D4.6

Real-time Executor Software Release Y3

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.6
Deliverable name	Real-time Executor – Software Release Y3
Date	December 2020
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 D4.6 *Real-time Executor Software Release Y3* of the Task 4.2 *Predictable Scheduling and Execution*.

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 . . . 5
3.3.1 Architecture . . . 5
3.3.2 Scheduling Semantics . . . 6
3.4 RCLC-Executor . . . 7
3.4.1 Requirement Analysis . . . 7
3.4.2 Features . . . 12
3.4.3 Executor API . . . 13
3.4.4 Examples . . . 14
3.4.5 Summary . . . 17
3.4.6 Future work . . . 18
3.4.7 Download . . . 18
3.5 Callback-group-level Executor . . . 18
3.5.1 API Changes . . . 19
3.5.2 Test Bench . . . 20
3.6 Related Work . . . 20
3.6.1 Fawkes Framework . . . 21
3.7 References . . . 23

1 Overview to Results

This document provides links to the released software and documentation for deliverable D4.6 *Real-time Executor Software Release Y3* 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.github.io/real-time_executor/

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

RCLC-Executor implementation in micro-ROS rclc fork:

- Git repository: https://github.com/micro-ROS/rclc
  Branch name: master
  Latest commit: 47972f8

RCLC-Executor implementation in ROS 2 rclc repository:

- Git repository: https://github.com/ros2/rclc
  Branch name: master
  Latest commit: 6e45370

Demo for RCLC-Executor in micro-ROS demo repositories:

- Git repository: https://github.com/micro-ROS/zephyr_apps
  Branch name: Foxy
  Latest commit: 209e2d9
- Git repository: https://github.com/micro-ROS/freertos_apps
  Branch name: Foxy
  Latest commit: 19e7d4a
- Git repository: https://github.com/micro-ROS/micro_ros_arduino
  Branch name: Foxy
  Latest commit: 489dfec

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/boschresearch/ros2_demos/tree/cbg_executor_demo/cbg_executor_demo

  Branch name: cbg_executor_demo

  Latest commit: 1aced36

3 Annex 1: Webpage on Real-Time Executor

Content of https://micro-ros.github.io/real-time_executor/ from 22nd December 2020.

3.1 Table of contents

- Introduction
- Analysis of rclcpp standard Executor
  - Architecture
  - Scheduling Semantics
- RCLC-Executor
  - Requirement Analysis
  - Features
  - Executor API
  - Examples
  - Download
- Callback-group-level Executor
  - API Changes
  - Test Bench
- Related Work
- 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 input data is collected in field tests, saved with ROS-bags and re-played, often results are different due to non-determinism of process scheduling.

Manually setting up a particular execution order of subscribing and publishing topics in the callbacks or by tweaking the 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 deplyed ROS 2 packages in the system.

Therefore the goal of the Real-Time Executor 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, like priorization. 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 term defined in rcl-layer and summarizes timers, subscriptions, clients and services etc..

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 proving real-time guarantees hard. 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++ appliations 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 framework and OS scheduler (semi-automated and optimized mapping is desired as well as generic, well-understood framework mechanisms)

Our approach is to provide Real-Time Executors on two layers as described in section Introduction to Client Library. One based on the rcl-layer written in C programming language and one based on rclcpp written in C++.

As the first step, we propose the LET-Executor for the rcl-layer in C, which implements static order scheduling policy with logic execution time semantics. In this scheduling policy, all callbacks are

executed in a pre-defined order. Logical execution time refers to the concept, that first input data is read before tasks are executed. Secondly, we developed a Callback-group-level Executor, which allows to prioritize a group of callbacks. These approaches are based on the concept of Executors, which have been introduced in ROS 2.

In the future, we plan to provide other Real-Time Executors for the rcl- and rclcpp-layer.

3.3 Analysis of rclcpp standard Executor

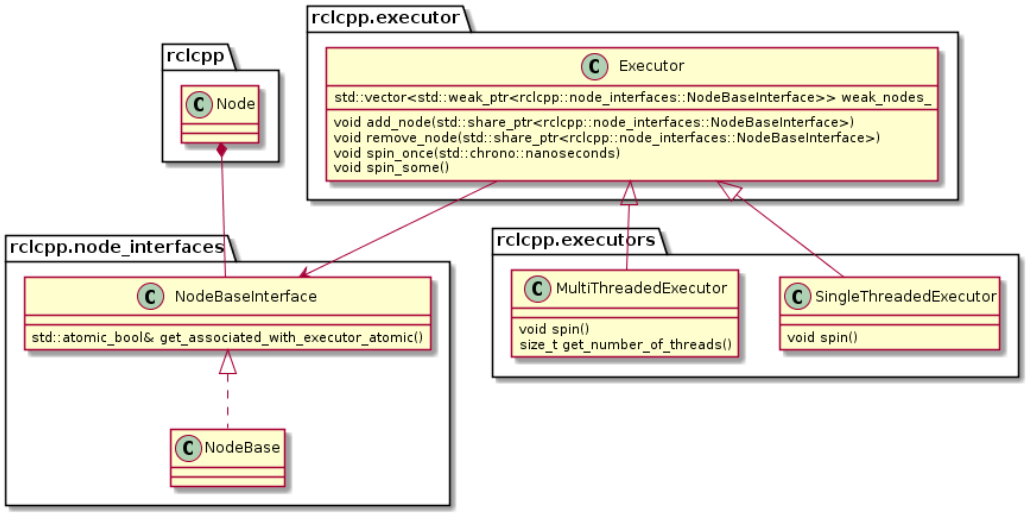
ROS 2 allows to bundle multiple nodes in one operating system process. To coordinate the execution of the callbacks of the nodes of a process, the Executor concept was introduced in rclcpp (and also in rclpy).

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 an in the order as they were registered at the Executor.

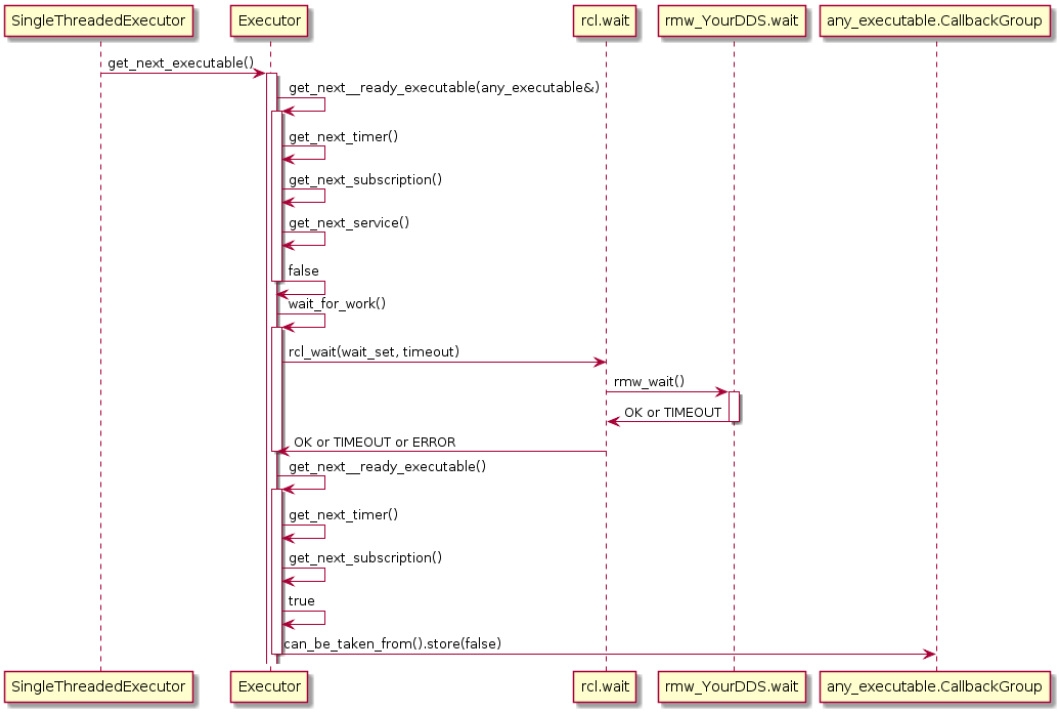
3.3.1 Architecture

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 RCLC-Executor

Here we introduce the rclc Executor, which is a ROS 2 Executor implemented based on and for the rcl API, for applications written in the C language. Often embedded applications require real-time to guarantee end-to-end latencies and need deterministic runtime behavior to correctly replay test data. However, this is difficult with the default ROS 2 Executor because of its complex semantics, as discussed in the previous section.

First, we will analyse the requirements for such applications and, secondly, derive simple features for an Executor to enable deterministic and real-time behavior. Then we will present the API of the RCLC-Executor and provide example usages of the RCLC-Executor to address these requirements.

3.4.1 Requirement Analysis

First we discuss a use-case in the embedded domain, in which the time-triggered paradigm is often used to guarantee deterministic and real-time behavior. Then we analyse software design patterns in mobile robotics which enable deterministic behavior.

3.4.1.1 Real-time embedded application used-case 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. Figure 1 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.

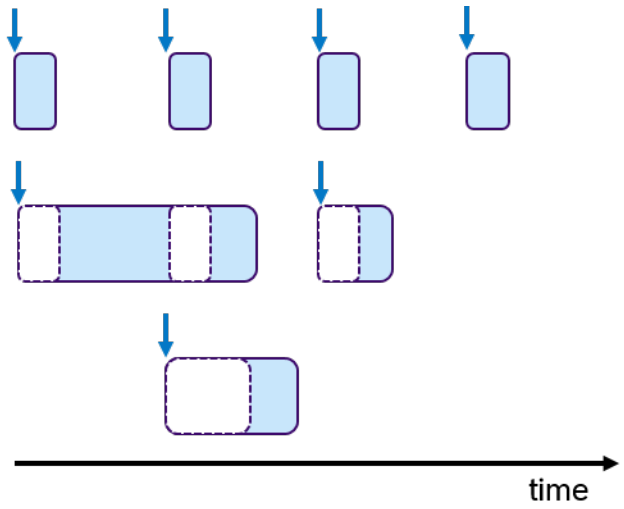


Figure 1: Fixed periodic preemptive scheduling

To each process one or multiple tasks can be assigned, as shown in Figure 2. These tasks are executed sequentially, which is often called cooperative scheduling.

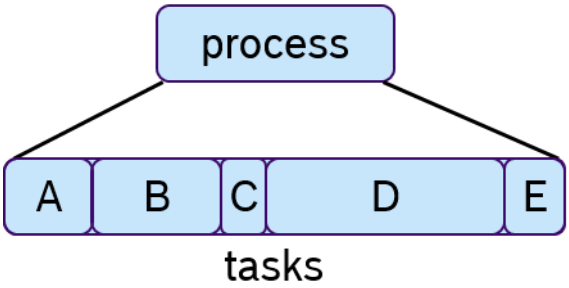


Figure 2: Processes with sequentially executed tasks.

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.

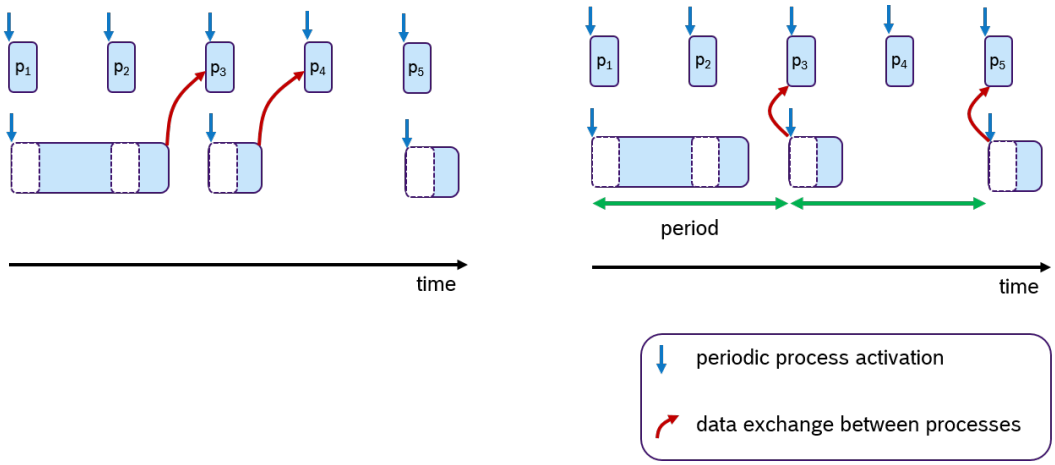


Figure 3: Data communication without and with Logical Execution Time paradigm.

An Example of the LET concept is shown in Figure 3. 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.

The described embedded use case relies on the following concepts: - 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 ROS2 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 wheter 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 of multiple callbacks - sequential processing of callbacks - data synchronization with LET semantics

3.4.1.2 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 aquire 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 Figure 4.

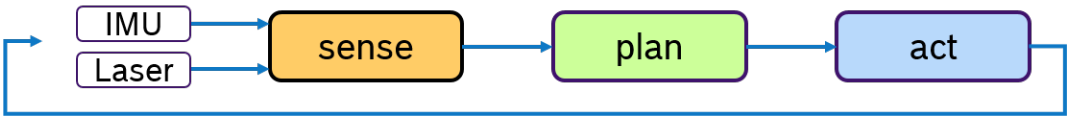


Figure 4: Multiple sensors driving a Sense-Plan-Act pipeline.

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 achive 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 ROS2-Executor. 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 of callbacks

3.4.1.3 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 availabe 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 Figure 5.

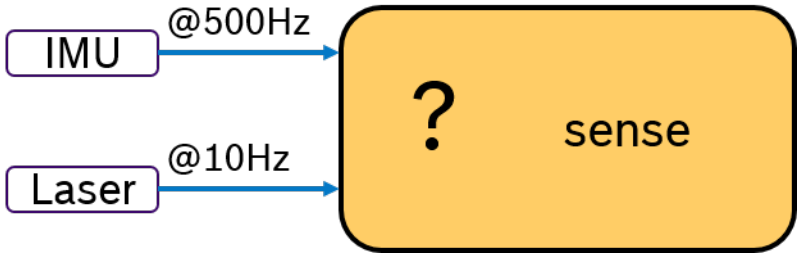


Figure 5: How to deterministically process multi-frequent sensor data.

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 evalute 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 Figure 6. 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.

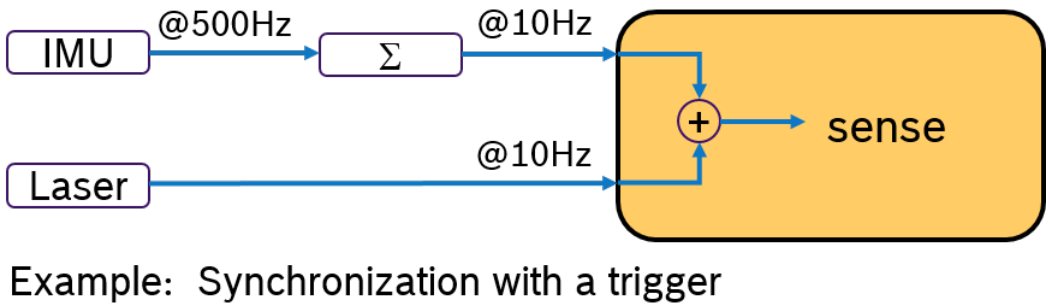


Figure 6: Synchronization of multiple input data with a trigger.

In ROS 2, this is currently not possible to be modeled because of the lack of a trigger concept in the ROS2 Executor. 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 activly request for IMU data only when a laser scan is received. This concept is shown in Figure 7. Upon arrival of a laser scan mesage, 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.

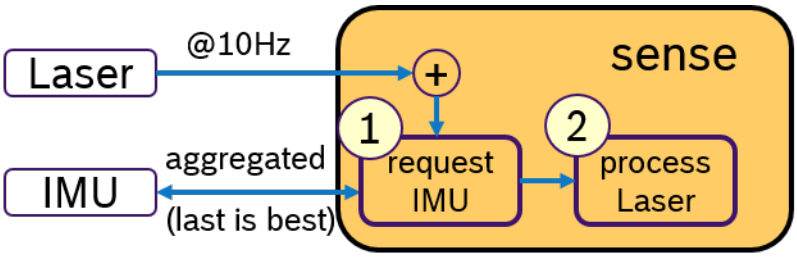


Figure 7: Synchronization with sequential processing.

Derived Requirements from both concepts: - triggered execution - sequential procesing of callbacks

3.4.1.4 High-priority processing path

Motivation:

Often a robot has to fullfill several activities at the same time. For example following a path and avoiding obstacles. While path following is a permanent activity, obstacle avoidance is trigged by the environment and should be immediately reacted upon. Therefore one would like to specify priorities to activities. This is depicted in Figure 8:

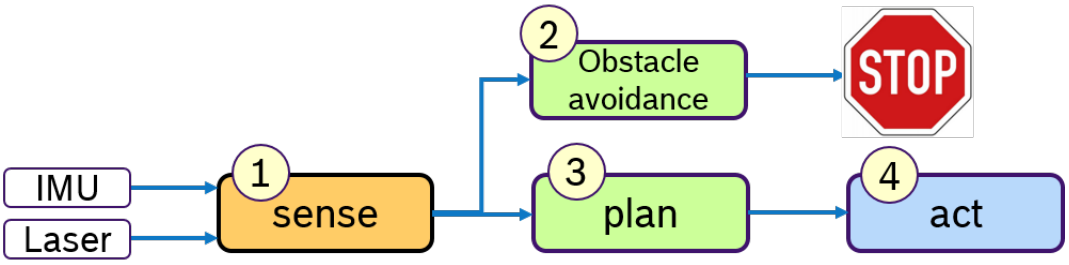


Figure 8: Managing high priority path with sequential order.

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 activites are processed in one thread.

Derived requirements: - sequential processing of callbacks

3.4.2 Features

Based on the real-time embedded use-case as well as the software architecture patterns in mobile robotics, we propose an Executor with the following main features: - user-defined sequential execution of callbacks - trigger condition to activate processing - data synchronization: LET-semantics or rclcpp Executor semantics

As stated before, this Executor is based on the RCL library and is written in C to nativly support microcontroller applications written in C. These features are now described in more detail.

The rclc-Executor supports all event types like the ROS 2 rclc executor, which are: - subscription - timer - service - client - guard condition

3.4.2.1 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).

3.4.2.2 Trigger condition

- Given a set of handles, a trigger condition based on the input data of these handles shall decide when the processing is started.
- 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
- ONE: fires when input data for a user-specified handle is available
- User-defined function: user can implement more sophisticated logic

3.4.2.3 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".

3.4.3 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 communcation 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.

3.4.3.1 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 indefinitly

3.4.4 Examples

We provide the relevant code snippets how to setup the RCLC-Executor for the embedded use case and for the software design patterns in mobile robotics applications as described above. #### Embedded use-case

With seqential 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);
```

3.4.4.1 Sense-plan-act pipeline 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;
rcle_let_executor_t exe_sense, exe_plan, exe_act;
// initialize executors
rclc_executor_init(&exe_sense, &context, 2, ...);
rclc_executor_init(&exe_plan, &context, 1, ...);
rclc_executor_init(&exe_act, &context, 1, ...);
// executor for sense-phase
rclc_executor_add_subscription(&exe_sense, &sense_Laser, &my_sub_cb1, ON_NEW_DATA);
rclc_executor_add_subscription(&exe_sense, &sense_IMU, &my_sub_cb2, ON_NEW_DATA);
rclc_let_executor_set_trigger(&exe_sense, rclc_executor_trigger_all, NULL);
// executor for plan-phase
rclc_executor_add_subscription(&exe_plan, &plan, &my_sub_cb3, ON_NEW_DATA);
// executor for act-phase
rclc_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);
}
```

3.4.4.2 Sensor fusion The sensor fusion synchronizing the multiple rates with a trigger is shown below.

```
...
rcl_subscription_t aggr_IMU, sense_Laser, sense_IMU;
rcle_let_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 sequetial

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;
rcle_let_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);
```

3.4.4.3 High priorty path 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);
```

3.4.5 Summary

The RCLC Executor is an Executor for C applications and can be used with default rclcpp Executor semantics. If additional deterministic behavior is necessary, the user can rely on pre-defined sequential execution, trigged execution and LET-Semantics for data synchronization. The concept of the rclc-Executor has been published in SLL2020.

3.4.6 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

3.4.7 Download

The RCLC-Executor can be downloaded from the micro-ROS rclc repository. In this repository, the rclc package provides the RCLC-Executor and the rclc_examples package provides several demos. Further more, the rclc Executor is also available from the ros2/rclc repository.

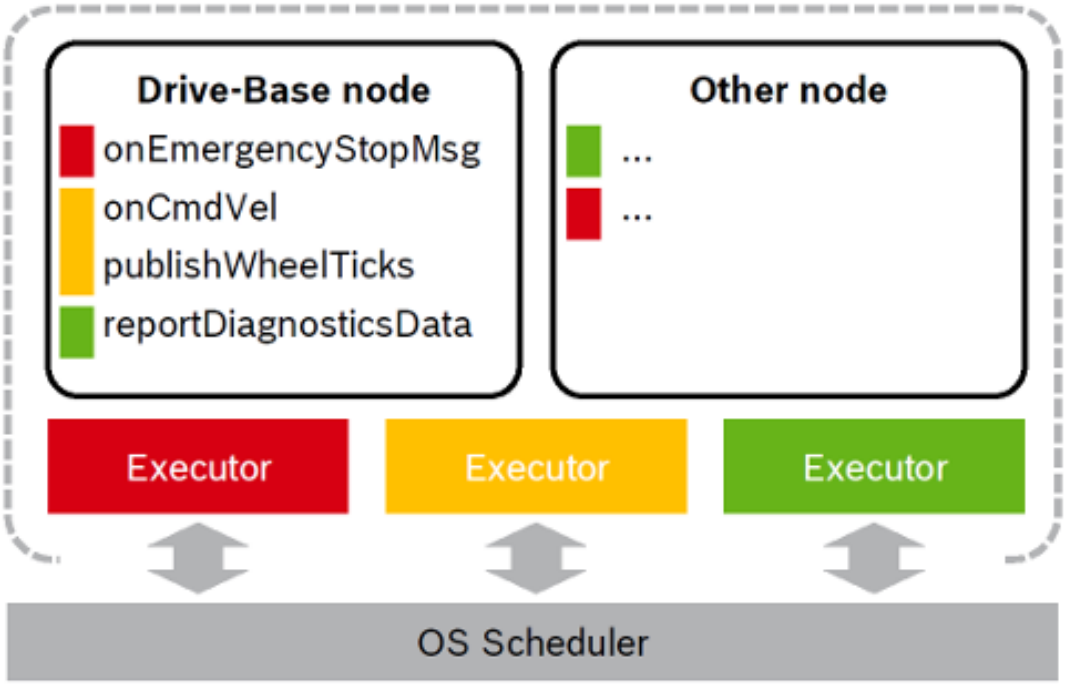
3.5 Callback-group-level Executor

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.

3.5.1 API Changes

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

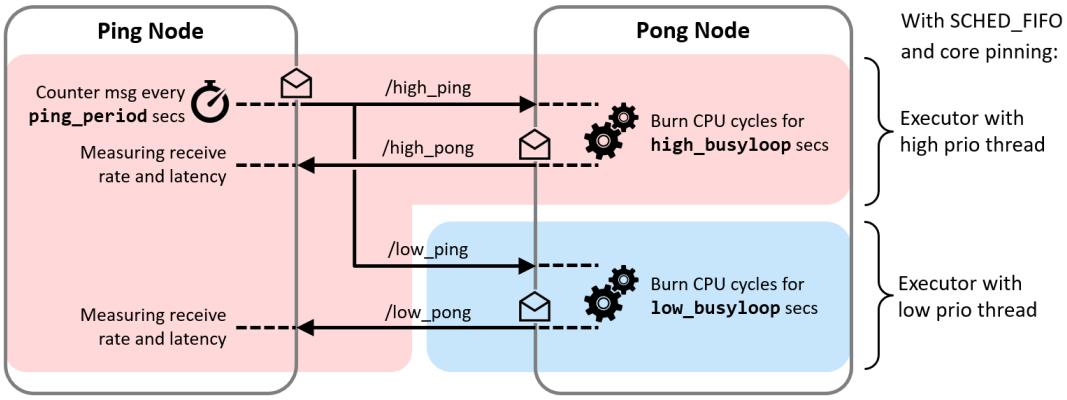
```
* Introduced an enum to distinguish up to three real-time classes (requirements) per node (Real
* Changed association with Executor instance from nodes to callback groups.
```

- 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.

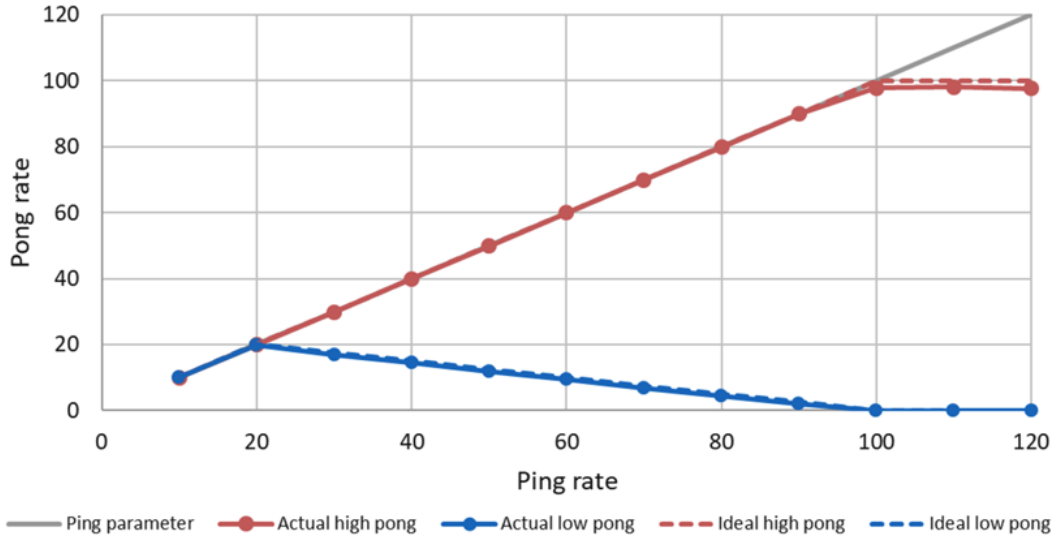
The callback-group-level executor has been merged into ROS 2 rclcpp in pull request 1218.

3.5.2 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 (10ms*20+40ms*20=1000ms). With higher frequencies the high priorty 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 ressources any more and cannot send any messages.

The test bench is provided in the cbg_executor_demo.

3.6 Related Work

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

3.6.1 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.

3.6.1.1 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

3.6.1.2 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")
```

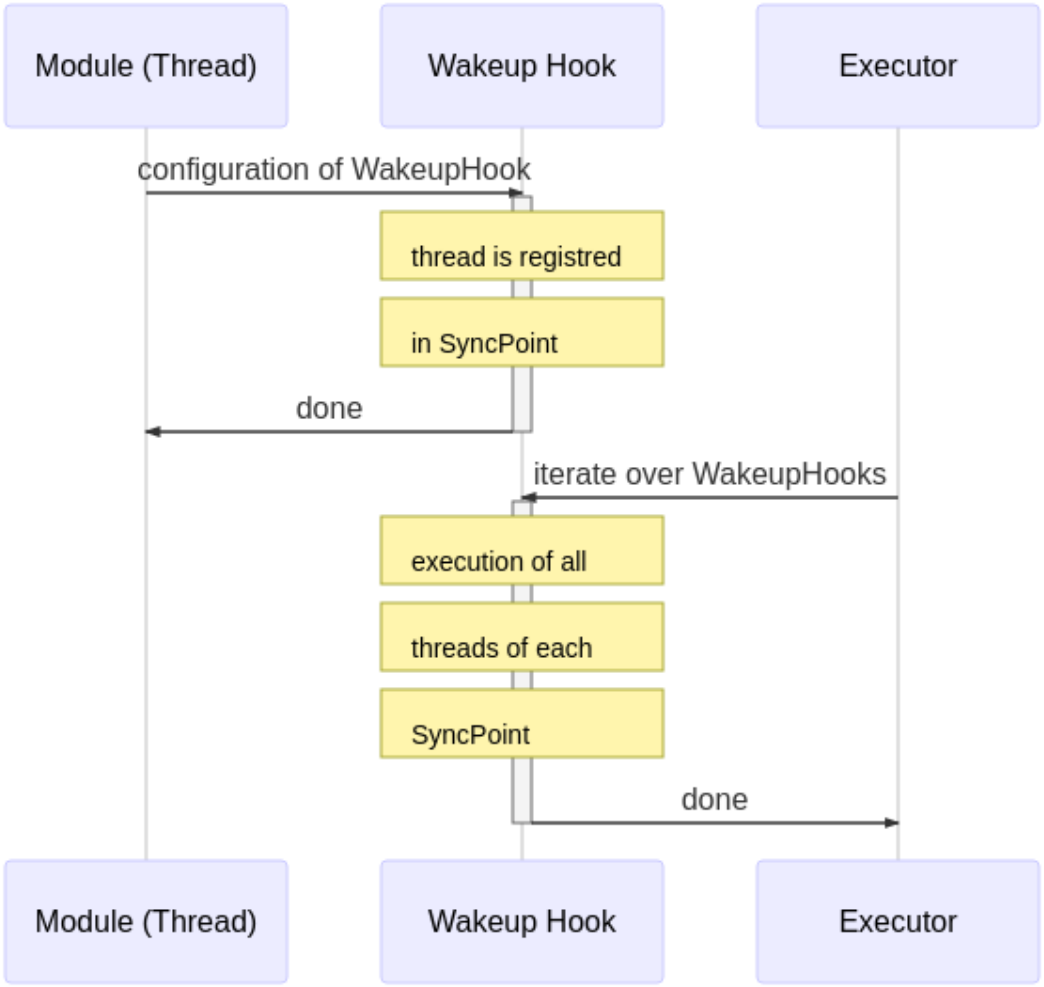
Similarly, if `NaoQiButtonThread` shall be executed in the SENSOR_PROCESS hook, the constructor is:

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

3.6.1.3 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++.

3.6.1.4 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 operation, 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.

3.7 References

- [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]
- [L2018] Ralph Lange: Callback-group-level Executor for ROS 2. Lightning talk at ROSCon 2018. Madrid, Spain. Sep 2018. [Slides] [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]
- [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]
- [BP2017] A. Biondi, P. Pazzaglia, A. Balsini, M. D. Natale: Logical Execution Time Implementation and Memory Optimization Issues in AUTOSAR Applications for Multicores, International Worshop on Analysis Tools and Methodologies for Embedded and Real-Time Systems (WATERS2017), Dubrovnik, Croatia.[Paper]
- [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.
- [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. EMSOFT 2001. Lecture Notes in Computer Science, vol 2211. Springer, Berlin, Heidelberg
- [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.

- [KZH2015] S. Kramer, D. Ziegenbein, and A. Hamann: Real World Automotive Benchmarks For Free, International Workshop on Analysis Tools and Methodologies for Embedded adn Real-Time Sysems (WATERS), 2015.[Paper] ## 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).