micro-ROS
ROSCon France

22-23 June 2021

Maria Merlan (eProsima)
Jan Staschulat (Bosch)
Pablo Garrido (eProsima)

AGENDA

micro-ROS Intro and RMW
Maria Merlan from eProsima
01

micro-ROS RCLC
Jan Staschulatt from Bosch
02

micro-ROS Live Demo
Pablo Garrido from eProsima
03

Questions and Answers
04

micro-ROS Intro and RMW

- What is micro-ROS?
- Purpose
- micro-ROS Layered architecture
- Middleware architecture

Who are we?

OFERA

funded by European Commission

EPROSIMA The Middleware Experts

BOSCH

FIWARE FOUNDATION

PIAP

µROS

Open-source project,
now benefiting from a huge participation from a growing community!

https://micro-ros.github.io/

Why micro-ROS?

XRCE (μC) Embedded world

Robotics trend evolves towards interconnected systems of CPUs and **multisensor-actuator (that run on low resource boards μC)**

New inherent challenges

Memory limitations, real-time systems, energy consumption, wide range of vendors.
Lack of common standard development framework

micro-ROS mission

Common framework ROS 2 based which Mission is to bring ROS 2 nodes into the embedded world (μC)

micro-ROS

Why micro-ROS?

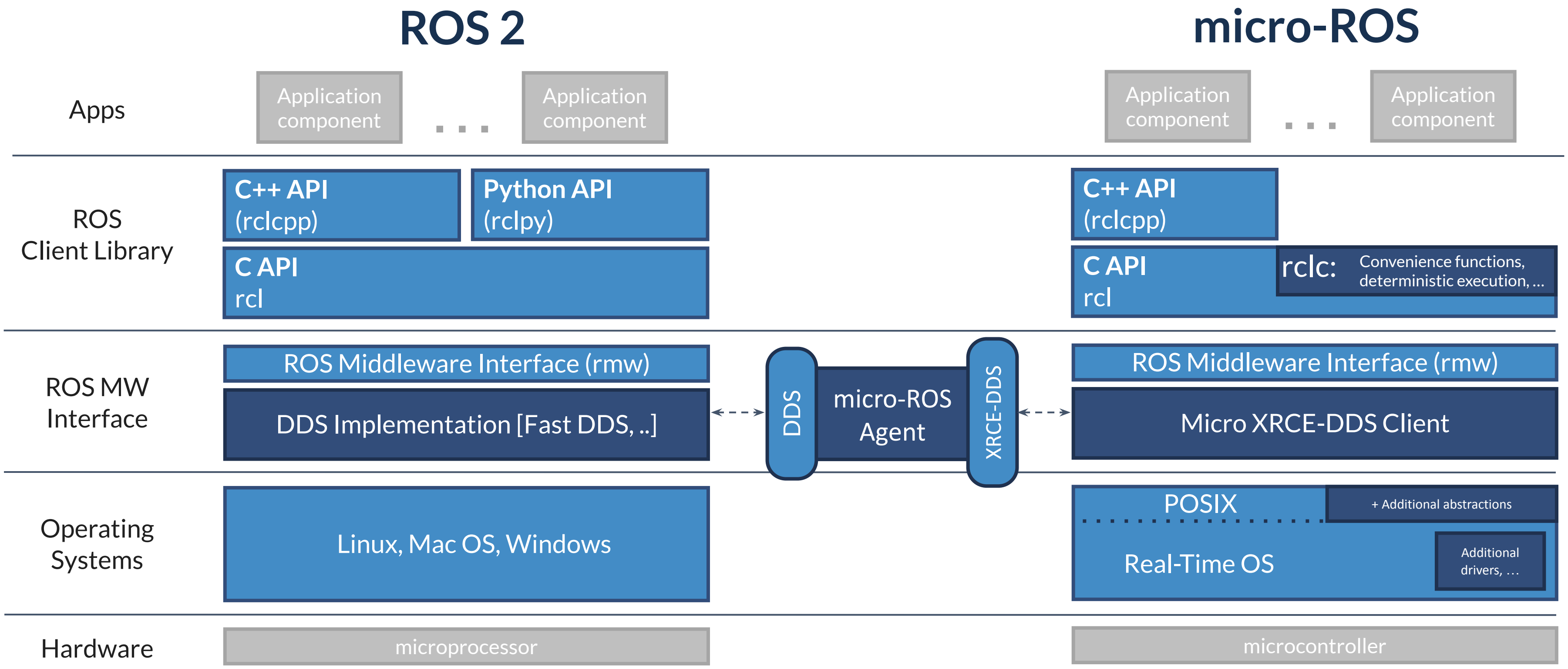
A solution for creating ROS 2 nodes into embedded devices

- **Accelerator** of application development via allowing the combination of CPUs and µC within any robotic system
- **Enabler** of affordable deployments (IoT, robotics, autonomous driving ,...)

Highlights

- **Mirroring ROS 2 for Embedded world**
  - Layer-compatible with ROS 2
  - Integrated into ROS 2 ecosystem
  - Allows to create a ROS 2 node with ~ all functionalities
  - Client-server logics (client fully dynamic memory free)
- **Widest range of use cases**
  - Middleware transports fully customizable
  - Runs on bare-metal, all RTOSs and all MCUs
  - Platform-versatile cross-compilation tools
- **Mature technology**
  - Benefits of full QoS support ROS 2
  - Now supporting **Foxy and Galactic and Rolling**
  - A growing community

micro-ROS layered architecture



µROS

Middleware architecture

Micro XRCE-DDS

- **Wire-protocol over Client-Server architecture**

    XRCE Client on low-resource consumption devices

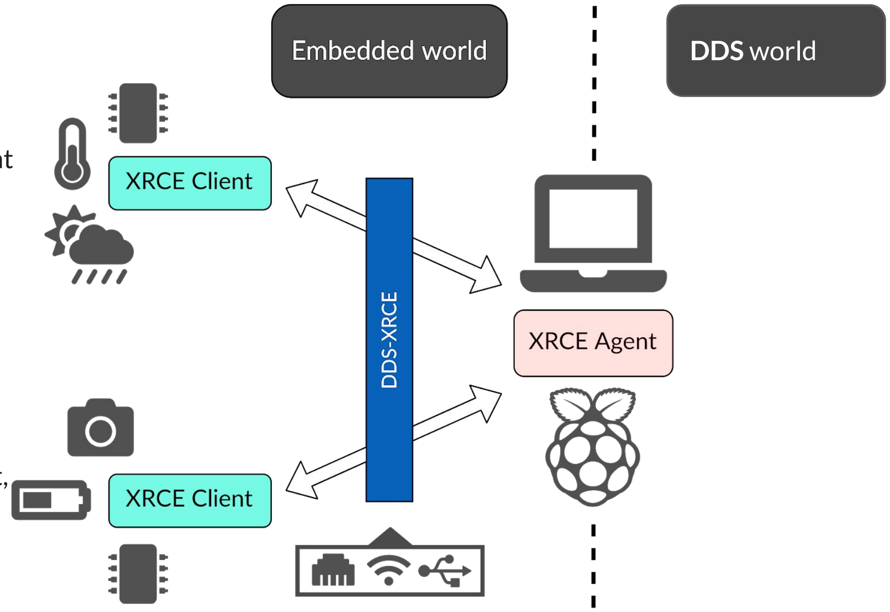
    XRCE Agent entity connected with DDS global data space that acts on behalf of Clients

- **Client fully static and dynamic memory free**

    75 KB of Flash memory and 3 KB of RAM

- **Real-Time and Deterministic - critical applications**
- **Transport-agnostic, customized by the user**

    Built-in support for serial transports, TCP, UDP over Ethernet, Wi-Fi, and 6LoWPAN, and Bluetooth



µROS

Memory optimization

- Implemented using Micro XRCE-DDS middleware in lower layers
- Allows smart configuration of memory resources (micro-ROS)
  - Static configuration
  - Parameter level

micro-ROS configurable parameters

- Max Publishers
- Max History
- Max Subscriptions
- Node name max length
- Max Clients
- Type name max length
- Max Services
- Max Nodes
- Max Topics
- Topic name max length

µROS

FULL PORTABILITY

Any RTOS and Bare metal Library Generator!

Any low-mid range MCU!

Typical features:

~ 150 KB of flash memory

> 25 KB of RAM memory

General purpose input/output pins

Peripherals: GPIO, USB, Ethernet, SPI, UART, I2C, CAN, etc

REFERENCE HW

- **Arduino Portenta**
- **Raspberry Pi Pico**
- **Arduino Nano RP2040 Connect** 1st Arduino with Raspberry Pi silicon
- **ESP-IDF v4.3 & ESP32-S2/C3**
- **Teensy 3.2 / 3.5 / 4.1 / 4.2**
- **OpenCR support**
- **STM32CubeMX & STM32CubeIDE**
- **Olimex LTD STM32-E407**
- **Crazyflie 2.1 drone, ...**

REFERENCE RTOS

- **Mbed RTOS 6.8 / 6.9 / 6.10**
- **FreeRTOS**
- **NuttX 10.0 / 10.1**
- **Zephyr RTOS 2.4 / 2.5**

Check full list of supported HW & RTOS
https://micro.ros.org/docs/overview/hardware/

AGENDA

micro-ROS Intro and RMW
Maria Merlan from eProsima
01

micro-ROS RCLC
Jan Staschulatt from Bosch
02

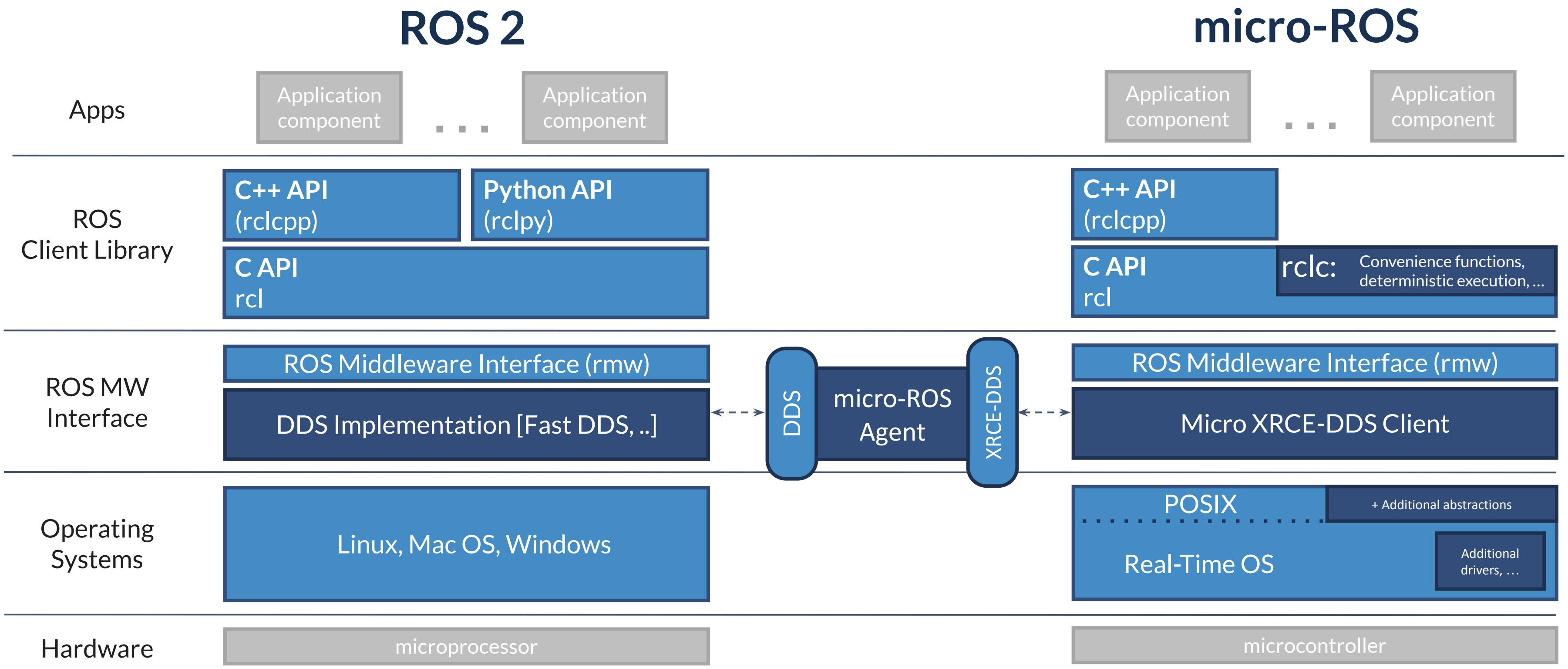
micro-ROS Live Demo
Pablo Garrido from eProsima
03

Questions and Answers
04

micro-ROS RCLC

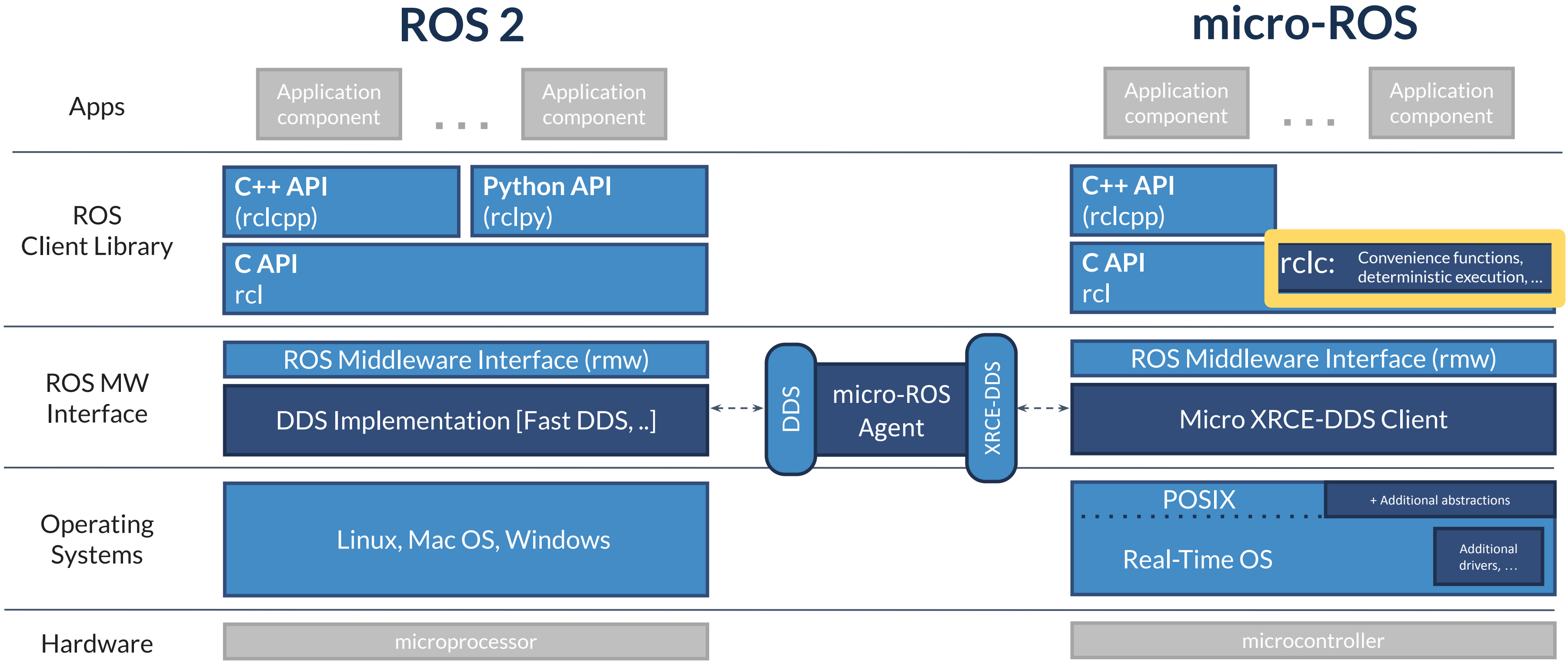
- ROS 2 basic concepts
- API Overview
- Executor
- Lifecycle
- Parameter

micro-ROS layered architecture



µROS

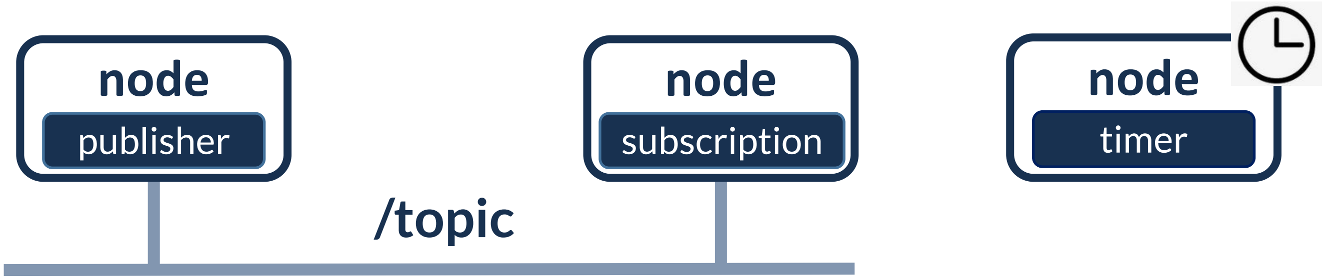
micro-ROS layered architecture



µROS

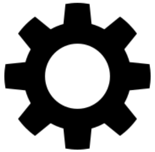
ROS 2: basic concepts

Pub-sub communication



Executor

spin()



- Checks for new messages
- Executes corresponding callbacks

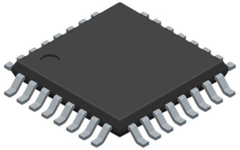
Why an RCLC API?

ROS 2 – RCLCPP drawbacks

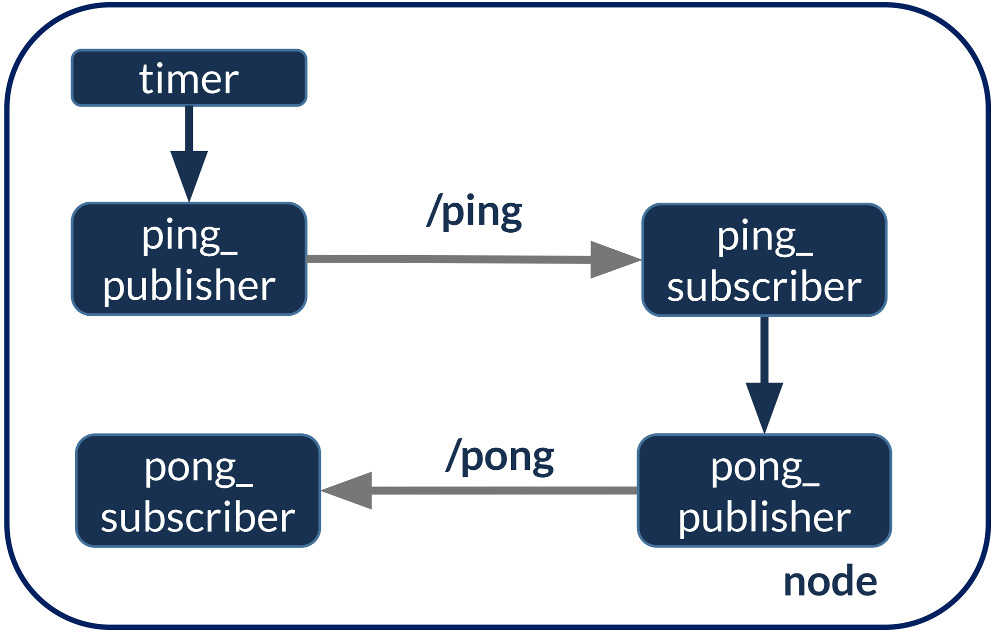
- API in C++ uses dynamic memory allocation
- Executor is not deterministic nor does it support real-time

Micro-ROS – RCLC benefits

- Thin layer on top of RCL (no additional data structures) feature-complete (publishers, subscriptions, timers, services/clients, guard conditions, parameters, lifecycle)
- Executor uses dynamic memory allocation only at startup
- Deterministic Executor with additional features to support real-time applications

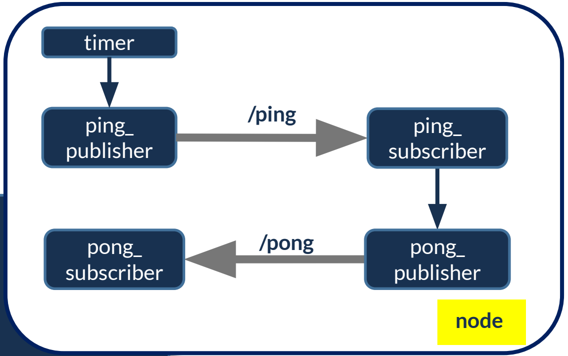


RCLC API: Overview

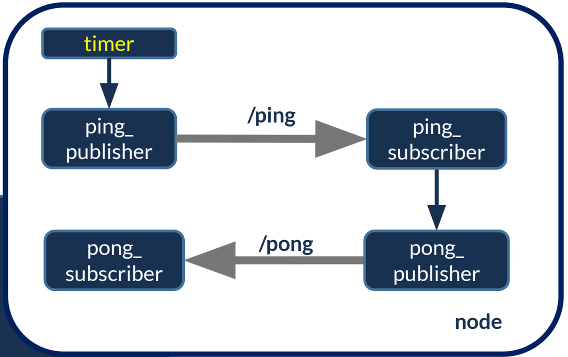


RCLC API: node

```
#include <rclc/rclc.h>
#include <rclc/executor.h>

void main()
{
...
rcl_allocator_t allocator = rcl_get_default_allocator();
rclc_support_t support;
rclc_support_init(&support, 0, NULL, &allocator);

rclc_node_init_default(&node, "pingpong_node", "", &support);
```



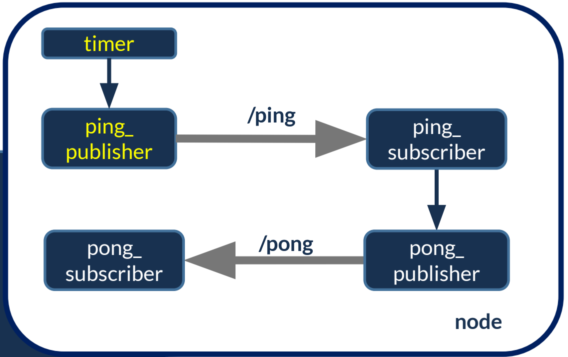
RCLC API: timer




```
rcl_timer_t timer = rcl_get_zero_initialized_timer();

rclc_timer_init_default(&timer,
      &support,
      RCL_MS_TO_NS(2000),
      ping_timer_callback);
```

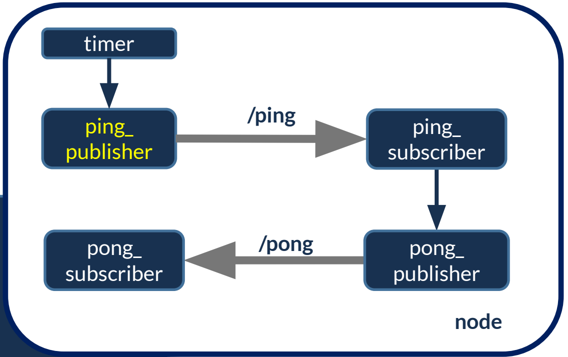
RCLC API: timer callback

```
void ping_timer_callback(rcl_timer_t * timer, int64_t last_call_time)
{
    (void) last_call_time;
    if (timer != NULL) {
        seq_no = rand();
        sprintf(outcoming_ping.frame_id.data, "%d_%d", seq_no, device_id);
        outcoming_ping.frame_id.size = strlen(outcoming_ping.frame_id.data);
        struct timespec ts;
        clock_gettime(CLOCK_REALTIME, &ts);
        outcoming_ping.stamp.sec = ts.tv_sec;
        outcoming_ping.stamp.nanosec = ts.tv_nsec;
        pong_count = 0;
        rcl_publish(&ping_publisher, (const void*)&outcoming_ping, NULL);
    }
}
```



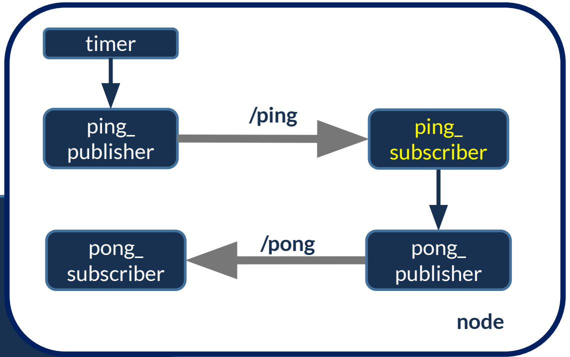
RCLC API: publisher

```
rcl_publisher_t ping_publisher;

rclc_publisher_init_default(&ping_publisher,
      &node,
      ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Header),
      "/microROS/ping");
```



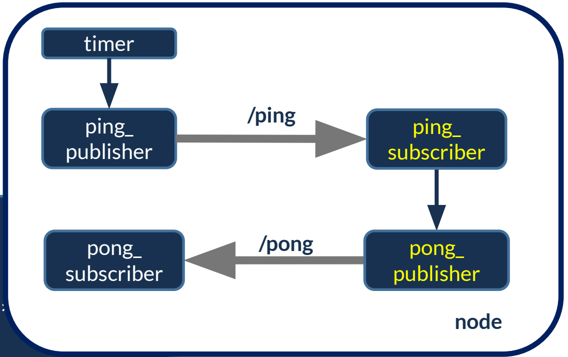
RCLC API: subscription

```
rcl_subscription_t ping_subscriber;

rclc_subscription_init_default(&ping_subscriber,
      &node,
      ROSIDL_GET_MSG_TYPE_SUPPORT(std_msgs, msg, Header),
      "/microROS/ping");
```



RCLC API: subscription callback

```
void ping_subscription_callback(const void * msgin)
{
    const std_msgs__msg__Header * msg = (const std_msgs__msg__Header *
    rcl_publish(&pong_publisher, (const void*)msg, NULL);
    }
}
```



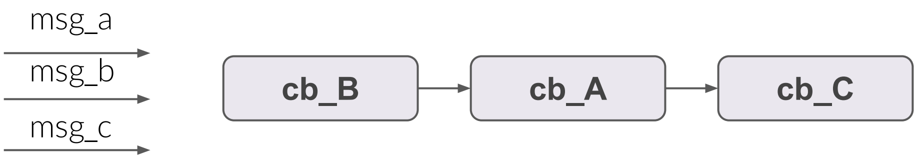
RCLC API: executor

```
rclc_executor_t executor = rclc_executor_get_zero_initialized_executor();
rclc_executor_init(&executor, &support.context, 3, &allocator));


rclc_executor_add_timer(&executor, &timer));
rclc_executor_add_subscription(&executor, &ping_subscriber, &incoming_ping,
&ping_subscription_callback, ON_NEW_DATA));

rclc_executor_spin(&executor);
```

RCLC Executor: determinism

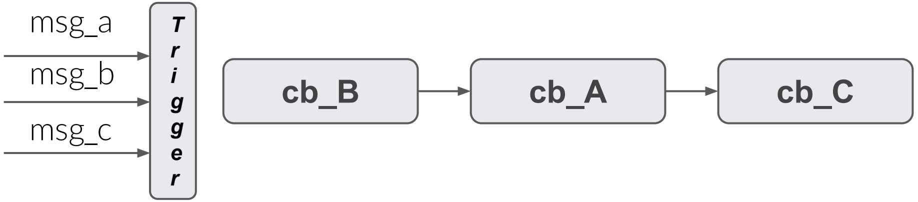
Deterministic behavior

- User-defined order of callback processing determines which callback is processed first

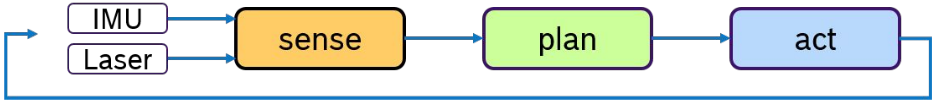


Domain-specific scheduling

- Trigger condition to support domain specific-scheduling (e.g. OR, AND, ONE)



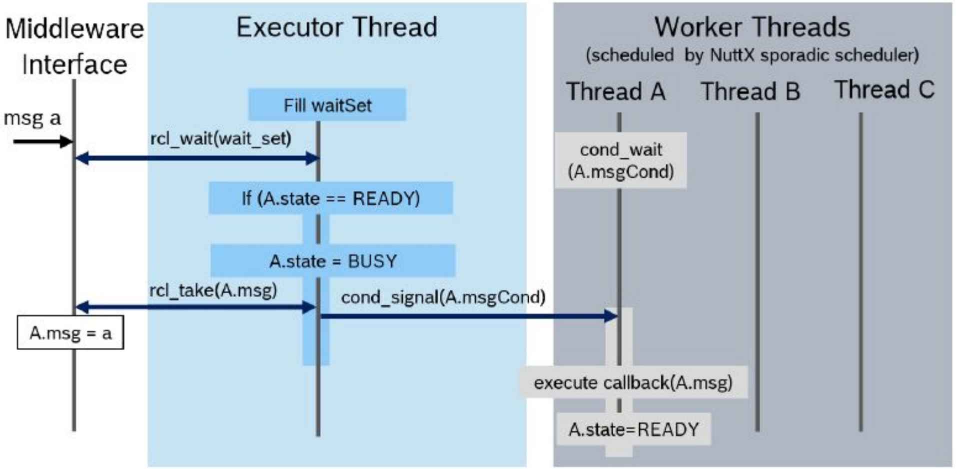
- Use cases
  - Sense-plan-act control loops
  - Synchronization of messages (sensor fusion)



RCLC Executor: real-time scheduling

Expose scheduling features of RTOS

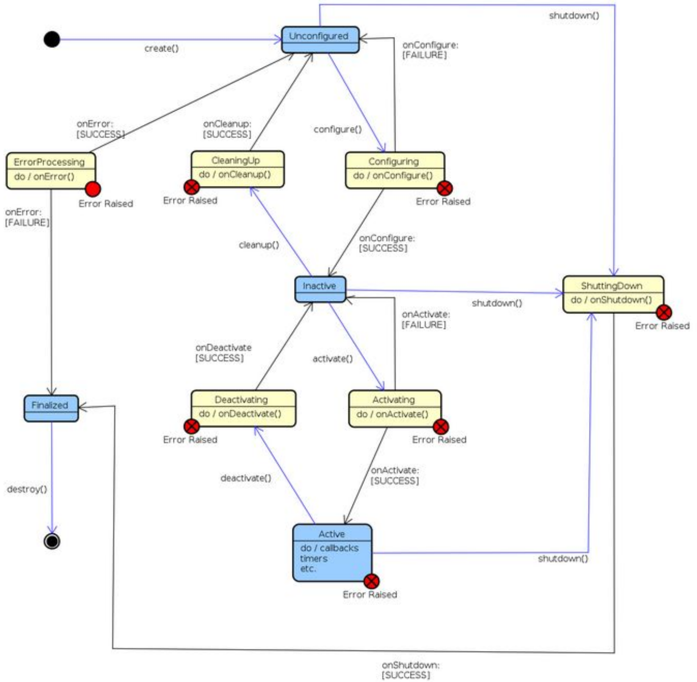
- callbacks are processed in worker thread
- Executor thread manages data exchange with middleware layer
- Assignment of RTOS priority to worker thread allows real-time scheduling of callback processing
- Status: proof-of-concept with budget-based scheduling of NuttX-OS (arXiv paper)



RCLC Lifecycle

Convenience function for **ROS 2 Lifecycle Node with rclc**
rclc lifeycle node bundles an rcl Node and the ROS 2 lifecycle state machine

- Greater control over the state of ROS system
  - ROS 2 standard node life cycle
  - configure, activate, deactivate, cleanup, …
  - integrated with launch, e.g., ensure all components active before any component begins executing its behavior
- Previously only available for C++ (rclcpp_lifecycle)
- Now available for C (rclc):
- Builds upon rcl_lifecycle (as does rclcpp_lifecycle)
  - **Transitions and callbacks** implemented, working, and tested
  - **Lifecycle services** implemented, pull request **pending**
    - Under discussion: Completely avoid dynamic memory allocation.
    - Not yet possible due to strings in lifecycle messages



RCLC Lifecycle

Initialisation:

```
rclc_node_init_default(&my_node, "lifecycle_node", …);
rcl_lifecycle_get_zero_initialized_state_machine();
rclc_make_node_a_lifecycle_node(&lifecycle_node, &my_node, …);
```

Transitions and Callbacks:

```
rclc_lifecycle_register_on_configure(&lifecycle_node, &my_on_configure);
rclc_lifecycle_change_state(&lifecycle_node, …TRANSITION_CONFIGURE, …);
```

Lifecycle services: *(pull request pending!)*

```
rclc_lifecycle_add_get_state_service(&lifecycle_node, &executor);
rclc_lifecycle_add_get_available_states_service(&lifecycle_node, &executor);
rclc_lifecycle_add_change_state_service(&lifecycle_node, &executor);
```

µROS

RCLC Parameter

```
rclc_parameter_server_init_default(&param_server, &node);

rclc_executor_t executor;
rclc_executor_init(&executor, &support.context, RCLC_PARAMETER_NUM + 1, &allocator);
rclc_executor_add_parameter_server(&executor, &param_server, on_parameter_changed);
rclc_executor_add_timer(&executor, &timer);

rclc_add_parameter(&param_server, "param1", RCLC_PARAMETER_BOOL);
rclc_add_parameter(&param_server, "param2", RCLC_PARAMETER_INT);
rclc_add_parameter(&param_server, "param3", RCLC_PARAMETER_DOUBLE);

rclc_parameter_set_bool(&param_server, "param1", false);
rclc_parameter_get_bool(&param_server, "param1", &value);
```

µROS

AGENDA

micro-ROS Intro and RMW
Maria Merlan from eProsima
01

micro-ROS RCLC
Jan Staschulatt from Bosch
02

micro-ROS Live Demo
Pablo Garrido from eProsima
03

Questions and Answers
04

Micro ROS Live Demo
Q&A

micro-ROS
ROSCon France

Thank you