Bartłomiej STADNIK[1], **Artur WYMYSŁOWSKI**[1]

Wrocław University of Science and Technology (1)

doi:10.15199/48.2024.12.48

Overview Analysis of Micro-ROS System as an Embedded Solution for Microcontrollers in Automatics and Robotics Applications

Abstract. *Micro-ROS extends ROS capabilities to resource-constrained devices like ESP32, enabling seamless integration into robotic systems. This paper compares micro-ROS with classic ROS 2, emphasizing its practicality for microcontroller applications. Future plans include evaluating micro-ROS on a mobile robot platform for performance and efficiency.*

Streszczenie. *Micro-ROS rozszerza możliwości środowiska ROS na urządzenia o ograniczonych zasobach, takie jak ESP32, umożliwiając płynną integrację z systemami robotycznymi. Niniejszy artykuł porównuje system micro-ROS z klasycznym ROS 2, podkreślając praktyczność w zastosowaniach z mikrokontrolerami. Plany na przyszłość obejmują ocenę systemu micro-ROS na platformie robota mobilnego pod kątem wydajności i efektywności.* ***(Analiza Przeglądowa Systemu Micro-ROS jako Rozwiązania dla Mikrokontrolerów w Aplikacjach Automatyki i Robotyki)***

Keywords: micro-ROS, ROS, ESP32, real-time systems
Słowa kluczowe: micro-ROS, ROS, ESP32, systemy czasu rzeczywistego

Introduction

Robot Operating System (ROS) is a commonly used framework across various robotic applications, serving as a robust toolkit for developers dealing with complex robotic designs. This standardized platform addresses the phases of research, prototyping, and deployment. The second iteration of the system, called ROS 2, signifies a leap forward addressing drawbacks and constraints of its predecessor. ROS 2 achieves this through the implementation of a more distributed design, the incorporation of real-time capabilities, and the deployment of advanced communication protocols. Typically, a robotic system often consists of a network of microcontrollers that implement functionalities wrapped within hard real-time control loops. With industry embracing ROS 2, the need arises to extend its capabilities to resource-constrained devices, which are not driven by a native Operating System (OS). This gap is to be filled by a relatively fresh solution referenced as micro-ROS system, a direct extension of ROS 2, which reuses its stack and combines components specifically tailored for microcontrollers' platform.

In this paper, we present a technical overview of the micro-ROS system and compare it with the classic ROS 2 implementation, especially with reference to microcontroller applications. It should be underlined that the micro-ROS system has been designed to work as a part of ROS 2 platform, which means that it inherits many of the features and capabilities of ROS 2. To exemplify the current state of the art, the system was deployed on an ESP32 embedded device using the available toolkit. In the future, it is planned to implement a micro-ROS system for a mobile robot platform as part of PhD thesis, aiming to evaluate the impact of transitioning from a traditional computer to a resource-constrained device while maintaining functionality at an equivalent level. Microcontrollers are known for their low power consumption and ability to operate in real-time, which is particularly beneficial for mobile robots. Extending such capabilities with features equivalent to those offered by classic ROS 2, can open up and extend possibilities in the field of robotics.

ROS and ROS 2 Basics and Overview

Despite its suggestion of name, ROS is not an operating system, but rather a middleware framework designed for robotics applications. In order to run it, users typically install ROS on top of a conventional operating system such as Linux, which serves as the underlying platform for executing ROS-based programs and managing hardware resources. The ROS system is quite popular among robotics researchers, engineers, and enthusiasts around the world. It is used by companies such as Boston Dynamics, Clearpath Robotics, and Fetch Robotics for developing advanced robotic solutions for commercial purposes. These systems excel in various tasks including navigation, perception, manipulation, coordination, and more.

The evolution of ROS [1] was marked by the incorporation of a standardized communication approach based on DDS (Data Distribution Service), an external middleware that forms the foundation of ROS 2 [2]. DDS enables the exchange of data in a real-time publisher-subscriber model, where a publisher sends data to subscribers that are linked by message queue. DDS provides message definition, message serialization, and control over different QoS (Quality of Service) of transportation to meet specific application requirements. A distinctive feature of DDS is its data-centric concept, designed to send updates only to the data that have changed, rather than transmitting the entire message with each update. This approach effectively reduces network traffic, a valuable feature within robotic systems characterized by rich and dynamic data flows.

With integrated abstraction layers, the complexity of DDS is hidden from ROS components, providing users with familiar ROS 1 like interface. ROS 2 does it by integrating the so-called RMW (ROS Middleware) on top of the DDS middleware, rather than directly using it. This enables the system to support the usage of multiple DDS implementations, allowing users to leverage different functionalities and properties offered by different vendors. On top of the RMW, the RCL (ROS Client Library) comes into play. Written in C language, this library provides essential access to ROS 2 features in the form of an interface. This is further wrapped as an API for languages like Python and C++, forming another abstraction layer. The architectural layout of ROS 2 is presented in the figure 1.

The presence of discovery system resulted in the decision to eliminate tools like ROS Master (centralized entity coordinating communication between nodes), promoting a more distributed architecture. When a new component appears in the system, it broadcasts information about its existence using the DDS API interface, meaning that nodes do not need to be aware of each other's existence. This design enables modularization of robotic systems, allowing components to run independently of each other, which can be par-

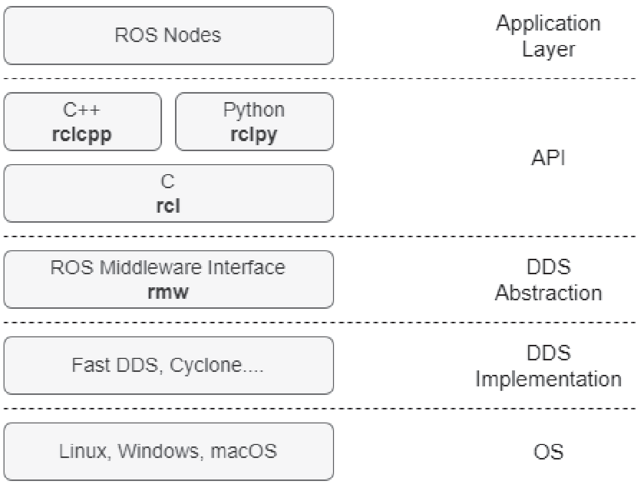


Fig. 1. ROS 2 architecture overview

ticularly beneficial in the case of a hardware-distributed system. Furthermore, there is no single point of failure that could cause a complete system breakdown. Individual nodes use a lifecycle management mechanism with an internal state machine capable of error handling and recovery.

Introduction to micro-ROS

In contrast to ROS 2, micro-ROS introduces the core concepts of the ROS 2 system to resource-constrained devices that would not have sufficient performance to run its classical counterpart. Employing a layered architecture, the micro-ROS stack reuses existing components of the ROS 2 to ensure compatibility between both systems and providing a common interface. Provided functionalities include node design, communication with the publish-subscribe or client-service model, lifecycle management, node graph, and more. Although both systems are comparable, there are major differences that originate in the requirements of their targeted platforms [3]. Working with devices limited in resources, such as microcontrollers, requires an optimized architecture with elements specifically tailored to fit their specific hardware [4]. The current architectural layout of the micro-ROS system is presented in the figure 2.

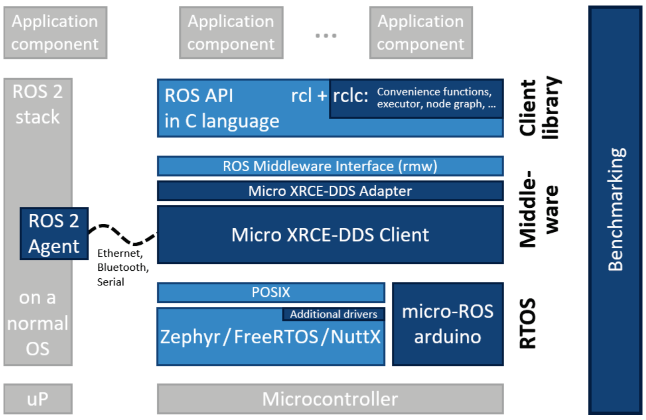


Fig. 2. Architecture of the micro-ROS stack [5]

Within the layers maintained by both micro-ROS and ROS 2, we can identify the components of RCL and RMW. To deliver features similar to RCLCPP, which is a C++ abstraction over RCL, an additional lightweight library RCLC was tailored to provide a complete and optimized C API for mechanisms such as executors or convenience functions. The combination of RCL and RCLC enables the system to operate without dynamic memory allocation after the initialization stage. This aspect ensures efficient use of limited resources and deterministic behavior, which is crucial for real-time systems.

Ultimately, what shapes the micro-ROS architecture is its middleware layer. It is based on Micro XRCE-DDS, a standard for Extremely Resource-Constrained Environments (XRCE) capable of bringing the DDS model to embedded systems. The middleware includes built-in support for serial transports. While XRCE-DDS is responsible for passing messages between two entities, it is complemented by the micro-ROS agent, which takes over discovery processes and QoS (Quality of Service) mechanisms that would be too computationally expansive to handle by an embedded device. The agent is running on a host with standard ROS 2, as presented in the figure 3.

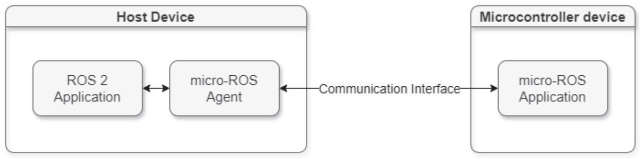


Fig. 3. ROS 2 to micro-ROS communication concept with Agent

The micro-ROS environment supports the assumptions of the real-time system by integrating selected open source real-time operating system, e.g. FreeRTOS [6]. As such, the remaining layers are run on top of RTOS, which enables compliance with time-critical requirements, a task rather hard to achieve on standard operating systems lacking deterministic scheduling. Furthermore, the usage of RTOS provides a POSIX interface, which is required for interoperability with upper-class layers. This standard ensures compatibility and portability across different operating systems, making programs easily portable between derivatives of Unix systems. Consequently, developers can use a single POSIX-compliant application and expect it to behave consistently across most Unix-like systems.

Executor of the micro-ROS System

With ROS applications being based on distributed node design, communication is handled by invoking callback for each message. The messages are buffered in DDS, and then the Executor is responsible for dequeuing them, thus dispatching to threads for execution. Prior to message handling, a wait-check is configured allowing the system to wait until specified conditions are met. However, in the default Executor of ROS 2, there is no further prioritization of messages and their scheduling is nonpreemptive round-robin type, meaning that deterministic execution is hard to fulfill. With the presence of time-critical callbacks, this behavior is prone to suffer from missed deadlines, affecting the response jitter.

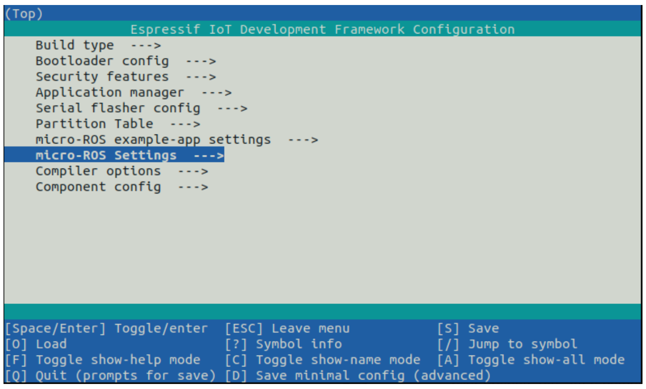
To resolve the said limitation, the RCLC Executor [7, 8] has been specifically implemented for micro-ROS. It is a feature-complete component that is based on the RCL layer and further expands the available tools with:

1. Sequential execution: enables for configuration of the processing order, in which callbacks should unfold during runtime.
2. Trigger execution: provides control over execution flow through trigger conditions; user can select logic dictating when processing of callback should start. The available conditions are: all, any, and one. In addition, a custom definition can be used.
3. LET-semantics for data synchronization: Logical Execution Time is a concept which reads all input data on a

periodic trigger of executor, and then processes all required callbacks. This removes the requirement of data synchronization for subsequent callbacks.

4. Multithreading support: for every callback dequeued by Executor, a new thread is spawned with provided scheduling policy. Executor threads can be prioritized according to real-time requirements.

During the initialization phase of the RCLC Executor, which is shaped by user-defined properties, essential dynamic memory is allocated, and the total number of callbacks is set by the user. This makes the Executor static, ensuring that no additional memory is allocated during run-time once initialized; no further callbacks can be added during run-time. The drawback of this solution is that it lacks flexibility. Since the total number of callbacks must be determined at initialization, any need for additional callbacks or changes in the callback configuration during runtime cannot be accommodated. This can be particularly limiting in dynamic environments where the workload or functional requirements may evolve.



```
(Top)
                    Espressif IoT Development Framework Configuration
    Build type  --->
    Bootloader config  --->
    Security features  --->
    Application manager  --->
    Serial flasher config  --->
    Partition Table  --->
    micro-ROS example-app settings  --->
    micro-ROS Settings  --->
    Compiler options  --->
    Component config  --->

[Space/Enter] Toggle/enter   [ESC] Leave menu           [S] Save
[O] Load                     [?] Symbol info            [/] Jump to symbol
[F] Toggle show-help mode    [C] Toggle show-name mode  [A] Toggle show-all mode
[Q] Quit (prompts for save)  [D] Save minimal config (advanced)
```

Fig. 4. Menuconfig interface of the ESP-IDF Development Framework

ESP32 Platform Support

The micro-ROS framework supports a number of microcontroller platforms, e.g., STM32L4, Raspberry Pi Pico, Espressif ESP32, etc. The ESP32 family stands out as a cost-effective and robust entry point for exploring the capabilities of the mentioned system. The ESP32 family are low-cost, low-power microcontrollers developed by Espressif Systems. They feature a dual-core processor, integrated WiFi and Bluetooth connectivity, GPIO pins, and other peripheries. These capabilities make the ESP32 platform an attractive choice for a wide range of embedded, robotic, and IoT systems. Complementing the device, the official ESP-IDF (Espressif IoT Development Framework) [9] is provided to simplify application development. This includes essentials like RTOS kernel and CMake based build system with capability to support external components.

The latest advancements within the micro-ROS build system includes a tool generating static libraries and headers, which are required for the external build systems. As such, it is possible to install the micro-ROS software as a component of ESP-IDF rather than performing a manual installation. This is the recommended approach due to better optimization within the platform. Furthermore, FreeRTOS is natively supported by the hardware and falls within the scope of supported RTOSes of the micro-ROS system. The resulting mix enables a system with complete ROS capabilities, real-time support, and the ability to exploit the built-in WiFi communication. This is appealing to users seeking to develop wireless applications within the ROS 2 ecosystem. Alternatively, the Arduino Portenta H7 board supports WiFi UDP transport, but does not include RTOS support.

The development process involves configuring the device using the Menuconfig tool prior to flashing. As presented in the figure 4, it provides an interactive interface for settings related to target hardware, internal components, compiler options, bootloader, and more. As part of the system, the micro-ROS software extends the basic configuration list with its internal parameters consisting of:

- The micro-ROS Agent IP and port: denoting information where agent is hosted within the network.
- The micro-ROS network interface: selection of the transport setting among the following: WLAN, Ethernet, or custom transport over UART.
- The micro-ROS middleware: selection of available vendor, the package is capable of running two different middlewares compatible with ROS 2 (figure 1):
  - eProsima Micro XRCE-DDS: default, which is was used for testing,
  - embeddedRTPS: an experimental implementation of a RTPS.

Installation and Testing of micro-ROS on ESP32

In this paper, we detail the micro-ROS installation process on ESP32 through the ESP-IDF framework. This involves pre-installing the necessary SDK (Software Development Kit) for further development purposes. This step can be performed via manual installation or, preferably, as an IDE extension for Eclipse or VSCode to ease the process. Once the ESP-IDF tool is present, one can source the environment from terminal using the `source $IDF_PATH/export.sh` command, enabling the use of the `idf.py` build management tool, which was written in Python language.

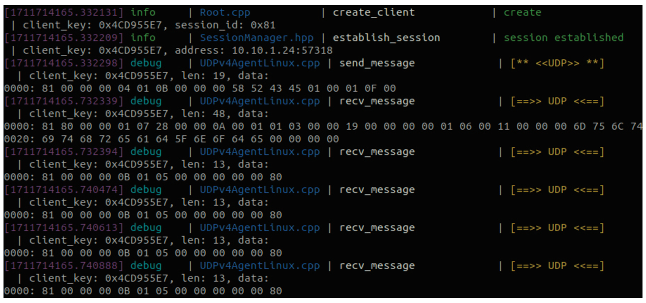
To start with, it is necessary to create a project with the `idf.py create-project <project_name>` command. The micro-ROS system is considered a component of our application and, as such, the subsequent step is to clone the relevant repository [10] directly into the components folder. To match specific ROS 2 distribution, particular branch can be selected denoting a version of ROS; as for the moment, the latest LTS (long-term support) *Humble* was used in the study. The cloned component includes several example applications that can serve as a starting point for further development.

In prior to the building process, the micro-ROS component can be configured within *colcon.meta* file where one can override predefined ROS-specific CMake arguments, such as enabling the trace tools. To build an application, several Python packages are required by the micro-ROS component. Instead of manually installing them, it is suggested to use a Docker container. Docker is a containerization platform that allows for packaging, deploying, and running applications in isolated environments called containers. The container is instantiated from the prepared image `microros/esp-idf-microros:latest` [12], providing a reproducible and isolated environment with all the requirements ready. With the workspace directory mounted to the created container, the entire build and run process can be executed with a single command `idf.py menuconfig build flash monitor`. This command initiates the following steps:

1. Menuconfig: opens up an interactive environment to setup micro-ROS configuration and WiFi credentials under micro-ROS settings. By default, the UDP connection will be used.
2. Building and flashing: compiles the project using the

cross-compilation toolchain, links libraries and dependencies, and generates firmware binaries. Once the process is successful, these binaries will flash onto the device, assuming that the chip is in download mode.
3. Monitoring: provides a way for developers to read the serial output of the device, enabling one to monitor the real-time behavior of an user application.

After successful deployment, the ESP32 is now in the state of an operational standalone system. However, to unlock its full potential, the ESP32 has the capability to connect to a selected WiFi network and search for an Agent at a specified address. Complementing the integration into a scalable system, it is required to run the micro-ROS Agent within a common network on a host device. The recommended approach is to run a container using the official Docker image `microros/micro-ros-agent:humble`, leveraging the host machine's network stack. Once the Agent is up and running, it will wait for a connection, and when acquired, the subsequent debug info will be displayed (figure 5). Now, having an ROS 2 application running on the host computer, it is possible for both devices to share their topics and exchange data. The micro-ROS Agent acts as a proxy, being responsible for interfacing any send messages.



```
[1711714165.332131] info     | Root.cpp          | create_client          | create
 | client_key: 0x4CD955E7, session_id: 0x81
[1711714165.332209] info     | SessionManager.hpp | establish_session     | session established
 | client_key: 0x4CD955E7, address: 10.10.1.24:57318
[1711714165.332298] debug    | UDPv4AgentLinux.cpp | send_message         | [** <<UDP>> **]
  | client_key: 0x4CD955E7, len: 19, data:
0000: 81 00 00 00 04 01 0B 00 00 00 58 52 43 45 01 00 01 0F 00
[1711714165.732339] debug    | UDPv4AgentLinux.cpp | recv_message         | [==>> UDP <<==]
  | client_key: 0x4CD955E7, len: 48, data:
0000: 81 80 00 00 01 07 28 00 00 0A 00 01 01 03 00 00 19 00 00 00 00 01 06 00 11 00 00 00 6D 75 6C 74
0020: 69 74 68 72 65 61 64 5F 6E 6F 64 65 00 00 00 00
[1711714165.732394] debug    | UDPv4AgentLinux.cpp | recv_message         | [==>> UDP <<==]
  | client_key: 0x4CD955E7, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1711714165.740474] debug    | UDPv4AgentLinux.cpp | recv_message         | [==>> UDP <<==]
  | client_key: 0x4CD955E7, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1711714165.740613] debug    | UDPv4AgentLinux.cpp | recv_message         | [==>> UDP <<==]
  | client_key: 0x4CD955E7, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
[1711714165.740888] debug    | UDPv4AgentLinux.cpp | recv_message         | [==>> UDP <<==]
  | client_key: 0x4CD955E7, len: 13, data:
0000: 81 00 00 00 0B 01 05 00 00 00 00 00 80
```

Fig. 5. ROS 2 Agent logs for a message interfacing with micro-ROS system

Testing Bench

The testing bench was prepared for the purpose of exemplifying the use cases of micro-ROS in automatics, robotics, and distributed systems, while showcasing its potential benefits compared to other solutions, such as MicroPython. The experiments aimed to demonstrate the advantages of using micro-ROS for reliable communication in resource-constrained environments. Both environments were evaluated within the same platform configuration; the prepared circuit sketch is presented in the figure 7. Tools and technologies used for testing and development were as follow:

- ESP32-WROOM-32E development board (figure 6),
- DHT22 (AM2302) sensor,
- ROS 2 Humble and micro-ROS Humble distributions,
- PC with Ubuntu 20.04 installed for project building and running ROS 2 Agent,
- Docker version 24.0.5 for containerization,
- MicroPython v1.23.0 for ESP32 [11].

In the context of our example application, we have a system responsible for managing and collecting data from a sensor connected to an ESP32 device. The selected sensor is a DHT22, which provides temperature and humidity measurements. It is compatible with the ESP32 platform and can be integrated using either MicroPython or FreeRTOS software, making it a versatile choice for various IoT applications. This flexibility eases integration with the ROS 2 ecosystem, as existing libraries can be used as a wrapper for the sensor.

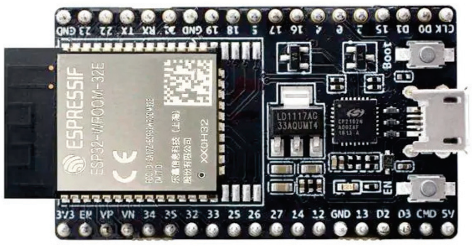


Fig. 6. Version of ESP32 device used for testing and development

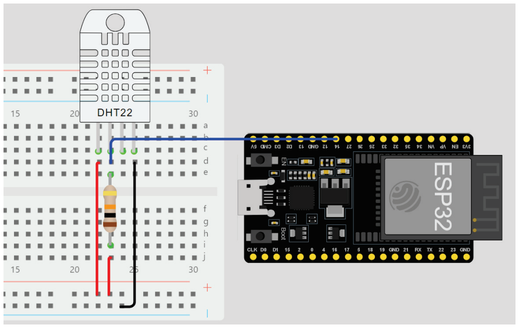


Fig. 7. Testing bench and its wiring

The DHT22 is a simple sensor with well-supported libraries and was chosen to demonstrate micro-ROS capabilities in an accessible manner. The goal was to highlight how micro-ROS can be used to streamline communication and data exchange between distributed systems, even in simple setups. Although more complex sensors such as IMUs could be used, the DHT22 provides a clear and easily reproducible example of micro-ROS in action. This allows us to focus on its key benefits in a straightforward manner, such as scalability, real-time communication, and seamless integration with ROS 2, ensuring that data formatting and communication are aligned with ROS 2 standards.

This setup showcases how micro-ROS and ESP32 can address automation and robotics challenges by enabling efficient communication and integration between distributed modules. For example, sensor data can be used to trigger real-world actions such as controlling HVAC systems based on temperature or activating alarms when humidity exceeds predefined thresholds. Micro-ROS ensures that this process is reliable and scalable, offering a solid framework for data management and communication across devices.

Application Based on micro-ROS System

The application was written with the help of the *rclc*, *freertos*, *uros_network_interfaces*, *zorxx_dht*[13] and *esp32* libraries. The *zorxx_dht* driver was integrated as a component within ESP-IDF, enabling communication with DHT. To aid in the distribution of data for subsequent analysis outside the microcontroller, a node was defined along with the necessary topics within the micro-ROS framework. The node handling was performed as a FreeRTOS task with its own stack size and priority. Concurrently, DHT data reading was also implemented as another FreeRTOSs task. Both tasks were synchronized by a semaphore, as they both access the same data stored locally for reading or updating purposes. FreeRTOS is an integral part of the ESP32 system when micro-ROS is deployed (as it provides POSIX-like interfaces), allowing efficient concurrent task management and essential tools for synchronization. This ensures data consistency, system reliability, and real-time support.

The development started with structuring the message containing DHT reading, suitable for the ROS system. The sensor data reading involves two variables that represent humidity and temperature. Since there is no such direct representation within standard ROS messages, a custom one was created. The process involved creating a new package with dependencies described as on the official site [14]. The new package is meant to be placed in the *extra_packages* folder of *micro_ros_espidf_component* component, as they get copied into the ROS workspace at build time. The content of *DhtData.msg* from custom *dht_sensor* package is presented on the listing 1. The message definition must be present on both sides of the communication; as such, a ROS 2 system needs to have a copy of this package built and sourced to interface with associated topics.

Listing 1. Definition of custom *DhtData.msg*

```
float64 humidity
float64 temperature
```

Listing 2. Publisher of type *DhtData* and *sensor_data* topic name

```
RCCHECK(rclc_publisher_init_best_effort(
  &publisher,
  &node,
  ROSIDL_GET_MSG_TYPE_SUPPORT(
    dht_sensor, msg, DhtData),
  "sensor_data"
));
```

The listing 2 presents the initialization of the publisher along with its configuration. To ensure efficient communication, a QoS profile was set as a predefined best effort policy. This policy prioritizes timely message delivery while maintaining a preference for the most recent data. We selected a message *DhtData* definition which was defined previously. ROS timer was used to periodically publish the most recent data, and the subscriber was used to capture data outside of the microcontroller; both the timer and the subscriber must be added to the Executor for callback handling (listing 3).

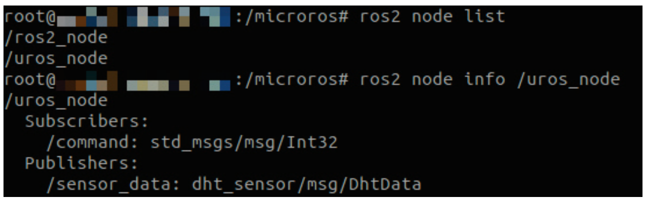
Listing 3. Adding handles to the Executor

```
RCCHECK(rclc_executor_add_timer(&executor, &timer));
RCCHECK(rclc_executor_add_subscription(
  &executor, &subscriber, &recv_msg,
  &subscription_callback, ON_NEW_DATA
));
```

With micro-ROS Agent running, the micro-ROS node and its associated topics become visible within the ROS 2 ecosystem (figure 8). As such, we established connections to these topics by creating the corresponding subscriber and publisher on the host running ROS 2. It is crucial to ensure alignment between the QoS profiles and message types across both ends of the communication channel. It should be underlined that in a practical scenario, ROS 2 can utilize the sensor data received from micro-ROS to generate and publish commands back to the microcontroller, allowing seamless interaction between the two systems (figure 9).

Application Based on MicroPython Embedded System

In addition to the micro-ROS setup, an alternative method was implemented using a MicroPython embedded system for microcontrollers [15]. This firmware is a compact and efficient version of the Python programming language suited for microcontrollers; installing it transforms the



```
root@[redacted]:/microros# ros2 node list
/ros2_node
/uros_node
root@[redacted]:/microros# ros2 node info /uros_node
/uros_node
  Subscribers:
    /command: std_msgs/msg/Int32
  Publishers:
    /sensor_data: dht_sensor/msg/DhtData
```

Fig. 8. Micro-ROS node visible within ROS 2; output of *ros2 node* command line tool

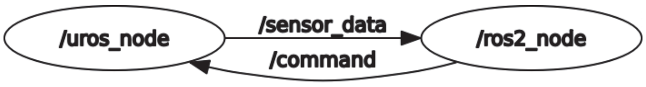


Fig. 9. *Rosgraph* tool output for application sharing nodes in micro-ROS and ROS 2 systems

device into a Python interpreter machine capable of running scripts directly on the hardware. The MicroPython installation process requires a USB connection. After downloading the firmware[11], the *esptool.py* tool can be used to erase the existing flash memory and then to deploy MicroPython onto the device. Then, the REPL (Python prompt) can be accessed.

The alternative approach uses sockets and WiFi to establish communication between a ROS 2 node running on a main computer and ESP32 device with MicroPython installed. The application involves a script periodically reading the sensor data from the ESP32 and transmitting it over a socket connection to the host machine. On the main computer, the ROS 2 node receives the data and propagates it to another node for further processing.

The script is responsible for establishing an ESP32 WiFi connection, configuring a socket server, and obtaining data from a connected DHT22 sensor. The required libraries (*network*, *socket*, *dht*) come pre-installed within the kit. The DHT sensor is initialized with the appropriate model and connected pin. Each sensor reading is formatted as JSON, encoded in UTF-8, and transmitted to the host computer through the socket connection (listing 4). On the host side, a ROS 2 node was written to listen for incoming data on the corresponding socket port, decoding them into known format. Finally, the sensor readings are parsed into a ROS message and then and published to a ROS 2 topic which can be listen to by other nodes.

Listing 4. Sending sensor reads as JSON via socket

```
sensor.measure()
data = {
    'temperature': sensor.temperature(),
    'humidity': sensor.humidity()
}
message = json.dumps(data)
client.send(message.encode())
```

Although this method demonstrates the flexibility and simplicity of using MicroPython for IoT applications, it presents several limitations compared to the micro-ROS approach:

1. Limited ROS integration: with MicroPython, developers must manually handle communication, data serialization, and parsing to establish ROS connectivity. Such setup requires a dedicated node running on the main computer rather than microcontroller, making auto-discovery and dynamic node connection less practical. In contrast, micro-ROS offers built-in support for ROS messages and communication protocols, leveraging the existing DDS system, simplifying data handling, and de-

ploying ROS node directly on the ESP32.

2. Scalability and maintainability: micro-ROS supports advanced ROS 2 features such as QoS settings, timers, and executors, which are not available in the MicroPython approach. These features enhance the scalability and maintainability of the application, allowing detailed control over communication and task scheduling.
3. Real-Time performance: micro-ROS is designed for real-time applications requiring deterministic behavior and low latency. The MicroPython method, while functional, may not meet the performance requirements of time-critical applications due to the overhead of Python interpreted nature and socket communication.

Conclusion

The micro-ROS system acts as a bridge, connecting the ROS framework concepts to microcontrollers through a layered architecture that leverages components from ROS 2. It offers a unified interface for node design, communication, and lifecycle management, facilitating the creation of scalable software with uniform communication mechanisms. The implementation of the system on the ESP-32 platform has been highlighted as a cost-effective and practical solution, especially in relation to connectivity with other devices, as it eases their integration into larger systems. The combination of inherent support for real-time control and capabilities, along with DDS communication mechanisms, promotes edge computing in automatics and robotics by allowing data processing, decision making, and control either directly on the robot or at the network edge. However, this versatility is somewhat limited by the required presence of ROS Agent which acts as a server interacting with DDS.

In contrast, using the MicroPython embedded system as an alternative approach showcases its own strengths, especially regarding flexibility and ease of use for IoT applications. However, it has several drawbacks compared to micro-ROS, such as the need for manual handling of communication and data serialization, restricted ROS integration, and limited scalability, real-time performance, and memory efficiency. These limitations highlight the advantages of micro-ROS, particularly when integrated with other ROS 2 applications, due to integrated support for ROS messages, advanced scalability and maintainability features, and enhanced real-time performance capabilities.

The presented procedure of the system installation and the corresponding test results show that the adoption of microcontrollers provides advantages like reduced power consumption and cost-effectiveness compared to conventional OS-driven devices. This positions micro-ROS as a perfect fit for mobile robots, where the importance of power efficiency and compactness cannot be overstated. Future research plan consists of the integration of the micro-ROS system for a mobile robot platform that can benefit from the new architecture, then investigating the impact of this transition on the robot's performance, power consumption, and overall efficiency. This presents the difficulty of adapting the existing code base to an environment with limited resources, yet maintaining functionality at a comparable level. However, overcoming these challenges will be essential in realizing the full potential of micro-ROS in mobile robotics or automatics applications.

Authors: *M. Sc. Bartłomiej Stadnik, Prof. Artur Wymysłowski, Wroclaw University of Science and Technology, Faculty of Electronics, Photonics and Microsystems, ul. Janiszewskiego 11/17, 50-372 Wrocław, Poland*

REFERENCES

[1] DiLuoffo, V., Michalson, W.R., Sunar, B., Robot Operating System 2: The need for a holistic security approach to robotic architectures, International Journal of Advanced Robotic Systems, 15, pp. 172988141877001, May 2018. `https://doi.org/10.1177/1729881418770011`.

[2] Abdallah, I. B., Bouteraa, Y., Mobayen, S., Kahouli, O., Aloui, A., Amara, M., and Jebali, M. Fuzzy logic-based vehicle safety estimation using V2V communications and on-board embedded ROS-based architecture for safe traffic management system in Hail city *Electronic Research Archive*, 31, 5083-5103, 07/2023. DOI: 10.3934/era.2023260

[3] Ralph Lange, Micro-ROS – bringing the most popular robotics middleware onto tiny microcontrollers [web page] `https://www.bosch.com/stories/bringing-robotics-middleware-onto-tiny-microcontrollers/`. [Accessed on 29 Feb. 2024.].

[4] Staschulat, J., Lütkebohle, I., Lange, R.: The rclc Executor: Domain-specific deterministic scheduling mechanisms for ROS applications on microcontrollers: work-in-progress, 2020 International Conference on Embedded Software (EMSOFT), pp. 18-19, 2020. DOI: 10.1109/EMSOFT51651.2020.9244014.

[5] Micro-ROS, Official site [web page] `https://micro.ros.org/`. [Accessed on 28 Apr. 2024.].

[6] Finocchiaro, F., Garrido-Sanchez, P., Moral Parras, J. A., Micro-ROS on FreeRTOS [web page] `https://www.freertos.org/2020/09/micro-ros-on-freertos.html`. [Accessed on 29 Feb. 2024.].

[7] Staschulat, J., Lange, R., Dasari, D. N.: Budget-based real-time Executor for Micro-ROS, arXiv preprint, arXiv:2105.05590, 2021. `https://arxiv.org/abs/2105.05590`.

[8] Micro-ROS Documentation, Client Library: Execution Management [web page] `https://micro.ros.org/docs/concepts/client_library/execution_management/`. [Accessed on 29 Feb. 2024.].

[9] Espressif, ESP-IDF development framework [web page] `https://www.espressif.com/en/products/sdks/esp-idf`. [Accessed on 28 Apr. 2024.].

[10] micro-ROS component for ESP-IDF, GitHub [web page] `https://github.com/micro-ROS/micro_ros_espidf_component`. [Accessed on 28 Apr. 2024.].

[11] MicroPython firmware, ESP32 / WROOM [web page] `https://micropython.org/download/ESP32_GENERIC/`. [Accessed on 6 July 2024.].

[12] Docker image of micro-ROS component for ESP-IDF, GitHub [web page] `https://hub.docker.com/r/microros/esp-idf-microros`. [Accessed on 28 Apr. 2024.].

[13] esp-idf Digital Humidity and Temperature Sensor Driver, ESP Component Registry [web page] `https://components.espressif.com/components/zorxx/dht`. [Accessed on 6 July 2024.].

[14] Micro-ROS Documentation, How to include a custom ROS message in micro-ROS [web page] `https://micro.ros.org/docs/tutorials/advanced/create_new_type/`. [Accessed on 6 July 2024.].

[15] MicroPython documentation, Overview [web page] `https://docs.micropython.org/en/latest/index.html`. [Accessed on 6 July 2024.].