D4.5

Real-time Executor Software Release Y2

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.5
Deliverable name	Real-time Executor – Software Release Y2
Date	December 2019
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 D4.5 *Real-time Executor Software Release Y2* 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 2

3.1 Table of contents . . . 3
3.2 Introduction . . . 3
3.3 Analysis of rclcpp standard Executor . . . 4
3.3.1 Architecture . . . 5
3.3.2 Scheduling Semantics . . . 6
3.4 Rcl LET-Executor . . . 6
3.4.1 Concept . . . 7
3.4.2 Example . . . 7
3.4.3 Download . . . 8
3.5 Callback-group-level Executor . . . 8
3.5.1 API Changes . . . 9
3.5.2 Meta Executor Concept . . . 10
3.5.3 Test Bench . . . 10
3.6 Related Work . . . 11
3.6.1 Fawkes Framework . . . 11
3.7 Roadmap . . . 14
3.8 References . . . 14
3.9 Acknowledgments . . . 15

1 Overview to Results

This document provides links to the released software and documentation for deliverable D4.5 *Real-time Executor Software Release Y2* 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

LET-Executor implementation in micro-ROS rclc fork:

- Git repository: https://github.com/micro-ROS/rclc
  Branch name: new_api_and_LET_executor
  Latest commit: 2103d7c

Demo for LET-Executor in micro-ROS-demo repository:

- Git repository: https://github.com/micro-ROS/micro-ROS-demos
  Branch name: rcl_executor_examples
  Latest commit: 6c22076

Callback-group-based Executor implementation in ROS 2 rclcpp fork:

- Git repository: https://github.com/micro-ROS/rclcpp
  Version: ROS 2 v0.6.1
  Branch name: cbg-executor-0.6.1

Demo for Callback-group-based Executor concept in micro-ROS-demo repository:

- Git repository: https://github.com/micro-ROS/micro-ROS_experiments/
  Branch name: cbg-executor-0.6.1
  Latest commit: a902f39

3 Annex 1: Webpage on Real-Time Executor

Content of https://micro-ros.github.io/real-time_executor/ from 19th December 2019.

3.1 Table of contents

- Introduction
- Analysis of rclcpp standard Executor
  - Architecture
  - Scheduling Semantics
- Rcl LET-Executor
  - Concept
  - Example
  - Download
- Callback-group-level Executor
  - API Changes
  - Meta Executor Concept
  - Test Bench
- Related Work
- Roadmap
- 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-prune, 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).

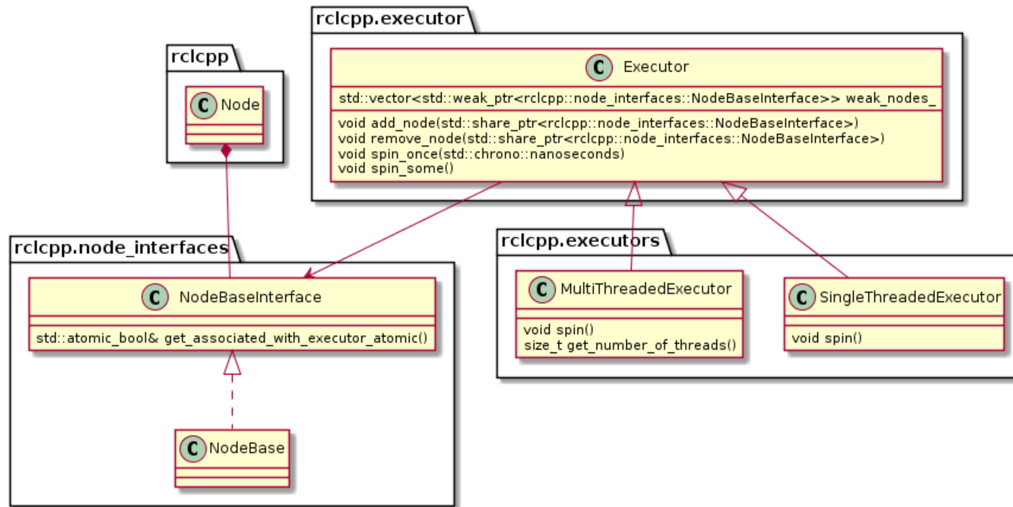
The ROS 2 design defines one Executor (instance of rclcpp::executor::Executor) per process, which is typically created either in a custom main function or by the launch system. The Executor coordinates the execution of all callbacks issued by these nodes by checking for available work (timers, services, 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 queues, which notifies it of any pending callback in the DDS queue. If there are multiple pend-

ing callbacks, the ROS 2 Executor executes them in an in the order as they were registred at the Executor.

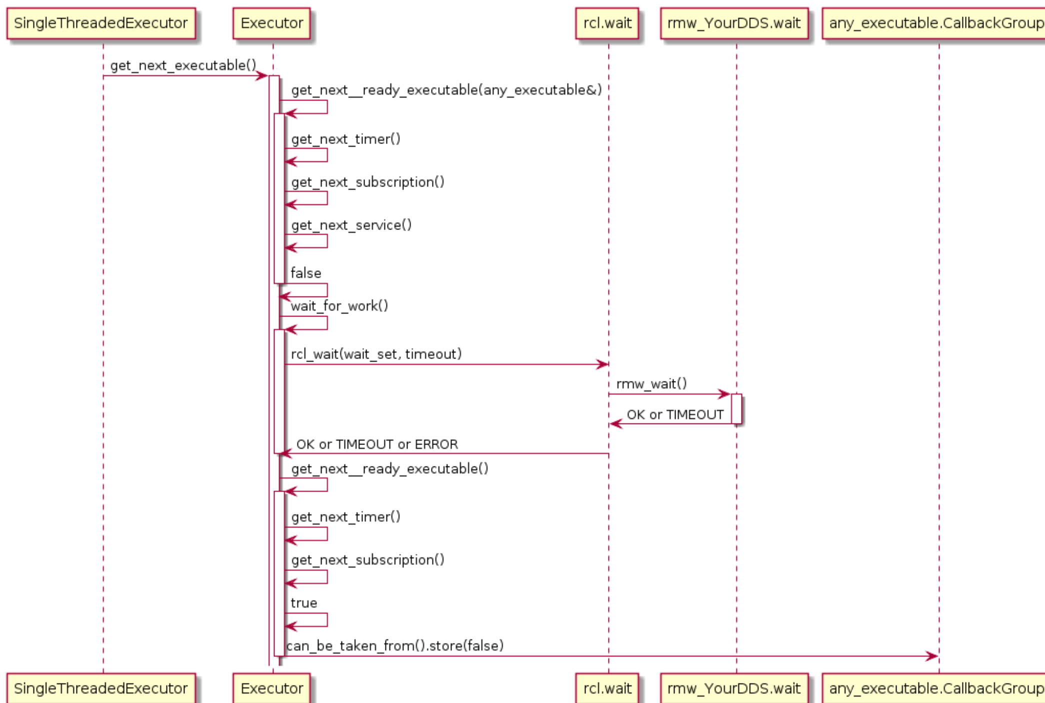
3.3.1 Architecture

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 Rcl LET-Executor

This section describes the rcl-LET-Executor. It is a first step towards deterministic execution by providing static order scheduling with a let semantics. The abbreviation let stands for Logical-Execution-Time (LET) and is a known concept in automotive domain to simplify synchronization in process scheduling. If refers to the concept to schedule multiple ready tasks in such a way, that

first all input data is read for all tasks, and then all tasks are executed. This removes any inter-dependence of input data among these ready tasks and hence input data synchronization is not necessary any more[BP2017] [EK2018].

3.4.1 Concept

The LET-Executor consists of tho phases, configuration and running phase. First, in configuration phase, the total number of handles are defined. A handle is the term in the rcl-layer to generalize *timers*, *subscriptions*, *services* etc.. Also in this phase, the execution order of the callbacks is defined. Secondly, in the running phase, the availability of input data for all handles is requested from the DDS-queue, then all received input data is stored and, finally, all callbacks corresponding to the handles are executed in the specified order. With this two-step approach, the LET-Executor guarantees a deterministic callback execution (pre-defined static order) and implements the LET semantics while executing the callbacks.

3.4.2 Example

We provide an example for using the LET-Executor. First, the callbacks *my_sub_cb* and my_timer_cb_ are defined for a subscription and a timer, respectivly. Then, the ROS *context* and the ROS *node* are defined as well as the subscription object *sub* and the timer object *timer*.

The Executor is initialized with two handles. Then, the *add* functions define the static execution order of the handles. In this example, the subscription *sub* shall be processed before the timer *timer*. Finally, the Executor is activated with the *spin_period* function, which is continuously called every 20 ms.

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

// define subscription callback
void my_sub_cb(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 sub;
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 a subscription
rcl_subscription_init(&sub, &node, ...);

// create a timer
rcl_timer_init(&timer, &my_timer_cb, ... );

// initialize executor with two handles
rcle_let_executor_init(&exe, &context, 2, ...);

// define static execution order of handles
rcle_let_executor_add_subscription(&exe, &sub, &my_sub_cb, ...);
rcle_let_executor_add_timer(&exe, &timer);

// spin with a period of 20ms
rcle_let_executor_spin_period(&exe, 20);
```

3.4.3 Download

The LET-Executor can be downloaded from the micro-ROS GitHub rcl_executor repository. The package rcl_executor provides the LET-Executor library with a step-by-step tutorial and the package rcl_executor_examples provides an example, how to use the LET-Executor.

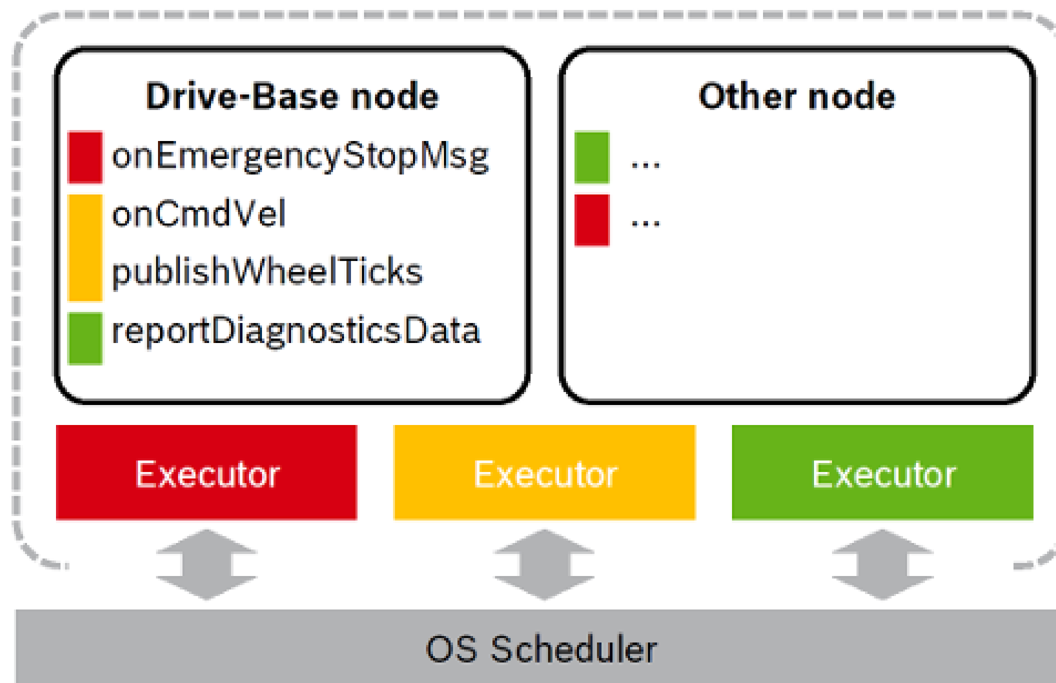
3.5 Callback-group-level Executor

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.

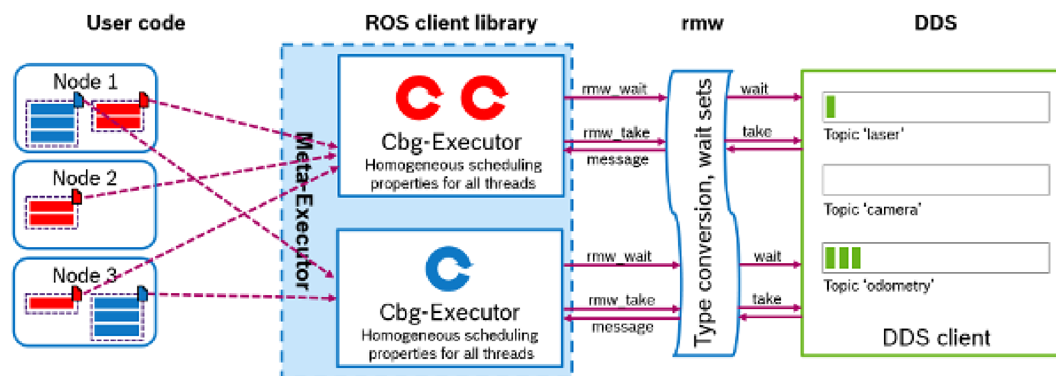
- include/rclcpp/node.hpp and include/rclcpp/node_interfaces/node_base.hpp
  - Extended arguments of create_callback_group function for the real-time class.
  - Removed the get_associated_with_executor_atomic function.

All the changes can be found in the branches cbg-executor-0.5.1 and cbg-executor-0.6.1 for the corresponding version 0.5.1 and 0.6.1 of the rclcpp in the fork at github.com/micro-ROS/rclcpp/.

3.5.2 Meta Executor Concept

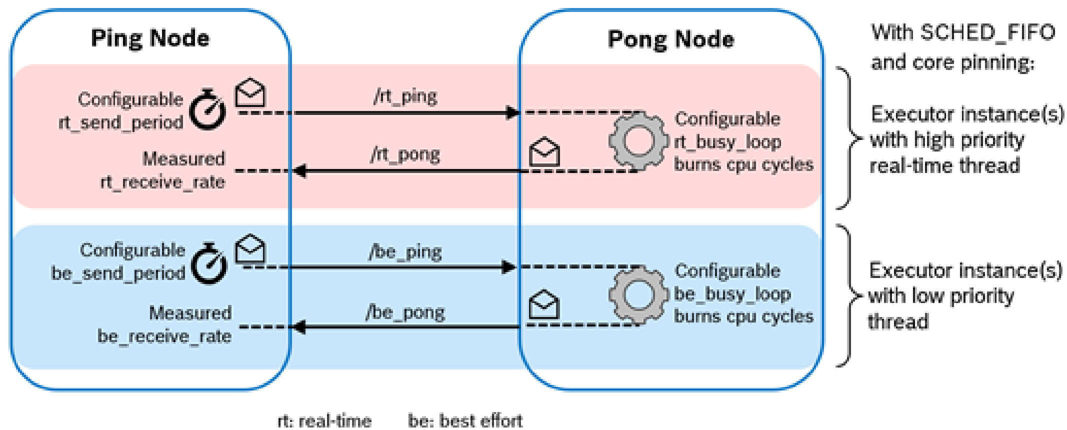
The idea of the Meta Executor is to abstract away the callback-group assignment, thread allocation and other inner workings of the Executors from the user, thereby presenting a simple API that resembles the original Executor interface. Internally, the Meta Executor maintains multiple instances of our Callback-group-level Executor (Cbg-Executor).

The Meta Executor internally binds these Executors to the underlying kernel threads, assigns them a priority, chooses the scheduling mechanism (e.g., SCHED-FIFO policy) and then dispatches them. When adding a node with its list of callback group and real-time profiles to the Meta Executor, it parses the real-time profiles and assigns the node's callback groups to the relevant internal Executors.

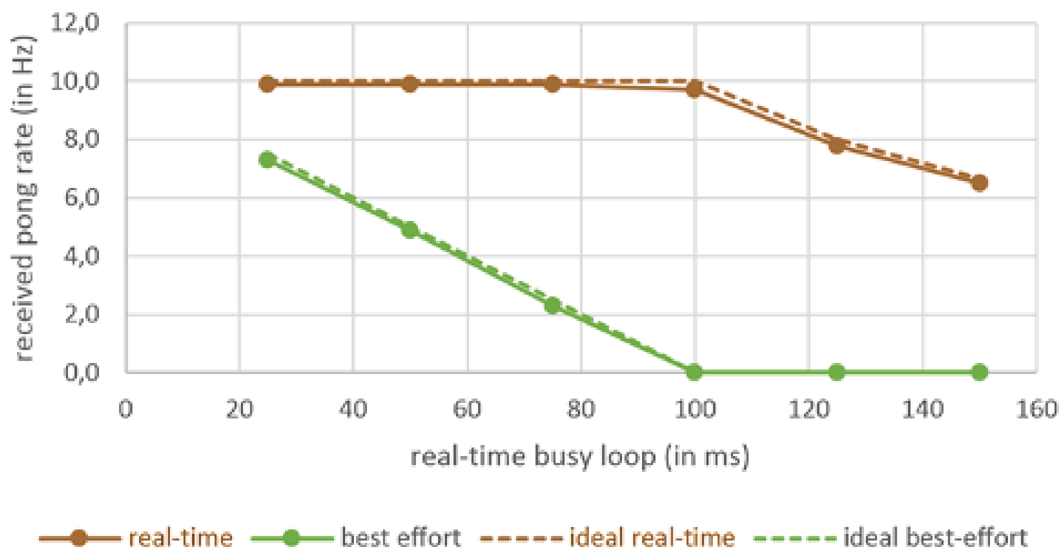


3.5.3 Test Bench

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 - here on ROS 2 v0.5.1 with the Fast-RTPS DDS implementation - on a typical laptop.



The test bench is provided in the bg-executor_ping-pong_cpp package of the micro-ROS_experiments repository.

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")
```

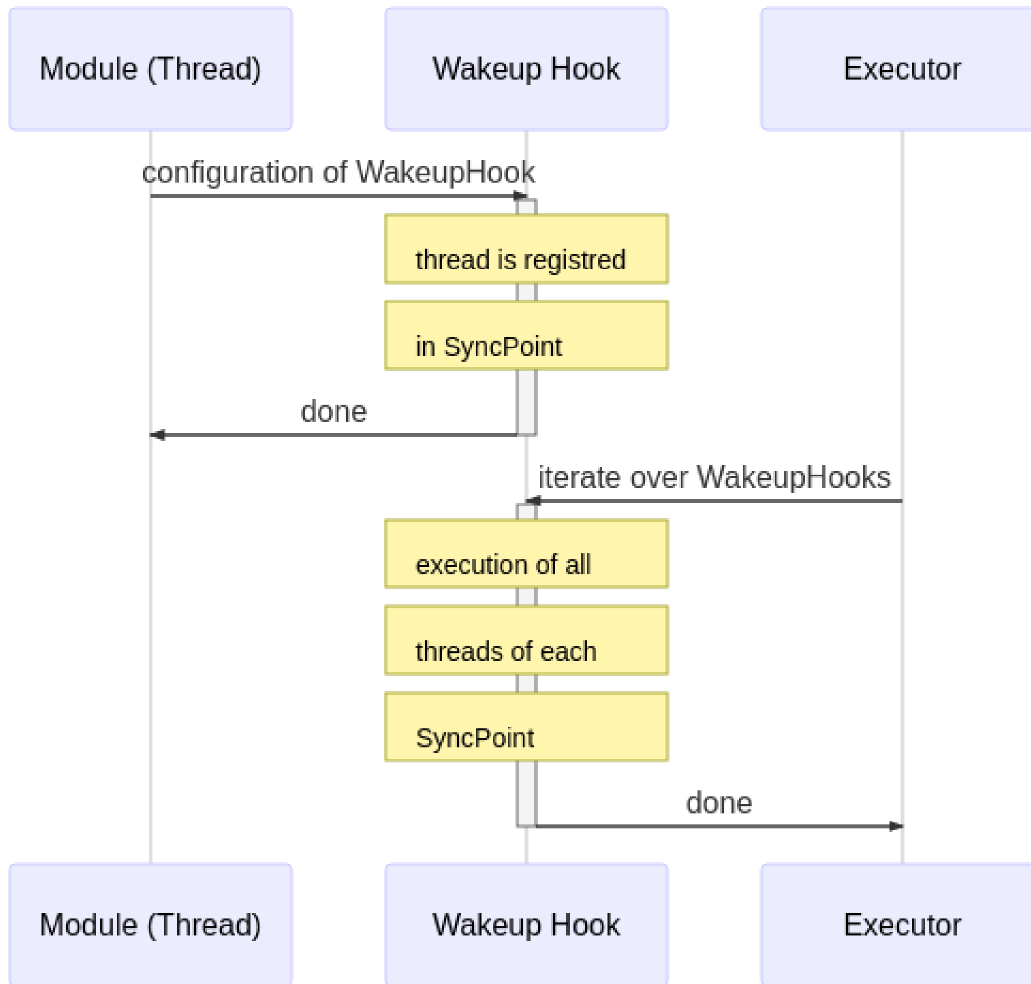
Similarly, if `NaoQiButtonThread` shall be executed in the SENSOR_PROCESS hook, the constructor is:

```
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 Roadmap

2018

- In-depth analysis of the ROS 2 Executor concept.
- Design and analysis of Callback-group-level Executor.
- Validated correct functioning of underlying layers and middleware with multiple Executor instances.

2019

- Design and implementation of LET Executor for rcl.
- Design and implementation of domain-specific Executor for rcl, featuring sense-plan-act semantics
- In-depth runtime analysis of default rclcpp Executor using ROS 2 tracetools developed in micro-ROS.
- Design and implementation of static Executor for rclcpp with improved performance.
- Research of concepts for model-based optimization of end-to-end latencies.

2020

- Integration with a selected advanced scheduling and resource monitoring mechanisms such as reservation-based scheduling.
- Integration with selected sleep modes and low-power modes.

3.8 References

- 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]

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).