasing node count or message size will impact the behavior of the baseline and proposed implementations. Finally, the deadline for each periodic callback is set to be equal to it's period.

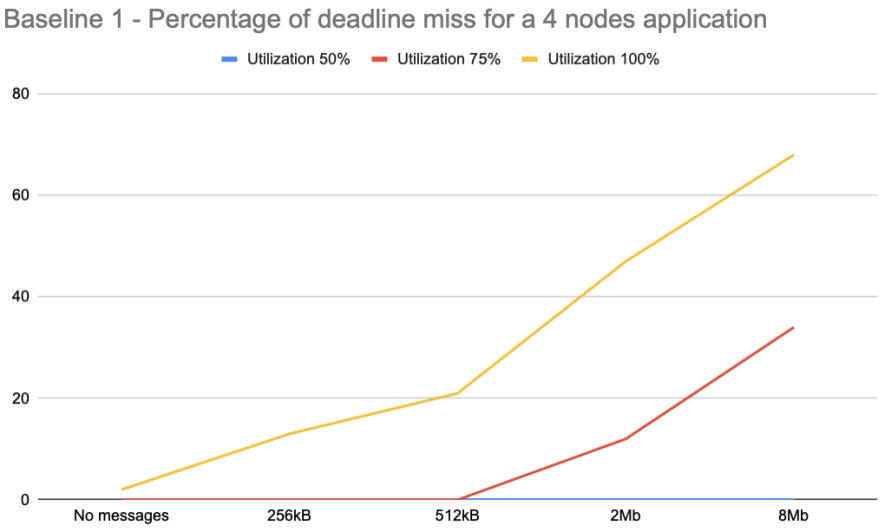



Figure 5.2

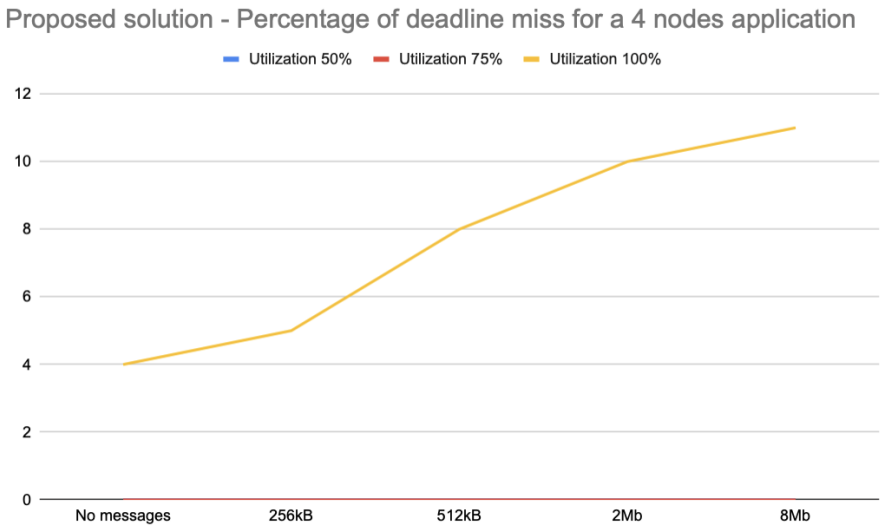



Figure 5.3

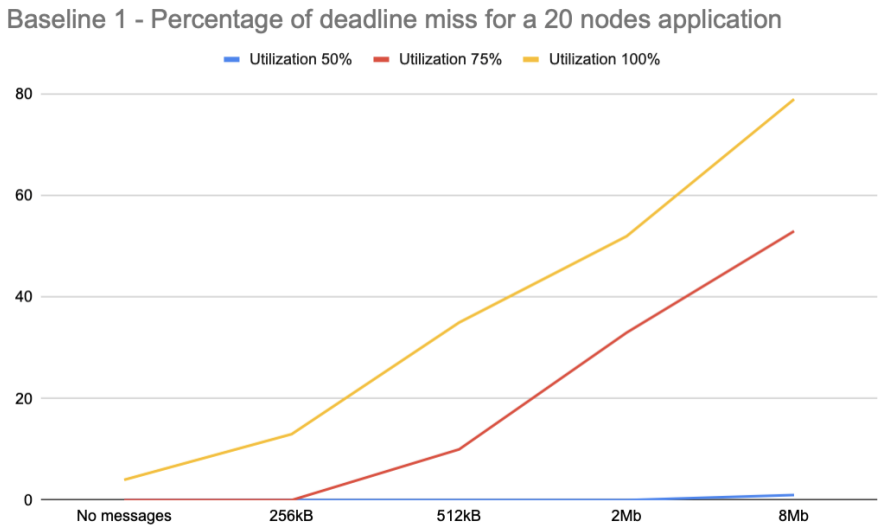



Figure 5.4

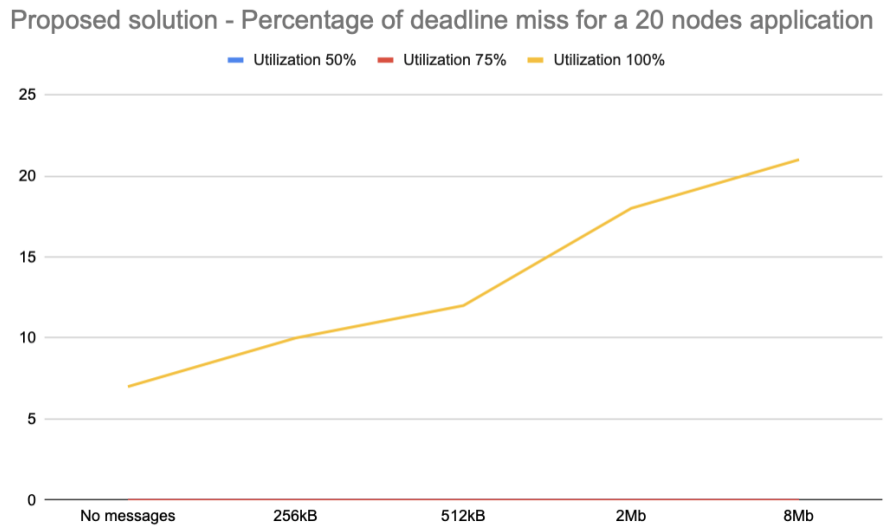



Figure 5.5

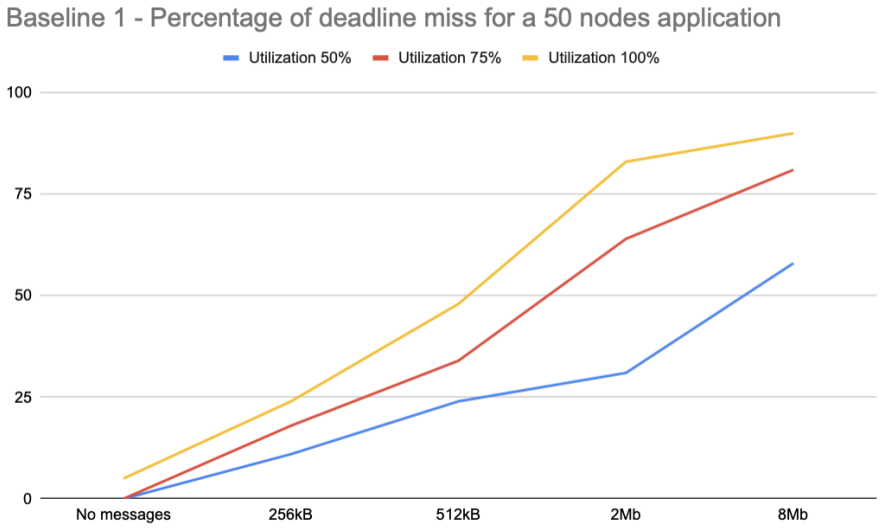



Figure 5.6

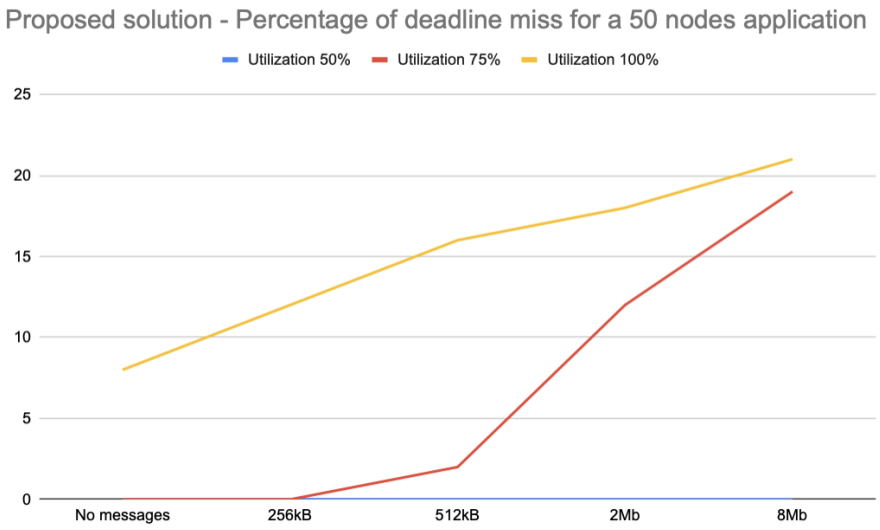



Figure 5.7

### Discussion of the results

The discussion of the results will be split into 3 parts, where results will be reviewed for applications with 4, 20 and 50 nodes respectively.

**Small applications with 4 nodes.** As expected, with low utilization and small message sizes being shared between nodes neither the baseline nor the proposed approach suffer from any deadline miss, as presented on 5.2 and 5.3. However, for bigger message sizes, such as 2Mb or 8Mb, even for a very small node

count, if the utilization is at 75%, the `baseline1` approach already starts suffering from deadline misses. This is a direct result of the significantly increased overhead of message transport compared to intra-process message transport allowed by the proposed approach. This becomes even more visible when running an application with a 100% utilization, where sending messages of small sizes (256Kb) already adds enough overhead to significantly disrupt the execution schedule (around 15% of callbacks miss their deadlines). On the other hand, the little increase in overall executor overhead causes some deadline misses for scenarios without any message exchange when utilizing the proposed `executor`, while the `baseline1` approach was able to stay below 2% of deadline misses.

**Applications with 20 nodes.** A very similar effect, to the one described for applications with 4 nodes, can be observed, as presented on 5.4 and 5.5. The impact on low utilization applications (50%) still remains negligible for both proposed `executor` and the `baseline1` approach. For 75% utilization, the `baseline1` approach already starts suffering from deadline misses for message sizes of 512Kb. For the `baseline1` approach and message sizes of 8Mb, the deadline miss percentage is 50% higher then for an application with 4 nodes. It should be also noticed that for applications without message exchange, with 100% utilization and ran using proposed `executor`, the deadline miss jumps from 4% to 7%.

**Large applications with 50 nodes.** The evaluations for such large applications show even more extreme differences between the `baseline1` and proposed approach, as presented on 5.6 and 5.7. It is worth noticing that, at this point, this would be a very rare setup in reality. Applications most often contain less then 50 nodes, exchange data much smaller than 8Mb and the callback periods are rarely lower than 50ms. However, presenting such examples demonstrates the growing returns of using the approach proposed in this thesis. Similarly to the previously discussed examples for applications with 4 nodes and 20 nodes, the `baseline1` approach suffers an increasing percentage of deadline miss with as the message size increases. Even for 50% utilization and message size of 512Kb, the number of deadline misses averages around 26%. The effect of the increased executor overhead is more visible for the proposed approach in this scenario as well. As presented in Sec.5.4.1, the proposed `executor` overhead increases with the number of subscribers in the application. Hence, as expected, for the test scenario with 50 nodes the proposed approach suffers from deadline misses for applications with utilization of 75% and 100%. The additional `executor` overhead is does not cause deadline misses for applications with 50% utilization.

### 5.4.3 Improving data freshness

#### Hypothesis

As described in Sec.2, the baseline ROS2 implementation always prioritizes periodic callback execution (timers) over subscription callbacks (delivering data). In case a periodic callback depends on data that is received in a subscription callback within same node, it may happen that the periodic callback (executed first) will use old data, while fresher data is already waiting to be delivered.

Our hypothesis is that the proposed improvements presented in this work should improve the following aspect by reducing such events to zero (allow the

periodic callbacks to always use the latest available data). Furthermore, we expect to see that for applications with harmonic periods the data freshness is dramatically reduced. As described in detail in Sec. 2, the producer and consumer will always want to run at the same point in time, hence when producer finishes execution and publishes the data, the data will not be delivered to the consumer first as the execution of it's timer based callback will be prioritized, causing execution using old data.

Furthermore, in a vanilla ROS and for applications with high utilization (where majority of time is spent on executing the periodic callbacks), sometimes the data delivery (i.e., execution of the subscription callback of a node) can be starved for prolonged periods of times. Forcing execution of subscription callbacks prior to periodic ones might cause a slight increase in deadline miss, however, given the fact that efficient intra-process data exchange is used, the overhead should be negligible.

**Setup**

In this test we try to measure the impact of two effects (described in detail in Sec. 2), namely subscription callback starvation and prioritization of timer over subscription callbacks, on the data freshness. In total 36 synthetic application parameters were generated to cover all possible combinations of node count, utilization, periods and message size. Applications with higher utilization should be more prone to subscription callback starvation. Applications with higher node counts (such as 50) should stress test the new proposed executor logic. By comparing different message size we should be able to observe if increased message transport time also has a negative on the data freshness. Lastly, by comparing two different possible period sets, we should be able to observe that applications with harmonic periods suffer from the effect of prioritizing timer callback over subscriptions causing decreased data freshness. The detailed input for the application generation script is presented in Tab. 5.10. The metric for this test is calculated using the following method. First, for each node, the number of executions using old data is divided by the total number of executions for that node; next, the obtained values (ranging from 0 to 1) are averaged out, giving us an average percentage of executions on old data. For the setup used in this test, where 50% of the nodes were subscribers, the worst case scenario is 50% executions on old data (all callback executions that use data produced by nodes that are publishers).

| parameter name | selected values |
|---|---|
| force all combinations | nodes, utilization, periods, message-size |
| nodes | 4 or 20 or 50 |
| utilization | 0.5 or 0.75 or 1.0 |
| periods | [10, 20, 40] or [10,22,47] |
| pubsub | 0.5, 0.5 |
| deadlines | period |
| deadlines_wcet | - |
| priorities | rate-monotonic |
| message_size | 512kB or 8Mb |

Table 5.10: **Synthetic application parameters for measuring the improvements in data freshness**

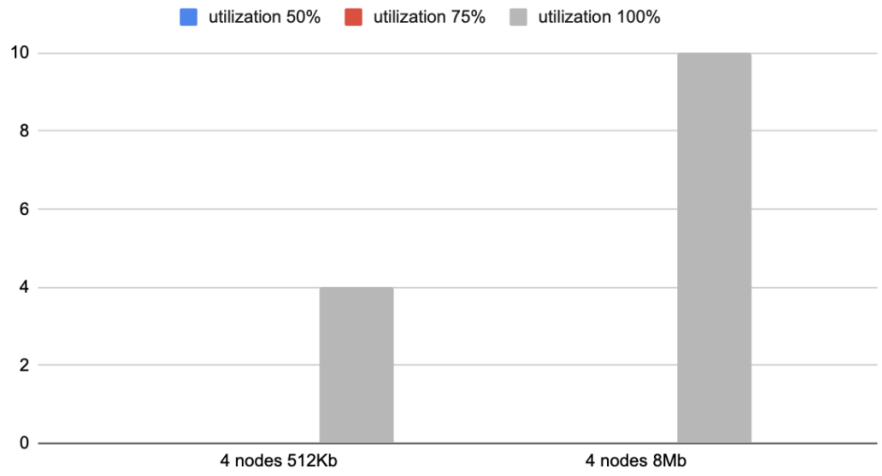



Figure 5.8: **Percentage of periodic computations using old data for 4 node applications**

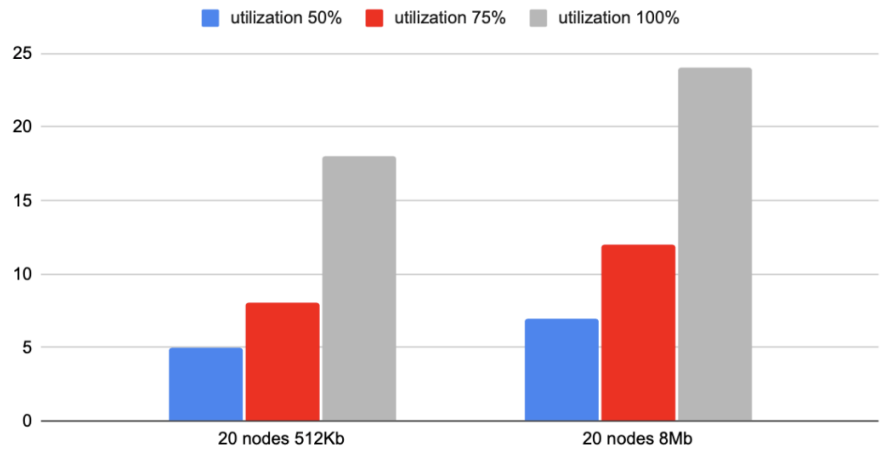



Figure 5.9: **Percentage of periodic computations using old data for 20 node applications**

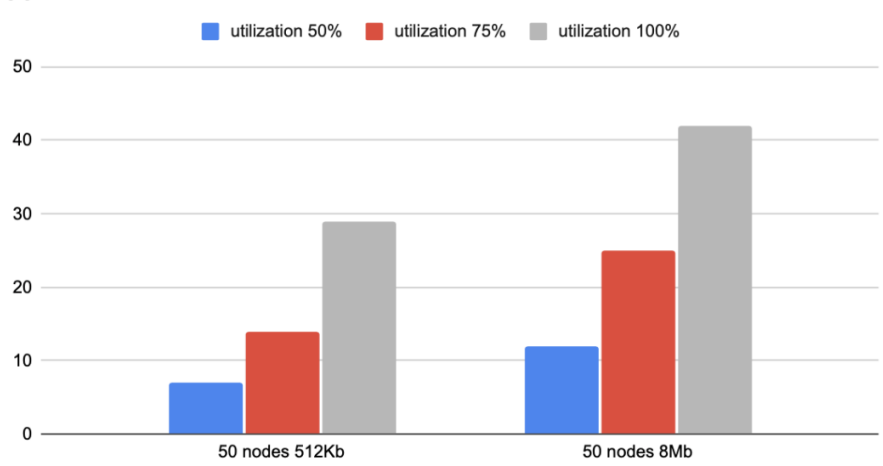



Figure 5.10: **Percentage of periodic computations using old data for 50 node applications**

### Discussion of the results

**Applications with harmonic periods.** First thing that should be noted to understand correctly the presented data is the clarification for lack of values for the proposed approach on Fig. 5.8, Fig. 5.9 and Fig. 5.10. The presented values represent the statistics for the `baseline1` approach only due for the following reasons. First, it was observed, that thanks to the additional logic described in Sec.4, the number of executions on old data was reduced to zero for all tested scenarios! Secondly, to speed up the evaluation procedure and narrow down the comparison to cover only the approaches that allow for callback prioritization, we did not include `baseline0` in this test.

Secondly, the measurements confirmed the hypothesis that for applications with harmonic periods all periodic callbacks that depend on subscribed data will run on old data (potentially except of the first execution prior to the first time data is published). For our scenario, where 50% of the nodes were subscribers, the effect of prioritizing timer based callbacks over subscription callbacks caused 50% of periodic callbacks to execute on old data.

**Applications with non-harmonic periods.** For applications with 4 nodes only utilization of 100% caused the subscription callbacks to be starved for long enough to not allow data delivery prior to periodic callback execution. The problem became more visible for applications with 20 nodes or 50 nodes where the chance of subscription callback starvation increase.

On Fig. 5.9, it can be seen that even for a utilization of 50% and 20 nodes in an application, when messages of 512Kb size are shared, 5% of the callbacks were ran on old data due to subscription callback starvation. Once again, for extreme cases of 100% utilization, message size of 8Mb and 50 nodes in an application, the subscription callback is deprived of execution time almost all the time, resulting in average 42% of periodic callbacks running on old data.

## 5.5 Application-aware scheduling

### Hypothesis

In Sec. 5.4.3 we checked if a callback was ran on old data (newer data was already produced but not delivered to the node). In this test, we extend the data freshness metric to exact data age which is the *time that passed since data was produced to the moment data was used for computation.* The application aware scheduling (a part of the improvements included in the enhanced `executor` implementation described in Sec. 4), allows for creating custom scheduling policies, including ones that control the execution of nodes that produce data for other nodes to ensure meeting data age constraints that might arise depending on the situation that the application is in (examples can be found in Sec. 2). In the case of the test apps, this data constraint will take effect for the whole duration of the app. But you can easily imagine that such constrain can appear only in certain moments during the application run time. Our hypothesis is that our the additional logic from the proposed approach, enhanced with custom scheduling logic that should ensure that the data age does not exceed 40ms, will allow always ensure that the data used for computation is not older than the established limit. This should apply in all scenarios, no matter the number of other nodes in the application. We expect the data age to significantly grow for the `baseline1` approach with the increased number of nodes. Furthermore, we expect that solely the addition of the logic that helps to prevent the negative effects of prioritizing timer over subscription based callbacks (as evaluated in Sec. 5.4.3), will also improve the data age metric. This is why, in the results we compare the `baseline1` approach together with our `proposed` solution with data age constraint monitoring enabled and disabled (disabled means only preventing the prioritization of timer based callbacks).

### Setup

This test required significantly simpler setup, compared with the previously described ones with only 3 application configurations. Each configuration was ran using `baseline1` approach , `proposed` approach and `proposed` approach with data age constraint monitoring enabled. The only difference between configurations was the number of nodes which was either 4, 20 or 50. On top of that, two other nodes were added to each application. Those nodes were the ones that we used to measure the data freshness. One node was a subscriber (listener) node running on highest priority in the application with a period of 100ms, the second node was a publisher (data producer) running on lowest priority in the application with a period of 50ms. The remaining parametrs for the generated application can be found in Tab. 5.11.

| **parameter name** | **selected values** |
|---|---|
| force all combinations | nodes |
| nodes | 4 or 20 or 50 |
| utilization | 0.75 |
| periods | 10, 20, 40 |
| pubsub | 0, 0 |
| deadlines | period |
| deadlines_wcet | - |
| priorities | rate-monotonic |
| message_size | 512kB |

Table 5.11: **Synthetic application parameters for measuring the improvements in data freshness for application aware scheduling**

| | Min | Q1 | Median | Q3 | Max | Mean | SD |
|---|---|---|---|---|---|---|---|
| baseline1 | 4 | 23 | 40 | 45 | 83 | 36.76 | 18.80 |
| proposed | 2 | 15 | 33 | 38 | 74 | 28.51 | 17.46 |
| proposed (data age constraint) | 4 | 14 | 17 | 36 | 39 | 22.86 | 11.64 |

Table 5.12: **Data age for the critical node (application with 6 nodes)**

| | Min | Q1 | Median | Q3 | Max | Mean | SD |
|---|---|---|---|---|---|---|---|
| baseline1 | 7 | 40.25 | 48 | 53 | 92 | 48.18 | 16.16 |
| proposed | 5 | 33.25 | 40.5 | 47.5 | 51 | 37.61 | 11.82 |
| proposed (data age constraint) | 5 | 12 | 13 | 35 | 40 | 18.86 | 11.86 |

Table 5.13: **Data age for the critical node (application with 22 nodes)**

### Discussion of results

By looking at Fig.5.11, Fig.5.12, Fig.5.13, we can see that for all the conducted tests, the `baseline1` approach did not allow to ensure the desired low (40ms) data age. The same also applies for the proposed approach with a standard fixed priority polity. However, due to the additional logic ensuring that the

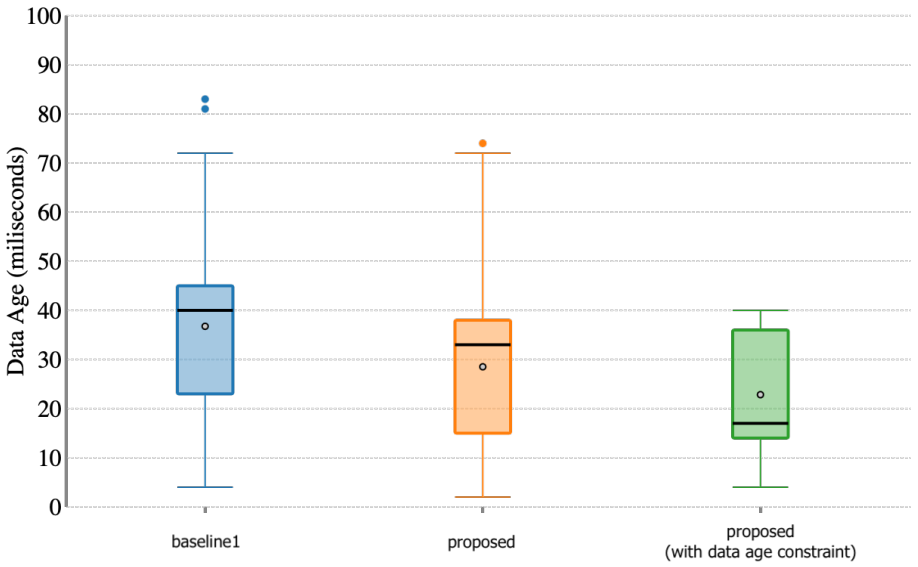



Figure 5.11: **Data age for the critical node (application with 4 nodes)**

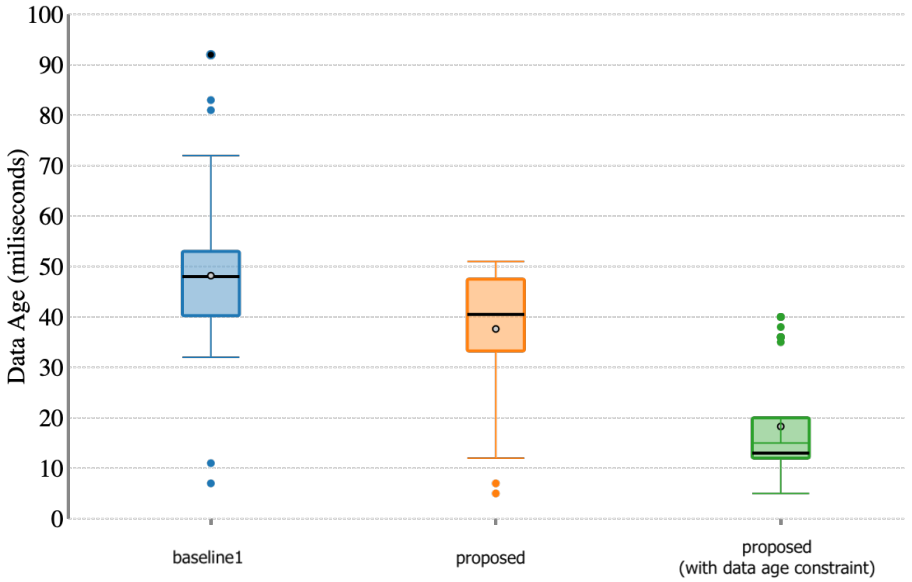



Figure 5.12: **Data age for the critical node (application with 22 nodes)**

relevand subscription callback is called prior to the timer callback, the average data age was 22% lower than for the `baseline1` approach for an application with 6 nodes. By looking at Fig.5.11, Fig.5.12, Fig.5.13, we can see that the difference between the `baseline1` and the proposed approach does not change with the increasing size of the application.

| | Min | Q1 | Median | Q3 | Max | Mean | SD |
|---|---|---|---|---|---|---|---|
| baseline1 | 22 | 45.5 | 63 | 70 | 101 | 60.05 | 18.03 |
| proposed | 29 | 58 | 60.5 | 63 | 92 | 58.63 | 10.57 |
| proposed (data age constraint) | 5 | 12 | 13 | 35 | 38 | 19.3 | 11.8 |

Table 5.14: **Data age for the critical node (application with 52 nodes)**

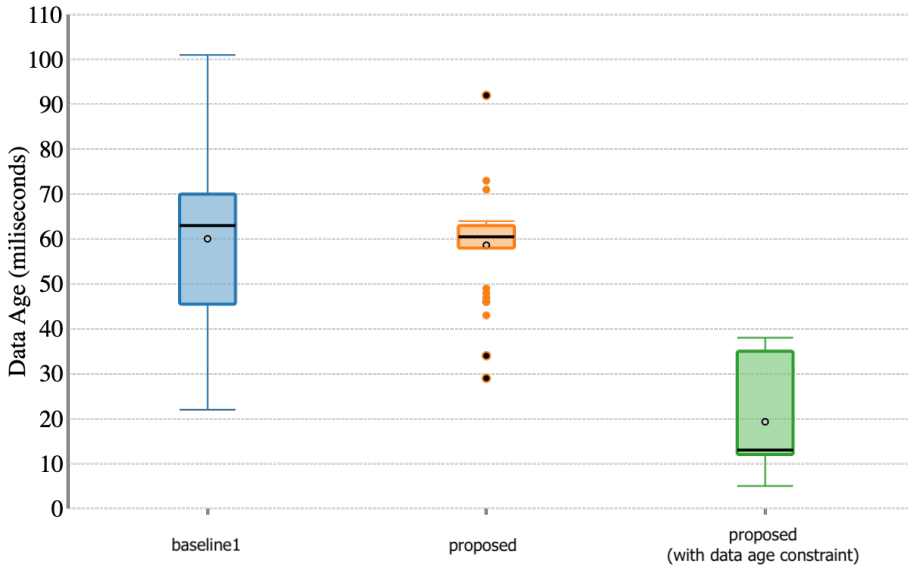



Figure 5.13: **Data age for the critical node (application with 52 nodes)**

By looking at the results for the proposed approach with data age constraint enabled we can see that, as expected, no matter the size of the application, the data age never exceeded the 40ms limit. With the increasing node count, the data age for both `baseline1` and proposed approach was growing. As a result, for the biggest tested application (52 nodes) the data age achieved by the proposed approach with data age constrain enabled was on average 67% lower.

# Chapter 6

# Conclusion

In this chapter we summarize the results of the presented work. We revisit each described aspect and look into potential next steps as we talk about the future work.

We started by doing a deep dive into the details of callback scheduling in the current ROS2 implementation. We investigated the flow of the ROS2 executor mechanism to discover potential challenges for implementing latency sensitive applications based on periodic tasks. We presented a dillema that ROS2 developers might face when aiming to achieve efficient data exchange between nodes together with callback prioritization capabilities. We discovered that those two aspects can not be combined and choosing just can cause unpredictable system behavior in terms of order of task execution or significant overhead caused by message transport. Furthermore, we present the need for application-aware scheduling and describe the limitations of ROS2 that so far did not allow to easily implement such functionality.

We followed up with a short test presenting noticeable differences between DDS based and intra-process based data exchange. This allowed us to confirm the hypothesis that with the growing number of callbacks in the system and increasing message size the message transport time might cause more constraints in terms of possible utilization.

In order to gain better insight into all the aspects of scheduling tasks in ROS2 we performed a review of a set of published work, both analytical and practical. We discovered that researchers so far focused mostly on analyzing and improving the end-to-end latency of callback chains while less atention was drawn towards purely periodic callback sets.

After establishing the areas of potential improvements we set out to implement a prototype of changes for the ROS2 executor that would allow to both utilize efficient intra-process data exchange and maintain the ability to schedule callbacks according to desired priorities. We implemented our changes into the heard of the executor logic to allow for executing callbacks based on a set of policies that we implemented as well. At the same time, we created an API for implementing any custom scheduling policy. This allowed us to also tackle the topic of application-aware scheduling and address scenarios where applications have to adjust to momentary changes in timing-constraints.

We concluded our work by performing a detailed analysis of all the aspects that we aimed to improve with our proposed solution. We find that even though

the proposed solution causes a 54% increase for medium sized applications (20 nodes), the overhead is still neglegible in the context of the whole application and the benefits of allowing for reduced message transport time overhead overweight the drawbacks of increased overhead. We find that with the proposed solution, for applications with utilization of 75% we were able to reduce the number of deadline misses from around 26% to 3%. Furthermore, we were able to improve the data freshness aspect by ensuring that no cases of execution on old data happen during the whole time of the application execution, compared to up to 42% for baseline approaches.

In order to efficiently perform the evaluation, a tool-set for creating artificial ROS2 applications was created together with script for executing the tests in larger batches.

## 6.1 Future work

It is important to notice that we realise that the proposed solution is soely a prototype and detailed checks would have to be executed prior to potentially submiting a pull-request to the main ROS2 repository. However it is one of our main goals to bring more visibility to the presented problems and make sure that the framework can be published. Allowing other members of the ROS2 community to experiment with the proposed approach would surely bring further insight and allow for fine tuning the solution. We would like to take a look into a set of existing industrial ROS2 based projects and investigate the potential gains of porting them over to the solution proposed in this thesis.

Our framework currently requires 2 isolated cores to allow uniterrupted execution. This is definitely a significant limitation and would potentially limit the number of platforms that can be targeted with our solution.

Despite dispatching work into multiple threads, our executor does not support concurrency and cannot make full use of multicore platforms.

# Bibliography


[1] Tobias Blass, Arne Hamann, Ralph Lange, Dirk Ziegenbein, and Björn B. Brandenburg. Automatic latency management for ros 2: Benefits, challenges, and open problems. In *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 264–277, 2021.

[2] Brian Gerkey. Ros2. `https://spectrum.ieee.org/ros-robot-operating-system-celebrates-8-years`, 2015.

[3] Alan Burns and Chris Dale. Scheduling and timing analysis for safety-critical real-time systems. *ELECTRONICS WORLD*, 116:18–20, 02 2010.

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

[5] Raimarius Delgado, Bum-Jae You, and Byoungwook Choi. Real-time control architecture based on xenomai using ros packages for a service robot. *Journal of Systems and Software*, 151, 05 2019.

[6] Carlos Gutiérrez, Lander Juan, Irati Ugarte, and Víctor Vilches. Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications, 09 2018.

[7] Ramyad Hadidi, Nima Ghalehshahi, Bahar Asgari, and Hyesoon Kim. Context-aware task handling in resource-constrained robots with virtualization, 04 2021.

[8] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. pages 1–10, 10 2016.

[9] Fulvio Mastrogiovanni, Ali Paikan, and Antonio Sgorbissa. Semantic-aware real-time scheduling in robotics. *Robotics, IEEE Transactions on*, 29:118–135, 02 2013.

[10] Mitra Nasri and Gerhard Fohler. Non-work-conserving non-preemptive scheduling: Motivations, challenges, and potential solutions. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 165–175, 2016.

[11] Open Robotics. Ros2. `https://docs.ros.org/en/foxy/#`, 2021.

[12] Charles Randolph. Improving the predictability of event chains in ros 2. Master thesis, Delft University of Technology, Delft, The Netherlands, 2021.

[13] Yukihiro Saito, Futoshi Sato, Takuya Azumi, Shinpei Kato, and Nobuhiko Nishio. Rosch:real-time scheduling framework for ros. pages 52–58, 08 2018.

[14] Hideki Takase, Tomoya Mori, Kazuyoshi Takagi, and Naofumi Takagi. mros: A lightweight runtime environment for robot software components onto embedded devices. pages 1–6, 06 2019.

[15] Yue Tang, Zhiwei Feng, Nan Guan, Xu Jiang, Mingsong Lv, Qingxu Deng, and Wang Yi. Response time analysis and priority assignment of processing chains on ros2 executors. In *2020 IEEE Real-Time Systems Symposium (RTSS)*, pages 231–243, 2020.

# Appendix A

# Sample usage of proposed API

Listing A.1: **Example usage of one of the prototyped APIs with access to scheduling strategies that allow to dynamically change the scheduling behavior at run-time**

```
auto scheduling_scenario_default = std::make_shared<
    SchedulingScenario>("default_scenario");
scheduling_scenario_default->add_timer("
    scheduling_id_timer3", 98, 50, {"subscribes":"
    topic1"});
scheduling_scenario_default->add_timer("
    scheduling_id_timer4", 97, 20, {"publishes":"
    topic1"});
scheduling_scenario_default->
    set_scheduling_policy(&npedf_policy);

auto scheduling_scenario_special = std::
    make_shared<SchedulingScenario>("
    special_scenario");
scheduling_scenario_special->add_timer("
    scheduling_id_timer3", 98, 40, {"subscribes":"
    topic1"});
scheduling_scenario_special->add_timer("
    scheduling_id_timer4", 97, 20, {"publishes":"
    topic1"});
scheduling_scenario_special->
    set_scheduling_policy_function(&
    custom_scheduling_policy);

auto scheduling_strategy = std::make_shared<
    SchedulingStrategy>();
scheduling_strategy->add_scenario(
    scheduling_scenario_default);
scheduling_strategy->add_scenario(
```

```
    scheduling_scenario_special);
scheduling_strategy->set_default_scenario("
    default_scenario");

auto node_handle_node3 = std::make_shared<node3
    >();
auto node_handle_node4 = std::make_shared<node4
    >();

node_handle_node3->set_scheduling_strategy(
    scheduling_strategy);
node_handle_node4->set_scheduling_strategy(
    scheduling_strategy);

rclcpp::executors::SchedulingExecutor executor;
executor.add_node(node_handle_node3);
executor.add_node(node_handle_node4);
executor.set_scheduling_strategy(
    scheduling_strategy);
executor.spin();
```