



D4.17

Real-time Executor Software Release Y4

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.17
Deliverable name	Real-time Executor – Software Release Y4
Date	December 2021
Dissemination level	public
Workpackage and task	4.2
Author	Jan Staschulat (Bosch)
Contributors	Ralph Lange (Bosch)
Keywords	micro-ROS, robotics, ROS, microcontrollers, scheduling, executor
Abstract	This document provides links to the released software and documentation for deliverable D4.17 <i>Real-time Executor Software Release Y4</i> of the Task 4.2 <i>Predictable Scheduling and Execution</i> .



Contents

1	Overview to Results	2
2	Links to Software Repositories	2
3	Annex 1: Webpage on Real-Time Executor	3
3.1	Table of contents	3
3.2	Introduction	4
3.3	Analysis of rclcpp standard Executor	6
3.3.1	Architecture	6
3.3.2	Scheduling Semantics	7
3.4	Analysis of processing patterns	8
3.4.1	Sense-plan-act pipeline in robotics	8
3.4.2	Synchronization of multiple rates	9
3.4.3	High-priority processing path	10
3.4.4	Real-time embedded applications	10
3.5	rclc Executor	12
3.5.1	Features	13
3.5.2	Executor API	17
3.5.3	Examples	18
3.5.4	Future work	22
3.5.5	Download	22
3.6	Callback-group-level Executor	22
3.6.1	API Changes	23
3.6.2	Test Bench	24
3.7	Related Work	24
3.7.1	Fawkes Framework	25
3.8	References	27
3.9	Acknowledgments	28

1 Overview to Results

This document provides links to the released software and documentation for deliverable D4.17 *Real-time Executor Software Release Y4* of the Task 4.2 *Predictable Scheduling and Execution*.

As an entry-point to all software and documentation, we created a dedicated webpage on the micro-ROS website: https://micro.ros.org/docs/concepts/client_library/execution_management/

The annex includes a copy of this webpage and a copy of the major documentation file of this software release.

2 Links to Software Repositories

rcl Executer repository:

- Git repository: <https://github.com/ros2/rcl>
 - Branch name: [master](#)
Lastest release: [v3.0.5-rolling](#)
Latest commit: [27b4d4f](#)
 - Branch name: [Galactic](#)
Lastest release: [v2.0.5-galactic](#)
Latest commit: [23d6167](#)
 - Branch name: [Foxy](#)
Lastest release: [v1.0.2-foxy](#)
Latest commit: [26ccfd6](#)

Demos for rcl Executer in micro-ROS demo repositories:

- Git repository: https://github.com/micro-ROS/micro_ros_setup#standalone-build-system-tools
Branch name: [Galactic](#)
Latest commit: [92599b1](#)
- Git repository: https://github.com/micro-ROS/zephyr_apps
Branch name: [Galactic](#)
Latest commit: [aadef04](#)
- Git repository: https://github.com/micro-ROS/freertos_apps
Branch name: [Galactic](#)
Latest commit: [7c24435](#)
- Git repository: https://github.com/micro-ROS/micro_ros_arduino
Branch name: [Galactic](#)
Latest commit: [9ca06b0](#)

- Git repository: https://github.com/micro-ROS/micro_ros_mbed
Branch name: [Galactic](#)
Latest commit: [6209542](#)
- Git repository: https://github.com/micro-ROS/micro_ros_azure_rtos_app
Branch name: [main](#)
Latest commit: [3e0f605](#)
- Git repository: https://github.com/micro-ROS/micro_ros_nuttx_app
Branch name: [Galactic](#)
Latest commit: [1f88263](#)

Callback-group-based Executor:

- Pull request on ROS 2 rclcpp repository: <https://github.com/ros2/rclcpp/pull/1218/commits>

Demo for Callback-group-based Executor:

- Git repository: https://github.com/ros2/examples/tree/master/rclcpp/executors/cbg_executor
Branch name: [master](#)
Latest commit: [447580d](#)

3 Annex 1: Webpage on Real-Time Executor

Content of https://micro.ros.org/docs/concepts/client_library/execution_management/ from 16th December 2021.

3.1 Table of contents

- [Introduction](#)
- [Analysis of rclcpp standard Executor](#)
 - [Architecture](#)
 - [Scheduling Semantics](#)
- [Analysis of processing patterns](#)
 - [Sense-plan-act pipeline in robotics](#)
 - [Synchronization of multiple rates](#)
 - [High-priority processing path](#)
 - [Real-time embedded applications](#)
- [rclc Executor](#)

- Features
 - * Trigger condition
 - * Sequential execution
 - * LET-Semantics
 - * Multi-threading and scheduling configuration
- Executor API
 - * Configuration phase
 - * Running phase
- Examples
 - * Sense-plan-act pipeline in robotics example
 - * Synchronization of multiple rates example
 - * High-priority processing path example
 - * Real-time embedded applications example
 - * ROS 2 Executor Workshop Reference System
- Future work
- Download

- Callback-group-level Executor
 - API Changes
 - Test Bench

- Related Work
 - Fawkes Framework

- References

- Acknowledgments

3.2 Introduction

Predictable execution under given real-time constraints is a crucial requirement for many robotic applications. While the service-based paradigm of ROS allows a fast integration of many different functionalities, it does not provide sufficient control over the execution management. For example, there are no mechanisms to enforce a certain execution order of callbacks within a node. Also the execution order of multiple nodes is essential for control applications in mobile robotics. Cause-effect-chains comprising of sensor acquisition, evaluation of data and actuation control should be mapped to ROS nodes executed in this order, however there are no explicit mechanisms to enforce it. Furthermore, when data recordings collected in field tests as ROS-bags are re-played, then the results are often surprisingly different due to non-determinism of process scheduling.

Manually setting up a particular execution order of subscriptions and publishing topics as well as defining use-case specific priorities of the corresponding Linux processes is always possible. However, this approach is error-prone, difficult to extend and requires an in-depth knowledge of the deployed ROS 2 packages in the system.

Therefore the goal of the Executor in micro-ROS is to support roboticists with practical and easy-to-use real-time mechanisms which provide solutions for: - Deterministic execution - Real-time guarantees - Integration of real-time and non real-time functionalities on one platform - Specific support for RTOS and microcontrollers

In ROS 1 a network thread is responsible for receiving all messages and putting them into a FIFO queue (in `roscpp`). That is, all callbacks were called in a FIFO manner, without any execution management. With the introduction of DDS (data distribution service) in ROS 2, the messages are buffered in DDS. In ROS 2, an Executor concept was introduced to support execution management. At the `rcl`-layer, a *wait-set* is configured with handles to be received and in a second step, the handles are taken from the DDS-queue. A handle is a generic term defined in `rcl`-layer for timers, subscriptions, services, clients and guard conditions.

The standard implementation of the ROS 2 Executor for the C++ API (`rclcpp`) has, however, certain unusual features, like precedence of timers over all other DDS handles, non-preemptive round-robin scheduling for non-timer handles and considering only one input data for each handle (even if multiple could be available). These features have the consequence, that in certain situations the standard `rclcpp` Executor is not deterministic and it makes guaranteeing real-time requirements very hard [CB2019]. We have not looked at the ROS 2 Executor implementation for Python Frontend (`rclpy`) because we consider a micro-controllers platform, on which typically C or C++ applications will run.

Given the goals for a Real-Time Executor and the limitations of the ROS 2 standard `rclcpp` Executor, the challenges are: - to develop an adequate and well-defined scheduling mechanisms for the ROS 2 framework and the real-time operating system (RTOS) - to define an easy-to-use interface for ROS developers - to model requirements (like latencies, determinism in subsystems) - mapping of ROS 2 framework and operating system schedulers (semi-automated and optimized mapping is desired as well as generic, well-understood framework mechanisms)

Our approach is to provide a real-time-capable Executor for the `rcl+rcl` layer (as described in section [Introduction to Client Library](#).) in the C programming language .

As the first step, we propose the `rcl` Executor for the `rcl`-layer in C programming language with several new features to support real-time and deterministic execution: It supports 1.) user-defined static sequential execution, 2) conditional execution semantics, 3) multi-threaded execution with scheduling configuration, and 4) logical execution semantics (LET). Sequential execution refers to the runtime behavior, that all callbacks are executed in a pre-defined order independent of the arrival time of messages. Conditional execution is available with a trigger condition which enables typical processing patterns in robotics (which are analyzed in detail in section [Analysis of processing patterns](#)). Configuration of scheduling parameters for multi-threaded application accomplishes prioritized execution. The logical execution time concept (LET) provides data synchronization for fixed periodic task scheduling of embedded applications.

Beyond the advanced execution management mechanisms for micro-ROS, we also contributed to improving and extending the Executor concept in `rclcpp` for standard ROS 2: the callback group-level Executor. It is not a new Executor but rather a refinement of the ROS 2 Executor API allowing to prioritize a group of callbacks which is not possible with the ROS 2 default Executor in its current Galactic release.

3.3 Analysis of rclcpp standard Executor

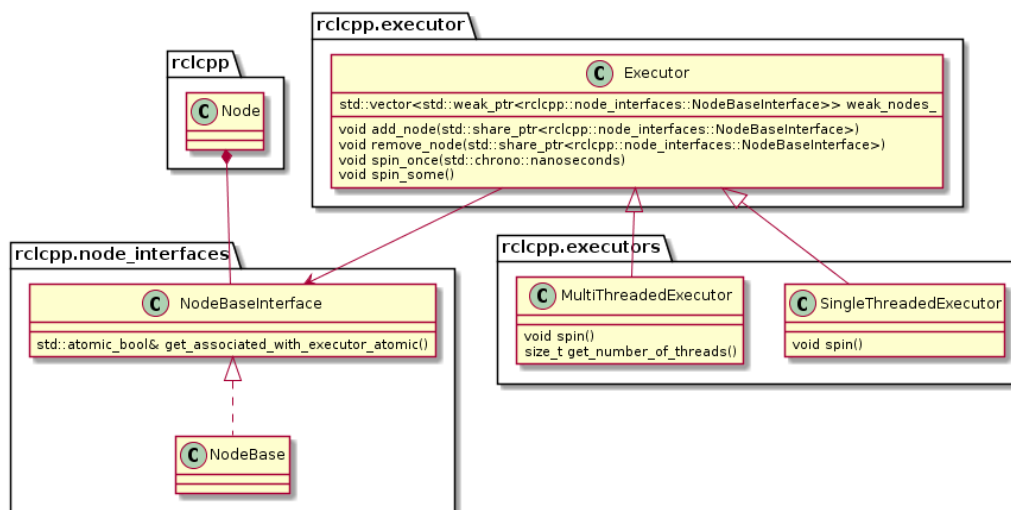
ROS 2 allows to bundle multiple nodes in one operating system process. To coordinate the execution of the callbacks of the nodes of a process, the Executor concept was introduced in rclcpp (and also in rclpy).

The ROS 2 design defines one Executor (instance of `rclcpp::executor::Executor`) per process, which is typically created either in a custom main function or by the launch system. The Executor coordinates the execution of all callbacks issued by these nodes by checking for available work (timers, services, messages, subscriptions, etc.) from the DDS queue and dispatching it to one or more threads, implemented in `SingleThreadedExecutor` and `MultiThreadedExecutor`, respectively.

The dispatching mechanism resembles the ROS 1 spin thread behavior: the Executor looks up the wait sets, which notifies it of any pending callback in the DDS queue. If there are multiple pending callbacks, the ROS 2 Executor executes them in the order as they were registered at the Executor.

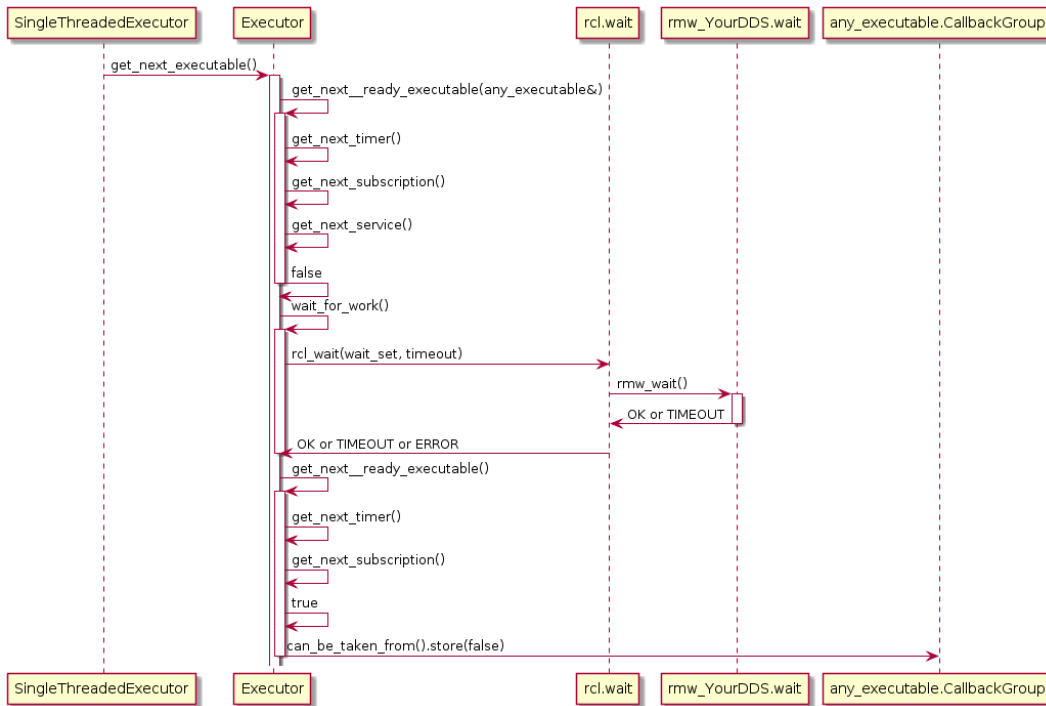
3.3.1 Architecture

The following diagram depicts the relevant classes of the standard ROS 2 Executor implementation:



Note that an Executor instance maintains weak pointers to the NodeBaseInterfaces of the nodes only. Therefore, nodes can be destroyed safely, without notifying the Executor.

Also, the Executor does not maintain an explicit callback queue, but relies on the queue mechanism of the underlying DDS implementation as illustrated in the following sequence diagram:



The Executor concept, however, does not provide means for prioritization or categorization of the incoming callback calls. Moreover, it does not leverage the real-time characteristics of the underlying operating-system scheduler to have finer control on the order of executions. The overall implication of this behavior is that time-critical callbacks could suffer possible deadline misses and a degraded performance since they are serviced later than non-critical callbacks. Additionally, due to the FIFO mechanism, it is difficult to determine usable bounds on the worst-case latency that each callback execution may incur.

3.3.2 Scheduling Semantics

In a recent paper [CB2019], the rclcpp Executor has been analyzed in detail and a response time analysis of cause-effect chains has been proposed under reservation-based scheduling. The Executor distinguishes four categories of callbacks: *timers*, which are triggered by system-level timers, *subscribers*, which are triggered by new messages on a subscribed topic, *services*, which are triggered by service requests, and *clients*, which are triggered by responses to service requests. The Executor is responsible for taking messages from the input queues of the DDS layer and executing the corresponding callback. Since it executes callbacks to completion, it is a non-preemptive scheduler. However it does not consider all ready tasks for execution, but only a snapshot, called *readySet*. This *readySet* is updated when the Executor is idle and in this step it interacts with the DDS layer updating the set of ready tasks. Then for every type of task, there are dedicated queues (timers, subscriptions, services, clients) which are processed sequentially. The following undesired properties were pointed out:

- Timers have the highest priority. The Executor processes *timers* always first. This can lead to the intrinsic effect, that in overload situations messages from the DDS queue are not processed.
- Non-preemptive round-robin scheduling of non-timer handles. Messages arriving during the processing of the *readySet* are not considered until the next update, which depends on the execution time of all remaining callbacks. This leads to priority inversion, as lower-priority callbacks may implicitly block higher-priority callbacks by prolonging the current processing of the *readySet*.

- Only one message per handle is considered. The readySet contains only one task instance, For example, even if multiple messages of the same topic are available, only one instance is processed until the Executor is idle again and the readySet is updated from the DDS layer. This aggravates priority inversion, as a backlogged callback might have to wait for multiple processing of readySets until it is considered for scheduling. This means that non-timer callback instances might be blocked by multiple instances of the same lower-priority callback.

Due to these findings, the authors present an alternative approach to provide determinism and to apply well-known schedulability analyses to a ROS 2 systems. A response time analysis is described under reservation-based scheduling.

3.4 Analysis of processing patterns

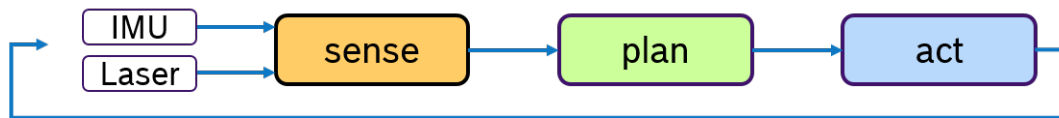
The development of an execution management mechanism for micro-ROS is based on an analysis of processing patterns commonly used in robotics and embedded systems. First, typical processing patterns in mobile robotics are presented which are used to implement deterministic behavior. Then, the processing patterns in the real-time embedded systems are analyzed, in which, as a main difference, the time-triggered paradigm is applied to accomplish real-time behavior.

3.4.1 Sense-plan-act pipeline in robotics

Now we describe common software design patterns which are used in mobile robotics to achieve deterministic behavior. For each design pattern we describe the concept and the derived requirements for a deterministic Executor.

Concept:

A common design paradigm in mobile robotics is a control loop, consisting of several phases: A sensing phase to acquire sensor data, a plan phase for localization and path planning and an actuation-phase to steer the mobile robot. Of course, more phases are possible, here these three phases shall serve as an example. Such a processing pipeline is shown in the next figure.



Typically multiple sensors are used to perceive the environment. For example an IMU and a laser scanner. The quality of localization algorithms highly depend on how old such sensor data is when it is processed. Ideally the latest data of all sensors should be processed. One way to achieve this is to execute first all sensor drivers in the sense-phase and then process all algorithms in the plan-phase.

Currently, such a processing order cannot be defined with the default Executor of rclcpp. One could in principle design a data-driven pipeline, however if e.g. the Laser scan is needed by some other callback in the sense-phase as well as in the plan-phase, the processing order of these subscribers is arbitrary.

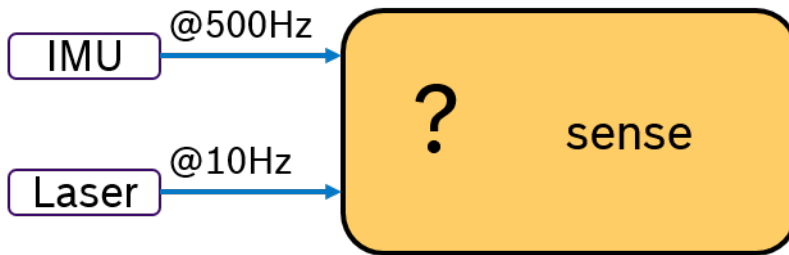
For this sense-plan-act pattern, we could define one executor for each phase. The plan-phase would be triggered only when all callbacks in the sense-phase have finished.

Derived requirements: - triggered execution

3.4.2 Synchronization of multiple rates

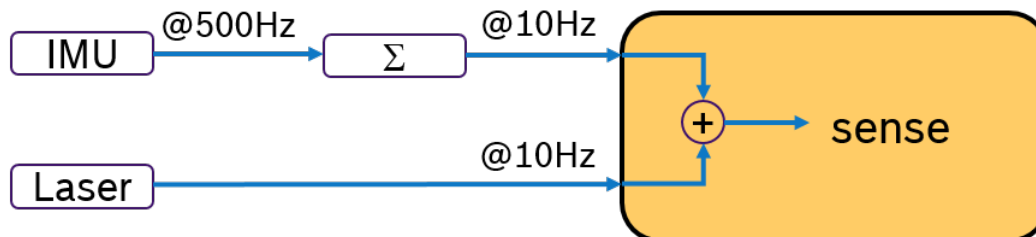
Concept:

Often multiple sensors are being used to sense the environment for mobile robotics. While an IMU sensor provides data samples at a very high rate (e.g., 500 Hz), laser scans are available at a much slower frequency (e.g. 10Hz) determined by the revolution time. Then the challenge is, how to deterministically fuse sensor data with different frequencies. This problem is depicted in the next figure.



Due to scheduling effects, the callback for evaluating the laser scan might be called just before or just after an IMU data is received. One way to tackle this is to write additional synchronization code inside the application. Obviously, this is a cumbersome and not-portable solution.

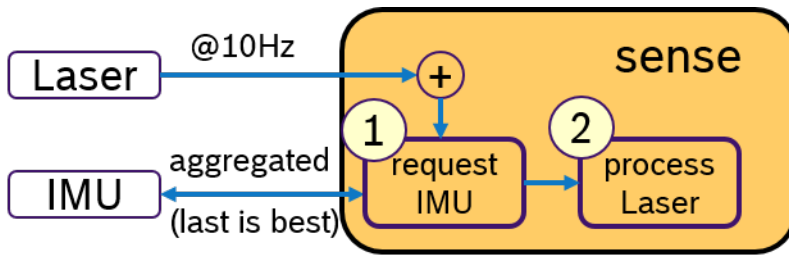
An Alternative would be to evaluate the IMU sample and the laser scan by synchronizing their frequency. For example by processing always 50 IMU samples with one laser scan. This approach is shown in the next figure. A pre-processing callback aggregates the IMU samples and sends an aggregated message with 50 samples at 10Hz rate. Now both messages have the same frequency. With a trigger condition, which fires when both messages are available, the sensor fusion algorithm can expect always synchronized input data.



Example: Synchronization with a trigger

In ROS 2, this is currently not possible to be modeled because of the lack of a trigger concept in the Executors of rclcpp and rclpy. Message filters could be used to synchronize input data based on the timestamp in the header, but this is only available in rclcpp (and not in rcl). Further more, it would be more efficient to have such a trigger concept directly in the Executor.

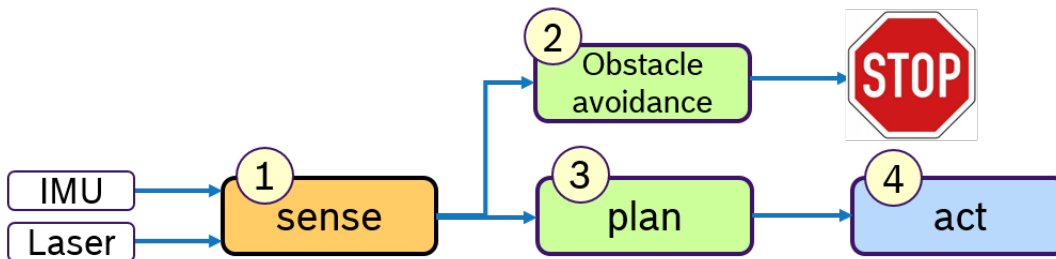
Another idea would be to actively request for IMU data only when a laser scan is received. This concept is shown in the next figure. Upon arrival of a laser scan message, first, a message with aggregated IMU samples is requested. Then, the laser scan is processed and later the sensor fusion algorithm. An Executor, which would support sequential execution of callbacks, could realize this idea.



Derived requirements: - triggered execution - sequential processing of callbacks

3.4.3 High-priority processing path

Concept Often a robot has to fulfill several activities at the same time. For example following a path and avoiding obstacles. While path following is a permanent activity, obstacle avoidance is triggered by the environment and should be immediately reacted upon. Therefore one would like to specify priorities to activities. This is depicted in the next figure:

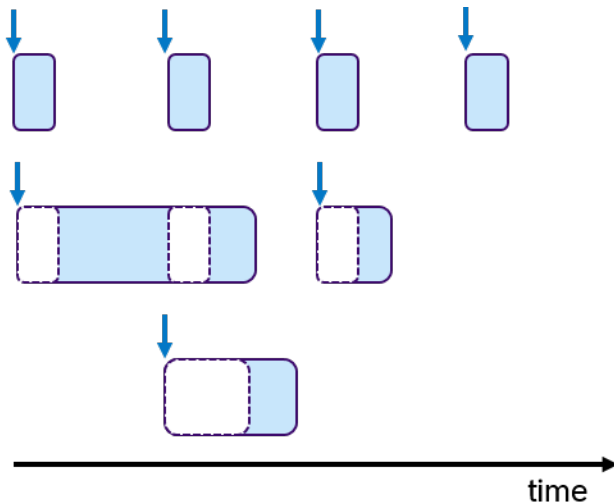


Assuming a simplified control loop with the activities sense-plan-act, the obstacle avoidance, which might temporarily stop the robot, should be processed before the planning phase. In this example we assume that these activities are processed in one thread.

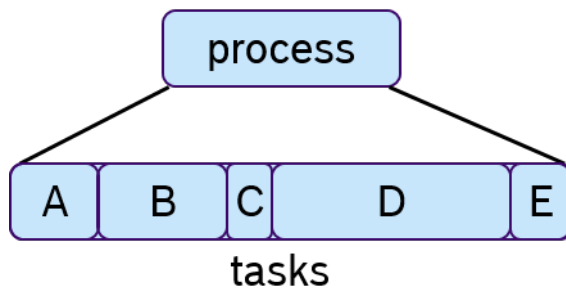
Derived requirements: - sequential processing of callbacks

3.4.4 Real-time embedded applications

In embedded systems, real-time behavior is approached by using the time-triggered paradigm, which means that the processes are periodically activated. Processes can be assigned priorities to allow pre-emptions. The next figure shows an example, in which three processes with fixed periods are shown. The middle and lower process are preempted multiple times depicted with empty dashed boxes.



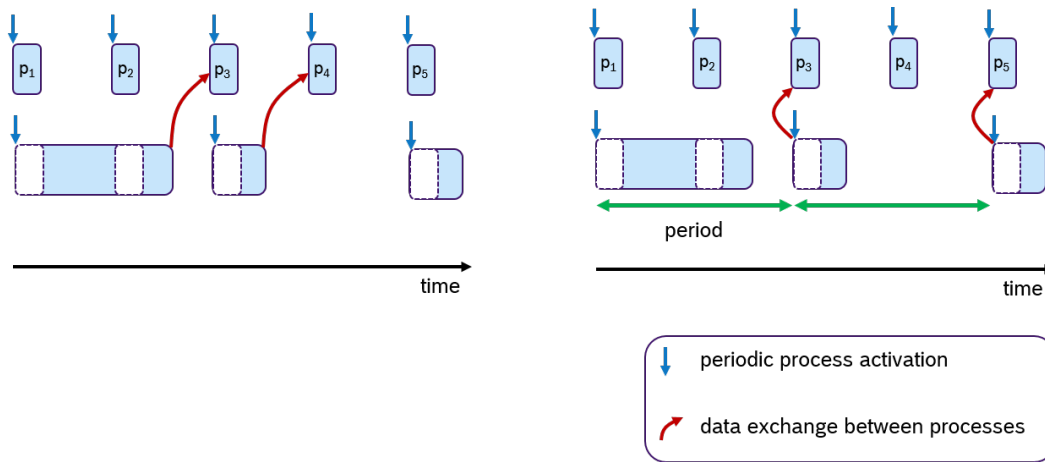
To each process one or multiple tasks can be assigned, as shown in the next figure. These tasks are executed sequentially, which is often called cooperative scheduling.



While there are different ways to assign priorities to a given number of processes, the rate-monotonic scheduling assignment, in which processes with a shorter period have a higher priority, has been shown optimal if the processor utilization is less than 69% [LL1973].

In the last decades many different scheduling approaches have been presented, however fixed-periodic preemptive scheduling is still widely used in embedded real-time systems [KZH2015]. This becomes also obvious, when looking at the features of current operating systems. Like Linux, real-time operating systems, such as NuttX, Zephyr, FreeRTOS, QNX etc., support fixed-periodic preemptive scheduling and the assignment of priorities, which makes the time-triggered paradigm the dominant design principle in this domain.

However, data consistency is often an issue when preemptive scheduling is used and if data is being shared across multiple processes via global variables. Due to scheduling effects and varying execution times of processes, writing and reading these variables could occur sometimes sooner or later. This results in an latency jitter of update times (the timepoint at which a variable change becomes visible to other processes). Race conditions can occur when multiple processes access a variable at the same time. So solve this problem, the concept of logical-execution time (LET) was introduced in [HHK2001], in which communication of data occurs only at pre-defined periodic time instances: Reading data only at the beginning of the period and writing data only at the end of the period. The cost of an additional latency delay is traded for data consistency and reduced jitter. This concept has also recently been applied to automotive applications [NSP2018].



An Example of the LET concept is shown in the Figure. Assume that two processes are communicating data via one global variable. The timepoint when this data is written is at the end of the processing time. In the default case (left side), the process p3 and p4 receive the update. At the right side of the figure, the same scenario is shown with LET semantics. Here, the data is communicated only at period boundaries. In this case, the lower process communicates at the end of the period, so that always process p3 and p5 receive the new data.

Concept: - periodic execution of processes - assignment of fixed priorities to processes - preemptive scheduling of processes - co-operative scheduling of tasks within a process (sequential execution) - data synchronization with LET-semantics

While periodic activation is possible in ROS 2 by using timers, preemptive scheduling is supported by the operating system and assigning priorities on the granularity of threads/processes that correspond to the ROS nodes; it is not possible to sequentially execute callbacks, which have no data-dependency. Furthermore data is read from the DDS queue just before the callback is executed and data is written sometime during the time the application is executed. While the `spin_period` function of the `rclcpp` Executor allows to check for data at a fixed period and executing those callbacks for which data is available, however, with this `spin`-function does not execute all callbacks irrespective whether data is available or not. So `spin_period` is not helpful to periodically execute a number of callbacks (aka tasks within a process). So we need a mechanism that triggers the execution of multiple callbacks (aka tasks) based on a timer. Data transmission is achieved via DDS which does not allow to implement a LET-semantics. To summarize, we derive the following requirements.

Derived requirements: - trigger the execution - sequential processing of callbacks - data synchronization with LET semantics

3.5 rclc Executor

The `rclc` Executor is a ROS 2 Executor based on the `rcl-layer` in C programming language. As discussed above, the default `rclcpp` Executor is not suitable to implement real-time applications because of three main reasons: timers are preferred over all other handles, no prioritization of callback execution and the round-robin to completion execution of callbacks. On the other hand, several processing patterns have been developed as best practices to achieve non-functional requirements, such as bounded end-to-end latencies, low jitter of response times of cause-effect chains, deterministic processing and short response times even in overload situations. These processing patterns are difficult to implement with the concepts available in the default ROS 2 Executor, therefore we have developed a flexible Executor: the `rclc` Executor.

3.5.1 Features

The rclc Executor is feature-complete, i.e. it supports all event types as the default ROS 2 Executor, which are: - subscriptions - timers - services - clients - guard conditions - actions - lifecycle

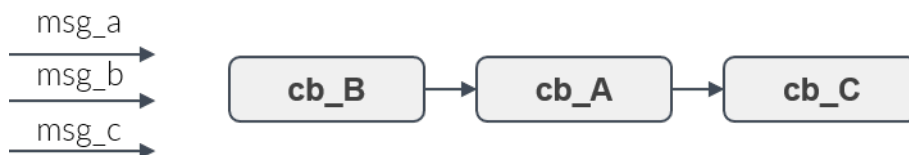
The flexible rclc Executor provides on top the following new features: - triggered execution - user-defined sequential execution - multi-threading and scheduling configuration (WIP) - LET-semantics for data synchronization of periodic process scheduling

First, a *trigger condition* allows to define when the processing of a callback shall start. This is useful to implement sense-plan-act control loops or more complex processing structures with directed acyclic graphs. Second, a user can specify the *processing order* in which these callbacks will be executed. With this feature, the pattern of sensor fusion with multiple rates, in which data is requested from a sensor based on the arrival of some other sensor, can be easily implemented. Third, the assignment of scheduling parameters (e.g., priorities) of the underlying operating system. With this feature, prioritized processing can be implemented. Finally, for periodic applications, the *LET Semantics* has been implemented to support data consistency for periodic process scheduling. These features are now described in more detail.

3.5.1.1 Sequential execution

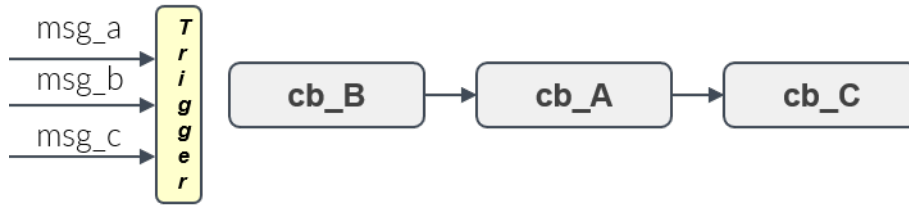
- At configuration, the user defines the order of handles.
- At configuration, the user defines whether the handle shall be called only when new data is available (ON_NEW_DATA) or whether the callback shall always be called (ALWAYS).
- At runtime, all handles are processed in the user-defined order
 - if the configuration of handle is ON_NEW_DATA, then the corresponding callback is only called if new data is available
 - if the configuration of the handle is ALWAYS, then the corresponding callback is always. If no data is available, then the callback is called with no data (e.g. NULL pointer).

The next figure shows three callbacks, A, B and C. Assume, they shall be executed in the order *B,A,C*. Then the user adds the callbacks to the rclc Executor in this order. Whenever new messages have arrived then the callbacks for which a new message is available will be always executed in the user-defined processing order.



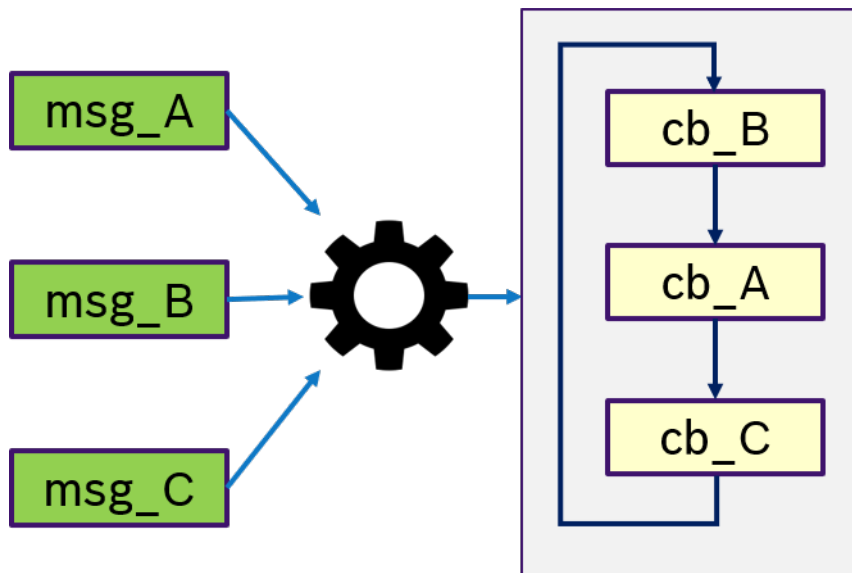
3.5.1.2 Trigger condition

- Given a set of handles, a trigger condition, which is based on the availability of input data of these handles, decides when the processing of all callbacks starts.

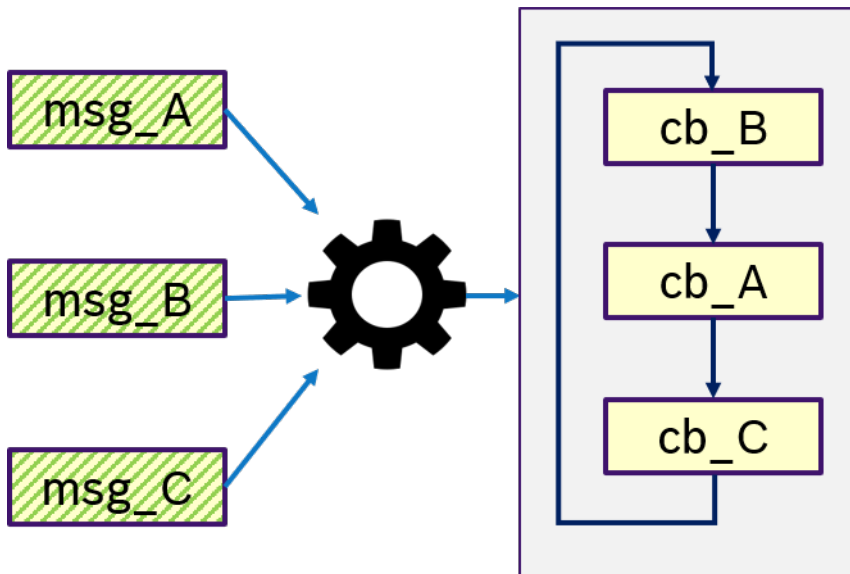


- Available options:
 - ALL operation: fires when input data is available for all handles
 - ANY operation: fires when input data is available for at least one handle (OR semantics)
 - ONE: fires when input data for a user-specified handle is available
 - User-defined function: user can implement custom logic

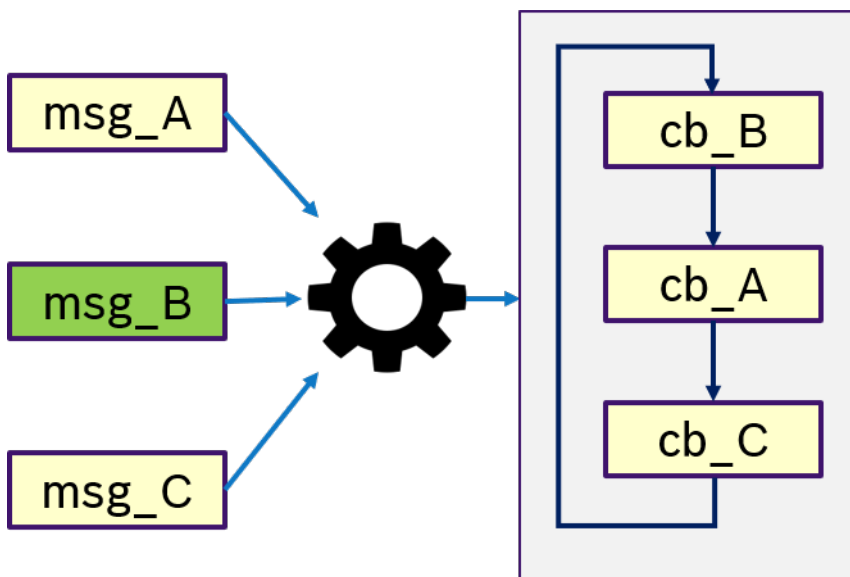
The next figure shows an example of the ALL semantics. Only if all messages *msg_A*, *msg_B*, *msg_C* were received, then trigger condition is fulfilled and the callbacks are processed in a user-defined order.



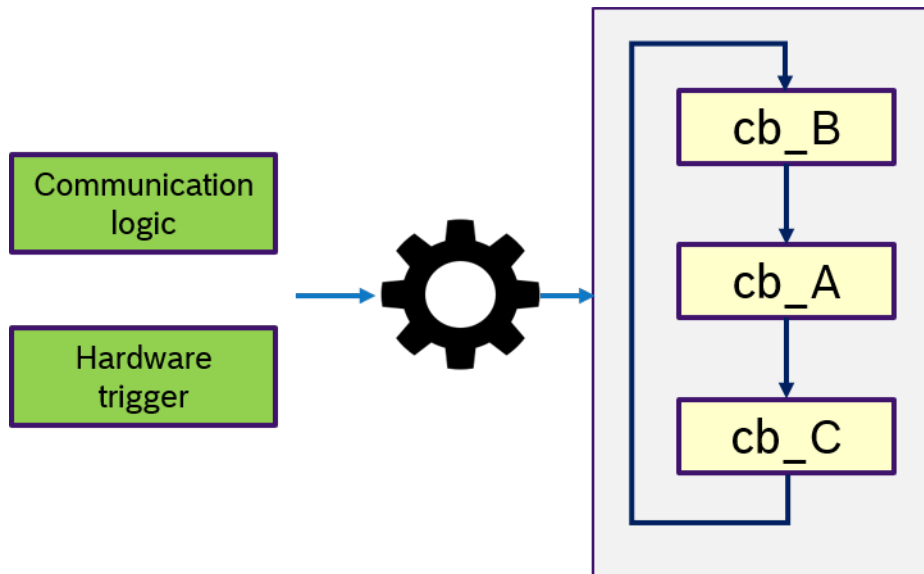
The next figure shows an example of the ANY semantics. That is, if any messages *msg_A*, *msg_B*, *msg_C* was received, then trigger condition is fulfilled and the callbacks are processed in a user-defined order. This is equivalent to OR semantics.



The next figure shows an example of the ONE semantics. That is, only if message `msg_B` was received, the trigger condition is fulfilled and (potentially all) callbacks are processed in a user-defined order.



The next figure describes the custom semantics. A custom trigger condition with could be a more complex logic of multiple messages, can be passed to the executor. This might also include hardware triggers, like interrupts.



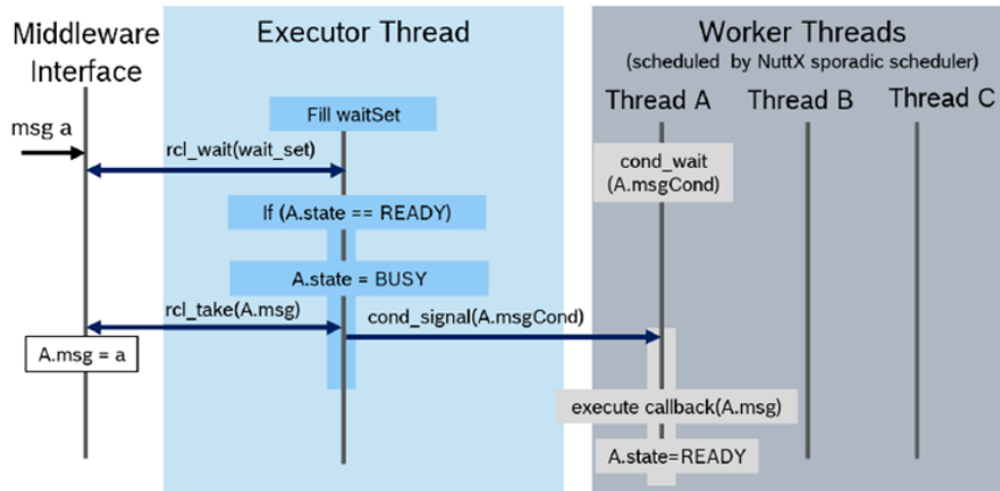
3.5.1.3 LET-Semantics

- Assumption: time-triggered system, the executor is activated periodically
- When the trigger fires, reads all input data and makes a local copy
- Processes all callbacks in sequential order
- Write output data at the end of the executor's period (Note: this is not implemented yet)

Additionally we have implemented the current rclcpp Executor semantics ("RCLCPP"): - waiting for new data for all handles (`rcl_wait`) - using trigger condition ANY - if trigger fires, start processing handles in pre-defined sequential order - request from DDS-queue the new data just before the handle is executed (`rcl_take`)

The selection of the Executor semantics is optional. The default semantics is "RCLCPP".

3.5.1.4 Multi-threading and scheduling configuration The rclc Executor has been extended for multi-threading. It supports the assignment of scheduling policies, like priorities or more advanced scheduling algorithms as reservation-based scheduling, to subscription callbacks. [Pull Request, Pre-print SLD2021]. The overall architecture is shown in the next figure. One Executor thread is responsible for checking for new data from the DDS queue. For every callback, a thread is spawned with the dedicated scheduling policy provided by the operating system. The Executor then dispatches new data of a subscription to its corresponding callback function, which is then executed in its own thread by operating system.



3.5.2 Executor API

The API of the rclc Executor can be divided in two phases: Configuration and Running. ##### Configuration phase During the configuration phase, the user shall define: - the total number of callbacks - the sequence of the callbacks - trigger condition (optional, default: ANY) - data communication semantics (optional, default ROS2)

As the Executor is intended for embedded controllers, dynamic memory management is crucial. Therefore at initialization of the rclc Executor, the user defines the total number of callbacks. The necessary dynamic memory will be allocated only in this phase and no more memory in the running phase. This makes this Executor static in the sense, that during runtime no additional callbacks can be added.

Then, the user adds handles and the corresponding callbacks (e.g. for subscriptions and timers) to the Executor. The order in which this takes place, defines later the sequential processing order during runtime.

For each handle the user can specify, if the callback shall be executed only if new data is available (ON_NEW_DATA) or if the callback shall always be executed (ALWAYS). The second option is useful when the callback is expected to be called at a fixed rate.

The trigger condition defines when the processing of these callbacks shall start. For convenience some default conditions have been defined: - trigger_any(default) : start executing if any callback has new data - trigger_all : start executing if all callbacks have new data - trigger_one(&data) : start executing if data has been received - user_defined_function: the user can also define its own function with more complex logic

With 'trigger_any' being the default, the current semantics of the rclcpp Executor is selected.

The data communication semantics can be - ROS2 (default) - LET

To be compatible with ROS2 rclcpp Executor, the existing rclcpp semantics is implemented as 'ROS2'. That is, with the spin-function the DDS-queue is constantly monitored for new data (rcl_wait). If new data becomes available, then is fetched from DDS (rcl_take) immediately before the callback is executed. All callbacks are processed in the user-defined order, this is the only difference to the rclcpp Executor, in which no order can be specified.

Secondly, the LET semantics is implemented such that at the beginning of processing all available data is fetched (`rcl_take`) and buffered and then the callbacks are processed in the pre-defined operating on the buffered copy.

3.5.2.1 Running phase As the main functionality, the Executor has a spin-function which constantly checks for new data at the DDS-queue, like the `rclcpp` Executor in ROS2. If the trigger condition is satisfied then all available data from the DDS queue is processed according to the specified semantics (ROS or LET) in the user-defined sequential order. After all callbacks have been processed the DDS is checked for new data again.

Available spin functions are - `spin_some` - spin one time - `spin_period` - spin with a period - `spin` - spin indefinitely

3.5.3 Examples

We provide the relevant code snippets how to setup the `rcl` Executor for the processing patterns as described above.

3.5.3.1 Sense-plan-act pipeline in robotics example In this example we want to realise a sense-plan-act pipeline in a single thread. The trigger condition is demonstrated by activating the sense-phase when both data for the Laser and IMU are available. Three executors are necessary `exe_sense`, `exe_plan` and `exe_act`. The two sensor acquisition callbacks `sense_Laser` and `sense_IMU` are registered in the Executor `exe_sense`. The trigger condition `ALL` is responsible to activate the sense-phase only when all data for these two callbacks are available. Finally all three Executors are spinning using a `while`-loop and the `spin_some` function.

The definitions of callbacks are omitted.

```
...
rcl_subscription_t sense_Laser, sense_IMU, plan, act;
rcl_executor_t exe_sense, exe_plan, exe_act;
// initialize executors
rcl_executor_init(&exe_sense, &context, 2, ...);
rcl_executor_init(&exe_plan, &context, 1, ...);
rcl_executor_init(&exe_act, &context, 1, ...);
// executor for sense-phase
rcl_executor_add_subscription(&exe_sense, &sense_Laser, &my_sub_cb1, ON_NEW_DATA);
rcl_executor_add_subscription(&exe_sense, &sense_IMU, &my_sub_cb2, ON_NEW_DATA);
rcl_executor_set_trigger(&exe_sense, rcl_executor_trigger_all, NULL);
// executor for plan-phase
rcl_executor_add_subscription(&exe_plan, &plan, &my_sub_cb3, ON_NEW_DATA);
// executor for act-phase
rcl_executor_add_subscription(&exe_act, &act, &my_sub_cb4, ON_NEW_DATA);

// spin all executors
while (true) {
```

```
rclc_executor_spin_some(&exe_sense);
rclc_executor_spin_some(&exe_plan);
rclc_executor_spin_some(&exe_act);
}
```

3.5.3.2 Synchronization of multiple rates example The sensor fusion synchronizing the multiple rates with a trigger is shown below.

```
...
rcl_subscription_t aggr_IMU, sense_Laser, sense_IMU;
rcl_executor_t exe_aggr, exe_sense;
// initialize executors
rclc_executor_init(&exe_aggr, &context, 1, ...);
rclc_executor_init(&exe_sense, &context, 2, ...);
// executor for aggregate IMU data
rclc_executor_add_subscription(&exe_aggr, &aggr_IMU, &my_sub_cb1, ON_NEW_DATA);
// executor for sense-phase
rclc_executor_add_subscription(&exe_sense, &sense_Laser, &my_sub_cb2, ON_NEW_DATA);
rclc_executor_add_subscription(&exe_sense, &sense_IMU, &my_sub_cb3, ON_NEW_DATA);
rclc_executor_set_trigger(&exe_sense, rclc_executor_trigger_all, NULL);

// spin all executors
while (true) {
    rclc_executor_spin_some(&exe_aggr);
    rclc_executor_spin_some(&exe_sense);
}
```

The setup for the sensor fusion using sequential execution is shown below. Note that the sequential order is sense_IMU, which will request the aggregated IMU message, and then sense_Laser while the trigger will fire, when a laser message is received.

```
...
rcl_subscription_t sense_Laser, sense_IMU;
rcl_executor_t exe_sense;
// initialize executor
rclc_executor_init(&exe_sense, &context, 2, ...);
// executor for sense-phase
rclc_executor_add_subscription(&exe_sense, &sense_IMU, &my_sub_cb1, ALWAYS);
rclc_executor_add_subscription(&exe_sense, &sense_Laser, &my_sub_cb2, ON_NEW_DATA);
rclc_executor_set_trigger(&exe_sense, rclc_executor_trigger_one, &sense_Laser);
// spin
rclc_executor_spin(&exe_sense);
```

3.5.3.3 High priority processing path example This example shows the sequential processing order to execute the obstacle avoidance `obst_avoid` after the callbacks of the sense-phase and before the callback of the planning phase `plan`. The control loop is started when a laser message is received. Then an

aggregated IMU message is requested, like in the example above. Then all the other callbacks are always executed. This assumes that these callbacks communicate via a global data structure. Race conditions cannot occur, because the callbacks run all in one thread.

```
...
rcl_subscription_t sense_Laser, sense_IMU, plan, act, obst_avoid;
rcle_let_executor_t exe;
// initialize executors
rclc_executor_init(&exe, &context, 5, ...);
// define processing order
rclc_executor_add_subscription(&exe, &sense_IMU, &my_sub_cb1, ALWAYS);
rclc_executor_add_subscription(&exe, &sense_Laser, &my_sub_cb2, ON_NEW_DATA);
rclc_executor_add_subscription(&exe, &obst_avoid, &my_sub_cb3, ALWAYS);
rclc_executor_add_subscription(&exe, &plan, &my_sub_cb4, ALWAYS);
rclc_executor_add_subscription(&exe, &act, &my_sub_cb5, ALWAYS);
rclc_executor_set_trigger(&exe, rclc_executor_trigger_one, &sense_Laser);
// spin
rclc_executor_spin(&exe);
```

3.5.3.4 Real-time embedded applications example With sequential execution the co-operative scheduling of tasks within a process can be modeled. The trigger condition is used to periodically activate the process which will then execute all callbacks in a pre-defined order. Data will be communicated using the LET-semantics. Every Executor is executed in its own tread, to which an appropriate priority can be assigned.

In the following example, the Executor is setup with 4 handles. We assume a process has three subscriptions sub1, sub2, sub3. The sequential processing order is given by the order as they are added to the Executor. A timer timer defines the period. The trigger_one with the paramter timer is used, so that whenever the timer is ready, all callbacks are processed. Finally the data communication semantics LET is defined.

```
#include "rcl_executor/let_executor.h"

// define subscription callback
void my_sub_cb1(const void * msgin)
{
    // ...
}
// define subscription callback
void my_sub_cb2(const void * msgin)
{
    // ...
}
// define subscription callback
void my_sub_cb3(const void * msgin)
{
    // ...
}
```

```
}

// define timer callback
void my_timer_cb(rcl_timer_t * timer, int64_t last_call_time)
{
    // ...
}

// necessary ROS 2 objects
rcl_context_t context;
rcl_node_t node;
rcl_subscription_t sub1, sub2, sub3;
rcl_timer_t timer;
rcle_let_executor_t exe;

// define ROS context
context = rcl_get_zero_initialized_context();
// initialize ROS node
rcl_node_init(&node, &context, ...);
// create subscriptions
rcl_subscription_init(&sub1, &node, ...);
rcl_subscription_init(&sub2, &node, ...);
rcl_subscription_init(&sub3, &node, ...);
// create a timer
rcl_timer_init(&timer, &my_timer_cb, ... );
// initialize executor with four handles
rclc_executor_init(&exe, &context, 4, ...);
// define static execution order of handles
rclc_executor_add_subscription(&exe, &sub1, &my_sub_cb1, ALWAYS);
rclc_executor_add_subscription(&exe, &sub2, &my_sub_cb2, ALWAYS);
rclc_executor_add_subscription(&exe, &sub3, &my_sub_cb3, ALWAYS);
rclc_executor_add_timer(&exe, &timer);
// trigger when handle 'timer' is ready
rclc_executor_set_trigger(&exe, rclc_executor_trigger_one, &timer);
// select LET-semantics
rclc_executor_data_comm_semantics(&exe, LET);
// spin forever
rclc_executor_spin(&exe);
```

3.5.3.5 ROS 2 Executor Workshop Reference System The rclc Executor has been presented at the workshop ‘ROS 2 Executor: How to make it efficient, real-time and deterministic?’ at [ROS World 2021](#) (i.e. the online version of ROSCon)[S2021]. A [Reference System](#) for testing and benchmarking ROS Executors has been developed for this workshop. The application of the rclc Executor on the reference system with the trigger condition can be found in the [rclc-executor branch of the Reference System](#).

The slides can be downloaded [here](#). All information and the videos and slides of the other talks of the workshop can be found at www.apex.ai/roscon-21.

3.5.4 Future work

- Full LET semantics (writing data at the end of the period)
 - one publisher that periodically publishes
 - if Executors are running in multiple threads, publishing needs to be atomic
- Multi-threaded executor with assignment of scheduling policies of underlying operating system. [[Pull Request](#), pre-print [SLD2021](#)].

3.5.5 Download

The rclc Executor can be downloaded from the [ros2/rclc repository](#). It is available for the ROS 2 versions Foxy, Galactic and Rolling. The repository provides several packages including the [rclc Executor](#) and an [rclc_examples package](#) with several application examples.

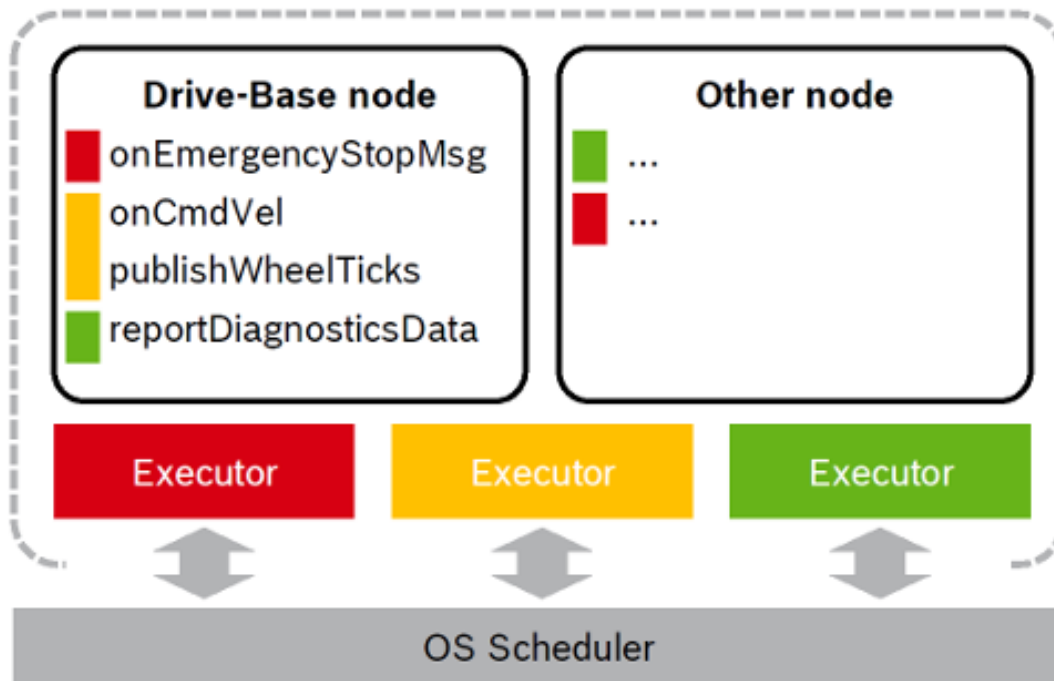
3.6 Callback-group-level Executor

The Callback-group-level Executor was an early prototype for a refined rclcpp Executor API developed in micro-ROS. It has been derived from the default rclcpp Executor and addresses some of the aforementioned deficits. Most important, it was used to validate that the underlying layers (rcl, rmw, rmw_adapter, DDS) allow for multiple Executor instances without any negative interferences.

As the default rclcpp Executor works at a node-level granularity – which is a limitation given that a node may issue different callbacks needing different real-time guarantees - we decided to refine the API for more fine-grained control over the scheduling of callbacks on the granularity of callback groups using. We leverage the callback-group concept existing in rclcpp by introducing real-time profiles such as RT-CRITICAL and BEST-EFFORT in the callback-group API (i.e. rclcpp/callback_group.hpp). Each callback needing specific real-time guarantees, when created, may therefore be associated with a dedicated callback group. With this in place, we enhanced the Executor and depending classes (e.g., for memory allocation) to operate at a finer callback-group granularity. This allows a single node to have callbacks with different real-time profiles assigned to different Executor instances - within one process.

Thus, an Executor instance can be dedicated to specific callback group(s) and the Executor's thread(s) can be prioritized according to the real-time requirements of these groups. For example, all time-critical callbacks are handled by an "RT-CRITICAL" Executor instance running at the highest scheduler priority.

The following figure illustrates this approach with two nodes served by three Callback-group-level Executors in one process:



The different callbacks of the Drive-Base node are distributed to different Executors (visualized by the color red, yellow and green). For example the `onCmdVel` and `publishWheelTicks` callback are scheduled by the same Executor (yellow). Callbacks from different nodes can be serviced by the same Executor.

3.6.1 API Changes

In this section, we describe the necessary changes to the Executor API: * [include/rclcpp/callback_group.hpp](#):

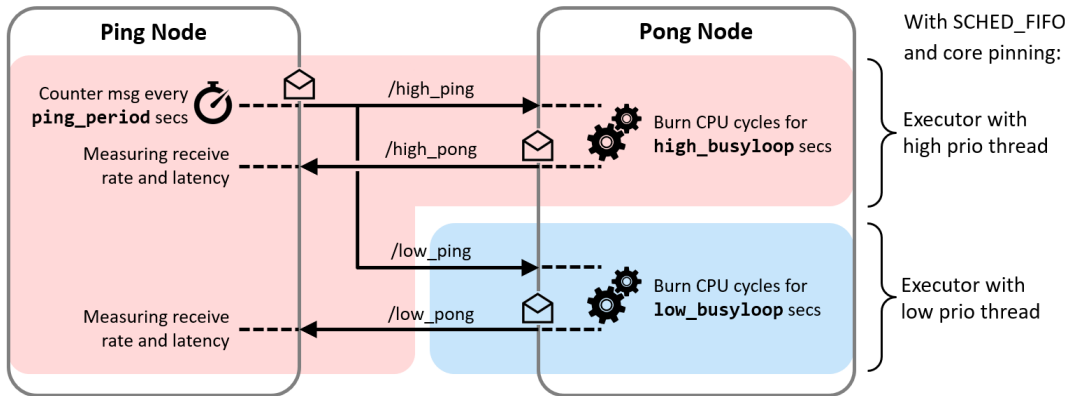
- * Introduced an enum to distinguish up to three real-time classes (requirements) per node (Real-time).
- * Changed association with Executor instance from nodes to callback groups.

- [include/rclcpp/executor.hpp](#)
 - Added functions to add and remove individual callback groups in addition to whole nodes.
 - Replaced private vector of nodes with a map from callback groups to nodes.
- [include/rclcpp/memory_strategy.hpp](#)
 - Changed all functions that expect a vector of nodes to the just mentioned map.
- [include/rclcpp/node.hpp](#) and [include/rclcpp/node_interfaces/node_base.hpp](#)
 - Extended arguments of `create_callback_group` function for the real-time class.
 - Removed the `get_associated_with_executor_atomic` function.

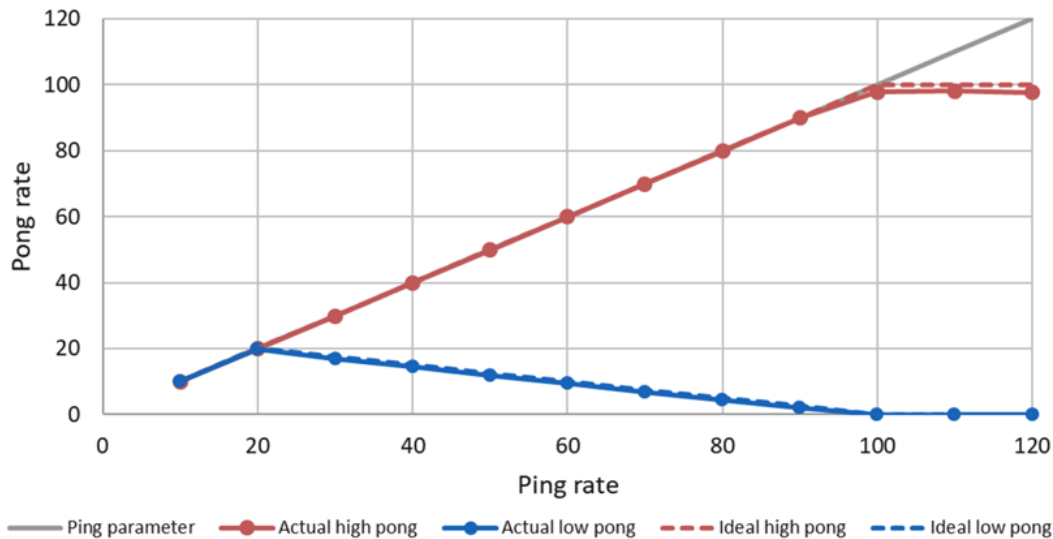
The callback-group-level executor has been merged into ROS 2 rclcpp in [pull request 1218](#).

3.6.2 Test Bench

As a proof of concept, we implemented a small test bench in the present package `cbg-executor_ping_pong_cpp`. The test bench comprises a Ping node and a Pong node which exchange real-time and best-effort messages simultaneously with each other. Each class of messages is handled with a dedicated Executor, as illustrated in the following figure.



With the test bench, we validated the functioning of the approach.



In this example, the callback for the high priority task (red line) consumes 10ms and the low priority task (blue line) 40ms in the Pong Node. With a ping rate of 20 Hz, the CPU saturates ($10\text{ms} \cdot 20 + 40\text{ms} \cdot 20 = 1000\text{ms}$). With higher frequencies the high priority task can continue to send its pong message, while the low priority pong task degrades. With a frequency of 100Hz the high priority task requires 100% CPU utilization. With higher ping rates it keeps sending pong messages with 100Hz, while the low priority task does not get any CPU resources any more and cannot send any messages.

The test bench is provided in the [cbg_executor_demo](#).

3.7 Related Work

In this section, we provide an overview to related approaches and link to the corresponding APIs.

3.7.1 Fawkes Framework

Fawkes is a robotic software framework, which supports synchronization points for sense-plan-act like execution. It has been developed by RWTH Aachen since 2006. Source code is available at github.com/fawkesrobotics.

3.7.1.1 Synchronization Fawkes provides developers different synchronization points, which are very useful for defining an execution order of a typical sense-plan-act application. These ten synchronization points (wake-up hooks) are the following (cf. [libs/aspect/blocked_timing.h](#)):

- WAKEUP_HOOK_PRE_LOOP
- WAKEUP_HOOK_SENSOR_ACQUIRE
- WAKEUP_HOOK_SENSOR_PREPARE
- WAKEUP_HOOK_SENSOR_PROCESS
- WAKEUP_HOOK_WORLDSTATE
- WAKEUP_HOOK_THINK
- WAKEUP_HOOK_SKILL

- WAKEUP_HOOK_ACT

- WAKEUP_HOOK_ACT_EXEC
- WAKEUP_HOOK_POST_LOOP

3.7.1.2 Configuration at compile time At compile time, a desired synchronization point is defined as a constructor parameter for a module. For example, assuming that `mapLaserGenThread` shall be executed in `SENSOR_ACQUIRE`, the constructor is implemented as:

```
MapLaserGenThread::MapLaserGenThread()  
  :: Thread("MapLaserGenThread", Thread::OPMODE_WAITFORWAKEUP),  
    BlockedTimingAspect(BlockedTimingAspect::WAKEUP_HOOK_SENSOR_ACQUIRE),  
    TransformAspect(TransformAspect::BOTH_DEFER_PUBLISHER, "Map Laser Odometry")
```

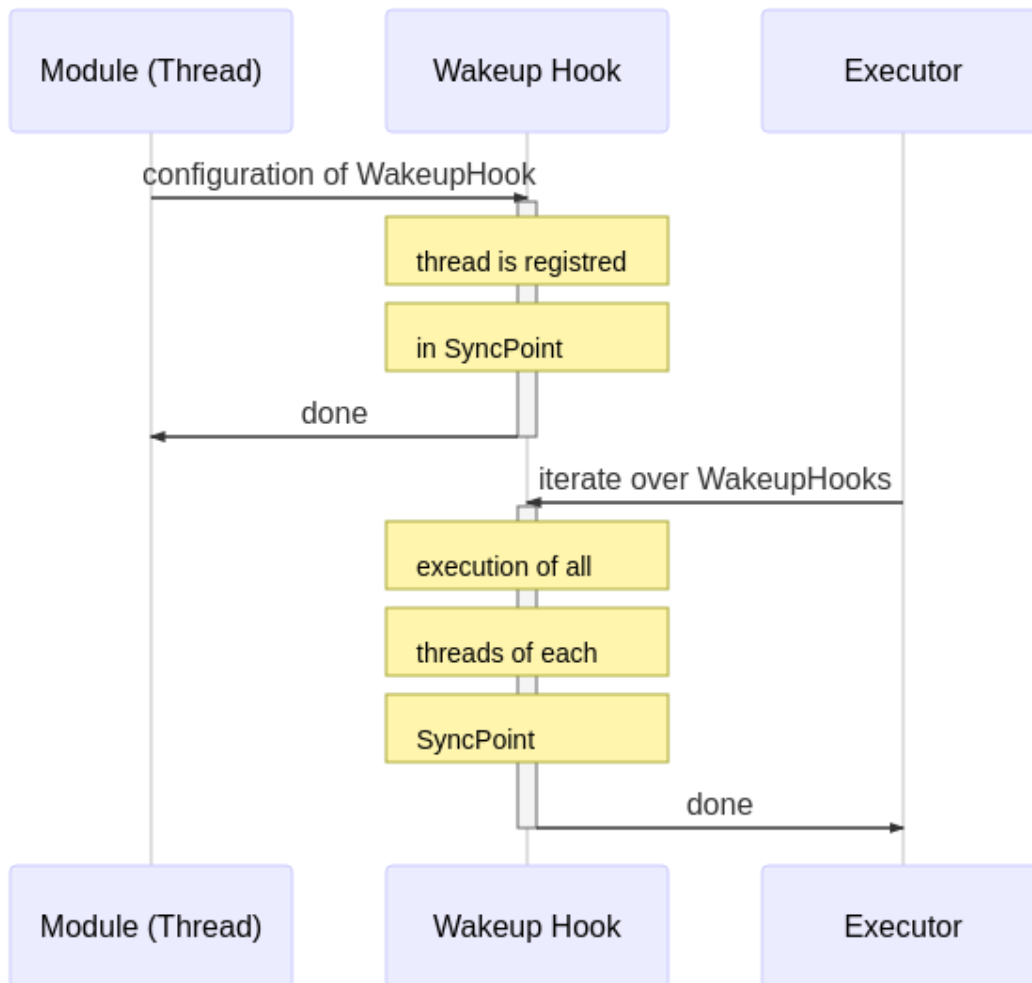
Similarly, if `NaoQiButtonThread` shall be executed in the `SENSOR_PROCESS` hook, the constructor is:

```
NaoQiButtonThread::NaoQiButtonThread()  
  :: Thread("NaoQiButtonThread", Thread::OPMODE_WAITFORWAKEUP),  
    BlockedTimingAspect(BlockedTimingAspect::WAKEUP_HOOK_SENSOR_PROCESS)
```

3.7.1.3 Runtime execution At runtime, the *Executor* iterates through the list of synchronization points and executes all registered threads until completion. Then, the threads of the next synchronization point are called.

A module (thread) can be configured independent of these sense-plan-act synchronization points. This has the effect, that this thread is executed in parallel to this chain.

The high level overview of the Fawkes framework is shown in the next figure. At compile-time the configuration of the sense-plan-act wakeup hook is done (upper part), while at run-time the scheduler iterates through this list of wakeup-hooks (lower part):



Hence, at run-time, the hooks are executed as a fixed static schedule without preemption. Multiple threads registered in the same hook are executed in parallel.

Orthogonal to the sequential execution of sense-plan-act like applications, it is possible to define further constraints on the execution order by means of a `Barrier`. A barrier defines a number of threads, which need to have finished before the thread can start, which owns the barrier.

These concepts are implemented by the following main classes:

- *Wakeup hook* by `SyncPoint` and `SyncPointManager`, which manages a list of synchronization points.
- *Executor* by the class `FawkesMainThread`, which is the scheduler, responsible for calling the user threads.
- `ThreadManager`, which is derived from `BlockedTimingExecutor`, provides the necessary API to add and remove threads to wakeup hooks as well as for sequential execution of the wakeup-hooks.
- `Barrier` is an object similar to `condition_variable` in C++.

3.7.1.4 Discussion All threads are executed with the same priority. If multiple sense-plan-act chains shall be executed with different priorities, e.g. to prefer execution of emergency-stop over normal oper-

ation, then this framework reaches its limits.

Also, different execution frequencies cannot be modeled by a single instance of this sense-plan-act chain. However, in robotics the fastest sensor will drive the chain and all other hooks are executed with the same frequency.

The option to execute threads independent of the predefined wakeup-hooks is very useful, e.g. for diagnostics. The concept of the Barrier is useful for satisfying functional dependencies which need to be considered in the execution order.

3.8 References

- [S2021] J. Staschulat, “Micro-ROS: The rclc Executor”, in Workshop ROS 2 Executor: How to make it efficient, real-time and deterministic? at ROS World, Oct. 2021, [[slides](#)] [[Video](#)]
- [SLD2021] J. Staschulat, R. Lange and D. N. Dasari, “Budget-based real-time Executor for Micro-ROS”, arXiv Pre-Print, May 2021. [[paper](#)]
- [L2020] Ralph Lange: Advanced Execution Management with ROS 2, ROS-Industrial Conference, Dec 2020 [[Slides](#)]
- [SLL2020] J. Staschulat, I. Lütkebohle and R. Lange, “The rclc Executor: Domain-specific deterministic scheduling mechanisms for ROS applications on microcontrollers: work-in-progress,” 2020 International Conference on Embedded Software (EMSOFT), Singapore, Singapore, 2020, pp. 18-19. [[Paper](#)] [[Video](#)]
- [CB2019] D. Casini, T. Blaß, I. Lütkebohle, B. Brandenburg: Response-Time Analysis of ROS 2 Processing Chains under Reservation-Based Scheduling, in Euromicro-Conference on Real-Time Systems 2019. [[Paper](#)] [[slides](#)]
- [L2018] Ralph Lange: Callback-group-level Executor for ROS 2. Lightning talk at ROSCon 2018. Madrid, Spain. Sep 2018. [[Slides](#)] [[Video](#)]
- [EK2018] R. Ernst, S. Kuntz, S. Quinton, M. Simons: The Logical Execution Time Paradigm: New Perspectives for Multicore Systems, February 25-28 2018 (Dagstuhl Seminar 18092). [[Paper](#)]
- [NSP2018] A. Naderlinger, S. Resmerita, and W. Pree: LET for Legacy and Model-based Applications, Proceedings of The Logical Execution Time Paradigm: New Perspectives for Multicore Systems (Dagstuhl Seminar 18092), Wadern, Germany, February 2018.
- [BP2017] A. Biondi, P. Pazzaglia, A. Balsini, M. D. Natale: Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores, International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS2017), Dubrovnik, Croatia. [[Paper](#)]
- [KZH2015] S. Kramer, D. Ziegenbein, and A. Hamann: Real World Automotive Benchmarks For Free, International Workshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS), 2015.



- [HHK2001] Henzinger T.A., Horowitz B., Kirsch C.M. (2001) Giotto: A Time-Triggered Language for Embedded Programming. In: Henzinger T.A., Kirsch C.M. (eds) Embedded Software. EM-SOFT 2001. Lecture Notes in Computer Science, vol 2211. Springer, Berlin, Heidelberg
- [LL1973] Liu, C. L.; Layland, J.:Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM, 20 (1): 46–61, 1973.

3.9 Acknowledgments

This activity has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement n° 780785).