D1.5

Overall Architecture Definition Revised

Grant agreement no.	780785
Project acronym	OFERA (micro-ROS)
Project full title	Open Framework for Embedded Robot Applications
Deliverable number	D1.5
Deliverable name	Overall Architecture Definition
Date	June 2019
Dissemination level	Public
Workpackage and task	WP1-Task 1.4
Author	Borja Outerelo Gamarra (eProsima)
Contributors	all partners
Keywords	micro-ROS
Abstract	This document provides a review over the proposed micro-ROS architecture.

Contents

1 Acronyms

Acronym	Explanation
API	Application Programming Interface
CDR	Common Data Representation
CPU	Central Processing Unit
DDS	Data Distribution Service
FIFO	First-In, First-Out
FSM	Finite-State Machine
HAL	Hardware Abstraction Layer
HW	Hardware
IEEE	Institute of Electrical and Electronics Engineers
IDL	Interface Definition Language
IP	Internet Protocol
IRQ	Interrupt Request
MCU	Microcontroller Unit
MPU	Microprocessor Unit
MTU	Maximum Transmission Unit
NTP	Network Time Protocol
OMG	Object Management Group
OS	Operating System
OSS	Open Source Software
QoS	Quality of Service
PTP	Precision Time Protocol
RCL	Ros Client Library
RCLCPP	Ros Client Library CPP
RMW	Ros Middleware
ROS	Robot Operating System
RPC	Remote Procedure Call
RTOS	Real Time Operating System
SHM	System modes and Hierarchy Model
SLAM	Simultaneous Localization and Mapping
TCP	Transmission Control Protocol
UART	Universal Asynchronous Receiver-Transmitter
UAV	Unmanned Aerial Vehicle
UDP	User Datagram Protocol
XRCE	eXtremely Resource Constrained Environments
YAML	YAML Ain't Markup Language
6LowPAN	IPv6 over Low power Wireless Personal Area Networks

2 Introduction

2.1 Summary

In this revised version of the overall architecture definition document, we revisited the initial proposed approach to the micro-ROS architecture, adding the changes and advances done during the project.

This document's starting point is the previous document: D1.4 Overall Architecture Definition – Initial. Taking that document as a base, we started to develop the micro-ROS architectural model. As in many other projects, during development some of the initial architectural design details were not clear enough, valid to our use case, or difficult to reach on the scope of the project, resulting in some architectural design changes. We present those architectural changes in the current document.

In this revised document, we have omitted the review of similar software done in the first part. Apart from that omission, the core structure of the document keeps untouched, incorporating the changes in it.

For architecture definition, we follow the approach taken in Simon Brown publications[1][2] and his C4 model[3]. In this architecture definition, we introduce functional behaviour along with the system specified at different levels of abstraction ("zoom levels"). The architecture definition starts with a context section where we set the scene for this architecture specification. Following the context, we present the primary functions of micro-ROS and what they do. Then we summarise quality attributes of micro-ROS and the constraints imposed to its development. To continue, we introduce the principles driving the design and development of micro-ROS.

As the central part of the architecture document, is the software architecture section, where we show the whole picture and complete it with an explanation of the interfaces of the system. Next follows a code section where we explain our layered code structure and roles of each one regarding the proposed architecture. As the final parts of this core section, we present the infrastructure and the deployment of the micro-ROS platform. Part of the deployment is a table defining profiles and a set of scenarios using those profiles.

As a closing part, we summarise the architectural changes introduced and where to find them in this document.

2.2 Purpose of Document

Architecture is the result of several choices to satisfy goals and constraints. In this document, we introduce goals and constraints we identified and explain the rationale behind our decisions, as information for future developers and users of micro-ROS. This document presents the second iteration of the micro-ROS architecture, showing the changes toward the initial design and the rationale behind them.

This document is intended to be a living document, evolving with the OFERA project, to reflect the experiences other contributors and we make.

At least one last revision already planned:

- Overall Architecture Definition - Final - 30.06.2020

2.3 Partners Involved

Short name	Full Name	Role
eProsima	Proyectos y Sistemas de Mantenimiento SL	Task lead
Bosch	Robert Bosch GmbH	Contributor
ALR	Acutronic Link Robotics AG	Contributor
PIAP	Przemyslowy Instytut Automatyki i Pomiarow PIAP	Verification

2.4 Main Changes

This section presents a summarised list of the most important changes from the previous version of the document:

Change	Section	Short description
Time	Timers and Clocks	Time synchronisation improvements based on experiments.
Real-Time Executors	Real-Time Executors	Real-Time ready executors. A ROS2 executors redesign with real-time capabilities in mind.
Peer-to-peer	micro-ROS peer-to-peer	Peer-to-peer functionality in micro-ROS. Micro XRCE-DDS peer-to-peer details.
Embedded Transform	Embedded Transform (TF)	Embedded Transform library.
Build System	Build System	Traditional ROS2 build system adaptation to a two steps build required for targeting MCUs.
Library layer	Library layer	RCLC layer was removed and replaced by a combination of RCL and modular extensions. Plus, RCLCPP will continue to be supported as a C++-based client library layer.
AL	RTOS Layer	AL are proposed as method of abstracting used RTOS to the upper layers.

3 micro-ROS Architecture

3.1 Context

Nowadays robotic systems are heterogeneously composed by a set of microcontroller-based sensors and actuators attached to a general propose computer. Take a UAV as an example, which is composed by a microcontroller-based Autopilot connected to multiple sensors, or the Bosch Indego lawnmower mixing safety-rated microcontroller with non-safety-rated microcontrollers.

Currently, the Robot Operating System 2 (ROS2), provides high flexibility to the users for extending robotic applications. This extension focuses on the integration of high-power/high-resources computers. Due to ROS2 design, this solution is not suitable for microcontroller-based devices. The following limitations are rooted in the ROS2 design:

- Usage of DDS communications infrastructure. Traditionally, DDS implementations have high resource requirements, do not support sleep cycles and requires considerable large MTUs.
- Implementation details on higher layers. As examples, we can take Node and Executors from ROS. Those ROS concepts highly depend on the ROS client library used. The existing ROS Client libraries are not best suited for microcontrollers.

Due to these core design decisions, the current ROS2 implementation is not capable of running on microcontrollers. This situation push developers to use their custom solutions, losing the benefits of a robust, highly adopted platform, ROS2. micro-ROS address the issue of a missing common platform to integrate microcontrollers in a robotic system. The micro-ROS design extends the existing ROS2 to micro-controllers. micro-ROS uses the same ROS2 concepts and interoperates with it. Integration of existing ROS2 concepts eases micro-ROS inclusion in the ROS2 ecosystem and its adoptions by the existing community.

The following context diagram shows how micro-ROS is related to other software systems and the different users of that ROS2 ecosystem. micro-ROS allows communication between microcontroller based solutions (micro-ROS applications) and general ROS2 solutions (ROS2 applications). In a nutshell, micro-ROS allows microcontrollers applications to communicate with ROS2 applications using client-server architecture, where the clients are in microcontroller based systems and the servers on the ROS2 side.

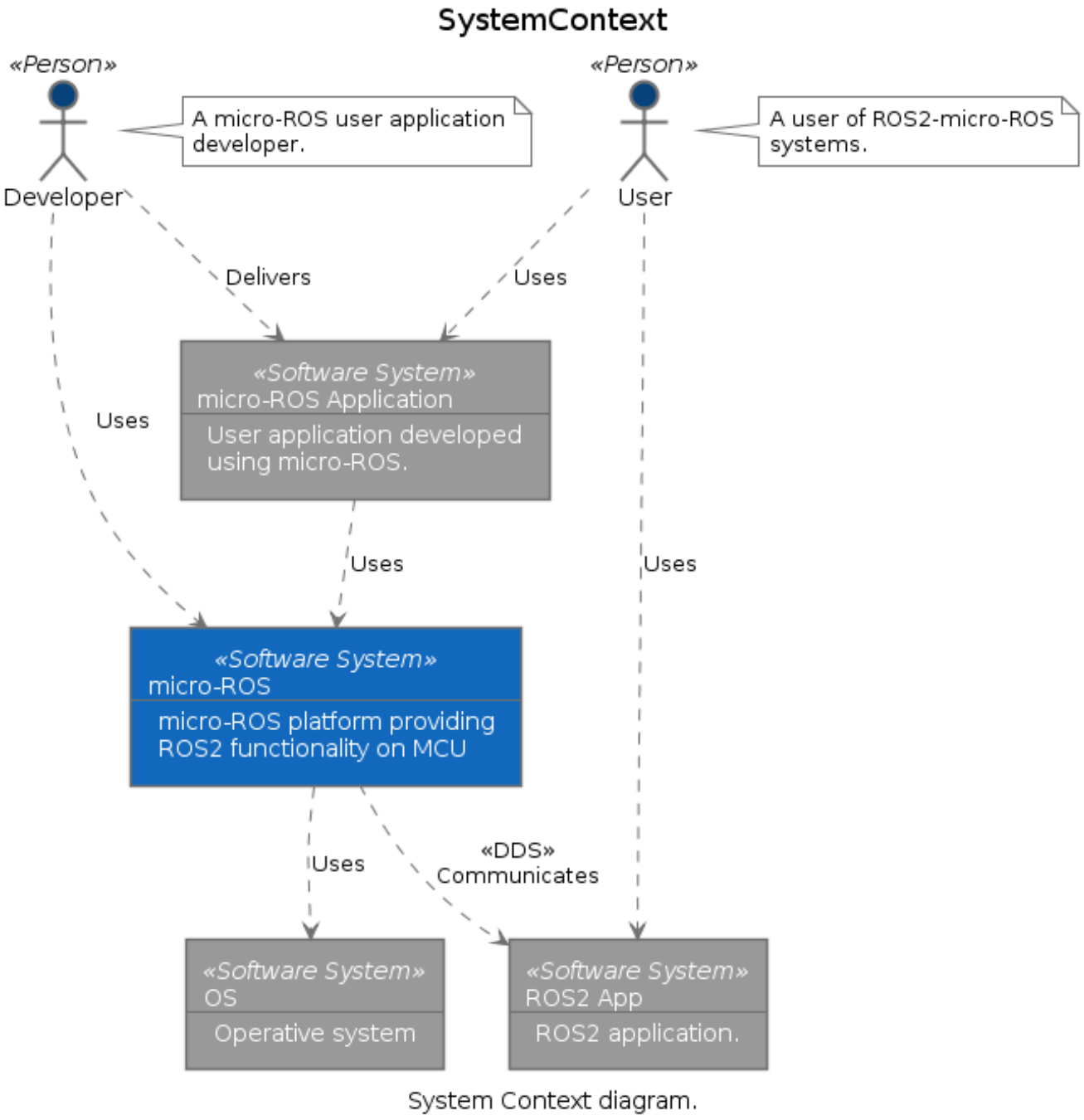


Figure 1: micro-ROS Context

3.1.1 Users

micro-ROS primary target users are Robotic system developers and embedded systems developers. micro-ROS developers could use the provided libraries and tools to develop micro-ROS applications. Developers could deliver those micro-ROS applications on a Robotic platform extending it with microcontroller-based applications.

The other kind of user is not using micro-ROS directly, but using systems developed with the help of

micro-ROS' technologies. This user interacts with both ends of the system, ROS2 applications, and micro-ROS applications.

3.1.2 External Systems

There are external systems integrated with micro-ROS. They are the systems represented by grey boxes on the micro-ROS context diagram.

1. **micro-ROS application**: It is a *User application* developed using micro-ROS and deployed on top of a real-time operative system running on a microcontroller. These applications are Low resource consumers and capable of performing sleep periods. These applications act as decoupled ROS components thanks to the communication provided by micro-ROS technologies.
2. **ROS2 application**: It is a *User application* developed using ROS2 concepts and technologies are deployed on regular computers with no resources restrictions. These ROS2 applications interact with micro-ROS applications thanks to micro-ROS technologies.
3. **OS**: micro-ROS target platforms are microcontrollers which are operated by an RTOS and provides an abstraction over the RTOS used. micro-ROS uses the underlying RTOS functionalities and exposes them to the user.

The usage of the micro-ROS platform allows communication between the two kinds of applications. Regular ROS2 uses DDS as the standard communication mechanism between ROS2 applications. These communications standards are also used on Robotic applications using micro-ROS.

3.2 Functional Overview

The micro-ROS platform takes most of the ROS2 existing functionality into a microcontroller system. This section provides a summary of the micro-ROS exposed functionality.

3.2.1 Node

Nodes, as in ROS2, are the core concept of the micro-ROS platform. micro-ROS applications are made of nodes. These nodes are orchestrated by micro-ROS scheduling.

Nodes are the entry point to users of the platform and they offer them essential features:

- Communication with other micro-ROS nodes and ROS2 nodes using Micro XRCE-DDS middleware.
- Publishing and Subscribing capabilities to Topics.
- Exposing and consuming services.
- Providing timers and timing facilities.
- Having associated configuration parameters.
- Logging capabilities.

Some nodes could have states observable from others. micro-ROS manages these nodes with states and notifies their observers on state changes and status. These states and their transitions follow the same ROS2 defined FSM.

3.2.2 Topic and Messages

Topics are data information that are exchanged between nodes. The former, are characterized by a topic name and an associated type.

micro-ROS topics are defined similarly to the ROS2 ones, using rosidl files. These topic definition files need to be processed to generate code supporting the type they define. From rosidl files, the build system generates type support consisting of the in-memory representation of the topic type. In addition to the in-memory representation, the topic needs code supporting middleware required to serialise/ serialise operations. The *code generation* is provided rosidl files in input and outputs middleware vendor-specific type support code. This way, this type of *support generation* is a process tied to the underlying middleware vendor. As for micro-ROS, the latter generates DDS-XRCE static type support. @[AMX rephrasing needed] rosidl_dds, an existing ROS2 package generates OMG IDL files from the ROS2 definition files and them from the generated IDL an implementation specific package produces the final code that the implementation uses for serialisation/deserialisation of the type. The usage of rosidl in future ROS2 versions (Starting from Dashing Diademata) will be changed to use OMG IDL 4.2 for interfaces definitions. As ROS2 is migrating to IDL4.2, micro-ROS will support IDL4.2.

3.2.3 Publisher and Subscription

Nodes' data is interchanged using publishing and subscribe interfaces provided by the underlying middleware. A node can create and hold multiple publisher and subscriptions. A node is capable of publishing user content as long as the topic is defined in the micro-ROS platform using previous mentioned micro-ROS tools. Node users can create subscriptions to Topics. Then, up to date data on callbacks are received via the preliminarily registered callback. Publications and subscriptions have a series of configurable middleware QoS to fit with micro-ROS applications requirements.

3.2.4 Service Client and Server

Nodes can expose services to be called by other nodes. A *user created services* receives requests and dispatches them via user's callback. In their turn, nodes that consume services and receive responses from the services attended by user callbacks.

3.2.5 Timers and Clocks

micro-ROS provides the user with time-based functionality: timers and clocks.

For timers, micro-ROS nodes provide callback based timers to the user. Due to real-time constraints, these timers follow real-time principles. To comply with these real-time requirement maintaining determinism on timer calls, low-level timer implementations, OS level or hardware timers are desirable.

Apart from timers, micro-ROS allows the user to provide different clocks, so micro-ROS time is measured based on different principles. Clock synchronisation between nodes is possible using custom time sources for clocks and adjust them using different synchronisation algorithms as could be NTP or PTP.

Currently an experimental implementation of PTP, IEE 1588 Precision Time Protocol has been done on top of the software stack. This implementation measures the round trip delay time from master and slave and from slave to master:

- measures clock-offset.
- measures sending delay.

Implementing such a synchronisation mechanism will allow synchronising master and slave clocks. There is still an on-going discussion on the best place to have this protocol implemented. To reduce delays, the closer to the hardware interface the better. Micro XRCE-DDS middleware implementation already has support for time messages. Thus, implementation on the middleware layer instead of application could be done. An NTP implementation has been done. It shall be located where the middleware provides an API to trigger time synchronisation. This time synchronization is entirely customizable by the end-user. This Micro XRCE-DDS time messages were introduced in a correction done to DDS-XRCE standard dated March 2019.

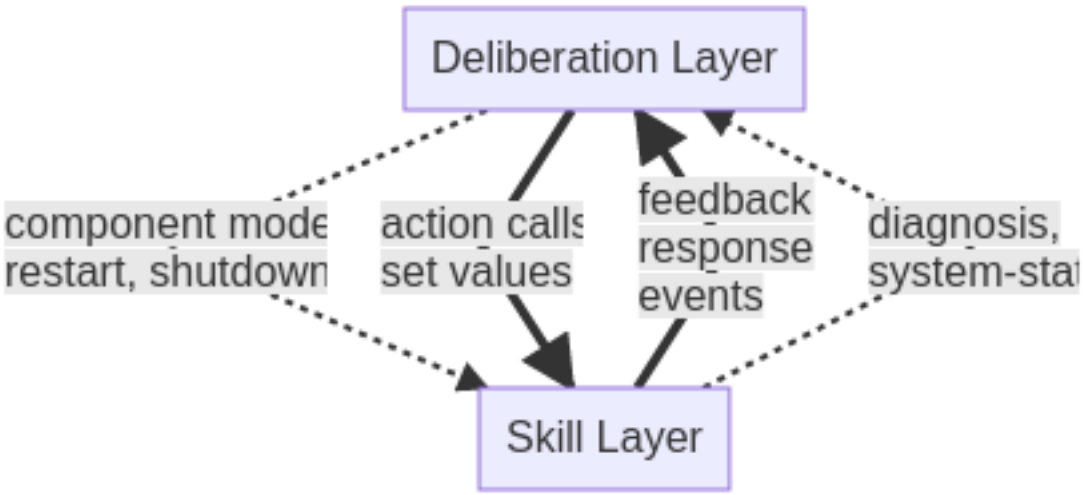
3.2.6 Lifecycle, System Modes and Parameters

Modern robotic software architectures follow a layered approach. The layer with the core algorithms for SLAM, vision-based object recognition, motion planning, etc. is often referred to as *skill layer*. To perform a complex task, these skills are orchestrated by one or more upper layers named *deliberation layer*(*s*). We observed three different but closely interwoven aspects to be handled on the deliberation layer:

1. **Task Handling**: Orchestration of the actual task, the *straight-forward*, *error-free* flow
2. **Contingency Handling**: Handling of task-specific contingencies, e.g., expected retries and failed attempts, obstacles, low battery.
3. **System Error Handling**: Handling of exceptions, e.g., sensor/actuator failures.

The mechanisms being used to orchestrate the skills are service and action calls, re-parameterisations, set values, activating/deactivating of components, etc. We distinguish between *function-oriented calls* to a running skill component (e.g., set values, action queries) and *system-oriented calls* to individual or multiple components (e.g., switching between component modes, restart, shutdown).

Analogously, we distinguish between *function-oriented notifications* from the skill layer in form a feedback on long-running service calls, messages on relevant events in the environment, etc. and *system-oriented notifications* about component failures, hardware errors, etc.



To easy handling of this complex communication, micro-ROS provides abstractions and framework functions for (1.) system runtime configuration and (2.) system error and contingency diagnosis, to reduce the effort for the application developer of designing and implementing the task, contingency and error handling. An example is given in the following:

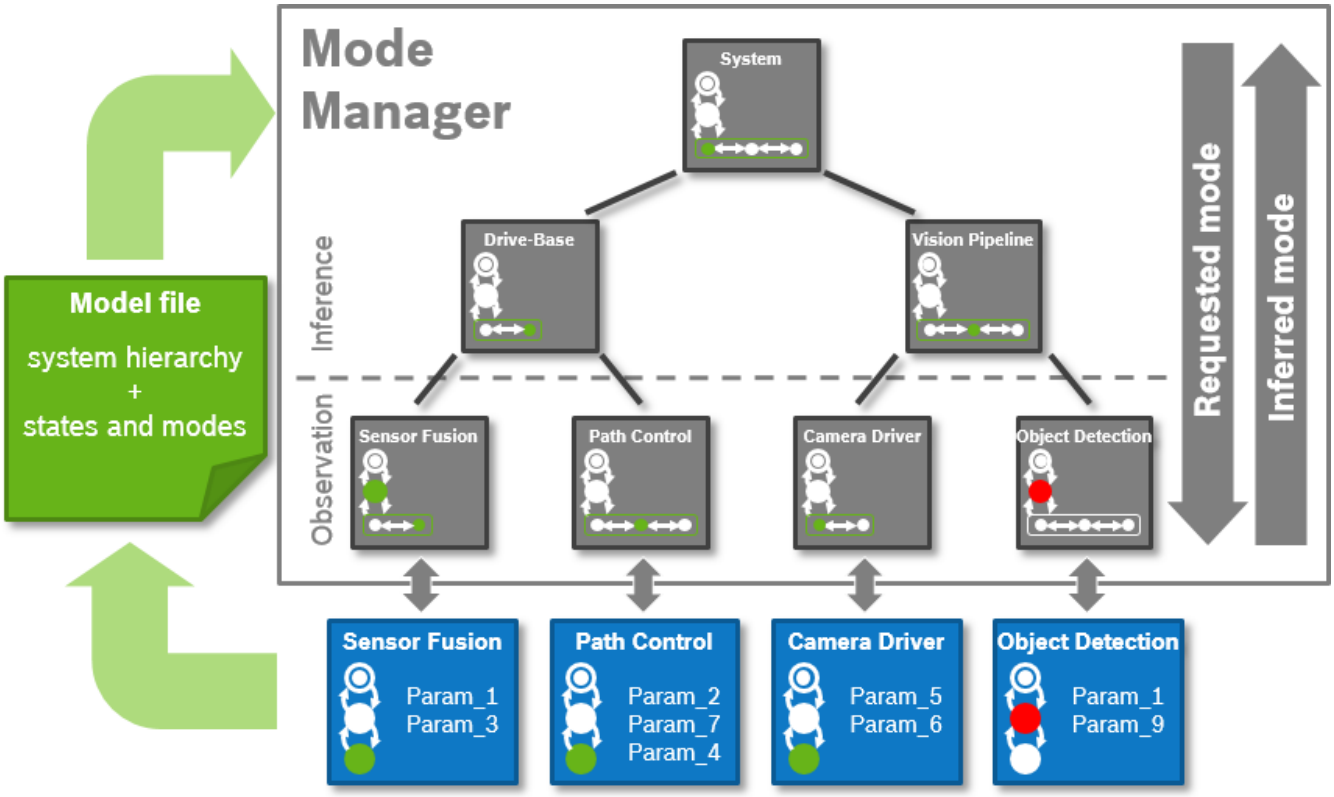


Figure 2: System modes architecture

The lifecycle and system modes management will support different expansion stages:

1. **Mode Management**

- introduces (sub-)systems
- extensions to ROS diagnostics
- introduced system mode manager

2. **Mode Management and Error Handling**

- modelling of system errors and error propagation
- extensions to ROS diagnostics
- extensions to mode manager

3. **Mode Verification and Validation**

- Verification and validation capabilities based on (1.) - (3.)

Please refer to Deliverable D4.8 "Lifecycle and System Modes" for a detailed description. The key elements of the first expansions state are:

1. Extensible concept to specify the runtime states of components, based on the node lifecycle FSM as defined in ROS2 [4].
2. A *system modes and hierarchy model* file (SHM file) that specifies system modes for all hierarchical (sub-)systems and components.
3. A *mode inference* mechanism, that infers (sub-)system states and modes based on the observed states and modes of their components.
4. A *mode manager* component that orchestrates state and mode change request for all (sub-)systems and components for system runtime configuration.

The corresponding functionality is not only be provided for micro-ROS but also for standard ROS 2 since system mode management involves all computing platforms of a robotic system.

3.2.7 Node Scheduling (aka Real-Time Executor)

Predictable execution under given real-time constraints is a crucial requirement for many robotic applications. ROS nodes are the primary interface to the user and need to be scheduled in a predictable way in order to receive and send communications data, update timers or perform logging operations. In ROS 2, such scheduling requirements are addressed by the Executor concept.

3.2.7.1 Analysis of the ROS 2 Executor

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 incoming messages in the DDS queue. If there are such messages, the ROS 2 Executor executes them by calling the corresponding callbacks with the message objects.

While ROS 1 processed the messages and timer events basically in a FIFO manner, the current ROS 2 Executor uses a complex schema mixing prioritization and round-robin. We recently analyzed and modeled this scheme in detail, cf. CasiniBlaß19. We discovered that timers are preferred over all other events from the DDS middleware. The implication is that in a high-load situation, only pending timers will be processed while incoming DDS-events will be delayed or starved. If no timer events are present, the messages from the different topics are processed in a round-robin fashion and not in the arrival order.

However, even a FIFO strategy would make it difficult to determine time bounds on the total execution time, which are necessary for the verification of safety and real-time requirements. Also, the Executor does not provide an interface 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.

3.2.7.2 Callback-group-level Executor

As a first step, we have proposed a callback-group-level Executor, which addresses some of these deficits. 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:

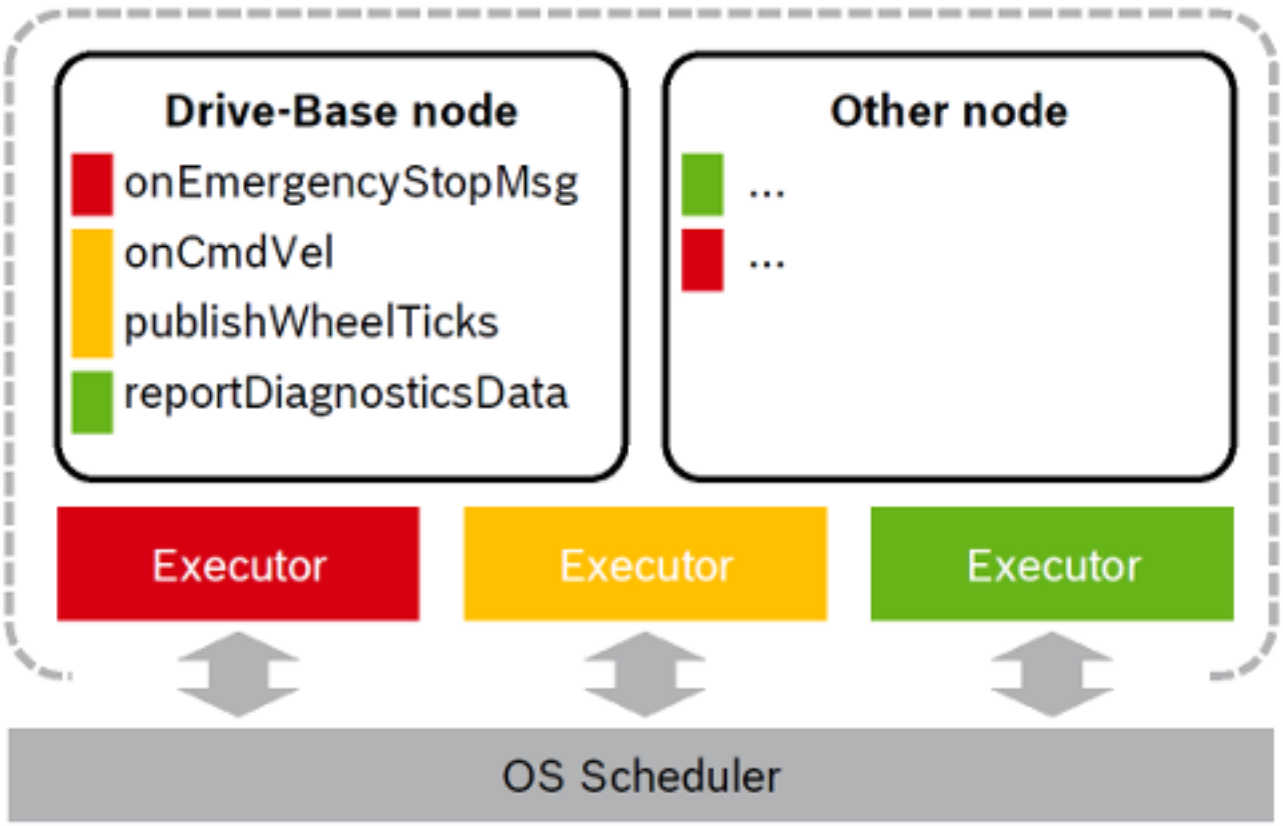


Figure 3: Sample system with two nodes and three Callback-group-level Executors in one process

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.

Meta Executor. 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, binds them to underlying kernel threads, assigns them priorities, 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.

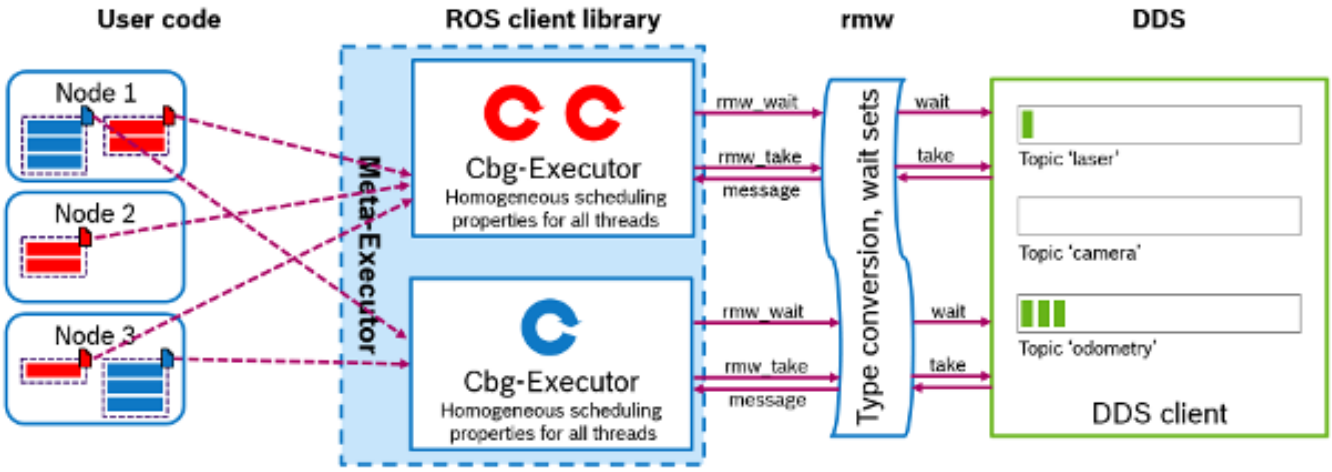


Figure 4: Illustration of the Meta-Executor concept

Further details on the Callback-group-level Executor implementation and the Meta Executor concept can be found in https://micro-ros.github.io/real-time_executor/

3.2.7.3 RCL-Executor

In a second step, we have implemented the RCL-Executor package, which provides different scheduling strategies. Currently, two scheduling policies are implemented: static-order and priority-based scheduling.

In static-order-scheduling, all events, including timers, are processed in a pre-defined order. In case different events (timers, subscriptions, etc.) are available at the DDS queue, then they are all processed in the pre-defined static order.

In priority-based scheduling, a priority is defined for each event. If multiple events are available at the DDS queue, then only the event with the highest priority is processed.

3.2.7.3.1 User API

The RCL-Executor is a library written in C based on the ROS client support library (RCL).

The RCL-Executor provides the following user interface:

- Initialization
  - Total number of handles
  - Scheduling policy (*STATIC*, *PRIORITY*)
- Configuration
  - rcle_executor_add_subscription()
  - rcle_executor_add_subscription_prio()
  - rcle_executor_add_timer()
  - rcle_executor_add_subscription_prio()
- Running
  - rcle_executor_spin_once()
  - rcle_executor_spin()

For the static-order scheduling, the user initializes the RCL-Executor with *STATIC* as scheduling policy and the total number of handles, i.e. types of events. Then, the handles are added the the RCL-Executor. The order of these method-calls defines the static-order for the scheduling. Processing the corresponding events is started by one of the *spin*-functions.

For priority-based scheduling, the user initializes the RCL-Executor with *PRIORITY* as scheduling policy and the total number of handles. The specific priority of each handle is specified in the respective *rcle_executor_add-*-prio* function.

As resources are very constrained on micro-controllers, specific attention has been paid to the memory allocation: Dynamic memory is only allocated during the initialization phase to reserve memory for all handles. Later on, no memory is allocated while scheduling the corresponding event.

The RCL-Executor and examples can be found in the repository rcl_executor.

3.2.8 Logging

Nodes provide named logger utilities where the user define their custom log writers controlling logs destinations.

3.2.9 Graph Introspection

micro-ROS and ROS2 nodes form a so-called system graph which is accessible with introspection tools. Introspection provides the user with services to query the current system graph.

3.2.10 Nodes Discovery

There are to ways to made Node discovery: statically or dynamically.

- Static discovery is achieved by middleware configuration. In this discovery, micro-ROS nodes are configured, so they know the existence of micro-ROS Agent beforehand.

- Dynamic discovery is a mechanism provided by middleware. The middleware used to allow nodes to be added on the fly to the system as they find and connect automatically with an existing Agent.

All discovery methods have the peculiarity that micro-ROS applications discover micro-ROS Agents, no the other way round. The discovery is a mechanism provided by the underlying middleware used, in this case, Micro XRCE-DDS.

3.2.11 micro-ROS Profiles

Along with the configuration mechanism mentioned, micro-ROS adds a profile system. This profile system allows the user to select how the micro-ROS platform behaves, so they are capable of configuring the platform accordingly with their needs and requirements. Multiple profiles exist, and they could be combined as the user wants to build time. These micro-ROS profiles reduce or extend the functionality of the platform.

3.2.12 micro-ROS Peer-to-peer

One of the particularities and main differences of micro-ROS compared to ROS2 is the client-server architecture. This architecture "force" the user to have a device big enough, resources-wise, to hold a micro-ROS Agent. However, DDS-XRCE standard address a peer-to-peer use case was, communication between extreme resource constrained devices is done using a new kind of lightweight agent. This peer-to-peer communication could be used at the micro-ROS level.

3.2.12.1 Peer-to-peer in Micro XRCE-DDS

Peer-to-peer communication could be defined as direct communication between applications hosted on a different system or in the same system.

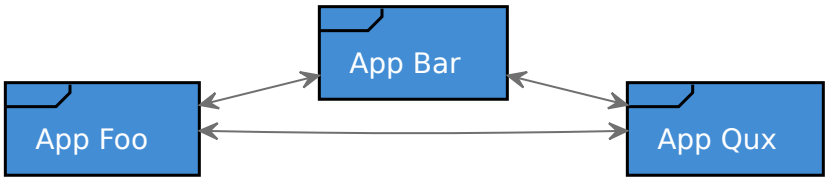


Figure 5: Peer-to-peer concept

The DDS-XRCE Standard[5] (see Sec. 10.5) addresses the peer-to-peer communication with the following architecture:

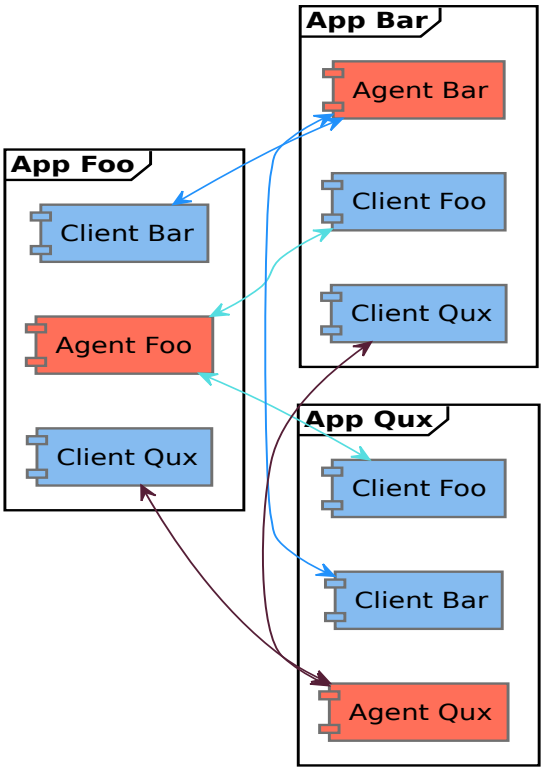


Figure 6: DDS-XRCE peer-to-peer architecture

However, this solution raises the following questions:

- The **internal management of topic**, that is:
  - How should an Agent act when a Client request for writing or reading a Topic?
  - What is the Clients' role? What do they publish or subscribe?
- The **internal communication between Agents and Clients**, that is:
  - How do Agents and Clients communicate inside the same application?

To answer the previous questions, the following architecture is proposed.

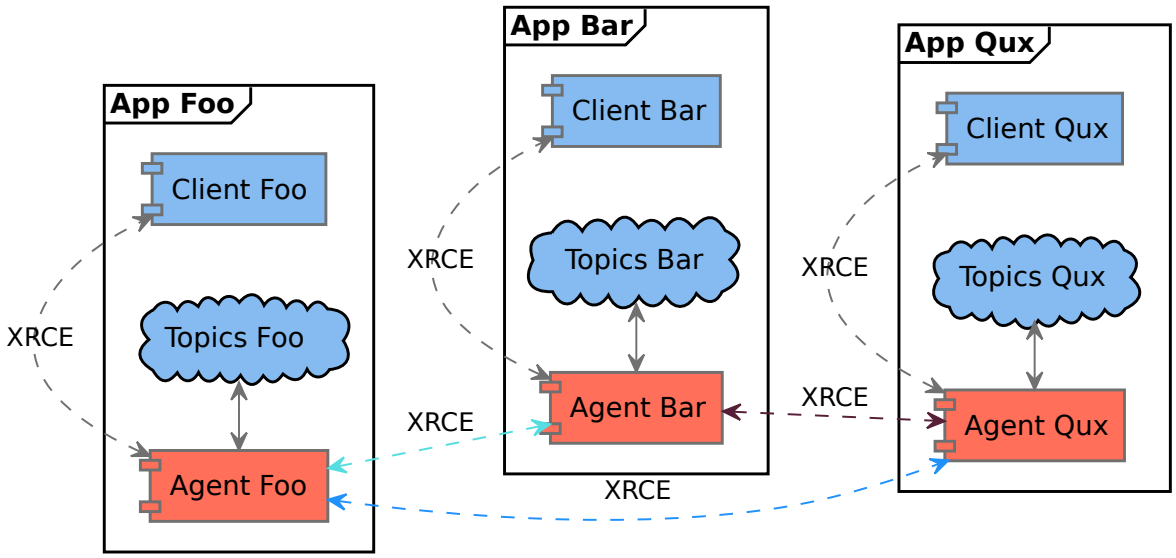


Figure 7: Proposed peer-to-peer architecture

In this architecture, an application is composed of a set of Clients and an Agent. The Clients communicate with the Agent through an **Inter-Thread Communication** offered by an **Agent's API**, while the Agent centralizes the application's data (pub/sub of Topics by Clients) through a **Centralized-Data Middleware**.

Regarding the communication between applications, the Agents are in charge of establishing communication between them using the XRCE protocol. Each Agent will establish a connection with each one of the other Agents using an **Internal-Client** which will subscribe to the Topics that its Agent manages.

Summarising, for being able to have a peer-to-peer communication, the following three elements are necessary: Centralized-Data-Middleware, Inter-Thread Communication and Internal-Client. In our case, all these elements are integrated into Micro XRCE-DDS implementation. Despite that DDS-XRCE standard address peer-to-peer communication, it is quite open to implementation, so any other implementation may provide a different approach.

3.2.13 Client Library

Apart from the core functionality, micro-ROS will extend the ROS2 Client Library functionality. In this section, we present some of the extensions micro-ROS does to the Client Library.

3.2.13.1 Embedded Transform (TF)

The TF transform graph, with its support for both a temporal history, and distributed information sources, has been a novel tool for robotics frameworks when it was released in 2008. Functionally, it is based in scene graph concepts known from computer graphics [6], but these only rarely offer distribution, and did not offer temporal histories at all (mainly, because this is not needed for frame-based rendering applications like in computer graphics). Distributed scene graphs have become more widely

available also in computer graphics. In robotics, work by de Laet et al. [7] has extended transforms graphs to also contain twist (i.e., angular motion) information, and to provide more compile-time error checking. This is currently not integrated with distribution mechanisms, but could be used on a single system. One persistent issue with transform graphs has been their resource use. ROS TF works through replicated copies of the entire transform tree at every node that uses it, and is implemented through unicast TCP connections between nodes. In systems with many dynamic parts, this has sometimes been called the TF firehose, because of the large stream of incoming messages. micro-ROS will go beyond this state-of-the-art by running the dynamic transform tree in an embedded device, while keeping resource use to a minimum based on an analysis of the spatial and temporal details actually necessary. Further, enabling real-time queries even in the face of concurrent updates through integration will be realized through integration with the micro-ROS real-time executor. It is also planned to integrate the embedded TF will with the node lifecycle to achieve further power-savings

3.3 Quality Attributes

This section provides information about the desired quality attributes of the micro-ROS platform.

3.3.1 Availability

micro-ROS should support systems composed by nodes with aggressive sleep cycles where nodes should wake up and be able to communicate with other nodes without the need of reconfiguration of micro-ROS Agent.

3.3.2 Extensibility

micro-ROS should be extensible with new packages the same fashion as ROS2 does.

Users should be able to add new topic types.

3.3.3 Monitoring and Management

micro-ROS should provide tools for monitoring and manage the system. One of those tools should manage middleware configurations and profiles also monitoring packages should be available to control nodes behaviour.

3.3.4 Interoperability

micro-ROS nodes should be interoperable with other ROS2 nodes, with nodes running on different hardware and different RTOS and with nodes using micro-ROS platform with different underlying middleware (As long as they use the same standard, DDS-XRCE).

3.4 Constraints

This section provides information about the constraints imposed on the development of the micro-ROS platform.

3.4.1 License

Since micro-ROS platform release is an OSS, under a commercially friendly license, it imposes a constraint. So, the consortium needs to make sure they hold all the necessary rights to license micro-ROS with such OSS license and to satisfy the license requirements of possible existing dependencies.

3.4.2 Time

As there are already scheduled three software deliverables, there is a time constraint to finish the development of the platform. These three delivering dates are 31.12.2018, 31.12.2019 and 31.12.2020. Consequently, The final date constraint is 31.12.2020.

3.4.3 Protocols

The ROS community adopted DDS, as communication standard since ROS2 thus, if micro-ROS provide a method to use the same DDS protocol in microcontrollers, it will achieve ROS2 native interoperability.

micro-ROS can provide DDS communication to low resource devices even though they do not use DDS protocol directly; it uses a recently approved standard, DDS-XRCE for that task. DDS-XRCE standard provides means to use DDS communications from within low resource devices; this protocol follows client-server architecture where the server acts as a bridge between its clients and DDS. The protocol works with client and server interchanging DDS-XRCE messages. In micro-ROS we use Micro XRCE-DDS as default DDS-XRCE implementation and core part of micro-ROS communications and ROS2 interoperability enabler.

3.4.4 Target Deployment Platform

micro-ROS deployment targets are systems based on microcontrollers where a RTOS is operating. For this project, the document *D2.1 Report on the reference hardware development platforms.* defines the target platforms. A common characteristic of the kind of platforms we target is that they are platforms with relatively low resources and mainly low power consumption.

3.4.5 Real Time

micro-ROS leading target platforms are microcontrollers operated by a RTOS. This means that micro-ROS should not break real-time principles:

- Determinism.
- Responsiveness.
- User control.
- Reliability
- Fail safe.

3.4.6 Memory Usage

Real-time constraint indirectly imposes a memory usage constraint as it is usually a source of failures on following real-time principles.

One of the main concerns and source of issues is dynamic memory usage, so dynamic memory is discouraged on real-time systems due to the indeterminism added by allocation methods and memory available on the constrained environments micro-ROS is used.

3.4.7 ROS2 Integration

The micro-ROS main target is to bring ROS2 capabilities to microcontrollers imposing an integration constrain where micro-ROS works integrated with an existing ROS2 system. ROS2 integration is achieved using the same standard messages over DDS and DDS-XRCE also, and integration of tools is desirable, meaning that micro-ROS should integrate with current existing ROS2 packages ecosystem.

3.4.8 Development Methodology

micro-ROS development should follow a development methodology defined by the Consortium. This methodology defines coding standards, tests methodologies, software integration and release models to follow. Also, code reviews and documentation standardisation should be addressed.

3.5 Principles

This section provides information about the principles adopted for the development of the micro-ROS platform.

3.5.1 Package Components and Layers

This architecture provides a "package by component" view of the micro-ROS platform. We choose this view as explaining components grouped by functionality is more straightforward and offers a more clear picture than arranging them by layers. Even though, the code follows a layering layout where each of the functional components spam over multiple abstraction layers, these are later explained in this document.

The micro-ROS components have:

- Concrete functional target.
- Cover, to some extent, original ROS2 abstraction concepts.
- User interface (micro-ROS application developer interface).
- Could be isolated for modular deployment and testing.

3.5.2 Modularity

To facilitate application in diverse scenarios, we emphasise a modular approach. Modularity should go beyond layering because this enables gradual and partial adoption into existing products.

For example, a middle layer such as serialisation should be usable without a particular lower transport layer. Similarly, functionalities such as the embedded transform library or the FIWARE context broker integration might be interesting, and furthering the goals of micro-ROS, even without the rest of micro-ROS.

3.6 Software Architecture

3.6.1 Containers

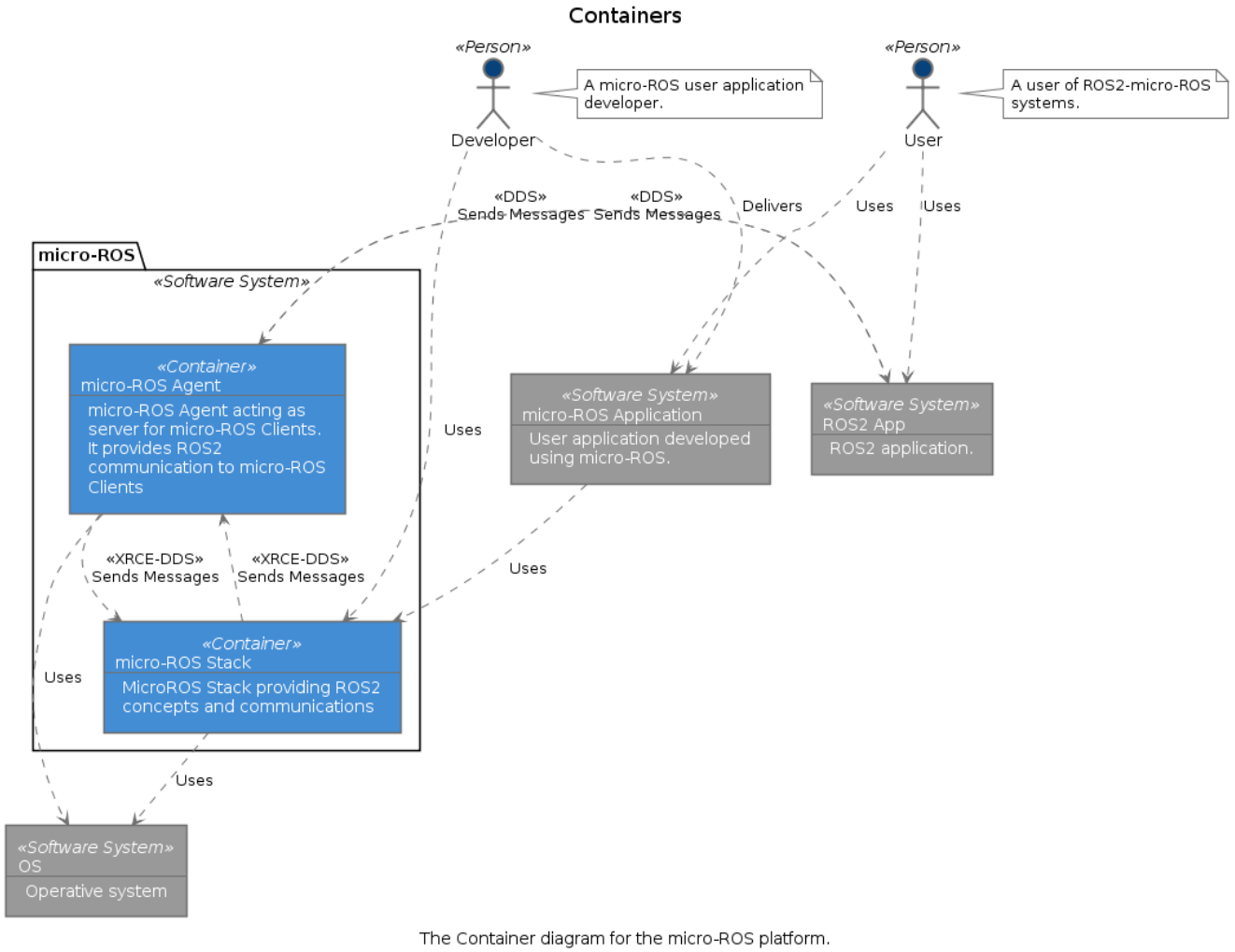


Figure 8: micro-ROS Containers

As stated before, micro-ROS follows a server-client architecture where a general purpose computer contains a server, and the microcontroller based systems contain the client applications. On that line, we have two essential parts of the architecture, a micro-ROS Agent and the micro-ROS Stack.

In the diagram:

- micro-ROS Agent: It is a C++11 server acting as the single point of access for the micro-ROS application to the DDS network shared with ROS2 applications.
- micro-ROS Stack: a group of C99 implemented packages that provide ROS2 concepts to its client micro-ROS application. This group of packages organise in a layered architecture. Each layer has particular functions, one layer deals with high-level concepts as nodes and executors and provides developer API, other layer deals with the communications with micro-ROS Agent, and a third one provides enough OS and hardware abstractions to avoid library clients to worry about platform details. These layers are defined later in this document.

The interface between a micro-ROS Stack and micro-ROS server uses the standard DDS-XRCE communication protocol, one of the many OMG specifications, which provides DDS similar interface to the micro-ROS Stack. It provides DDS object management interfaces as well as publish/subscribe; these interfaces are available to micro-ROS thanks to the middleware implementation used here, Micro XRCE-DDS which architecture is out of the scope of this document.

micro-ROS Stack, a part of hiding the middleware implementation, hides its users from details on OS interfaces and hardware. The micro-ROS Stack is intended to run over an RTOS on limited resources devices.

3.6.2 Components - micro-ROS Stack

The following diagram shows the micro-ROS Stack components from a functional point of view.

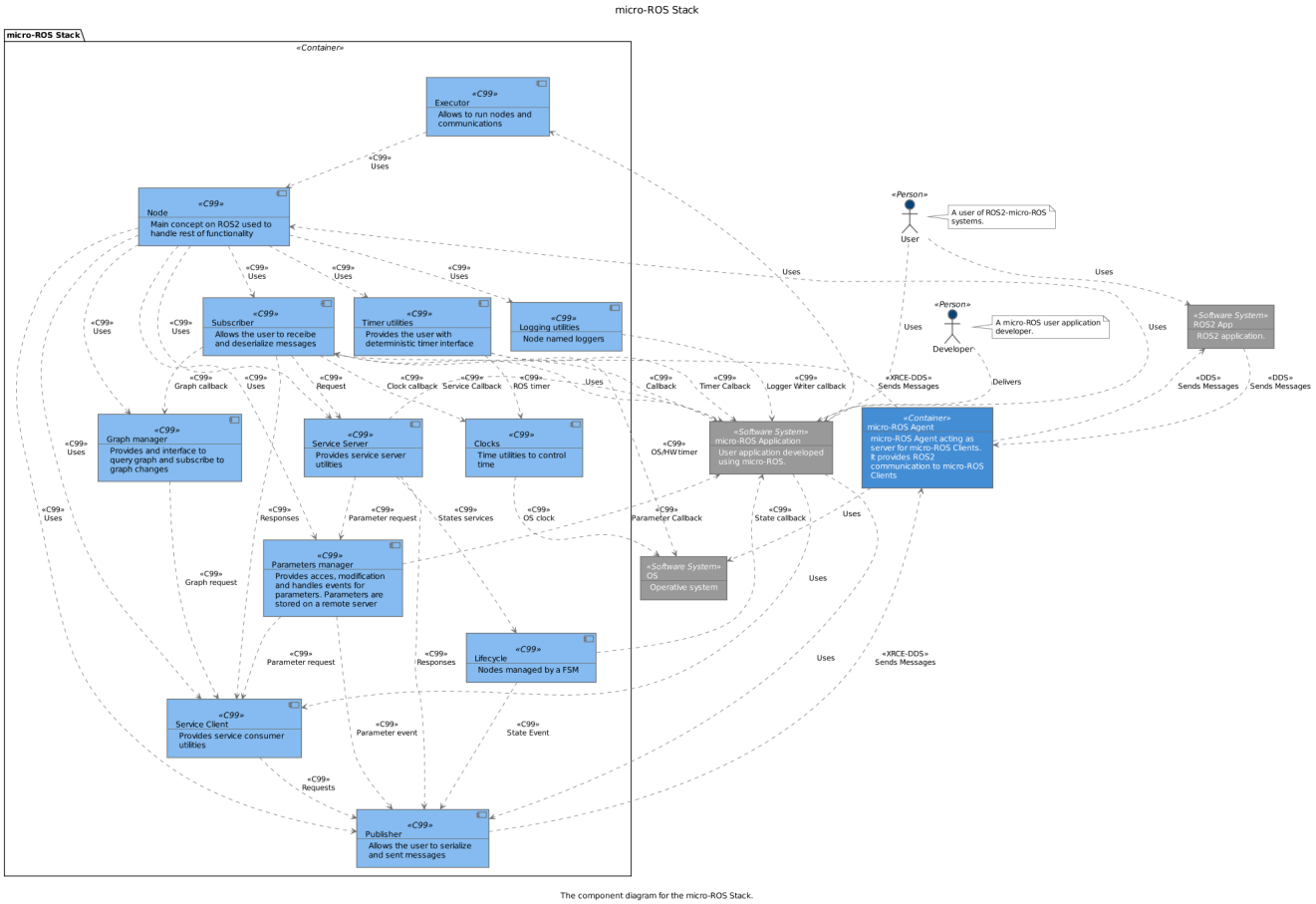


Figure 9: micro-ROS Stack all components

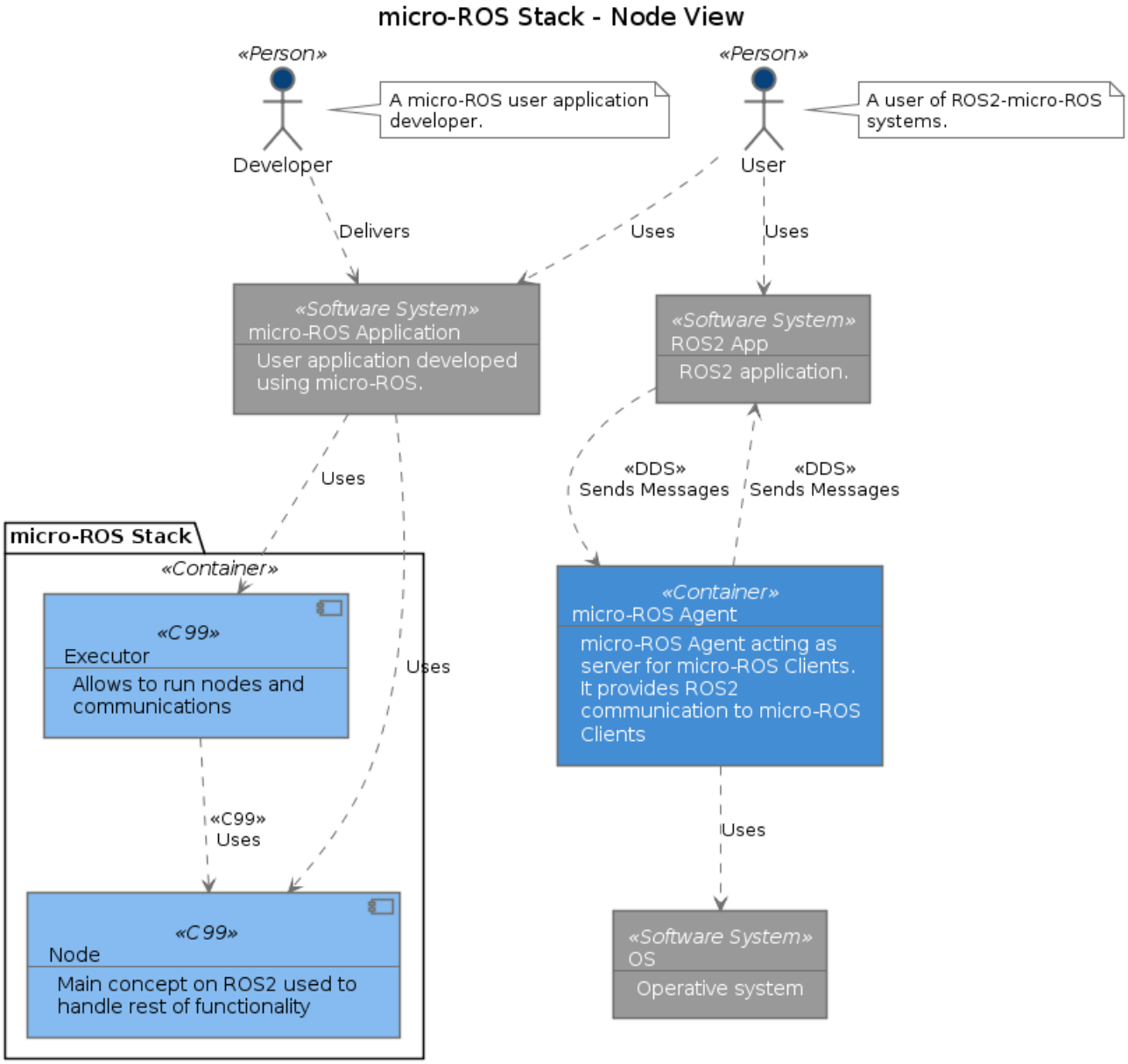


Figure 10: micro-ROS Stack node components

These are each component:

Node: The central concept of micro-ROS platform and the entry point of the micro-ROS Stack, constituting the basic building block for micro-ROS applications allowing the users to have access to all the other components. The users can associate components with the nodes, so when the executor runs the node, its contained components get updated. For communicating with other nodes in the system, micro-ROS nodes or ROS2 nodes, this component uses communication components. Nodes expose and consume services and could have configuration parameters associated with it.

Executor: Node execution triggers nodes and pushes components updates. In a micro-ROS environment,

there is only one kind of executor, single thread sequential executor. The executor runs all its nodes updating all its timers, publications, subscriptions, attending to service calls and triggering the needed events.

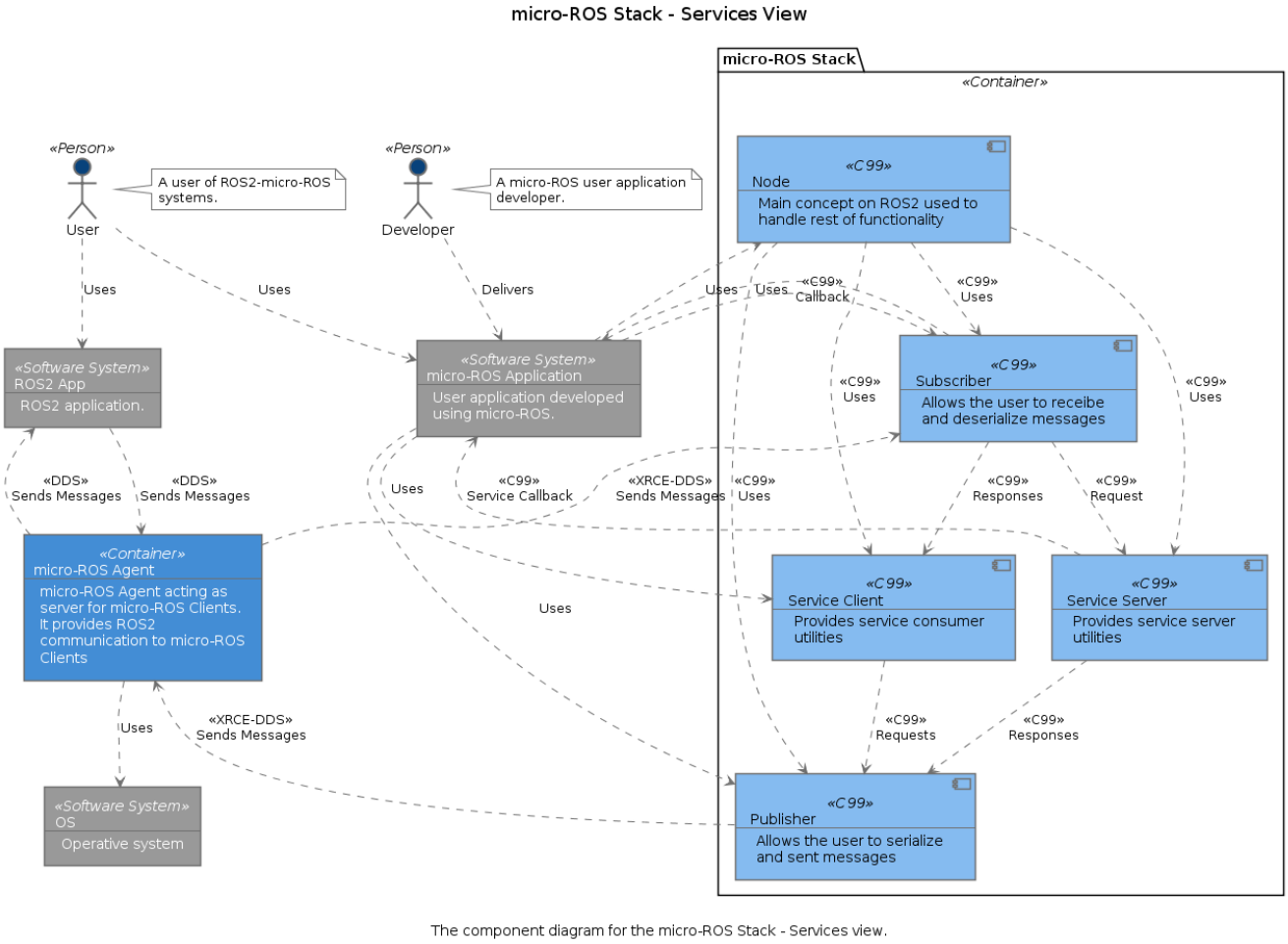


Figure 11: micro-ROS Stack services components

Publisher: One of the core functionality of micro-ROS is to provide communication with existing nodes; this is achieved, in part, using publishers. Publisher allows writing data to DDS global data space. This data is exposed in DDS, by the micro-ROS Agent on behalf of the publishing micro-ROS node which handles the data to micro-ROS Agent using DDS-XRCE protocol. Once in DDS space, all other nodes have access to the published data. This component is, along with subscriber, the only one with direct communication with micro-ROS Agent and constitute the basis for all the other components functionality. Publishers could communicate with other nodes within local space as could be nodes within the same process or same MCU and also can communicate directly with other micro-ROS nodes without the need to reach DDS space; this will consist on peer to peer communication which is provided by the middleware underneath.

Subscriber: This component is the other part of the core functionality of micro-ROS, allowing the reception of data coming from publishers from other nodes. This data is exposed in DDS, by ROS2 nodes or by a micro-ROS Agent on behalf of other micro-ROS nodes and it is read through micro-ROS Agent

using DDS-XRCE protocol which ends passing it to the subscriber. Again almost all the other components build on top of this one. Subscribers could communicate with other nodes within local space as nodes within the same process or same MCU. Subscribers could communicate with other nodes within local space as could be nodes within the same process or same MCU and also can communicate directly with other micro-ROS nodes without the need to reach DDS space; this will consist on peer to peer communication which is provided by the middleware underneath.

Service_server: micro-ROS nodes can create services servers which exposes a service callback to other nodes. This service implementation is a composition of a subscription and a publisher, receiving requests and sending responses, respectively.

Service_client: on the other end of service, it is always a service client. micro-ROS nodes use other nodes services thorough service clients who are composed by, a publisher sending user requests and a subscription receiving service call responses.

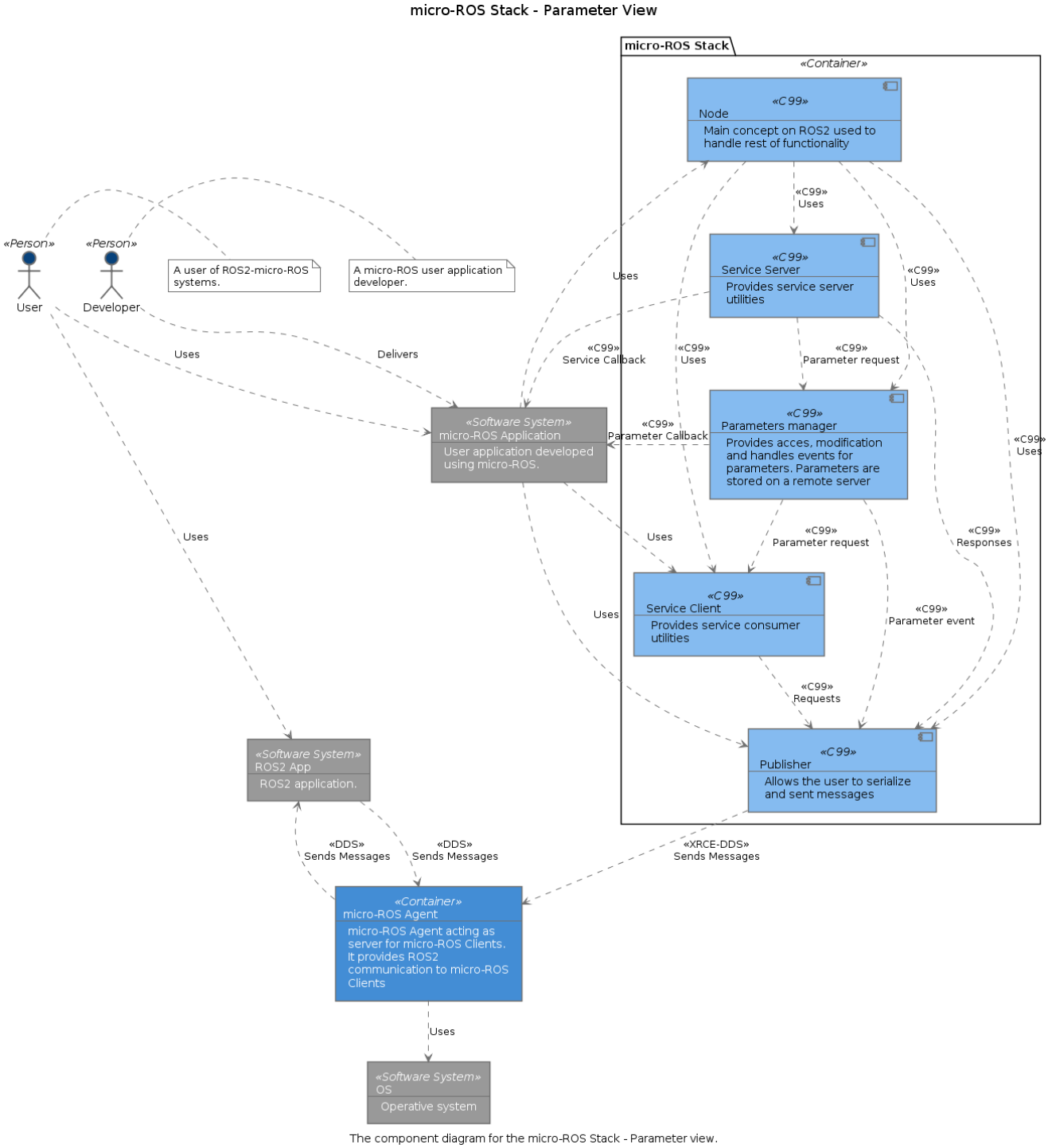


Figure 12: micro-ROS Stack parameters components

Parameters Manager: This component has the responsibility of the storage and handle of nodes specific parameters connecting directly with the parameter server residing on the micro-ROS Agent. Parameter manager acts as a cache to that parameter server on micro-ROS Agent performing transparent user synchronisation between them. Users can use the parameter manager interface to add, change and query node parameters.

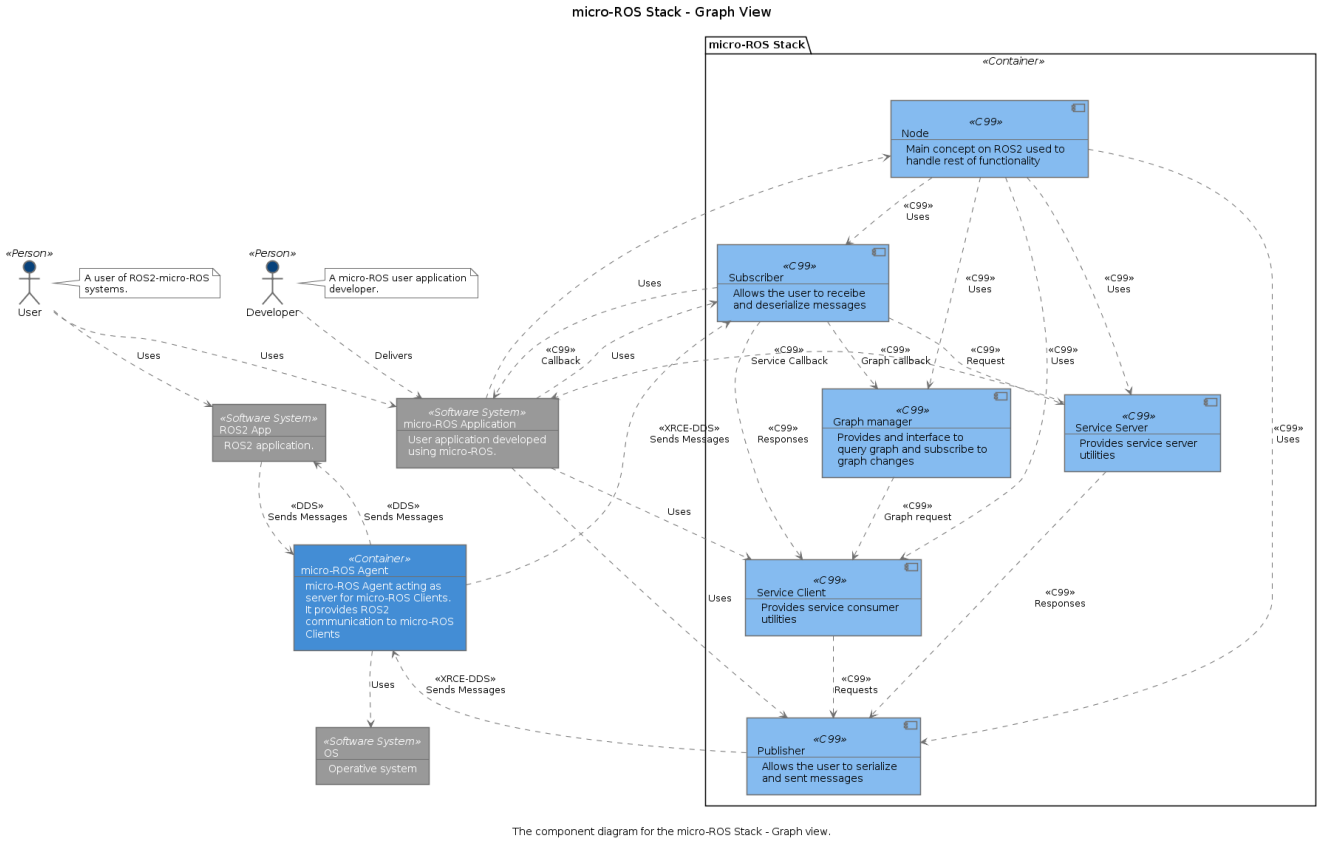


Figure 13: micro-ROS Stack graph components

Graph Manager: This component is responsible for providing needed tools for the introspection of the node graph, which reflects current nodes and entities available on the whole system, including ROS2 nodes and ROS2 created objects. Graph manager as parameters manager uses the same client-server architecture where graph manager queries on micro-ROS Agent graph server. Users of graph manager can subscribe to graph changes updates which the graph server triggers on micro-ROS Agent once a change in node graph is detected.

Mode Manager: The mode manager is a ROS node that accepts an SHM file (see above) as command line parameter. It parses the SHM file and creates the according services, publishers, and subscribers to manage the system, its components, and its modes.

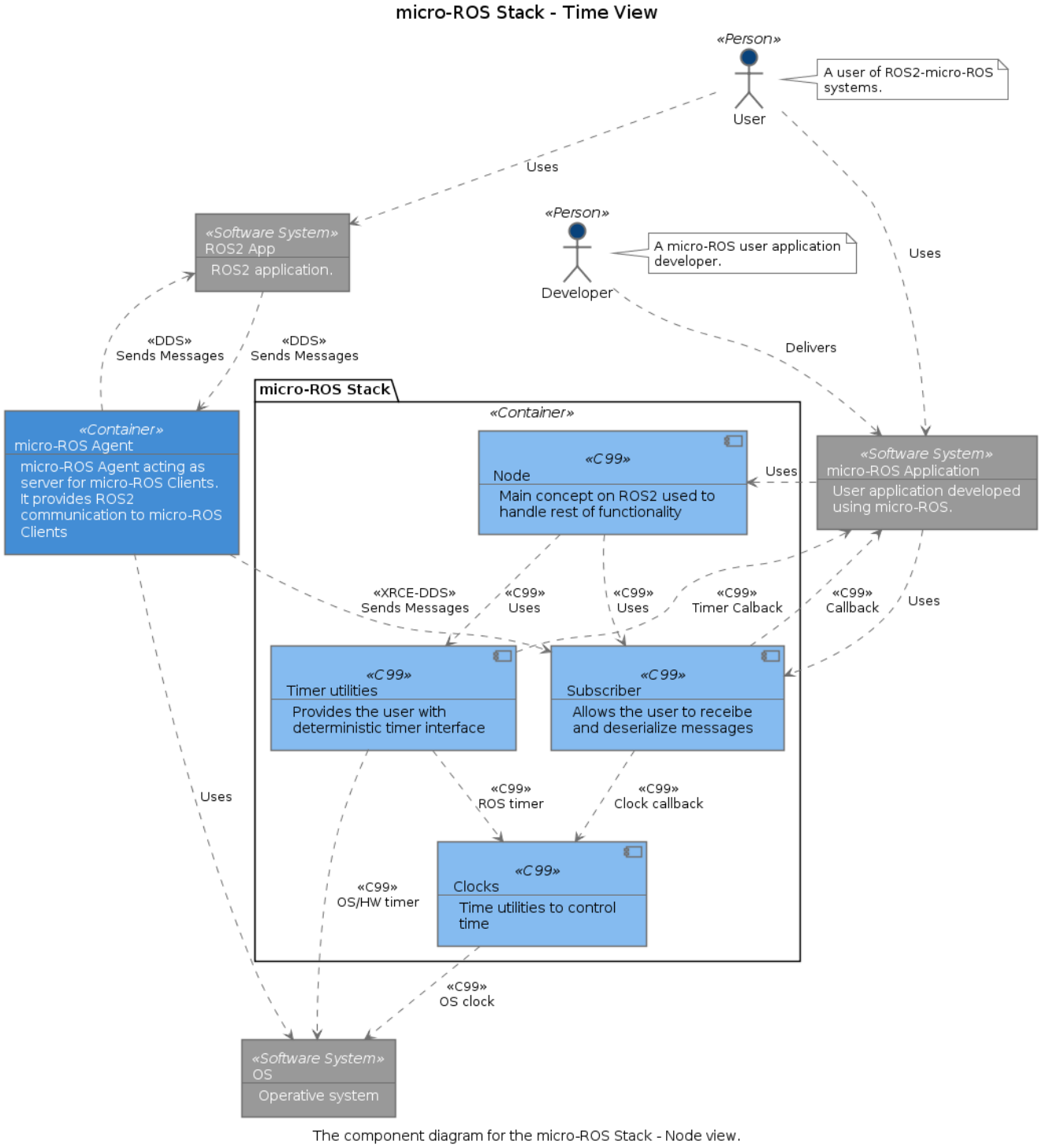


Figure 14: micro-ROS Stack time components

Timers: micro-ROS provides a different kind of timer abstractions to be chosen by the user accordingly to their requirements.

Among these timers, there are OS and HW timers and also a micro-ROS specific timer. This micro-ROS timer is implemented based on a time difference, calculated according to a micro-ROS provided clock.

As almost all timers do, when a timer expires, it calls the user-provided callback.

Clocks: micro-ROS provides a way to set how nodes measure time. This time measure is done based on a time source. Different time sources are available: system clock, steady clock or micro-ROS clock. The micro-ROS clock takes the time to read from a time server that could be running on a different node. This micro-ROS clock allows to have full control over the time, pause, stop, go forward, go back, speed up. However, the use of micro-ROS clock could break real-time deadlines and determinism. This clock and time source system allow node clock synchronisation between nodes, and it is done using a defined time server and a custom time source.

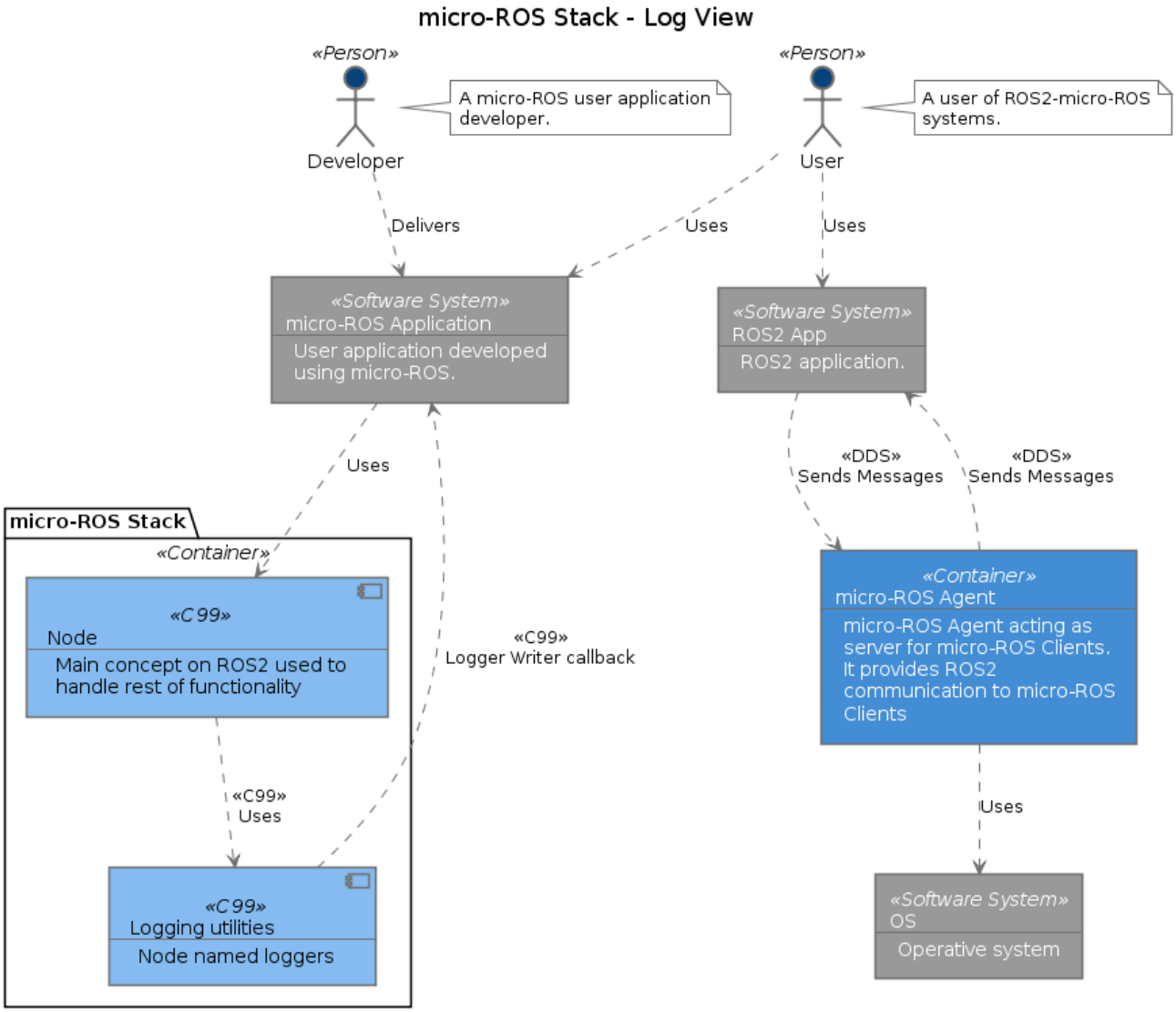


Figure 15: micro-ROS Stack log components

Logging utilities: There is a logger per each node which writes logging info thanks to the use of logger writers. This logger writer allows logging using different mechanisms. For example, a DDS topic publication could be the target of a micro-ROS logging writer. The application defines the logger writers

depending on its purpose and requirements.

3.6.3 Components - micro-ROS Agent

The following diagram shows the micro-ROS Agent components.

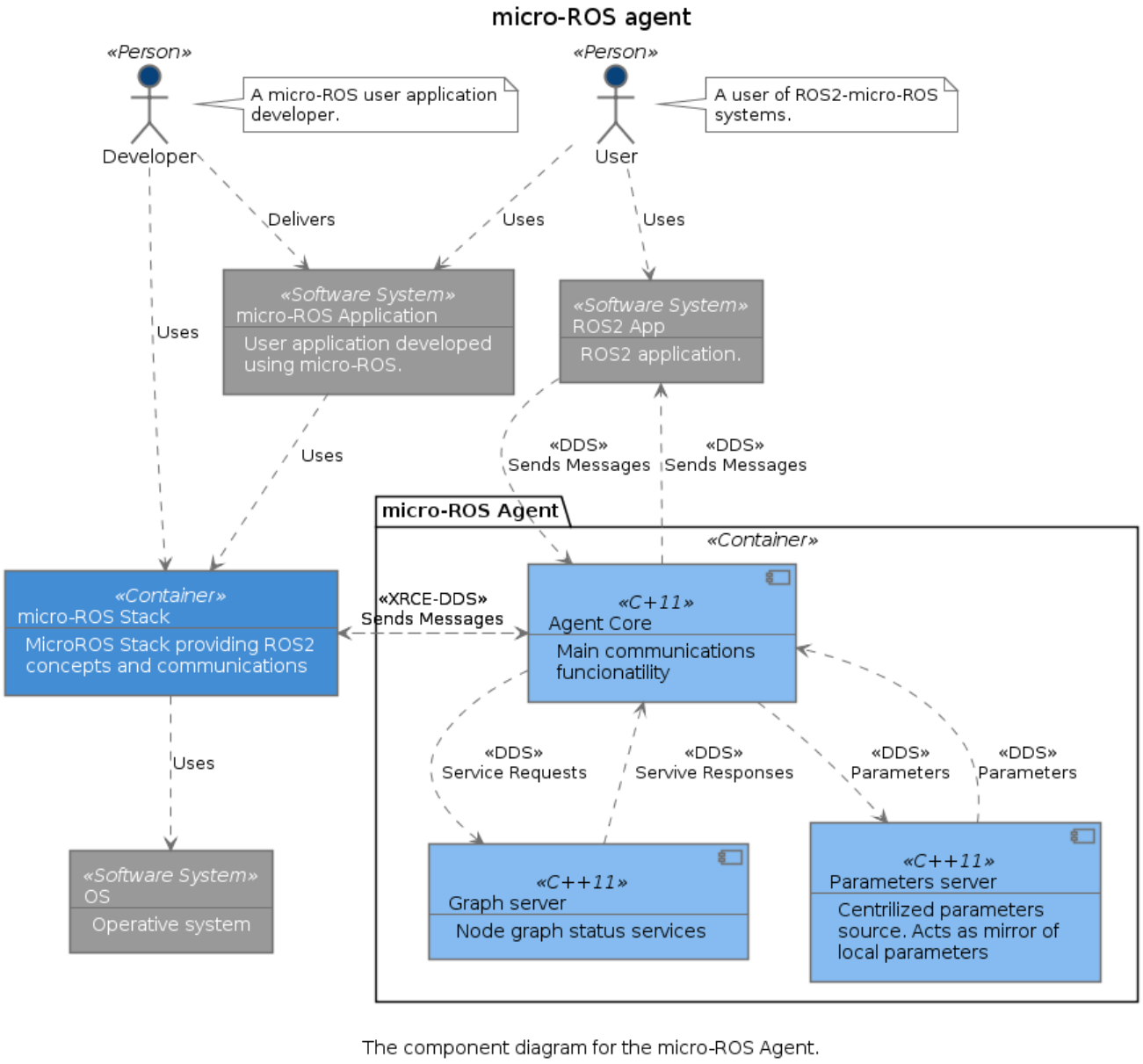


Figure 16: micro-ROS Agent components

Agent core: this component provides micro-ROS Agent main functionality providing the mechanism to communicate DDS with micro-ROS Stack. An agent core could hold any number of micro-ROS Stack connections keeping the state of micro-ROS clients, freeing them of that duty, allowing them to, for example, follow sleep cycles.

Parameter server: micro-ROS nodes use parameters for configurations in a client-server architecture. This parameter server provides those nodes of their configurations, storing parameters per node and providing interfaces to access and modify them. This parameter server is as wholly decoupled part from micro-ROS Agent core which provides high deployment flexibility, for example allowing the parameter server to be on a wholly separated node.

Graph server: micro-ROS Agent graph server provides introspection tools to micro-ROS Clients. As a client-server architecture, these introspection tools allow clients to query the system graph. Part of the system graph this server provides is queries on topic names and types, services names and types, node names, and publishers and subscribers counts.

3.7 External Interfaces

3.7.1 Node Interface

Node provides the entry point to the users and interfaces over the other components:

- **Node with micro-ROS application**: Nodes are the building blocks for micro-ROS applications. These applications can create nodes. The application can create other kinds of micro-ROS entities using this created nodes:

  ```
  * Publishers.
  * Subscriptions.
  * Service clients.
  * Service servers.
  * Parameters manager.
  * Graph manager.
  * Timing Services and clocks.
  * Logger.
  ```

  Along with creation, micro-ROS Node interface allows the use of other components. When the micro-ROS scheduler run node update, nodes process all communications and components the nodes have updating them. During this update, nodes call registered user callbacks if needed.

- **Node with publisher and subscriptions**: Node allows the creation of publishers and subscriptions, storing and handling them. Nodes trigger publications and subscriptions callbacks when executors update them.

- **Node with parameter manager**: Nodes provide the user with an interface to its parameter manager to be able to query and modify specific node parameters.

- **Node with services**: Node provides the creation of services servers and clients. Nodes expose user-created services and use user callbacks when services request are detected and also handle service consumption for the user, and it sends user requests.

- **Node with timing services**: Nodes provide an interface to create timers associated with them. Nodes updates micro-ROS controlled timers (no HW or OS) and triggered their callbacks if needed.

- **Node with logging utility**: Nodes have an associated logger providing an interface with them, allowing the use of different log levels and destinations.
- **Node with graph manager**: Nodes provide the user with introspection tools which query services and subscriptions with the graph manager.

3.7.2 Publisher and Subscribers

We group these two components interface under the same category. Publishers and subscribers interface with micro-ROS agents and on the other end with user code.

- **Publisher with micro-ROS Agent**: Publishers send user messages over topics; this publication is achieved using DDS-XRCE protocol between micro-ROS Stack and micro-ROS Agent. This way, a publisher can send a message over a DDS topic using micro-ROS Agent as a bridge between micro-ROS Stack inbound message using XRCE-DSS and DDS world. In this communication, data is serialised using DDS-XRCE defined serialisation protocol, CDR. Publishers, using Micro XRCE-DDS middleware, send CDR serialised DDS-XRCE to micro-ROS Agent, who publish CDR serialised DDS message for the rest of system nodes.
- **Subscription with micro-ROS Agent**: Subscribers receive up to date data from the micro-ROS Agent in the same fashion publishers send data, using DDS-XRCE. micro-ROS Agent sends data back to subscribers if they previously have requested that data, this send back is triggered during subscription owner node update. micro-ROS Agent sends DDS-XRCE message containing the user relevant topic message which is CDR serialised, so micro-ROS deserialise it upon reception.
- **Subscription with micro-ROS Application**: Once the micro-ROS scheduler executes a node subscription, if any new message is there, the subscription notifies the user, making use of a previously configured callback.

3.7.3 Service, Server and Client

micro-ROS offers an RPC interface. Service servers and clients interfaces this mechanism following traditional RPC request/response pattern.

- **Server with subscriber**: The user can create micro-ROS service servers, which generate a subscription to the service request topic.
- **Server with micro-ROS application**: Upon reception of a message in a service subscription, micro-ROS trigger the user service implementation. This user implementation processes all the input and generates a response for the client.
- **Server with publisher**: Along with a subscriber, a micro-ROS service server creates a dedicated publisher. This dedicated publisher pushes all the responses generated on user service implementation back to the service client.
- **Client with micro-ROS application**: The interface between this to elements is a two ways interface. Server-client users send requests to a service server, and they receive the responses. Users receive server responses by calls to callbacks they previously provided.

- **Client with publisher**: micro-ROS clients integrate a publisher of service requests internally. Users trigger this publication by a user call in which they provide a callback. Such callback handles the response once it arrives.
- **Client with subscriber**: service requests receive responses on a service responses topic. A client of a service subscribe to this topic and handle the responses calling to user's callbacks.

Service client interface with the micro-ROS application could be either asynchronous or synchronous. Upon asynchronous implementation, the application should decide and configure the number of requests waiting for responses. Synchronous responses handling should be made trying to adjust as max as possible to the real-time requirements of the application.

3.7.4 Parameters Manager

Parameters manager interfaces with multiple components:

- **Parameter manager with micro-ROS application**: micro-ROS users can subscribe to parameters changes triggered by external changes on parameters. These changes are changes on the node parameter manager or alterations propagated from parameter server on the micro-ROS Agent. Whatever the source of the change, the user gets noticed of these changes by a call to their callback. Parameter manager also provides users interface to add, remove, edit or query parameters.
- **Parameter manager with publisher**: Parameter manager uses publishers to send notifications on parameters changes. A change of parameters triggers an event publication.
- **Parameters with service client**: Parameter manager provides the user methods to call parameters services from other nodes. This interface allows the parameter manager to consume other parameter managers or parameter server services. This interface, as a regular micro-ROS service client, ends up using a publication and a subscription to services topics.
- **Parameters with service server**: Parameters manager expose different methods to access, modify and query parameters from other nodes. The interface to access parameters from other nodes uses micro-ROS services.

3.7.5 Graph Manager

Graph manager interfaces with node and with graph server stored in micro-ROS Agent.

- **Graph manager with micro-ROS application**: micro-ROS users can query the status of the system node graph. Graph manager provides the users with information on the different graph actors: nodes, publisher/subscribers, services and topics.
- **Graph manager with service_client**: The graph manager uses services provided by micro-ROS Agent. The graph server exposes these services on micro-ROS Agent.
- **Graph manager with subscriptions**: The user can subscribe to graph changes on the graph server. This subscription is provided locally by the graph manager, and it uses callbacks to notify users of graph events.

3.7.6 Timers and Clocks Interfaces

Timers and clocks interfaces with mainly external software and hardware systems.

- **Timers with micro-ROS application**: The user can create timers and provide them with a callback method to be called once the timer expires. Timers could be different depending on the needs. There are micro-ROS timers, RTOS timers or hardware timers.
- **Timers with OS**: By default, timers try to use OS provided timers. Doing so makes sure timers follow the real-time scheduling of each OS.
- **Timers with Hardware**: Timers implementation could be done directly with hardware technologies, hardware timers. Implementing hardware timer is only done when the OS does not support timers. This implementation is dependent on the hardware platform using IRQ to handle these timers.
- **Clock with Subscriber**: Clock main functionality is to provide time provided by a time source. When the time source is chosen to be a micro-ROS clock, time source is a based on a time server from another node which publishes time updates over DDS. Nodes subscribed to the time topic use it as the time source.
- **Clock with OS**: Clock abstractions could use this interface to access to OS clocks.

3.7.7 Executor

Executor unique interfaces are with the micro-ROS application and with node. The interface with the micro-ROS application allows the user to add, remove and query nodes within the executor. Once the user has added all the desired nodes, they can trigger a spin method to update all the containing nodes. Spin methods use the interface with nodes. This node side interface allows the executor to update communications, trigger callbacks and update node timers.

RTOS abstraction

In commonly used software stacks, like Micro XRCE-DDS, the middleware accesses functions of the operating system directly. Such functions typically include scheduling, memory management and communication. This dependability is hindering porting micro-ROS to different hardware platforms which usually come with their real-time operating system. To improve the portability and usability of micro-ROS, an RTOS abstraction layer is introduced, as shown in section RTOS Layer.

RTOS Syntax Encapsulation

The RTOS abstraction shall encapsulate the RTOS functions and shall provide generic access methods for the higher application layer for

- Scheduling (create a task, set priority, stop task).
- Memory management (memory allocation).
- Communication (e.g. TCP/IP, serial other protocols).
- Power management (sleep or wake up routines).
- Hardware timer use.

This could be implemented by two different approaches a) a code generator or by b) an API. In the code generator approach, low-level RTOS calls will be generated for middleware access. RTOS specific syntax is hidden in the code generator. No changes or extensions of the RTOS is necessary.

In the API approach, a POSIX-like API is expected by the middleware and needs to be provided by the RTOS. For POSIX-like OS, this may be simple, however, the effort might be more considerable for non-POSIX like OS, like micrium-OS.

Smart Scheduling

The usability on application level is enhanced through such RTOS abstractions, not only because RTOS-specific syntax is hidden, but most importantly by separating requirements from implementation detail. The user shall define performance application requirements, such as criticality, deadline and rate. The RTOS abstraction layer shall analyse these requirements and shall assign the appropriate scheduling parameters offered by the operating system. For example, based on the criticality level of an application and all other application requirements, an appropriate task priority shall be selected.

3.7.8 Lifecycle and System Modes

The interfaces exposed by nodes concerning the node lifecycle are:

- **Lifecycle sub-states input:** A list of sub-states of the standard *active* state is read for each node from a *system modes and hierarchy model* file (SHM file) at start-up.
- **Lifecycle publisher** Changes of the lifecycle state of a node are published on a standard topic.
- **Lifecycle query services** This service interface allows querying available states, current state and available transitions.
- **Lifecycle transition service** Also, each node provides a service to trigger a transition changing the node state.

The interfaces exposed by the mode manager are:

- **System model input** At start-up, a model of the application-specific system hierarchy in the form of sub-systems and nodes and modes on the system and sub-systems level is read from the SHM file.
- **System mode publisher** Mode changes are published on a standard topic.
- **Mode query services** This service interface allows to query available (sub-)system modes and available transitions.
- **Mode change services** Nodes (in particular from the executive/deliberation layer) may request mode changes via this interface.

3.7.9 Logging Utilities

Logging interfaces depend highly on logging writer implementation.

- **Logging utilities with micro-ROS application**: Logger triggers a call to the user defined log writer callback.

3.7.10 Agent Core

Agent core is the central piece of the micro-ROS Agent. It has multiple interfaces:

- **Agent core with micro-ROS stack**: Agent core sends and receives data from micro-ROS Stack using DDS-XRCE protocol.
- **Agent core with ROS2 application**: Agent core sends and receives data from other ROS2 nodes using the DDS protocol.
- **Agent core with parameters server**: Agent core sends and receives data from parameters servers using DDS protocol. External ROS2 applications could act as parameter server.
- **Agent core with graph server**: Agent core sends and receives data from graph servers using the DDS protocol. Same as the parameter server, an external ROS2 applications could act as parameter server.

3.7.11 Parameter Server

Parameter server provides services and events interfaces:

- **Parameter server with agent_core**: Parameter server uses DDS to provide access to parameters to the micro-ROS client using Agent core as an enabler.

3.7.12 Graph Server

This component provides interfaces that make introspection possible:

- **Graph server with agent core** Graph server provides introspection service. This service server, upon reception of an introspection operation, queries the graph maintained in the agent core and then it sends the response using agent core communications.

3.8 Code

3.8.1 micro-ROS Stack

micro-ROS Stack code is structured in a layered fashion, as shown in the following diagram.

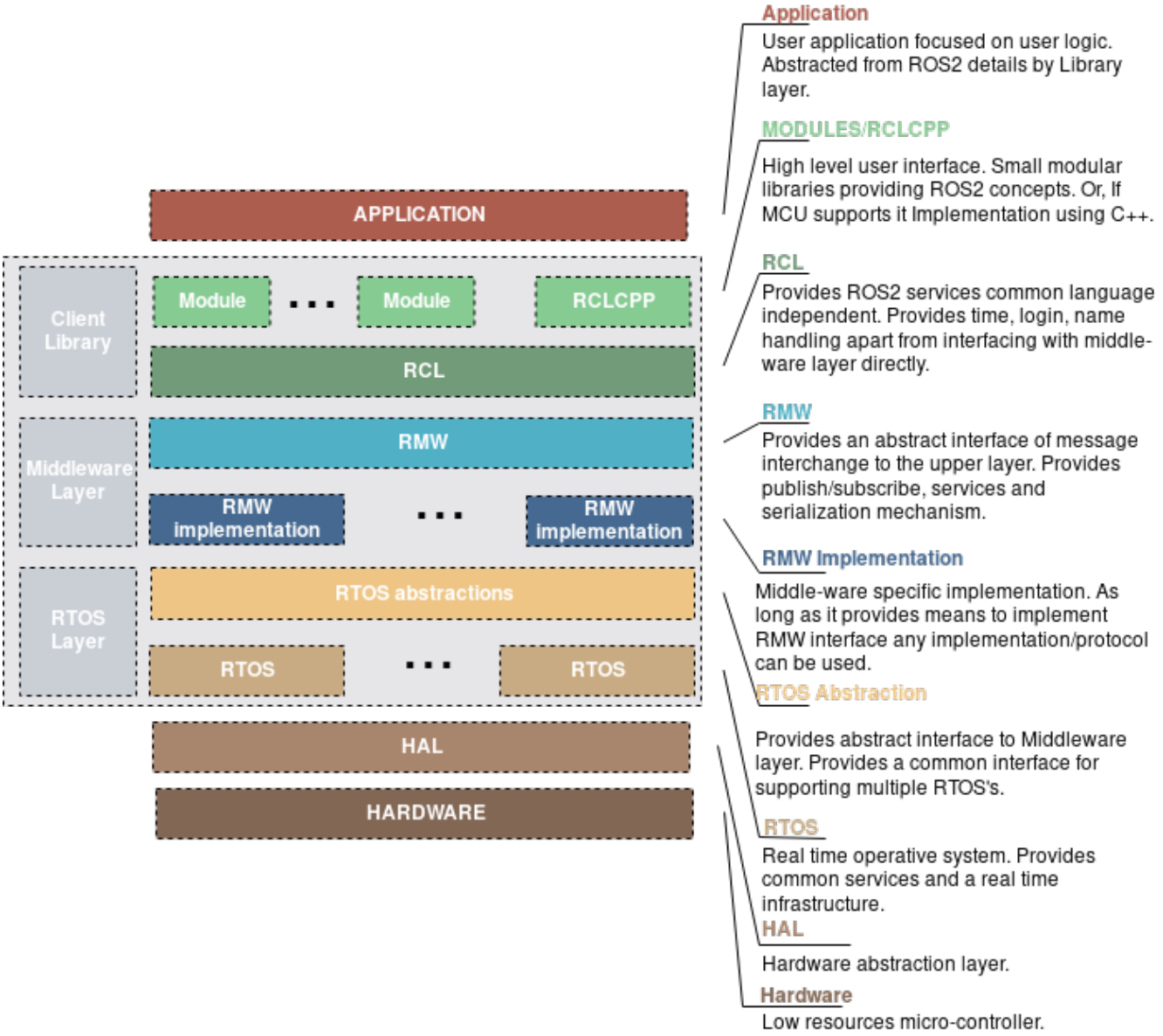


Figure 17: micro-ROS client

3.8.1.1 RTOS Layer

A micro-ROS client runs on top of a real-time operative system. By default, we use NuttX RTOS. One of the advantages of using NuttX is that mainly provides a POSIX-like interface, which makes easy to port code from Linux aimed source-code. As micro-ROS layers aim to be exchangeable, for example, changing the used middleware, the same applies to the RTOS.

But exchanging effortlessly the RTOS is not a trivial task. This is due to the fact that it provides many resources to the upper layers, such as transport interfaces, clock resources, power management or scheduling routines.

In order to make RTOS exchangeable, we came up with the idea of using abstraction layers (AL) for each of these resources. Thanks to these abstraction layers, every RTOS that is properly adapted and

abstracted with the AL could be used seamlessly under the micro-ROS stack. With purpose in mind, we identified and analyzed the next AL in the deliverable 2.2 "Report on the identified OS-independent abstractions":

- Peripheral AL: which handles peripheral access.
- Power AL: which handles power management routines.
- Scheduling AL: which gives access to the primitives of RTOS scheduler.
- Timer AL: which gives access to hardware timers of the RTOS under use.
- Time AL: which gives access to the basic time source RTOS is using (identified after deliverable writing).

Once these ALs are properly analyzed, checking that performance-wise do not affect the behavior of the RTOS, and implemented, the middleware could make use of these micro-ROS AL to build upon, and not needing to have specific implementations for each RTOS that micro-ROS uses.

The impact of those AL it is not only limited to the middleware implementation, as it will offer resources to the micro-ROS top layers. For example, the scheduling primitives could allow users to set the policies that they desire at ROS client library level for executing properly their application.

3.8.1.2 Middleware Layer

RMW layer is composed of an abstract API (RMW) and its implementation (RMW implementation).

This layer provides basic middleware functionality: publish/subscribe, services (request-reply) and serialisation of message types. The upper layers use this middleware API on the micro-ROS library stack. This API abstraction allows hiding implementation and vendor specific details to the user.

This abstract API has a default implementation using eProsimas's XRCE protocol implementation, Micro XRCE-DDS. This implementation resides on the rmw_implementation layer. This rmw_implementation layer is responsible for providing discovery as well as inter-process communication optimisation avoiding unnecessary wire interactions.

The middleware API provided by RMW layer to the upper layers should be similar to the one produced by its analogous to ROS2.

3.8.1.3 Library Layer

This layer provides micro-ROS API to the user application. This layer is composed of a set of libs, RCL and a set of modules providing a higher level of abstraction concepts.

RCL library provides primary and shared services. This library implements logging, the handle of parameters, time functionality, names and namespaces handling. These services are the same services this layer provides in ROS2.

Regarding the final user interface on top of RCL, micro-ROS follows a double-tracked approach:

1) Use rcl as C-based Client Library for micro-ROS by enriching it with small, modular libraries for parameters, graph, logging, clock, timers, execution management, lifecycle and system modes, TF, diagnostics, and power management.

2) Analyze fitness of rclcpp for use on microcontrollers, in particular regarding memory and CPU consumption as well as dynamic memory management.

The original approach using RCLC has been discarded by this more proper use of existing ROS2 packages. More on the rationale of this decision can be found in the project website.

3.8.1.4 Layers and Components

As a link between components and layers, the following diagram shows micro-ROS Stack components over the layers with they are related.

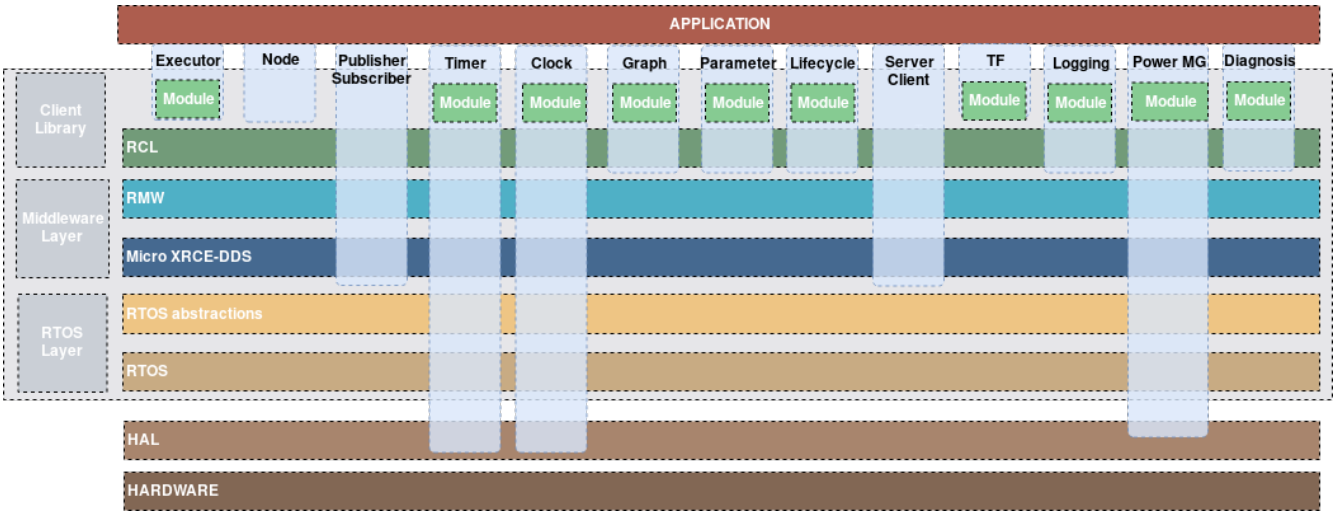


Figure 18: micro-ROS layered components

3.8.2 micro-ROS Agent

micro-ROS Agent is a ROS2 package. As the essential part of the package, there is micro-ROS Agent Core which provides a Micro XRCE-DDS Agent allowing communication between DDS and DDS-XRCE. Optionally and depending on the profiles and configuration required, along with micro-ROS Agent core users could use a parameter server and a graph server. Both servers provide ROS2 server interface to query parameters and node graphs, respectively.

3.9 Infrastructure Architecture

This section provides information about the infrastructure architecture of the micro-ROS platform.

3.9.1 Infrastructure

micro-ROS platform primary purpose is to connect ROS2 applications with applications running on microcontrollers. Following this crucial principle, we based the infrastructure on microcontrollers. This

project already define reference platforms in the *D2.1 Report on the reference hardware development platforms.* Document.

A sample of infrastructure is:

- Reference board: Olimex E407.
  - Real-time Operating System: NuttX.
  - I/O: Serial or radio based. (UART, 6LowPAN)
- Main computer: General purpose computer.
  - Operating System: Linux or Windows.
  - I/O: Serial or radio based. (UART, 6LowPAN)

This infrastructure is a sample, as the micro-ROS platform is mainly a generic framework, the final infrastructure depends on the user application.

3.10 Deployment

This section provides information about the mapping between the software and the infrastructure.

3.10.1 Deployment

This diagram represents schematically how is a micro-ROS platform deploy.

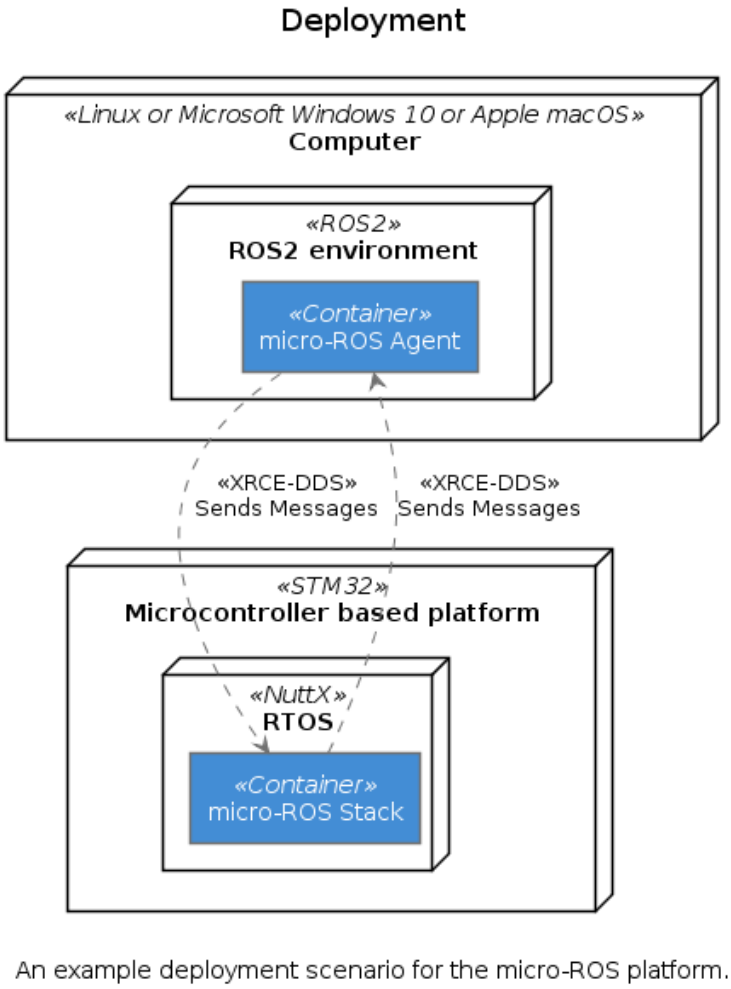


Figure 19: micro-ROS Deployment

The diagram represents a sample of deployment. In this sample, we show a single microcontroller added to a ROS2 system. Multiple microcontrollers deployment is possible.

3.10.2 Build System

The micro-ROS platform uses the same build system as ROS2. Using colcon as build dependencies manager allows sharing packages with ROS2 implementation. micro-ROS adds multiple packages to existing ROS2 package collection. These micro-ROS dedicated packages have the same configuration options as the ROS2.

This has some complications regarding, and the usage of different RTOS build systems. Trying to alleviate the micro-ROS users from this cumbersome a new build approach will be used. This approach is to create an RTOS ROS2 package, in this concrete case NuttX, including a set of MCU configurations. This way, the regular ROS developer can keep using colcon as before not needing to worry too much on the work done for him.

The current approach is to have two different build workspaces:

A) For the code meant to be deployed to the MCU.

B) For the code running on the host.

The proposed build steps could be described as follows

1) Run a Configuration package that configures the firmware for the target MCU, and this package is placed in the host. This configuration includes the target board and its configuration, which workspace packages to include in the firmware.
2) During configuration, an MCU workspace is created and configured with the required packages.
3) RTOS is configured, and cross-compilation tools are generated according to the target.
4) At build, time host workspace and MCU workspace will be built and linked separately, taking into account the final target.

3.10.3 Profiles

This lead to a modular build which only builds those system parts needed by profile configuration.

The following table shows a possible list of the profiles and the components.

Table 4: PS-Pure Sender, PR-Pure Reader, CS-Configurable Sender with diagnostics, AS-Ad-hoc disverable Sender, SS-Static configured Sender, MS-Managed Service, SN-Simulation Node.

micro-ROS/Profiles	PS	PR	CS	AS	SS	MS	SN
nodes	x	x	x	x	x	x	x
lifecycle						x	
publish	x		x	x	x		
subscribe		x					
services						x	
logging			x				
parameters			x				
discovery				x			

Timers, clock and executor are not affected by middleware profiles.

New profiles could be defined by the user to adjust to their applications needs.

3.10.3.1 Middleware Profiles

Apart from micro-ROS profiles, DDS-XRCE standard defines a set of profiles modularising the middleware capabilities. In the micro-ROS platform, these profiles are also configurable per middleware implementation.

- Read Access profile. Provides the clients with the ability to read data on pre-configured Topics with pre-configured QoS policies.

- Write Access profile. Provides the clients with the ability to write data on pre-configured Topics with pre-configured QoS policies.
- Configure Entities profile. Provides the clients with the ability to define DomainParticipant, Topic, Publisher, Subscriber, DataWriter, and DataReader entities using pre-configured QoS policies and data-types.
- Configure QoS profile. Provides client with the ability to define QoS profiles to be used by DDS entities.
- Configure types profile. Includes middleware client the ability to set data types to be used for DDS topics.
- Discovery access profile. Provides the middleware clients to discover entities.
- File-based configuration profile. micro-ROS Agent core could be configured using a file system.
- UDP Transport profile. It provides UDP transport.
- TCP Transport profile. It provides TCP transport.
- Complete profile. It allows the complete functionality of the middleware.

micro-ROS profiles require some of the specific middleware profiles. This profile dependencies are resolved automatically at build time selecting those middleware profiles required by micro-ROS profiles.

This table represents dependencies between micro-ROS components and Micro XRCE-DDS profiles.

Table 5: RA-Read access, WA-Write access, CE-Configure Entities, CQ-Configure QoS, CT-Configure Types, DA-Discovery Access, FC-File-based Configuration, UDP-UDP transport, TCP-TCP transport, CP-Complete Profile.

micro-ROS/Micro XRCE-DDS	RA	WA	CE	CQ	CT	DA	FC	UDP	TCP	CP
nodes										
lifecycle	x	x								
publish		x								
subscribe	x									
services	x	x								
parameters	x	x								
discovery					x					

Timer, clock, logging and executor are not dependent on any middleware profile. As a note, the middleware logging profile refers to the middleware implementation logging system, not to micro-ROS loggers.

3.10.4 Test System

The intended modularity of micro-ROS, apart from helping with deployment configurations, it dramatically helps to the testability of the whole system. micro-ROS should provide test per each component. Components interfaces are tested to verify the correctness of the functionality provided by it. Unit tests are recommended but not mandatory. The micro-ROS packages also have a test so that the users can verify the correctness of the package interfaces.

4 Appendix

4.1 A1 Related Documents

Previous Architecture Definition version: *D1.4 Overall Architecture Definition Initial.*

Scenarios and requirements definition: *D1.7 Reference Scenarios and Technical System Requirements Definition.*

Hardware platform: *D2.1 Report on the reference hardware development platforms.*

Lifecycle and System Modes definition: *D4.8 Lifecycle and System Modes.*

References

[1] S. Brown, *Software architecture for developers: Volume 1 - technical leadership and the balance with agility*. Leanpub, 2018.

[2] S. Brown, *Software architecture for developers: Volume 2 - visualise, document and explore your software architecture*. Leanpub, 2018.

[3] S. Brown, 'The c4 model for software architecture context, containers, components and code'. 2018 [Online]. Available: https://c4model.com/

[4] 'ROS2 design'. Open Source Robotics Foundation, Inc. [Online]. Available: http://design.ros2.org/

[5] OMG (Object Management Group), 'DDS For Extremely Resource Constrained Environments'. 2019 [Online]. Available: https://www.omg.org/spec/DDS-XRCE/About-DDS-XRCE/

[6] T. Foote, 'Tf: The transform library', in *2013 ieee conference on technologies for practical robot applications (tepra)*, 2013, pp. 1–6.

[7] T. De Laet, H. Bruyninckx, and J. De Schutter, 'Rigid body pose and twist scene graph founded on geometric relations semantics for robotic applications', in *2013 ieee/rsj international conference on intelligent robots and systems*, 2013, pp. 2398–2405.