D2.10

Report on RTOS scheduling

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D2.10
Deliverable name	Report on RTOS scheduling
Date	December 2018
Dissemination level	public
Workpackage and task	2.4
Author	Acutronic Robotics
Contributors	Jan Staschulat (Bosch)
Keywords	micro-ROS, robotics, ROS, microcontrollers, scheduling, real-time
Abstract	This document explains the need of scheduling mechanisms, reviews different scheduling algorithms and shows the scheduling offered by the RTOSes analyzed in the project scope.

Contents

1 Summary **2**

2 Acronyms and keywords **2**

3 Introduction **2**

4 Common scheduling mechanisms **3**

5 Micro-ROS software classification **4**

6 Scheduling in popular RTOSes **5**

7 Analysis of RTOS schedulers in micro-ROS **6**

7.1 FreeRTOS . . . 6

7.1.1 Scheduling algorithms . . . 7

7.2 NuttX . . . 7

7.3 Scheduling abstraction layer . . . 9

7.3.1 Creation of a task . . . 10

7.3.2 Delete of a task . . . 10

7.3.3 Pause the execution of a task . . . 10

7.3.4 Periodically wakes up . . . 10

7.3.5 Set/Get priority of the Task . . . 10

7.3.6 Get the ID of the Task . . . 10

7.4 Scheduling mechanism extension . . . 11

8 Conclusion **12**

1 Summary

This report describes the importance of scheduling mechanisms in real-time systems, in which deterministic response times of the software components is a requirement especially in safety critical use-cases. First, common scheduling mechanisms are presented which have been known for decades. Then, we compare different real-time operating systems (RTOS) in respect to scheduling features. To be more specific, the scheduling properties the RTOSes deployed in the micro-ROS project, FreeRTOS and NuttX, are presented in detail. Finally, the scheduling abstraction layer is described. We conclude that further work needs to be invested to implement consistent scheduling mechanisms in the ROS-stack before extending scheduling techniques in the RTOS.

2 Acronyms and keywords

Term	Definition
RTOS	Real-Time Operating System
DDS	Data Distribution Service
XRCE	Extremely Resource-Constrained Environments
ROS	Robot Operating System
UDP	User Datagram Protocol
TCP	Transmission Data Protocol
6LOWPAN	IPv6 over Low Power Wireless Personal Area Networks
CPU	Central Processing Unit
RAM	Random Access Memory
MCU	Microcontroller unit
AL	Abstraction layer

3 Introduction

Schedulers are software entities which manage the execution of processes using different techniques or objectives. Their main task is to decide which process to execute at each time in the processing unit. In real-time embedded systems, for example robotics, a scheduling analysis must be applied to ensure that all processes can meet their deadlines. Hard deadlines are required in some use-cases, where safety functionality of the system is at stake. These processes could be of different nature, such us communication, data processing or control tasks.

In micro-ROS, a complete software stack, that comprises of an RTOS, an ROS middleware-stack and the micro-ROS based application code, typically runs on a single device. Several scheduling methods are needed to execute all software entities of the system.

This document gives an overview of the scheduling methods and provides an insight of the schedulers used by popular RTOSes. Finally, the conclusions are presented together with the future steps

to be performed in order to have an appropriate mechanisms to ensure the correct function of micro-ROS in terms of process deadlines.

4 Common scheduling mechanisms

When a single processor has to execute a set of concurrent tasks, the CPU has to be assigned to the various tasks according to a predefined criterion, called a scheduling policy. The set of rules that, at any time, determines the order in which tasks are executed is called a scheduling algorithm. There have been several scheduling algorithms proposed to improve the efficiency and predictability of real-time systems [B05].

A very simple scheduling policy is static scheduling. Static algorithms are those in which scheduling decisions are based on fixed parameters, assigned to tasks before their activation. For example, in a complete static scheduler the order of tasks are fixed and they are executed sequentially after each other. The benefit of fixed order comes with the price of non-deterministic reaction times, as the execution time of tasks may vary. Another simple static scheduling policy is **Round-Robin scheduling**. Pre-defined time-slices are assigned to each process in equal portions and in circular order, handling all processes without any priority. In general Round Robin scheduler are an instance of cyclic scheduling and is a technique to guarantee equal progress for each task.

Fixed priority based scheduling is a scheduling policy in which each task is assigned a pre-defined priority. At run-time always the task with the highest priority is executed. Without preeemptions this scheduling policy is also called **FIFO scheduling**. FIFO scheduling is sometimes combined with Round Robin scheduling for tasks with equal priority.

The Rate Monotonic (RM) scheduling algorithm is a preemptive fixed priority based scheduling with the rule that assigns priorities to tasks according to their request rates. Tasks with higher request rates (that is with shorter periods) will have higher priorities. With a preemptive algorithm, the running task can be interrupted at any time to assign the processor to another active task, according to a predefined scheduling policy. RM is such a preemptive scheduling technique, in which the currently executing task is preempted by a newly arrived task with a shorter period. In 1973, Liu and Layland [LL73] showed that RM is optimal among all fixed-priority assignments, in the sense that no other fixed-priority algorithms can schedule a task set that cannot be scheduled by RM. Also they have derived a least upper bound of the processor utilization for a generic task set of n periodic tasks. That is, given a set of n periodic tasks, a feasible schedule can be found, that will always meet all deadlines if the CPU utilization is below a specific bound. If the number of tasks is infinite, this bound is about 0.6931. Therefore, a rough estimate is that RM can meet all deadlines if the CPU utilization is less than 69.32%. However this condition is not a necessary one. It cannot be said that a system with a higher utilization is not schedulable with RM.

In 1974, Horn presented an **Earliest Deadline First (EDF) scheduling** algorithm, which can schedule a task set with a CPU utilization of 100%. In EDF always the task with the earliest deadline will be scheduled. EDF is a dynamic scheduling algorithm, because at run-time the remaining time to the tasks deadline needs to be evaluated. The algorithm is therefore more difficult to implement [Hor74].

The drawback of fixed priority-based scheduling and EDF is, that the real-time guarantee for one task depends the the resource requirements of other tasks in the system. If this set of tasks is extended by a new task, then the CPU utilization is higher. This has consequently an impact on the

schedulability of the existing task set. Independence and composability in terms of resource allocation can be obtained by **reservation-based scheduling (RBS)**. In this scheduling policy, in a given time interval each task is assigned a pre-defined time-slice (budget), in which it can execute. The budget can be simply replenished when the time interval has elapsed. In more advanced versions, un-used time-budgets can be used by other tasks as well (called slack stealing). With this approach the CPU utilization can be partitioned among tasks sets and overload situations have no impact on other tasks [AB04].

[B05] Giorgio C. Buttazo: Hard Real-Time Computing Systems, Predictable Scheduling Algorithms and Applications, Springer, 2005.

[LL73] Liu, C. L. and Layland, James W..Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, J. ACM, Jan. 1973, Vol 20 Nr. 1, 46-61.

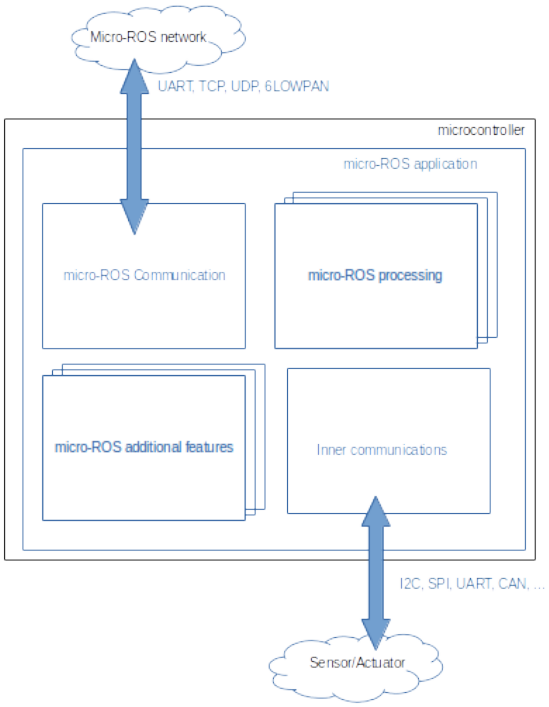
[Hor74] W. Horn. Some simple scheduling algorithms. Naval Research Logistics Quarterly, 21, 1974.

[AB04] Abeni, L and Buttazzo, G. Resource reservation in dynamic real-time systems. Real-Time Systems 27(2):123-167, 2004.

5 Micro-ROS software classification

A standard micro-ROS device will be comprised typically of the following functionalities in an environment where it is integrated with additional micro-ROS and ROS-devices:

- Communication tasks with other micro-ROS devices: comprising of a communication channel, where different protocols could be used, such as serial, UDP/TCP or 6LOWPAN. At middleware layer, this entity is responsible for communicating with a Micro XRCE-DDS Agent, which provides the capability of establishing communications with the rest of micro-ROS devices as well as with the ROS ecosystem.
- Inner communication tasks: comprising of the communication to be performed by the microcontroller to the connected sensors or actuators. The aim is to integrate these devices in the micro-ROS domain. The communication will allow reading sensor data or making control of an actuator.
- Processing tasks: comprising of the sensor and actuator driver or control tasks, as well as the micro-ROS application code execution, where ROS concepts are implemented, such as topic creation, publishing or subsciption to topics.
- Features: such us power management, CPU load or RAM usage, or debugging.



The scheduling mechanisms used in micro-ROS shall handle the features described above and shall provide the computing resources for these tasks such that all real-time requirements are met.

6 Scheduling in popular RTOSes

In micro-ROS software stack, the use of RTOS is contemplated. The features they offer in terms of characteristics is diverse. These differences starts from the targets MCUs that are able to run the RTOS, until the features they offer. The same applies to the scheduling mechanisms offered by the different RTOSes. Here we refer to the comparison table which was presented in the deliverable 2.1 *Report on the reference hardware development platforms*, and highlight only the scheduling methods of the compared RTOSes. The variety of these scheduling features make it difficult to find a common abstraction layer for a sound scheduling mechanism in micro-ROS. Such an abstraction shall enable switching the RTOS in the micro-ROS stack with no effort.

OS	NuttX	Micrium	ARM Mbed 5	FreeRTOS	OSEK Erika	Zephyr	RIOT
Scheduling							
Priority-based	FIFO	Yes	Yes	Yes	Yes	Yes	Yes
Round-Robin	Yes	Yes	Yes	Yes	ND	Yes	Co-op
EDF	No	No	No	No	Yes	No	No
RBS	No	No	No	No	Yes	No	No
Resource Mngt.							
Mutex	PI	PCP	PI	PI, PCP	IPC	PI	PI

Abbreviations:

- PI: Priority Inheritance
- PCP: Priority Ceiling Protocol
- IPC: Immediate Priority Ceiling
- ND: Not documented
- Co-op: Co-operative

While fixed-priority based scheduling as well as Round Robin scheduling are supported by most RTOS, the more advanced scheduling techniques which provide higher utilization (EDF) or guaranteed time-bandwidth (RBS) are only supported by a few. We need to analyze which policies are best suited for the tasks in the micro-ROS context.

7 Analysis of RTOS schedulers in micro-ROS

In this section we analyze which scheduling techniques are offered by the project's default RTOSes, which are NuttX and FreeRTOS. NuttX offers a powerful set of scheduling techniques. This RTOS is complient to Linux as much as possible, using POSIX standards as development principle. On the other hand, FreeRTOS uses a much simpler techniques.

7.1 FreeRTOS

FreeRTOS offers basic functionality for handling threads, in comparison to NuttX. In this RTOS, there are two kinds of tasks: standard tasks and idle tasks. The standard tasks are the applications user can execute. In FreeRTOS, the following parameters need to be specified for a tasks:

- A unique name for the task.
- A function that is attached to the task.
- The priority of the task.
- The stack size of the task.

From all these parameters, the most important one in respect to scheduling is the task priority. The priority is a positive number that indicates to the scheduler the importance of a task. This number is set by the user and can take values from 0 to `configMAX_PRIORITIES - 1`, where `configMAX_PRIORITIES` is a positive integer value stored in the configuration file `FreeRTOSConfig.h`. Low numbers are assigned to low priority tasks.

On the other hand, there is an Idle Task, which is a task that is created and executed at the startup of the RTOS. This task has the lowest priority and it enters into run mode only when there is no other task running. This task has priority number 0 assigned. The default objective of this task is to clean-up the resources the executed tasks has used, once these task has been deleted. Also, it is possible to implement power management logic in this task. It's important to avoid adding too many functionalities in this task, because it is only executed when no other task is running. This means, it might not receive sufficient execution time by the scheduler, making it not work correctly.

If a task has the highest priority and doesn't have any internal delay or request for external data, it could claim all of the processing time. To avoid this situation, FreeRTOS offers several configuration options to the developer. One of those is `TaskDelay`, which allows to pause the task for a pre-defined number of system ticks. This is useful because a minimal computational time can be guaranteed to other tasks and the continuous use of the MCU by the same task is avoided. Another option is `TaskDelayUntil`. This is similar to `TaskDelay`.

Basic features of a standard task are, that it can be created, run and deleted. The priority of a task can be modified at run time.

Finally, FreeRTOS offers a thread extension for enhancing thread functionality. This extension allows having pthread-like API in FreeRTOS. This tool needs to be analyzed to see if it would help to match NuttX and FreeRTOS interfaces, allowing to improve the implementation of the Scheduling abstraction layer.

7.1.1 Scheduling algorithms

Once we have understood better how FreeRTOS uses the tasks, it is time to understand how these tasks are handled. In order to determine which task to execute, scheduler algorithms are used. FreeRTOS give us the possibility to choose between three different scheduling algorithms.

By default, *Prioritized Preemptive Scheduling with Time Slicing* algorithm is used. With this algorithm, the highest priority task will be running first, but if it consumes more processing time than the assigned time slice, the scheduler will execute the next highest priority task.

The second scheduling option is the *Prioritized Preemptive Scheduling (without Time Slicing)*. This algorithm will run first the highest priority task, and it will be running until completion or until something stops it, for example, it could enter in pause mode or wait for data.

The third algorithm that it is available is *Cooperative Scheduling*. This algorithm doesn't automatize any context change. This means that the tasks are executed sequentially. They *cooperate* with each other in the sense that they stop or yield to their self to allow the access of computing resources to other tasks.

Finally, in all the algorithms, if two tasks have the same priority, *Round Robin* algorithm is applied, to distribute the execution time of each task. In this case, the tasks are executed during a pre-defined times-slice and, once the time-slice is over, the scheduler switches to next task.

7.2 NuttX

NuttX provides an API for tasks and a scheduler like in FreeRTOS, but in addition it also supports the use of threads inside the tasks.

In NuttX two different entities can be scheduled: tasks and threads. *Tasks* have their own resources, comprised by a private memory range only this task has access permissions. Even though, that each task is isolated, they can interact with each other. Additionally, NuttX implements a concept called application, that tries to be equivalent to a typical OS application. Behind the application, a task that is linked to the application is created and registered at boot time. This way the application can be initialized and executed from the console. Tasks could be of two types: kernel tasks or standard tasks.

The kernel task is equivalent to the Idle task in FreeRTOS but has some differences: It is not a unique task, but is composed of multiple tasks that manages the cleaning of the kernel, the scheduling policy or the power management among many others. This task is implemented in the kernel of each architecture, and in some cases it is possible to modify it. For example, the power management is easy to modify, but others like the scheduling policy is more difficult to change. The standard tasks are used by the user to implement applications.

The second entities are *threads*. A thread is similar to a task. But the main difference is that a thread runs inside of a task, sharing its resources. The threads can collaborate with each other, but they are executed in parallel. As with tasks, scheduling algorithms are also applied to control the thread execution. Additionally, NuttX allows creating child tasks from parent tasks. But we don't have clear if this child task uses parent task reserved memory or if they take it from the available memory of the system.

When a task is created, the following configurations need to be given:

- unique name to the task
- an attached function
- a priority
- a stack size.

As in other RTOSes, the priority is the parameter for the scheduler, giving to higher priority tasks more computational time.

NuttX offers a list of parameters to control a task, which are described in the following folders:

- task
- scheduler

Inside the scheduler folder all NuttX implementation regarding scheduling can be found. These are some of the main options:

- clock: This is a basic tool time handling. It is used to control the time task must run, when to set up timers or used by the watchdog.
- errno: This is a basic debugging option, to debug the tasks and threads. It allows us to know the exactly error that happened.
- group: It allows to create groups of threads and control them at the same time.
- irq: Interruptions between threads an tasks, and the way to handle it.
- mqueue: Message queues to work with in the tasks or threads.
- pthread: In this folder we can find all the options available to work with the threads. For example mutex, allows starting threads or killing threads. The NuttX pthreads are based on POSIX pthreads.
- sched: This folder implements the logic for the scheduling.
- semaphore: Semaphores to control the access to the data between the tasks and threads.
- signal: Implements the same behavior as in Linux. You can pause or kill a task sending an appropriate signal.

- task: In this folder we can find all the options available to create and control tasks.
- Watchdogs: watchdog implementation.

As we can see, NuttX has rather a big quantity of code regarding scheduling and threads/tasks. But we are going to focus on the tools that has equivalence with FreeRTOS. Similar to `TaskDelay` in FreeRTOS, NuttX makes use of `nxsig_sleep` function. This function sets the thread or task in pause for a pre-defined amount of milliseconds. Once you delete a task or thread under NuttX, the `idle` task will also clean the resources used by that thread or task and free it again. Additionally, it is also possible to get and change the priority of a task in run-time.

There are two interesting functions only available in NuttX, called `sched_lock` and `sched_unlock`. The first function blocks the scheduler avoiding a context switch until the unblocking of the scheduler happens, using the `sched_unlock` function.

NuttX implements the the following scheduling algorithms that can be set in `menuconfig` configuration menu:

- FIFO: First task In First task Out. This is a very basic static scheduling algorithm that when it finishes the processing of the task, it executes the next highest priority task.
- Sporadic Schedule: A task using sporadic scheduling continues executing until it blocks or is preempted by a higher-priority thread. Also, the task's priority can oscillate dynamically between a foreground or normal priority and a background or low priority.
- Time Slicing: the used time-slice is defined in NuttX's `menuconfig`. This time is the same for all the threads/tasks.
- Round Robin: This scheduling algorithm is used for tasks with the same priority.

The approach NuttX uses, is Linux oriented. The NuttX developers have tried to implement complex concepts such as signals, threads inside of task and the control of group of threads. But in comparison with FreeRTOS, the main difference resides in the scheduling internal logic. This logic aimes to be complient as much as possible to a general purpose operating system (like Linux), however, being a step ahead in terms of complexity.

7.3 Scheduling abstraction layer

The scheduling abstraction layer(AL) is a piece of code that abstracts scheduling functionality from the RTOSes used in the micro-ROS project. They are composed by the abstraction layer API and the actual implementation in each RTOS, called *adapters*. For more information about this topic, please refer to deliverable number 2.2, *Report on the identified OS-independent abstractions for micro-ROS compatibility*.

The aim of the scheduler abstraction layer is to unify the scheduling mechanisms used under the micro-ROS software stack. This is an ongoing work, where improvements of the AL is expected once we determined the feasibility and benefit of such a software layer. In order to give an intuition of the AL state, the implemented functions and a short explanation of them are now listed.

7.3.1 Creation of a task

```
uros_task_create(char *name, void *fun, uint8_t priority, uint16_t stack_size)
```

It creates a task where the name is given as char vector, sets the attached function, sets a stack size and the priority of the stack.

7.3.2 Delete of a task

```
uros_task_delete(void *thread_id)
```

It deletes the task, given by the id of it.

7.3.3 Pause the execution of a task

```
uros_task_delay(uint16_t time_ms)
```

It delays the execution of a task a determined time measured in milliseconds.

7.3.4 Periodically wakes up

```
uros_task_periodic(uint16_t time_ms)
```

Periodically wakes up a task.

7.3.5 Set/Get priority of the Task

```
uros_task_getpriority(void *thread_id)
uros_task_setpriority(void *thread_id,int8_t priority)
```

Get and sets the priority of a given task.

7.3.6 Get the ID of the Task

```
uros_task_getid()
```

It gets the id of a task.

7.4 Scheduling mechanism extension

Once we have a clear understanding of the RTOS scheduling options and the RTOS abstraction layers, the scheduling mechanisms in the ROS-stack need to be analyzed and taken into consideration to guarantee real-time performance requirements for the application.

The scheduling mechanisms in ROS-stack e.g. for receiving topics or executing cyclic activities, are distributed over several layers: rclc/rclcpp/rclpy (ROS language specific Client Library), RCL (ROS Client Library), RMW (ROS MiddleWare), DDS (Data Distribution Service) and OS (operating system).

In ROS2, a ROS Executor was introduced to coordinate the execution of the callbacks of the nodes of a process. It was implemented in rclcpp and also in rclpy. In the present Executor concept there is no notion of prioritization 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.

Therefore, we have proposed a Callback-group-level Executor (see Deliverable D4.4), which provides, firstly, an API to express real-time requirements on callback level (e.g. priorities) and secondly, a re-designed Executor which adheres to the real-time requirements in its scheduling decisions. Nevertheless, this is only a first step.

Incoming messages (topic subscriptions) of a ROS-node can be configured to be processed by a single thread or by a thread pool. These threads are created by the Executor on rclcpp/rclpy layer. While multi-threading improves fair progress, it could lead to re-ordering of topic evaluation. This is especially cumbersome when analyzing bag-files, because the original execution in the field cannot exactly be re-played do to these non-deterministic scheduling effects.

Further more, an analysis of the interface between RMW to DDS revealed that processing timers had a higher priority than processing incoming topics (subscription) and services. This has the impact that a time-critical topic might be delayed because of executing some activity, which is just time-triggered by a timer.

The complexity of such an implementation regarding scheduling makes it very hard to develop a sound model based on known scheduling theory. Firstly, a clear concept for real-time scheduling across all layers is missing and secondly, on different layers several artifacts need to be re-designed before techniques from well-known scheduling theory can be applied. Currently there is also a publication in preparation, which describes these findings in more detail.

Recently, the limitations of ROS2 regarding hard real-time have been pointed out by Apex.AI [HNS18] at ROScon 2018. They presented Apex.OS as an automotive ROS2 for safety-critical applications, which provides hard real-time, robustness and security as well as certification, which is currently missing in ROS2.

Based on our own analysis regarding real-time and scheduling of the ROS-stack as well as the work presented by Apex.AI, we came to the conclusion, that we need to address the scheduling issues first in the ROS-stack before providing proper scheduling abstractions on the real-time operating system.

Additionally, the RTOS performance and tools should be analyzed deeper, in order to see how all the software stack could be tuned for deterministic response.

Based on this information, the consortium has considered delaying the work regarding scheduling extension. In future, this work would be performed and create the optional deliverable.

[HNS18] Christopher Ho, Sumanth Nirmal, Juan Puablo Samper, Serge Nikulin, Anup Pemmaiah, Dejan Pangercic and Jan Decker, ROS 2 on Autonomous Vehicles, ROSCon, Oct 2018.

8 Conclusion

In this deliverable, we have reviewed common scheduling mechanisms, described the micro-ROS software classification, compared the different scheduling policies offered by various real-time operating systems and described the scheduling algorithms of NuttX and FreeRTOS as the chosen RTOS of the micro-ROS project.

As this deliverable shows, the scheduling mechanisms for achieving real-time and deterministic software response times is not trivial. It is determined by several factors, such as how the scheduling mechanisms are implemented in the ROS2-stack but also how they are mapped to scheduling policies offered in the RTOS. Additionally, the available scheduling algorithms and resource management need to be looked at in detail for real-time robotics application.

In this project, this task is even harder due to the fact that the software stack is distributed over multiple layers or modules and that their are several dependencies between them. This brings the necessity of creating a better vertical software design regarding real-time topics, from bottom-up, so to speak from the RTOS to the the application layer.

In order to have a better intuition of which are the changes and modifications that are required in the software stack, further research and tests need to be performed. This affects to:

- RTOS: The RTOS layer needs to be evaluated. It is required to analyze wether the scheduling algorithms and task features are sufficient to guarantee real-time behavior of micro-ROS applications. Additionally, the suitability of the abstraction layers also needs to be analyzed. It could be that additional abstraction layers could downgrade the overall real-time performance of the RTOS.
- Middleware: Middleware abstractions and middleware implementation need to be analyzed. This could also bring improvements to these layers, to meet real-time capabilities.
- micro-ROS client libraries: rcl, rclc and rclcpp need to be analyzed. These layers might need enhancements to provide real-time capabilities.
- micro-ROS application code: Once the software stack is tested, test-benches need to be performed for the application, in order to test if micro-ROS applications meet their real-time requirements. Additionally, real-time design rules and guidelines should be also released, in order to offer micro-ROS developers guidance to properly develop real-time compliant applications.