D4.4

Real-time Executor Software Release Y1

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D4.4
Deliverable name	Real-time Executor – Software Release Y1
Date	December 2018
Dissemination level	public
Workpackage and task	4.2
Author	Ralph Lange (Bosch)
Contributors	Jan Staschulat (Bosch)
Keywords	micro-ROS, robotics, ROS, microcontrollers, scheduling, executor
Abstract	This document provides links to the released software and documentation for deliverable D4.4 *Real-time Executor Software Release Y1* 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 Introduction and Goal . . . 3
3.2 Background: ROS 2 Executor Concept . . . 3
3.2.1 Description . . . 3
3.2.2 Architecture . . . 3
3.2.3 Analysis . . . 4
3.3 Callback-group-level Executor . . . 5
3.3.1 API Changes . . . 6
3.3.2 Meta-Executor Concept . . . 7
3.3.3 Test Bench . . . 8
3.4 Roadmap . . . 8
3.5 Related Work . . . 9
3.5.1 Sense-Plan-Act Cycle in Fawkes . . . 9
3.6 References and Links . . . 11
3.7 Acknowledgments . . . 11

4 Annex 2: README.md from cbg-executor_ping-pong Package **11**

4.1 Running the test bench . . . 12
4.2 Implementation Details . . . 13

1 Overview to Results

This document provides links to the released software and documentation for deliverable D4.4 *Real-time Executor Software Release Y1* 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://microros.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

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

- Git repository: https://github.com/microROS/rclcpp
  Version: ROS 2 v0.5.1
  Branch name: cbg-executor-0.5.1
  Squashed commit: dd31084
- Git repository: https://github.com/microROS/rclcpp
  Version: ROS 2 v0.6.1
  Branch name: cbg-executor-0.6.1
  Squashed commit: 08e9a1e

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

- Git repository: https://github.com/microROS/micro-ROS-demos
  Package name: cbg-executor_ping-pong_cpp
  Package path: ./Cpp/cbg-executor_ping-pong
  Major commits: a80e395, 29bb2c9

Markdown source of the microros.github.io/real-time_executor/ webpage:

- Git repository: https://github.com/microROS/microros.github.io
  Webpage path: ./real-time_executor/
  Major commits: 63de055, 74ed24c

3 Annex 1: Webpage on Real-Time Executor

Content of https://microros.github.io/real-time_executor/ from 14th December 2018.

3.1 Introduction and Goal

Predictable execution under given real-time constraints is one of the hallmarks of embedded systems, but is also a complex subject that can cause many errors. The real-time executor aims to provide a convenient API for the developers of micro-ROS based components and systems to simplify scheduling and to reduce errors, taking advantage of typical processing patterns in robotics applications.

The approach is based on the concept of executors, which have been introduced in ROS 2. The real-time executor shall provide fine-grained control of the mapping of callbacks to the scheduling primitives and mechanisms of the underlying RTOS, even across multiple nodes.

3.2 Background: ROS 2 Executor Concept

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).

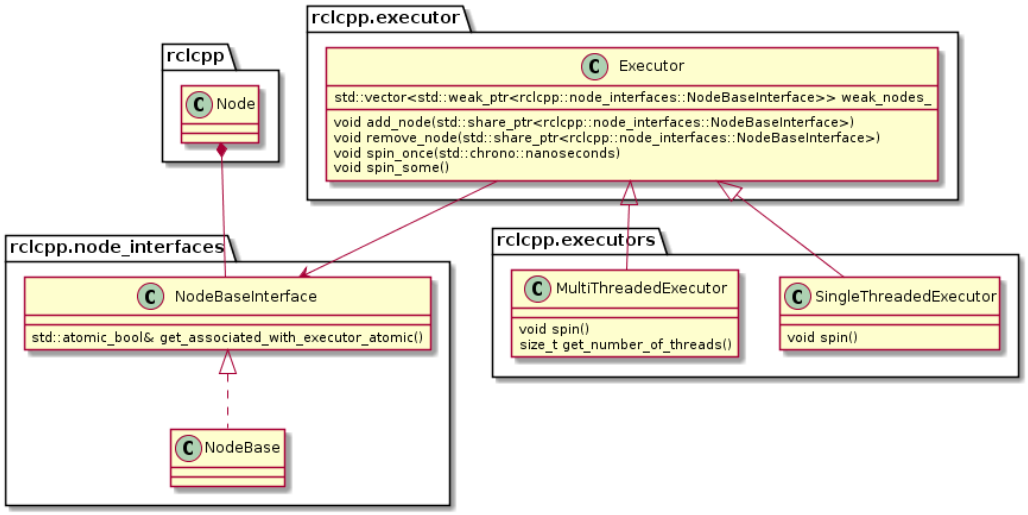
3.2.1 Description

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 pending callbacks, the ROS 2 Executor simply executes them in a FIFO manner.

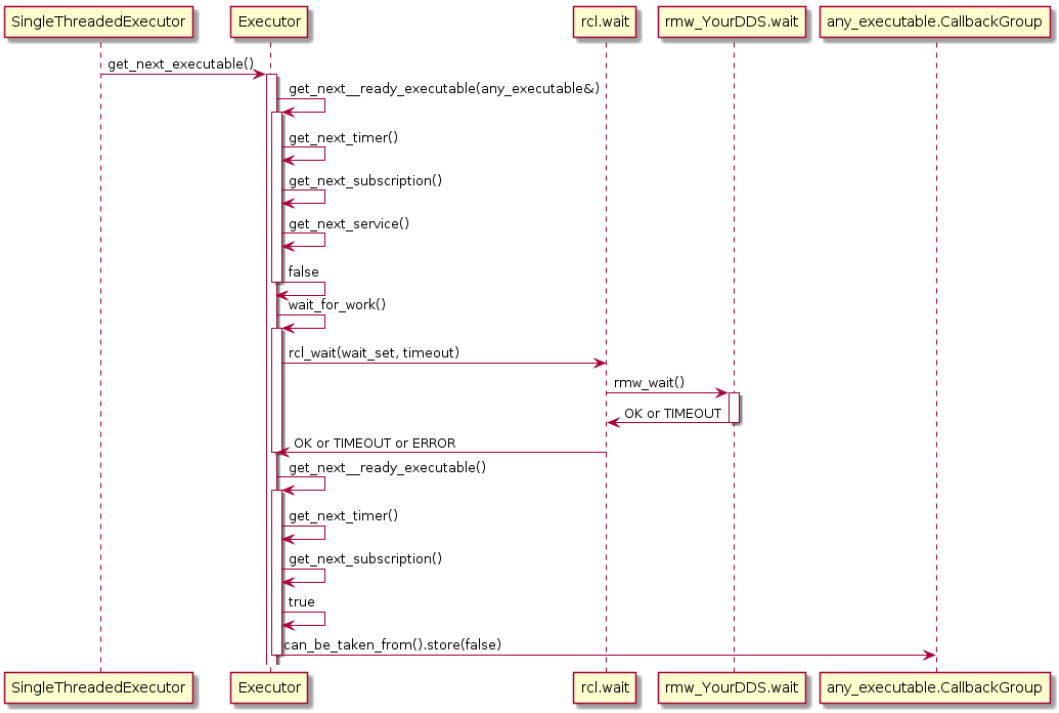
3.2.2 Architecture

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



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:



3.2.3 Analysis

In the present Executor concept there is no notion of 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 degraded performance since they are serviced later than non-critical callbacks. Additionally, due to the FIFO

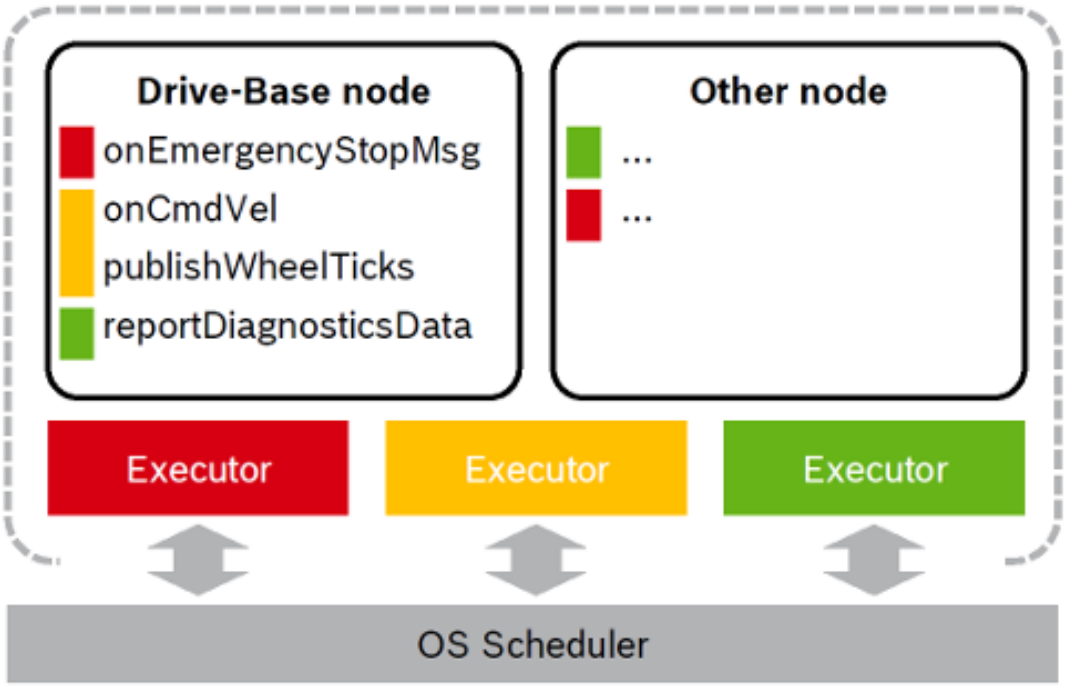
mechanism, it is difficult to determine usable bounds on the worst-case latency that each callback execution may incur.

3.3 Callback-group-level Executor

In order to address the challenges mentioned above, some changes are imminent: Firstly, an API to express real-time requirements on a callback level is needed and secondly, the Executor must be redesigned to respect these real-time requirements in its scheduling decisions. As the current ROS 2 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 ROS 2 Executor 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:



In this section, we describe the necessary changes to the Executor API and report on first experiments with it.

3.3.1 API Changes

- include/rclcpp/callback_group.hpp:

  Introduced enum to distinguish up to three real-time classes (requirements) per node.

```
enum class RealTimeClass
{
  RealTimeCritical,
  SoftRealTime,
  BestEffort
};
```

  Also, changed association with Executor instance from nodes to callback groups.

```
class CallbackGroup
{
  ...

  RCLCPP_PUBLIC
  std::atomic_bool &
  get_associated_with_executor_atomic();

  ...
}
```

- include/rclcpp/executor.hpp

  Added functions to add and remove individual callback groups in addition to whole nodes.

```
class Executor
{
  ...

  RCLCPP_PUBLIC
  virtual void
  add_callback_group(
    rclcpp::callback_group::CallbackGroup::SharedPtr group_ptr,
    rclcpp::node_interfaces::NodeBaseInterface::SharedPtr node_ptr, bool notify = true);

  RCLCPP_PUBLIC
  virtual void
  remove_callback_group(
    rclcpp::callback_group::CallbackGroup::SharedPtr group_ptr,
    bool notify = true);

  ...
}
```

Replaced private vector of nodes with a map from callback groups to nodes.

```
typedef std::map<rclcpp::callback_group::CallbackGroup::WeakPtr,
  rclcpp::node_interfaces::NodeBaseInterface::WeakPtr,
  std::owner_less<rclcpp::callback_group::CallbackGroup::WeakPtr>> WeakCallbackGroupsToNod
WeakCallbackGroupsToNodesMap weak_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.

```
create_callback_group(
  rclcpp::callback_group::CallbackGroupType group_type,
  rclcpp::callback_group::RealTimeClass real_time_class =
  rclcpp::callback_group::RealTimeClass::BestEffort);
```

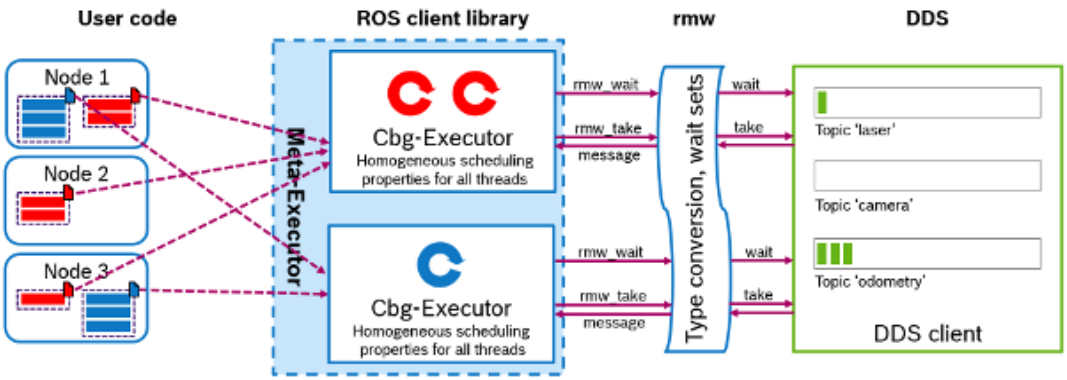
  Removed the get_associated_with_executor_atomic function.

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/microROS/rclcpp/.

3.3.2 Meta-Executor Concept

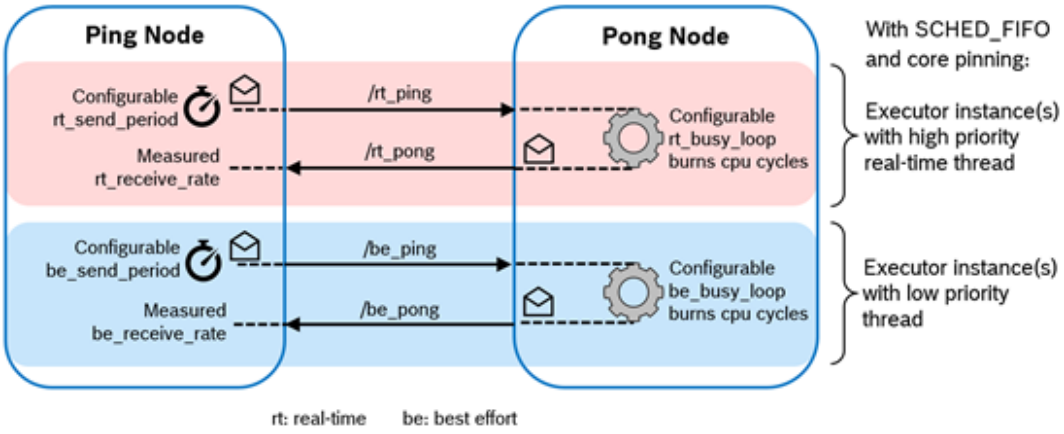
The idea of the Meta Executor is to abstract away the callback-group assignment, thread allocation and other inner workings of the Executors from the user, thereby presenting a simple API that resembles the original Executor interface. Internally, the Meta Executor maintains multiple instances of our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

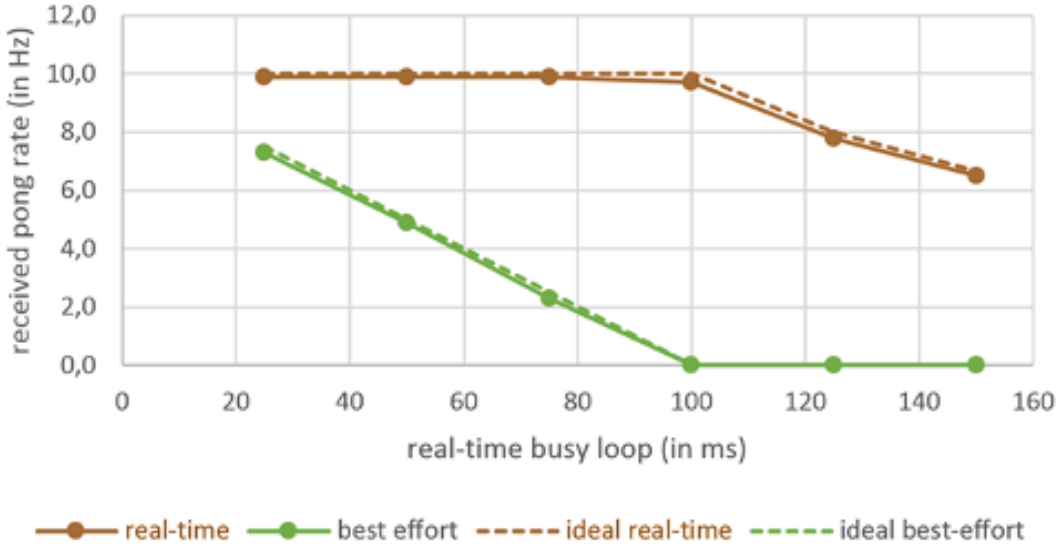


3.3.3 Test Bench

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-demos repository.

3.4 Roadmap

2018

- In-depth analysis of the ROS 2 Executor concept.
- Designed Callback-group-level Executor and implemented it - at first, prototypically, for ROS 2.

- Designed the Meta Executor concept.
- Validated correct functioning with underlying layers and middleware.

2019

- Implementation of the Callback-group-level Executor and Meta Executor for urcl.
- 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.

3.5 Related Work

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

3.5.1 Sense-Plan-Act Cycle in Fawkes

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.

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

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")
```

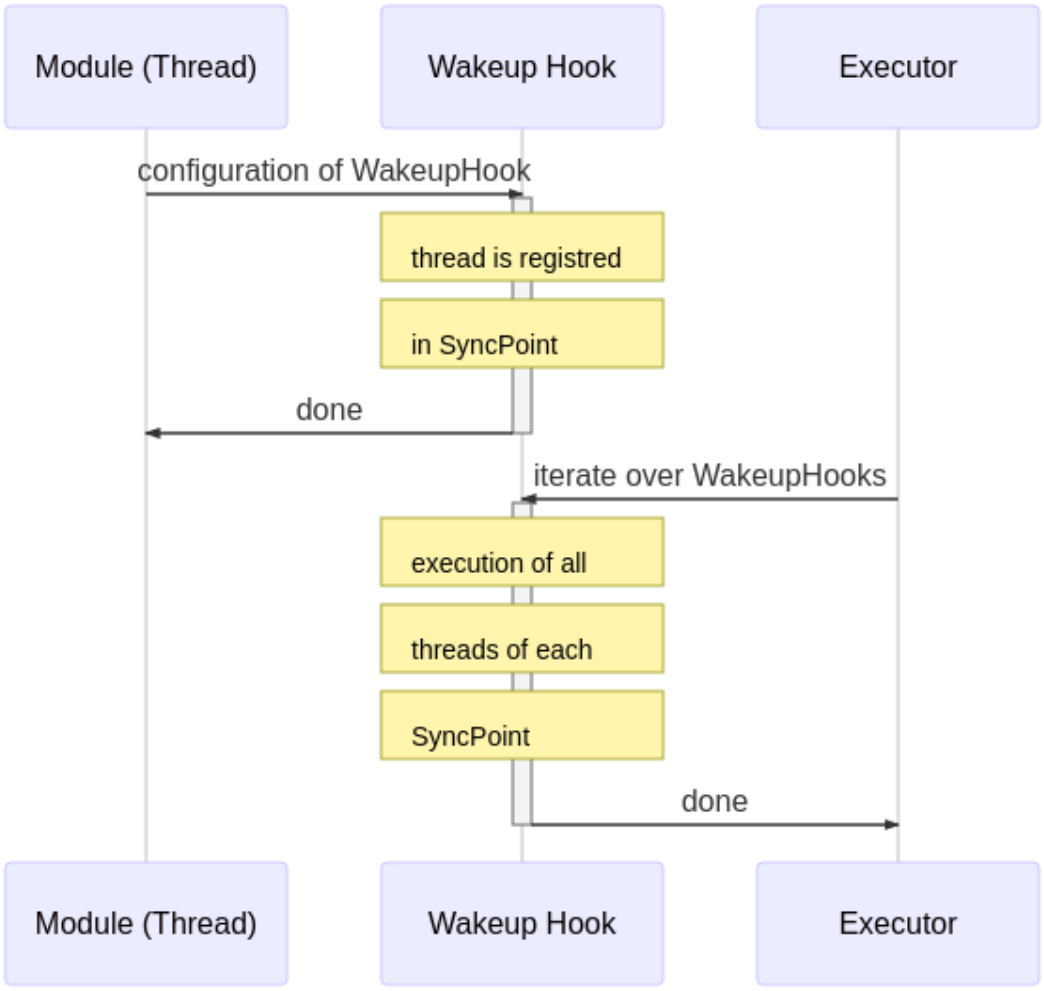
Similarly, if `NaoQiButtonThread` shall be executed in the SENSOR_PROCESS hook, the constructor is:

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

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++.

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.

3.6 References and Links

- Ralph Lange: Callback-group-level Executor for ROS 2. Lightning talk at ROSCon 2018. Madrid, Spain. Sep 2018. [Slides] [Video]

3.7 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).

4 Annex 2: README.md from cbg-executor_ping-pong Package

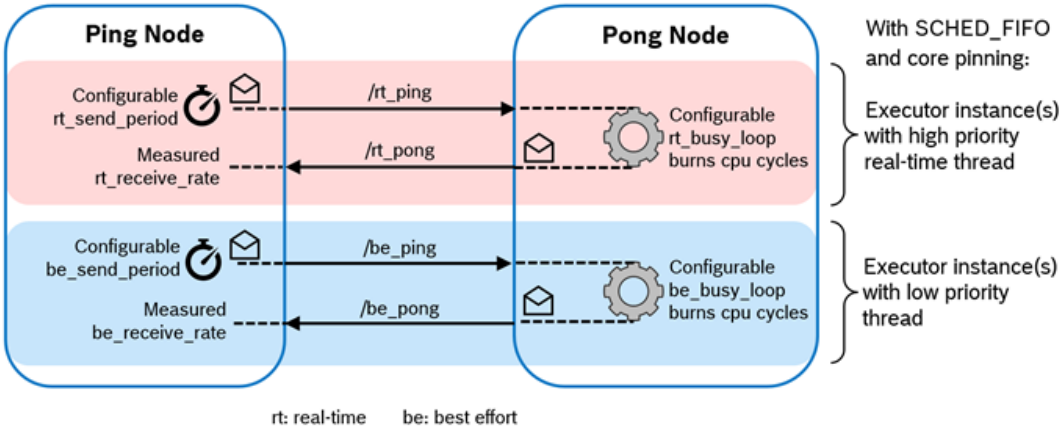
Content of https://github.com/microROS/micro-ROS-demos/blob/master/Cpp/cbg-executor_ping-pong/README.md from 14th December 2018.

This package provides a small example for the use of the Callback-group-level Executor concept.

The Callback-group-level Executor leverages the callback-group concept 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 requiring specific real-time guarantees, when created, may therefore be associated with a dedicated callback group. With this in place, the Executor class and depending classes (e.g., for memory allocation) were enhanced to operate at a finer, callback-group-level 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 one or few specific callback groups and the Executor's thread (or threads) can be prioritized according to the real-time requirements of these groups. For example, all time-critical callbacks may be handled by an "RT-CRITICAL" Executor instance running at the highest scheduler priority.

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.



The Ping node can be configured to send messages at a configured rate. The Pong node takes these ping messages and replies each of them. Before sending the reply, it can be configured to burn cycles (thereby varying the processor load) to simulate some message processing. We provide bash scripts to test intra-process and inter-process communication scenarios, wherein the nodes are co-located either in one process or two processes, respectively. These scripts also vary the message rates and processor loads. After each run, the Ping node outputs the measured throughput of real-time and best effort messages.

4.1 Running the test bench

After building the test bench with colcon, the Ping and Pong nodes may be either started in one process or in two processes. Please note that the test bench requires sudo privileges to be able to change the thread priorities using `pthread_setschedparam(..)`. Start the executable by

```
ros2 run cbg-executor_ping-pong_cpp ping-pong args...
```

where `args...` are five or six arguments as follows:

```
[type] [rt_ping_period_us] [be_ping_period_us] [rt_busyloop_us] [be_busyloop_us] [cpu_id]

type: determines the nodes included in this process:
  i: ping node only
  o: pong node only
  io: ping node and pong node

rt_ping_period_us: microseconds between publishing of ping messages by real-time thread
                   in ping node
be_ping_period_us: microseconds between publishing of ping messages by best-effort
                   thread in ping node
rt_busyloop_us:    microseconds of computation by real-time thread in pong node before
                   answering with pong
be_busyloop_us:    microseconds of computation by best-effort thread in pong node before
                   answering with pong
cpu_id (optional): pins both, real-time thread and best-effort thread, to the given cpu
```

When using type `i`, a second process with type `o` has to be started simultaneously.

The shell run* scripts in this folder run various experiments in sequence. For pertinent results, we propose to use the PREEMPT_RT patch for the Linux kernel.

4.2 Implementation Details

The algorithms of the Ping node and of the Pong node are factored out into classes *PingSubNode* and *PongSubNode* - configurable with regard to the real-time profile and the topic prefix. Thus, the Ping node contains two instances of the *PingSubNode* and the Pong node contains two instances of *PongSubNode*. (And the test bench could be easily extended to more than two ping-pong paths.)

The PingSubNode contains a timer for sending the ping messages and a subscription for the corresponding pong messages. Also, it records the number of messages being sent and received and measures the roundtrip time.

The PongSubNode contains a subscription for the ping messages and a publisher for the corresponding pong messages. On receiving a ping message, it calls the `PongSubNode::burn_cpu_cycles()` functions to simulate a given processing time before replying with a pong.

The Ping and Pong nodes, the two executors, etc. are composed and configured in the `main(..)` function of main.cpp. This function also starts and ends the experiment for a predefined duration and prints out the throughput and latency statistics.