Mälardalen University
School of Innovation Design and Engineering
Västerås, Sweden

Thesis for the Degree of Master of Science in Engineering - Robotics
30.0 credits

MICRO-ROS FOR MOBILE ROBOTICS SYSTEMS

Peter Nguyen
pnn16004@student.mdu.se

Examiner: Baran Cürüklü
Mälardalen University, Västerås

Supervisor: Jonas Larsson
Mälardalen University, Västerås

Company Supervisor: Jon Tjerngren
ABB Corporate Research, Västerås

10/06/2022

Abstract

The complexity of mobile robots increases as more parts are added to the system. Introducing microcontrollers into a mobile robot abstracts and modularises the system architecture, creating a demand for seamless microcontroller integration. The Robot Operating System (ROS) used by ABB's new mobile robot, the mobile YuMi prototype (mYuMi), allows standardised robot software libraries and packages to simplify robotic creations. As ABB is porting over from ROS1 to ROS2, the ROS2 compatible Microcontroller Robot Operating System (micro-ROS) will be incorporated into the system to smoothly integrate microcontrollers into mYuMi. In order to display the validity of micro-ROS, this project used tracing and latency measurements with external applications to test the remote communication between mYuMi using ROS2 and microcontrollers using micro-ROS, with three different microcontrollers tested. The communication was evaluated in different scenarios with a test bench, using ping pong communication to get the round-trip time. A reinforcement of the test results was presented by demonstrating the use of micro-ROS live in a prototype developed, where mYuMi controlled a 1D rangefinder and an RC servo motor by utilising two microcontrollers. The results concluded that the micro-ROS delay could be analysed in theory with external applications, equivalent micro-ROS functionality should apply to most microcontrollers, and the test results and prototype displayed the potential of micro-ROS matching ROS2 in terms of delay and stability.

Acknowledgements

The contribution of the people that helped and advised me has been essential for the success of this project. My supervisor at ABB, Jon Tjerngren, has been involved during every step of this process, from planning ideas to implement, to handling ABB formalities, and instructing me on how to work with ABB's mobile YuMi prototype. My appreciations goes out to Jon Tjerngren and any other people at ABB that helped me.

A big thank you to my MDU supervisor, Jonas Larsson, for guiding me throughout this thesis with report writing and university requirements, and for reassuring me that I was on the right track and on pace with the dates set. I am likewise thankful for the extra help received from other teachers and students at MDU.

I also want to express my gratitude towards the support at micro-ROS and Percepio, and anyone else online who assisted me with my many technical problems.

Contents

List of Figures

List of Tables

MDU	Mälardalen University
OFERA	Open Framework for Embedded Robot Applications
RT	real-time
OS	operating system
RTOS	real-time operating system
ROS	Robot Operating System
ROS1	Robot Operating System 1.0
ROS2	Robot Operating System 2.0
micro-ROS	Microcontroller Robot Operating System
RMW	ROS middleware
MCU	microcontroller unit
PC	personal computer
mYuMi	mobile YuMi prototype
F767ZI	STM32F767ZI
32S	NodeMCU-32S
Due	Arduino Due
IDE	integrated development environment
QoS	quality-of-service
EOL	end-of-life
LTS	long-term support
RQ	research question
DOF	degrees of freedom
RTT	round-trip time
RTPS	Real-Time Publish-Subscribe
CTF	Common Trace Format
DDS	Data Distribution Service
DDS-XRCE	DDS for extremely resource-constrained environments
Micro XRCE-DDS	Microcontroller DDS-XRCE
SBC	single-board computer
SLAM	Simultaneous Localisation and Mapping
AMCL	Adaptive Monte Carlo Localisation
SRT	Secure Reliable Transport
WCRT	worst-case response time
SADF	Scenario Aware Data Flow
SB	Shadow Builder
TFA	Trace Framework Abstraction

1 Introduction

The creation of mobile robots is a complex task. Components such as sensors, software, batteries, and motors need to cooperate in a structured manner for a system to work properly. If everyone used the same tools, components, and interfaces to create robots, it would allow people to smoothly integrate other people's work into their own and instead focus on new ideas and projects.

The free, open-source software Robot Operating System (ROS) is a middleware framework for standardising the creation of robotic systems, allowing developers to create deployable packages and libraries for widespread usage, facilitating the process of assistance and cooperation [1].

The first version of ROS, Robot Operating System 1.0 (ROS1), was released in 2007 and has built a large community over the years. Even though the community has worked hard to integrate new features and updates, the base platform's flaws are becoming increasingly apparent [2]. The final distribution of ROS1 has its end-of-life (EOL) in May 2025 [3].

A new version of ROS was released in late 2017, named Robot Operating System 2.0 (ROS2) [4]. ROS2 ports over the core functionalities and features of ROS1 and addresses its drawbacks, such as real-time (RT) and reliability [5]. Though the ROS2 functionalities, packages, and libraries are well-suited for a personal computer (PC), this is not the case for limited embedded devices.

As the complexity of mobile robot architecture is steadily increasing, the demand for flexible microcontroller unit (MCU) integration is higher than ever before. MCU-based solutions allow the robot structure to be handled in multiple parts while taking a load off the main processing unit, increasing the system's modularity and making for portable and less expensive solutions.

Microcontroller Robot Operating System (micro-ROS) brings the main concepts of ROS2 to MCUs as a low-power and lightweight rendition [6]. By using microcontrollers with micro-ROS, calculations and communication are internal to the MCUs, meaning components and microcontrollers are readily replaceable. micro-ROS is also seamlessly integrated into all ROS2 systems.

The development of micro-ROS is funded by the EU in a project called Open Framework for Embedded Robot Applications (OFERA) [7]. A funding period was set from January 2018 to a one-year extended December 2021. The project is a collaboration between 6 companies to bring MCUs to ROS2. This consortium consists of eProsima, Acutronic robotics, PIAP, BOSCH and FIWARE, with eProsima being the lead developers.

1.1 Aim

The purpose of this research was to investigate the quality-of-service (QoS) and capabilities of micro-ROS in mobile robotic systems using different microcontrollers. The focus was on analysing and testing the communication between MCUs with micro-ROS and a PC with ROS2. A prototype developed proved that the connection works in practice.

1.2 Motivation

This project is a thesis proposal from ABB to evaluate and present micro-ROS when used together with a version of the ABB robot YuMi[1], helping ABB decide the usability of micro-ROS in their systems. The project is also conducted as a 30 hp master's thesis for Mälardalen University (MDU).

1.3 Scientific Contribution

An analysis similar to this is made in a recent micro-ROS benchmark by OFERA [8]. But unlike this project, the evaluation by OFERA is between two microcontrollers of the same type, with a PC in between acting as the agent [9]. Multiple MCU and communication parameters are measured by using hardware functions and internal kernel to trace the microcontrollers.

In this project, different microcontrollers are analysed by tracing both the MCU and the ROS2 side with external applications, focusing on QoS parameters. The communication middleware is also directly monitored by an application displaying live statistics, providing an internal validation of the communication. In this setup, the microcontroller is readily replaceable, the installation is more accessible, and data is effortlessly visualised on a PC.

[1] https://cobots.robotics.abb.com/en/robots/yumi/

2 Problem Formulation

This research is in collaboration with ABB to introduce micro-ROS into their mobile robots. ABB is currently in the process of porting over from ROS1 to ROS2. To modularise and abstract the system architecture, MCUs with micro-ROS will control external sensors and actuators. Due to uncertainties about the reliability and validity of micro-ROS, ABB set the following objectives:

- Deploy micro-ROS on a STM32 microcontroller to evaluate the QoS and analyse communication with a ROS2 PC.
- Extend the deployment of micro-ROS to ESP32 and Arduino type MCUs.
- Develop a Sense-Plan-Act prototype using ROS2 with micro-ROS and visually demonstrate it in action.

From the problem formulation and the objectives set, the following research questions (RQs) were formed:

RQ1. What are the limitations to communication analysis between micro-ROS and ROS2?

RQ2. Can equivalent forms of micro-ROS functionality be deployed on STM32, ESP32, and Arduino development boards?

RQ3. Will micro-ROS be able to match ROS2 in terms of stability and delay when used jointly in a Sense-Plan-Act prototype with the new ABB robot, the mobile YuMi prototype?

3 Background

Background knowledge for related concepts and tools used are provided in this section. A basic understanding of ROS, microcontrollers, and tracing is required to comprehend the rest of the thesis work presented.

3.1 Quality-of-service

The QoS for communication is the performance evaluation of different parameters, with features such as delay and stability generally considered [10]. In this context, the definition of *delay* is the travel time for data between endpoints, while *stability* is the delay variation over a period of time. Delay and stability can also be referred to as latency and jitter, respectively.

3.2 Operating system

There are many variations of operating systems (OSs), but it is generally described as a computer program that administers system software and hardware and facilitates general utilities [11]. The most well-known type of OSs are the general-purpose ones used in PCs, such as Windows, macOS and Linux. Most appliances with a computer also use an OS, e.g. consoles, TVs, and mobile phones. There are also OSs for specific use cases, like RT and embedded systems.

3.3 Real-time operating system

A real-time operating system (RTOS) is a deterministic scheduler following critical timing constraints [12]. The task scheduling most commonly used is preemptive and event-driven, i.e. tasks with higher priority levels take precedence and allow interrupting the lower ones for execution time. One of the most popular RTOSs available is FreeRTOS[2]. It is a free and open-source RTOS with low memory consumption, mainly used in microcontrollers. FreeRTOS is written in C and is available on several platforms, including STM32, ESP-IDF, and micro-ROS.

3.4 Robot Operating System

ROS implements the basic functionalities of an OS and focuses on deadlines and response time like an RTOS, but does not fit either category [1]. It is better described as a collection of tools, libraries, and packages for developing robotic software. ROS introduces the concepts of nodes, topics, services, parameters, actions, and launch files, being essential parts of ROS1 and ROS2 [4].

The difference between ROS1 and ROS2 is the node architecture [5]. ROS1 utilises a master node as the central hub for communication and uses a custom transport protocol and node discovery method. ROS2 removes the dependency on the master node, preventing master node problems from incurring system-wide crashes. In place of custom message handling, ROS2 uses the industry-standard message protocol Data Distribution Service (DDS)[3] as its ROS middleware (RMW).

Updates to ROS are released as distributions [3]. The packages in a distribution follow the same version to make it as stable as possible, and any further updates are pushed onto the next release. ROS2 currently has two stable releases, Foxy Fitzroy and Galactic Geochelone [13]. Foxy is the long-term support (LTS) release with its EOL in May 2023, and Galactic is the latest release with its EOL in November 2022. A new distribution is planned to release May 23rd 2022.

3.5 Microcontroller Robot Operating System

micro-ROS is a microcontroller version of ROS2, prioritising memory efficiency and RT [6]. The three POSIX-based RTOSs, FreeRTOS, Zephyr and NuttX, are supported by micro-ROS [14], with the non-OS systems Arduino and Mbed also supported. The RMW for micro-ROS is built upon the same base as ROS2, resulting in micro-ROS being readily incorporated into any ROS2 system.

There are multiple ways of putting micro-ROS onto MCUs. The general method is to create a micro-ROS workspace and compiling it with ROS2 [15]. However, this only works with officially

[2]https://www.freertos.org/RTOS.html
[3]https://www.dds-foundation.org

and community supported hardware, which are limited [16]. Another option, depending on the board, is to use external tools supported by micro-ROS; these are Espressif, Zephyr, Arduino and STM32Cube [17]. If the above does not work, a more complicated approach is available by generating micro-ROS as a custom static library [18].

3.6 ROS concepts

Nodes are processes, usually in charge of one given task to increase modularity [19]. The nodes can communicate with multiple other processes simultaneously to receive, process, and send data. Mobile robotic systems with ROS2 use a multitude of nodes that needs to work in unison. A ROS2 script can contain multiple nodes.

The most common type of communication between nodes is through topics [20]. Topics are hubs receiving messages published by nodes, broadcasting the messages to any nodes subscribed to the selected topic. These nodes are called publishers and subscribers, respectively. Topics are specified by name and optionally a namespace that can be remapped[4] to connect separate systems and repurpose nodes.

Services are functions requested by service clients, able to input and output data [21]. A service can serve many clients simultaneously, similarly to topic subscribers, but unlike topics, there is only one service server, which is active only when serving requests.

A launch file can start-up and initialise multiple processes and executables simultaneously, including nodes and other launch files [22]. Launch files allow for the convenient configuration of start-up settings such as parameters and remapping. Either Python, XML, or YAML are used to write launch files, with the possibility to change between these languages when using multiple launch files.

QoS settings in ROS establish the communication aspects between nodes [23]. These settings are configured using policies. One example is the reliability policy, with the two different modes, best effort and reliable. Best effort is similar to UDP in that messages are not guaranteed to arrive. Reliable is the opposite, ensuring the arrival by resending any failed messages, like TCP. When all the policies are put together, a QoS profile is formed, with profile presets available for different use cases. The sensor data profile is used when messages must arrive in time, and only recent data is desired, utilising the best effort policy. The default profile uses the reliable policy.

3.7 Communication

DDS is a Real-Time Publish-Subscribe (RTPS) connectivity framework created by the Object Management Group (OMG). It provides a portable network for use in mission-critical areas. A lightweight implementation was created for DDS for extremely resource-constrained environments (DDS-XRCE)[5].

The Microcontroller DDS-XRCE (Micro XRCE-DDS)[6] protocol allows low-power device clients to communicate with DDS servers through an Agent. Micro XRCE-DDS was developed by eProsima as a microcontroller version of DDS-XRCE and is used in micro-ROS.

A micro-ROS Agent[7] sets up a server between micro-ROS nodes and DDS networks. It allows the nodes to send and receive messages by tracking and advertising the nodes to ROS2. The micro-ROS Agent is essentially a Micro XRCE-DDS Agent created inside a ROS2 node.

[4]https://docs.ros.org/en/foxy/How-To-Guides/Node-arguments.html#name-remapping
[5]https://www.omg.org/spec/DDS-XRCE/
[6]https://www.eprosima.com/index.php/products-all/eprosima-micro-xrce-dds
[7]https://github.com/micro-ROS/micro-ROS-Agent

3.8 YuMi

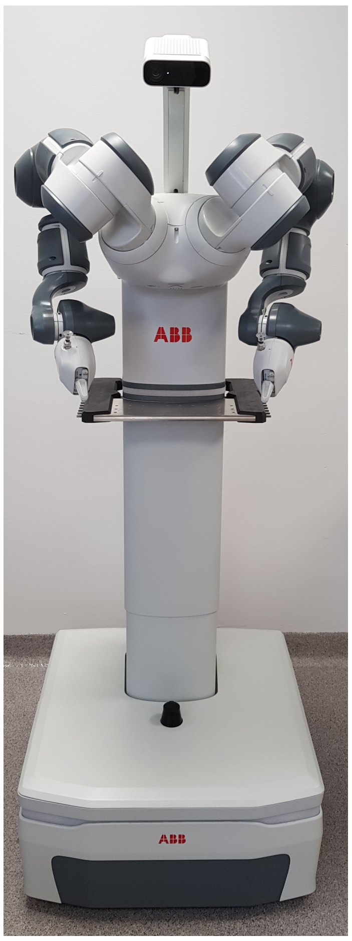


Figure 1: The mobile YuMi prototype (mYuMi), a second generation of the mobile YuMi platform. The upper body and the arms are a part of YuMi IRB 14000.

A new type of robot was presented in 2015 by ABB, the YuMi IRB 14000 [24]. The IRB 14000 is a collaborative robot designed to work alongside people safely and reliably, using one or two limbs with 7 degrees of freedom (DOF) to emulate human-like arm movement. In 2019 this would be taken further by integrating a modified version of IRB 14000 on an autonomous robot vehicle, creating the mobile YuMi platform. The mobile platform is expected to play an essential role within the future medical and laboratory industry. ABB's second generation of the mobile platform is currently in the works, the mobile YuMi prototype (mYuMi), seen in figure 1. mYuMi communicates using ROS and has most of its functionality transferred over from ROS1 to ROS2. The mYuMi robot is equipped with the following actuators:

- IRB 14000:
  - 7DOF arms $\times 2$
- SmartGrippers $\times 2$:
  - Finger servos $\times 2$
  - Vacuum modules $\times 2$
- Mobile base:
  - Omnidirectional wheels $\times 4$

  - Telescopic pillar

Using these sensors:

- Microsoft Azure Kinect RGB-D camera
- Texas Instrument IWR6843AOP radar
- SICK TiM7781S 2D lidars ×2
- Cognex AE3 2D cameras ×2

Note: The radar is not yet integrated into the ROS2 system, and the 2D cameras are inside the SmartGrippers.

3.9 Microcontroller units

STM32 refers to STMicroelectronics' 32-bit microcontrollers using the ARM Cortex-M type processors [25]. The variations of the MCUs are grouped into high performance, mainstream, ultra-low-power, and wireless. The naming scheme used indicates the type and core of the microcontroller, e.g. STM32F3[8] are the mainstream microcontrollers using Cortex-M4 processors, and STM32F7[9] are the high performance microcontrollers using Cortex-M7 processors. Development boards also have a full name, such as STM32F767ZI (F767ZI)[10], indicating line, number of pins, and flash size.

Espressif's ESP32 type microcontrollers are popular due to built-in Bluetooth and Wi-Fi connectivity [26]. ESP32 also has access to SPI, SDIO, I2C and UART communication, making integration into most systems possible. There are many variations of ESP32 type MCUs created by different manufacturers. One such example is the NodeMCU-32S (32S)[11] by Ai-Thinker, with functionality and hardware similar to Espressif's ESP-WROOM-32[12].

Arduino is a company developing software and hardware for microcontrollers [27]. Arduino can also refer to the microcontrollers and the programming language designed. The Arduino language uses C and C++, used primarily in non-OS environments. Various Arduino type MCUs are available, with notable variations in their specifications. One example is the Arduino Due (Due)[13], the first Arduino board to use a 32-bit ARM processor. It uses an Atmel SAM Cortex-M3 type processor, suitable for larger projects.

3.10 micro-ROS integration

To use micro-ROS in STM32 microcontrollers, micro-ROS utilities for STM32CubeMX[14] is recommended [17]. The utilities use a Dockerfile[15] to statically compile micro-ROS, allowing all STMicroelectronics boards virtual support of micro-ROS. Currently, FreeRTOS is required for the utilities to work.

For ESP32 microcontrollers, micro-ROS can be compiled with ESP-IDF as a component[16]. The component is recommended over the general micro-ROS method due to better optimisations for ESP32. Users can choose whether to include FreeRTOS.

Arduino projects using no OS can use micro-ROS as a precompiled library[17]. There are several different boards supported and some maintained by the community. It is possible to use the library inside PlatformIO[18], making the setup much more accessible.

[8] https://www.st.com/en/microcontrollers-microprocessors/stm32f3-series.html
[9] https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html
[10] https://www.st.com/en/microcontrollers-microprocessors/stm32f767zi.html
[11] https://docs.ai-thinker.com/en/esp32/boards/nodemcu_32s
[12] https://www.espressif.com/en/products/modules/esp32
[13] https://docs.arduino.cc/hardware/due
[14] https://github.com/micro-ROS/micro_ros_stm32cubemx_utils
[15] https://docs.docker.com/engine/reference/builder
[16] https://github.com/micro-ROS/micro_ros_espidf_component
[17] https://github.com/micro-ROS/micro_ros_arduino
[18] https://platformio.org

3.11 Tracing

Tracing is a method of recording process execution data and is commonly used for operating systems [28]. The purpose of tracing is to monitor a system while incurring as little overhead as possible to not affect the results in, for example, benchmarking. A popular tracing tool is the open-source Linux tracer LTTng[19]. LTTng traces the OSs user libraries, user applications, and kernels with insignificant overhead. LTTng is user friendly and generates traces in the Common Trace Format (CTF)[20]. CTF is a trace format standard written in binary, compatible with applications written in C and C++. The supported tool for reading CTF traces is Babeltrace[21], converting data to readable information.

3.12 Analysis tools

Tracealyzer[22] by Percepio, is an RT trace visualiser compatible with many platforms. Traces can be recorded with either snapshots or streaming and works with most transports. One of the platforms supported is FreeRTOS, offering direct support for STM32 and ESP32 microcontrollers.

Ozone by SEGGER is a visual embedded debugger with RT functionalities. Debugging is possible in environments using C or C++, working with RTOS systems as well as bare-metal ones. Ozone is compatible with most hardware, including ARM-based MCUs like F767ZI and Due. A convenient feature in Ozone is monitoring and displaying variables during run-time.

The ros2_tracing[23] package logs ROS2 communication data using LTTng together with Babeltrace. ros2_tracing provides tools to post-process and understand the execution of ROS2, and data can conveniently be visualised by external tools. More information in section 4.4.

eProsima's Fast DDS Monitor[24] visualises communication in RT between publishers and subscribers on a DDS network. The graphical interface can monitor several communication characteristics and statistically pre-process the data for display.

[19]https://lttng.org
[20]https://diamon.org/ctf
[21]https://babeltrace.org
[22]https://percepio.com/tracealyzer
[23]https://gitlab.com/ros-tracing/ros2_tracing
[24]https://www.eprosima.com/index.php/products-all/eprosima-fast-dds-monitor

4 Related Work

This section covers the related works in terms of State-of-the-Art and State-of-Practice, with each work categorised into different areas. A discussion of the related works is made in subsection 6.1.

4.1 Mobile Robotics

Phueakthong et al. implemented an autonomous differential drive robot [29]. The system used ROS2 and micro-ROS to communicate and focused on using inexpensive embedded components. The autonomous part was fully accomplished by utilising ROS2 package functionality. A Raspberry Pi 4 single-board computer (SBC) was used as the central processor, installed with ROS2 Foxy and Ubuntu MATE 20.04. The SBC was connected to a YDLIDAR X4 laser scanner, Raspberry Pi Camera Module V2, and Raspberry Pi Pico RP2040 MCU. The mobile robot used a three-layer structure, with the laser scanner on top to collect distance data. The SBC and the camera resided in the middle layer, with the camera at the front to observe the terrain. At the bottom layer, the MCU controlled an encoded motor connected to two wheels. The MCU received instructions from the SBC to set a motor velocity and then sent back the wheel odometry data to the SBC. The MCU used micro-ROS to communicate via topics to the ROS2 side. The Simultaneous Localisation and Mapping (SLAM) was accomplished using the ROS2 packages Cartographer, Adaptive Monte Carlo Localisation (AMCL), and Navigation2. A static grid map was generated using laser scan and odometry data with Cartographer. AMCL approximated the robot's pose with a particle filter, taking in the map, sensor, and odometry data. The SLAM was completed with Navigation2, which plans a path and publishes the corresponding speed to the motor topic. The path planner used A* global planning to handle known obstacles and DWB local planning for dynamic interferences.

4.2 Mathematical Validation

A model of Micro XRCE-DDS and an extensive analysis is presented by Dehnavi et al. [30], identifying issues with timings, available analysis methods, and dependencies in portable DDS variations. So far, this is the first and only comprehensive evaluation of Micro XRCE-DDS in RT systems. In addition, a bare-metal adaptation of Micro XRCE-DDS is suggested to combat these issues. The adaptation is used with a custom platform known as CompSOC on a Xilinx PYNQ-Z2 board with Ubuntu. The Micro XRCE-DDS Agent is hosted by a soft RT Cortex-A9 ARM processor on Secure Reliable Transport (SRT) Linux, while multiple clients run on parallel hard real-time RISC-V processors. Hypotheses claiming that the clients are guaranteed the worst-case response time (WCRT) and the Agent to encounter probabilistic response time in SRT Linux are proven using the model and the analysis. The modelling is split into two parts. Scenario Aware Data Flow (SADF) is used to probabilistically model Micro XRCE-DDS and its dynamism in separate scenarios, with scenarios being publisher, subscriber, and application (publisher and subscriber). The model allows the difference in response time, points of failure, and data loss to be examined while preserving determinism for timing analysis. Using Markov chains, a long-term throughput analysis was accomplished by modelling the SADF execution scenario switching. The Markov model computed the scenario-throughputs independently to be used as reward functions. The performance evaluation proposed using the WCRT of each component in the SADF models, as the values could be used in the long-term throughput analysis. The results of the suggested Micro XRCE-DDS adaptation conform to the proposed evaluation.

4.3 Real-Time Evaluation

An evaluation of ROS2 RT performance was conducted by Puck et al. [31]. Analysis of time synchronised publisher and subscriber nodes in RTLinux was used to determine latency, jitter, and response time. Instead of a mathematical validation of the system, a one-week stability evaluation was conducted, with measurements externally validated with an oscilloscope and the function stack trace call analysed with ros2_tracing. The experimental results indicate that ROS2 fulfils hard RT conditions when specialised configurations are used. The configurations and methods to achieve RT in ROS2 include changing the DDS RMW, swapping the default allocator, using the sensor data QoS profile, hard coding node priority, and mapping and pre-allocating memory.

4.4 System Analysis

In a paper by Bédard et al. [32], the ros2_tracing tool is explained and justified. The primary justification is the use of less runtime overhead than other solutions, with better use in RT analysis. A more extensive evaluation is also supported with the built-in functionality of instrumentation, usability tools, and test utilities. This paper states other methods are either too specific in their analysis or too hard to use, with ros2_tracing providing execution traces that are conveniently converted into statistics and graphs. The capabilities of ros2_tracing were tested with the similar tool performance_test[25], benchmarking the overhead for end-to-end latency. ros2_tracing was also tested on a ROS2 application replicating autonomous vehicle control. The callback execution was calculated on a node subscribed to several topics, then displayed as durations and intervals. This was combined with ROS2 and OS trace data using LTTng[19], to visualise thread states over time. The experimental results complied with the claims made.

4.5 Tool Comparison

A case study and a new means of tracing ROS2 are introduced by Li et al. [33]. The following ROS2 tracing tools were compared: ros2_tracing, performance_test, ros2_performance[26], and buildfarm_perf_test[27]. ros2_tracing can measure and visualise callback execution and message parameters but only indirectly evaluate DDS. performance_test determines most attributes of ROS2 topics but cannot handle more than one publisher, meaning end-to-end latency cannot be measured. ros2_performance can analyse regular nodes and topics specified, but not any custom nodes. buildfarm_perf_test analyses the CPU and packets in RMW implementations, but since results vary between RMWs, end-to-end latency cannot be measured in ROS2. The comparison determined the best tool to be ros2_tracing. A deeper investigation into ros2_tracing was made to find other limitations and possible solutions, of which two major problems were discovered. ros2_tracing can only determine the callback execution for all running nodes, not any specific ones, and the delays between callbacks or nodes cannot be estimated. A new method based on ros2_tracing, Autoware_perf, is proposed to solve these problems. The first solution is to inject a function that selects between nodes before calculating the callback. The second problem is handled by tracing multiple event points in the underlying code. However, Autoware_perf is customised for a specific application and is not open source yet.

4.6 Performance Benchmarking

This part summarises the RT analysis and the tools used in the consecutive benchmark deliverables by PIAP for the OFERA micro-ROS project [34]. The RT benchmarking methodology was to trace the MCUs and the RTOS using hardware functions and internal kernel, logging the data to a console, and then externally validating with an oscilloscope. The RT performance of micro-ROS was measured in terms of the min, max, and average of both latency and bitrate, using a ping pong application to calculate the round-trip time (RTT). Determinism was evaluated using the task scheduler, with one task being the ping pong application, a second task as idle, and a third task running with low priority. Communication was initially set up to be amongst one Olimex STM32-E407 and a PC, but was later changed to be between two Olimex STM32-E407, with an Agent PC mediating. The communication mediums measured were ethernet (UDP), serial (UART), and 6LoWPAN. The benchmarking tool used was custom made for this analysis, the Shadow Builder (SB). SB is a code instrumentation tool hosting other tools and applications. The part of SB that utilises the tool plugins is the Trace Framework Abstraction (TFA), overseeing and passing along plugin tags for parsing and compilation. TFA is set up so that plugins can be specified by the user. CTF was one plugin used to benchmark, running on top of NuttX RTOS by fetching data traces from user-specified probes.

[25] https://gitlab.com/ApexAI/performance_test
[26] https://github.com/irobot-ros/ros2-performance
[27] https://github.com/ros2/buildfarm_perf_tests

5 Limitations

Both ROS2 and micro-ROS are still relatively early into their development, with micro-ROS being the newer one. The documentation for micro-ROS leaves much to be desired, as seen from the lack of recent scientific micro-ROS papers[28]. If the user gets stuck on an issue, it can take too much time to solve if no external help is received. From what is currently known, no one at MDU or ABB has extensive knowledge of micro-ROS.

There are features and functionalities in ROS2 Galactic yet to be implemented in the Galactic version of micro-ROS, such as DDS-security, logging, and parameter clients [35]. As mentioned in 4.2, there is a lack of a method for analysing micro-ROS. This project instead monitors the MCU and ROS2 data flow, meaning the micro-ROS parameters are not explicitly examined, leading to uncertainties about the QoS.

The microcontrollers have different limitations depending on the inherent hardware, software, and progression of the MCU-specific micro-ROS integrations. The F767ZI does not have support for UDP in micro-ROS[14] and cannot use the micro-ROS library for Arduino[17], like 32S and Due can. The advantages F767ZI has are the most powerful hardware[10], more software freedom with STM32CubeIDE, and built-in debugger, making it easier to debug and monitor. 32S has access to the micro-ROS component for ESP-IDF[16], the micro-ROS integration with the most progression, and can use Wi-Fi and Bluetooth, but is the least powerful hardware-wise out of the three [26]. Due cannot use FreeRTOS or other RTOSs like the other two, making it hard to monitor, and it also has the least software freedom due to the micro-ROS library for Arduino being the most restrictive. F767ZI and 32S also support being compiled with ROS2, unlike Due [16].

[28]https://scholar.google.com/scholar?as_ylo=2021&q=micro-ros&hl=sv&as_sdt=0,5

6 Method

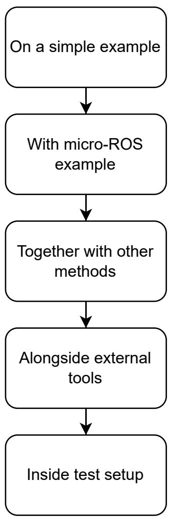


Figure 2: Test case scenarios for new methods.

The basis for this project is the case study of the current State-of-Practice and State-of-the-Art. From the related works and the objectives set, a problem formulation was created, producing RQs to guide the selection of methods. The steps below summarise the research methodology with the waterfall model [36]:

1. Identify problem or feature needed.
2. Find and examine scientific papers or books on the topic.
3. Investigate similar methods, solutions, or implementations.
4. Discuss feasibility and relevance with supervisor.
5. Confirm compatibility with the test cases in figure 2.
6. Establish inclusion and document it.

6.1 Reasoning

With inspiration taken from the work by Phueakthong et al. [29], the prototype uses ROS2 and micro-ROS in a mobile robot, connecting inexpensive components via MCUs. Instead of using ROS2 packages to navigate, this project uses simple custom functions and existing functionality to move the robot, as the Sense-Plan-Act is mostly a proof of concept for micro-ROS. Third-party navigation is not a focus since ABB already has an existing implementation using the same packages as Phueakthong et al.

Dehnavi et al. [30] and Puck et al. [31] use contrasting methods of RT analysis. Dehnavi et al. uses models and formulas to validate their system, while Puck et al. uses testing with a one-week stability evaluation and external oscilloscope validation to determine latency, jitter, and response time. Since this project needs to display micro-ROS usability with the prototype, the methods used by Puck et al. are implemented into this project, as focusing on testing seems like the more appropriate choice.

The ros2_tracing tool used by Puck et al. is discussed by Bédard et al. [32] and Li et al. [33]. Bédard et al. explains the functionality of ros2_tracing and justifies it over other existing methods. Li et al. selected ros2_tracing as the best among current ROS2 tracing tools, but a case study reveals several limitations of ros2_tracing. The main problem is that ros2_tracing cannot directly measure the DDS RMW, meaning data sent to the MCUs are not explicitly evaluated. This project's solution is utilising Fast DDS Monitor to analyse the transmission delay of ROS2.

Another type of analysis is covered by the deliverables in section 4.6 Performance Benchmarking [34], leaning more towards the work by Puck et al. than Dehnavi et al. Like Puck et al.'s project, an oscilloscope is used for external validation, which this project also utilises. Other methods inspired by Performance Benchmarking are microcontroller tracing, assessing scheduler determinism, console logging, and RTT measurements with ping pong, being the related work most similar to this project. However, key differences are the tracing and communication methods. Performance Benchmarking uses code injection to manipulate hardware functions and internal kernel, while this project utilises hardware capabilities with external applications to trace the MCUs, which is easier to apply on different microcontrollers. In Performance Benchmarking, communication is analysed between two MCUs whereas this project does it between MCU and PC, which Performance Benchmarking did initially. MCU to MCU most likely gets a better estimation of micro-ROS, but since this project researches the use of micro-ROS in a real robot, analysing communication between MCU and PC seems more appropriate.

6.2 System

The majority of the robot architecture has already been decided by ABB, but some additional elements are introduced in this project. Relevant parts of the existing robot and new components are presented here.

6.2.1 Hardware

The mobile robot this thesis worked with is mYuMi. mYuMi uses an Intel NUC PC[29] as the central processor for all communication. Linked to the PC are F767ZI and Due microcontrollers. The Due handles the internal LED-loops in mYuMi while F767ZI handles the complex computations, such as kinematics and control theory. Instructions from the F767ZI are sent to four STM32F3s in charge of mYuMi's omnidirectional wheels. A new microcontroller introduced into mYuMi is 32S, an ESP32. It uses its built-in wireless capabilities to connect to mYuMi remotely. External components tested in this project are a 1D rangefinder[30] and an RC servo motor[31].

6.2.2 Software

The mYuMi PC uses ROS2 Galactic with Ubuntu 20.04 LTS (Focal Fossa). The initial plan was to use Foxy, but Galactic was chosen due to better stability and robustness when using third-party navigation. This version of Ubuntu Linux was chosen because it is recommended for both the Galactic and Foxy versions of ROS2. Integrated into the system is the Galactic version of micro-ROS, connecting and controlling external sensors and actuators through MCUs.

6.3 Tools

The tools covered in section 3 are used to program and analyse the system. The micro-ROS specific tools build and flash the applications onto the MCUs, while Tracealyzer and Ozone live monitors the MCU behaviour. To visualise the communication in RT, Fast DDS Monitor pre-processes the data for statistical analysis. The post-processing is then handled with ros2_tracing by analysing the recorded ROS2 trace and visualising any communication aspects.

micro-ROS is tested at a desktop setup using F767ZI, 32S, and Due microcontrollers. A custom ABB PC with 64 GB RAM is installed with the tools and the system software, connected to the MCUs via USB and the high-speed debugger J-Link[32]. The oscilloscope used was Picoscope 3206[33].

Since Tracealyzer provides system-wide tracing with FreeRTOS, it is put on F767ZI and 32S as they are directly supported. Due is not compatible with Tracealyzer due to not using an OS, so Ozone is used instead as it provides similar capabilities and functions on non-OS systems.

[29] https://www.intel.com/content/www/us/en/products/details/nuc.html

[30] http://www.robot-electronics.co.uk/htm/srf04tech.htm

[31] https://hitecrcd.com/products/servos/sport-servos/digital-sport-servos/hs-5495bh-hv-digital-karbonite-gear-sport-servo/product

[32] https://www.segger.com/products/debug-probes/j-link/

[33] https://www.picotech.com/oscilloscope/3000/usb3-oscilloscope-logic-analyzer

6.4 Setup

Communication for the MCUs follows the same structure for sensors and actuators, with one micro-ROS node using best effort QoS policy publishers and subscribers to communicate. The ROS2 nodes use the sensor data QoS profile and act as mediators between micro-ROS and the nodes in mYuMi. A launch file initialises and configures the nodes and starts the micro-ROS Agent so the micro-ROS nodes can connect to the ROS2 network in mYuMi.

An analysis test bench is set up to test all the MCUs and components together using different transmission frequencies, conducted in a laboratory where mYuMi is stationed. The testing process is simplified by having the MCUs connect to the mYuMi PC from the desktop setup instead of integrating everything into mYuMi. This connection is set up over UDP, linking the components remotely to mYuMi.

Without modifications, connecting micro-ROS via UDP is possible if the board is supported and has built-in connectivity. 32S is connected with Wi-Fi since the micro-ROS ESP-IDF component supports UDP. F767ZI has an Ethernet socket, but the STM32 micro-ROS integration does not support UDP. In contrast, the micro-ROS integration for Arduino supports UDP, but Due does not have built-in connectivity. Instead, the boards are connected to mYuMi via the desktop PC by rerouting the serial connection over IP, i.e. port forwarding.

6.5 Integrated development environment

The MCUs are programmed inside integrated development environments (IDEs) able to build, flash, and debug the firmware of the projects. For F767ZI, STM32CubeIDE[34] is used, as it is compatible with the micro-ROS utilities[14] and is a convenient Eclipse-based[35] environment with support for all STM32 microcontrollers.

32S is configured inside Eclipse using the ESP-IDF Eclipse Plugin[36] to support ESP-IDF-based projects, working with the micro-ROS component[16]. The version of the ESP-IDF module used was v4.3[37], being compatible with Tracealyzer and micro-ROS.

Two different IDEs are used for Due because not all functionalities needed were present in both. PlatformIO[18] inside Visual Studio Code[38] and Arduino IDE are compatible with many different MCUs, including Due, and supported by the micro-ROS library[17]. However, PlatformIO cannot generate the correct debug symbols for Ozone, while Arduino IDE[39] is unable to debug projects. Thus both are needed, with PlatformIO used mainly.

6.6 Procedure

Procedure is how the tools interact with the system to produce results. This section outlines the general idea for the implementation.

6.6.1 Analysis

A ping pong application is used to test the latency of the communication. The initiator sends a ping, and the receiver responds with a pong, but a final acknowledgement peng, is also sent from the initiator, allowing both sides to measure the RTT. Communication is initialised by micro-ROS when sending sensor data and initialised by ROS2 for actuator commands, with the information sent using the pings. The transmission of data is analysed using Tracealyzer and Ozone to track incoming micro-ROS data, ros2_tracing recording ROS2 data over the OS, and Fast DDS Monitor monitoring data over the RMW. External validation is conducted as well using the oscilloscope.

[34]https://www.st.com/en/development-tools/stm32cubeide.html
[35]https://www.eclipse.org/
[36]https://github.com/espressif/idf-eclipse-plugin
[37]https://github.com/espressif/esp-idf/tree/release/v4.3
[38]https://code.visualstudio.com/
[39]https://www.arduino.cc/en/software

6.6.2 Prototype

A prototype is created with micro-ROS and ROS2 for testing in mYuMi. It is an extension of the ping pong application so that mYuMi can interact with the external components. The ROS2 nodes in the ping pong application are separate entities from any existing mYuMi nodes, interfacing the communication between micro-ROS and internal mYuMi functions. The nodes also do message type conversion and control that both sides are responding.

The 1D rangefinder is used to guide the height of the telescopic pillar relative to a workstation. This adjustment is helpful for mYuMi when using the cameras inside the SmartGrippers, as they are sensitive to height differences.

The RC servo motor takes input from mYuMi and adjusts the motor gear to the corresponding angle. This functionality is a proof of concept for actuator use in micro-ROS, with RC servos commonly used in robotics.

7 Ethical and Societal Considerations

Since testing will be conducted inside a laboratory environment on a robot, there are no concerns regarding personal information. A possible concern is safety since defective code on a mobile robot could lead to unpredictable behaviour and possibly hurt someone, though chances are minimal due to safety measures put in mYuMi.

On a related note, due to the lack of security support in the micro-ROS middleware mentioned in section 5, the possibility of a security breach has enormous implications on safety and reliability since mYuMi is planned to work close to people with no boundaries. However, this issue is only present on the client-side as ROS2 implements security measures that micro-ROS can use.

The positive contribution of this research is advancing the development of mYuMi with micro-ROS, leading to modular and less expensive solutions with microcontrollers. This contribution could lead to easier mass production of mYuMi due to costs and faster deployment into real-world settings.

8 Implementation

This section details how the methods are implemented. A representation of how the system is connected can be seen in figure 3.

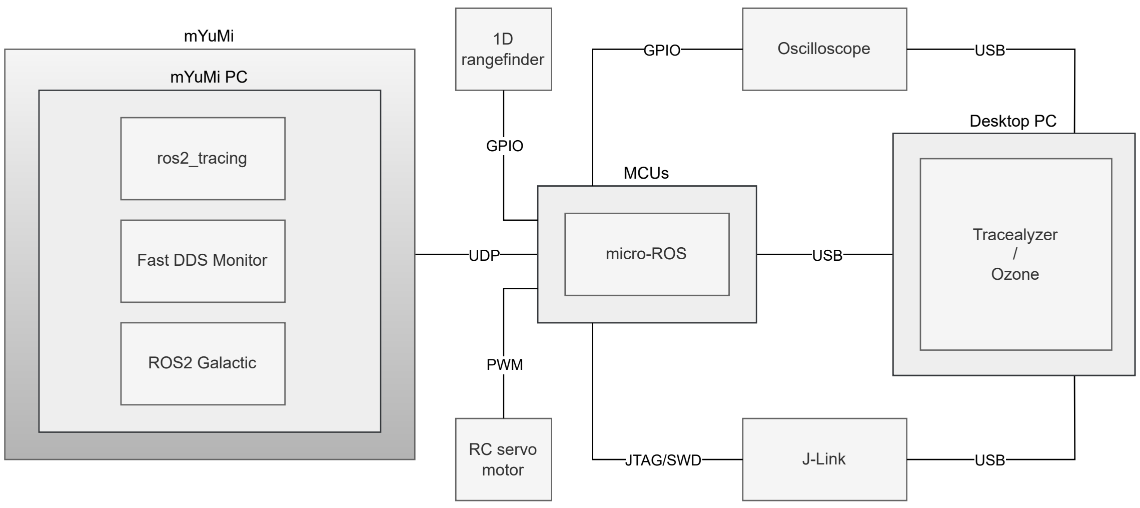


Figure 3: The physical connections and software used by mYuMi and the workstation are seen on the left and right. The UDP connection between mYuMi and the MCUs is the ping pong transmission, where the MCUs are F767ZI, 32S, and Due. For F767ZI and Due, the UDP connection is port forwarded through the desktop PC. The microcontrollers are connected two at a time, one for the 1D rangefinder and the other with the RC servo motor.

8.1 Communication

The micro-ROS setups for the MCUs are mostly the same, initialising different parameters and variables to allow micro-ROS to connect to the Agent and communicate with ROS2. The slight setup difference is in the transport initialisation, varying between chosen communication type and micro-ROS integration.

F767ZI uses UART serial transport via USB, requiring extra files inside the source folder and the corresponding code to set up the transport and allocate memory for micro-ROS inside the FreeRTOS tasks. 32S uses Wi-Fi configured with `menuconfig`[40] to connect with UDP to mYuMi, defining the RMW options and UDP addresses in the initialisation. Due uses USB serial transport, only needing to specify which transport to use in addition to the standard micro-ROS setup, as all functionality is built into the micro-ROS Arduino library.

Since the F767ZI and Due are connected serially to the desktop PC, the PC sets up a UDP connection with `netcat`[41] to stream the serial data to mYuMi.The mYuMi PC then reroutes the data to a virtual serial port using `socat`[42], emulating a direct connection to the boards.

The initialisation in ROS2 is less complicated than in micro-ROS as it does not need to set up an Agent connection or initialise the transport layer because the micro-ROS Agent handles all of this as a separate ROS2 node. The Agent looks for a device by specifying a device name or port for serial and IP, respectively, and continuously scans for a device after being started. A connection is then established as soon as the micro-ROS side has completed the Agent set up correctly, which can be confirmed from the terminal output of the Agent. The Agent makes micro-ROS reachable from the ROS2 network by interfacing any messages sent.

[40] https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/kconfig.html
[41] https://nc110.sourceforge.io/
[42] http://www.dest-unreach.org/socat/

8.2 Analysis

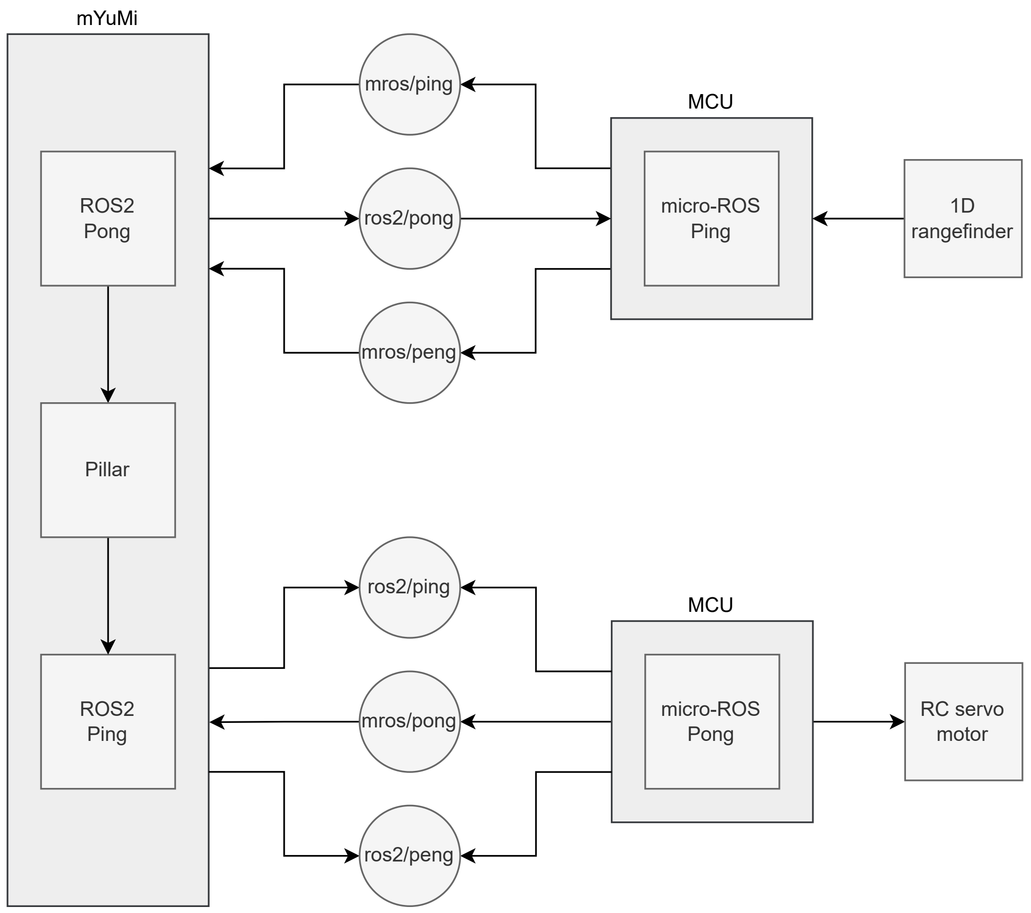


Figure 4: The ping pong communication setup during analysis. Ping, pong, and peng topics are divided into different namespaces depending on the transmitter.

The system is analysed by applying the tools on a ping pong transmission between micro-ROS and ROS2. Topic communication between mYuMi and the microcontrollers can be seen in figure 4. The implementation of the analysis and test setup is detailed in the following parts.

8.2.1 ROS2

The analysis tools applied to the mYuMi PC are ros2_tracing and Fast DDS Monitor. ros2_tracing is launched alongside the ping pong application on the ROS2 side, recording OS data in the background and saving the traces inside the workspace. Process execution can only be taken from incoming data with ros2_tracing, meaning the ROS2 subscribers are the ones observed. For ros2_tracing to function inside a ROS2 workspace, LTTng[19] has to be installed while building the ros2_tracing package using a CMake[43] file to set the correct flags.

The default RMW in Galactic is not compatible with Fast DDS Monitor and was thus changed to the one used by Foxy[44]. Due to limitations with Fast DDS Monitor, the topics to monitor and statistics to calculate need to be manually selected on each startup. Another problem is that Fast DDS Monitor must be used with ROS2 installed from source, separate from the Debian ROS2 Galactic installation currently in mYuMi, and built with a flag[45] to enable the statistics package.

[43] https://cmake.org/
[44] https://fast-dds.docs.eprosima.com/en/latest/fastdds/ros2/ros2.html
[45] https://fast-dds-monitor.readthedocs.io/en/latest/rst/ros/galactic/galactic.html

8.2.2 micro-ROS

Live performance analysis and variable monitoring are conducted on the desktop PC using Tracealyzer and Ozone with a J-Link debugger. Ozone is set up by specifying the microcontroller core and providing the debug file of the program flashed onto the board. A connection to the MCU via Ozone can either be made by downloading and resetting the program, or attaching to the program while it is running.

In order to use Tracealyzer streaming with FreeRTOS, code instrumentation in the form of extra files and startup functions is required [37]. STM32 microcontrollers must include the FreeRTOS Trace Recorder component Tracealyzer provides and specify processor family, FreeRTOS version, and tracing mode in the accompanying configuration file. The component is then enabled by calling a component initialisation function on startup, making J-Link streaming possible with the Tracealyzer application if the PSF Streaming and J-Link settings are configured correctly.

ESP32 microcontrollers must also include the FreeRTOS Trace Recorder component, but the one specific to ESP-IDF. In the configuration file, only the version of ESP-IDF used must be specified, as the other settings are the same for most ESP32-type MCUs. With the `menuconfig`, some ESP-IDF project configurations must be set to enable tracing. These include setting FreeRTOS to only run on the first core, enabling Application Level Tracing to trace memory, and configuring the Percepio Trace Recorder component to work with streaming. The initialisations of the component must be put directly in the startup function of the ESP-IDF installation, with methods differing between versions of ESP-IDF. Streaming is then possible by configuring the PSF Streaming and GDB settings correctly, but while streaming, an external session of OpenOCD[46] must be active.

The variable monitored by Tracealyzer and Ozone is `state`. The `state` variable is set to `1` when data is sent and reset to `0` when the corresponding data is received. For the transmitter, no more pings are sent when `state` is equal to `1`, but if more than one second passes, `state` is reset back to `0` so pings can be sent again. The `state` variable is also used to GPIO trigger a pin that the oscilloscope monitors.

The only requirement for variable monitoring in Ozone is that variables have to be global. Tracealyzer can monitor local variables but has to call a Tracealyzer specific print function after a variable is updated to monitor it. The main advantage of Tracealyzer in this project is that it also monitors and visualises task execution and context switching by tracing the RTOS.

8.2.3 Arrangements

The desktop PC is placed on a workstation inside a laboratory to analyse mYuMi and test the prototype, with MCUs connected two at a time to use both components. A micro-USB cable is connected to the PC to power the microcontrollers and transmit serial data, and a J-Link is also connected for debugging and tracing the microcontrollers.

Due to the J-Link being of an older hardware version, it is not compatible with the Cortex-M7 processor F767ZI uses, and therefore the built-in debugger in F767ZI is converted[47] to a J-Link for use with Tracealyzer. Since J-Link can connect with either SWD or JTAG to the MCUs, SWD is chosen for F767ZI and Due as it has better compatibility with ARM architectures. JTAG is used with 32S as it does not have inherent support for SWD.

8.2.4 Testing

The ping pong nodes in ROS2 used to test, continuously stream data between micro-ROS and mYuMi to assess the functionality and stability of the system. The pong node sets the pillar's height using the relative height of the 1D rangefinder, while the ping node sets the angle of the RC servo motor using the relative height range of the pillar.

In order to make the long-term testing more durable, a few of the tool functionalities are disabled. Recording kernel events is disabled in ros2_tracing to use less storage, Fast DDS Monitor uses the historical latency data instead of a live display to be less resource intensive, and Tracealyzer disables all visualisations to avoid overloading the RAM.

[46] `https://openocd.org/`
[47] `https://www.segger.com/products/debug-probes/j-link/models/other-j-links/st-link-on-board/`

8.3 Prototype

The prototype implements the areas needed by mYuMi to demonstrate the functionality of micro-ROS working with ROS2 in a live setting. The 1D rangefinder and RC servo motor are implemented in micro-ROS on F767ZI, 32S, and Due to display micro-ROS connecting external components with different MCUs. mYuMi is indirectly connected to the components through UDP using ROS2, connected to two of the microcontrollers at a time to display a series of actions listed in an application script.

8.3.1 1D rangefinder

The 1D rangefinder SRF04[30] activates when a 10 µs pulse is sent to the trigger input. This activation triggers the echo pin to high, starting transmission of an 8 cycle 40 kHz ultrasound burst, rebounding on the first available surface. The built-in operational amplifier circuit then awaits the returning echo to deactivate the echo pin, with the time between echo pin high and low being the pulse width P_W. If the pulse width is measured in microseconds, equation 1 transforms it into a distance D in centimetres, with SRF04 capable of measuring between 1–300 cm.

$$D = \frac{P_W * V}{2} \tag{1}$$

V is the speed of sound, approximately 0.0343 cm/µs. It is divided by 2 since the echo must return after colliding with a surface.

The F767ZI uses a global interrupt to monitor the rising and falling edge of the echo pin, returning the time difference in a global variable whenever the 1D rangefinder is triggered. Since STM32 microcontrollers have to configure clock channels manually, the prescaler was set as 72 to get the clock timer with 1 MHz speed, providing measurements in µs.

32S uses an ultrasonic component[48] to control the 1D rangefinder. With the component, the trigger and echo pins must be put in the initialisation function, and then the measurement function handles everything when called, outputting a distance in centimetres.

Due configures the pins and triggers the 1D rangefinder, and then uses the Arduino function `pulseIn()`[49] to handle the echo signal, returning a pulse width in microseconds. The distance can then be calculated with equation 1.

8.3.2 RC servo motor

The RC servo motor HS-5495BH[31] reacts when a PWM signal, a duty cycle D_C, between 800–2200 µs is sent to the signal wire, adjusting and maintaining the gear between 0–180°, a rotation α. Equation 2 converts an angle $\angle$ to the corresponding duty cycle that the RC servo motor can read.

$$D_C = \frac{\Big(\big(\angle + \max(\alpha)\big) * \big(\max(P_W) - \min(P_W)\big)\Big)}{2 * \max(\alpha)} + \min(P_W) \tag{2}$$

With the built-in functions in STM32Cube[50], a clock channel can be configured to generate a PWM signal after configuring the channel inside STM32CubeIDE, with the same configurations as the 1D rangefinder. The channel is then set directly with a duty cycle to adjust the motor angle.

The ESP-IDF motor control PWM component MCPWM[51] is used by 32S to set the rotation angle of the motor. The component specifies the pin, configures the PWM signal, and initialises the component. Then the angle, along with the PWM unit, timer, and generator, can be sent to the component function controlling the RC servo motor.

Due uses the Servo library[52] to control the motor. With this library, only the pin used needs to be sent to the library initialiser, with the option to redefine the pulse width range. The motor is then adjusted by sending in an angle to the library function handling the conversion and output.

[48] https://github.com/UncleRus/esp-idf-lib/tree/master/components/ultrasonic
[49] https://www.arduino.cc/reference/en/language/functions/advanced-io/pulsein/
[50] https://www.st.com/en/ecosystems/stm32cube.html
[51] https://docs.espressif.com/projects/esp-idf/en/v4.3/esp32/api-reference/peripherals/mcpwm.html
[52] https://www.arduino.cc/reference/en/libraries/servo/

8.3.3 mYuMi

The ping and pong nodes enable the mYuMi functions to interact with the components through micro-ROS. The pong node uses multithreading to continuously read data from the 1D rangefinder and mYuMi's pillar height topics, while running a service adjusting the pillar height based on a requested height D_R to set the 1D rangefinder. The 1D rangefinder is placed on a workstation that mYuMi tries to adjust to, measuring the relative height to mYuMi's arm stretched above the workstation. With equation 3, an adjustment σ of the current pillar height is calculated and added in a while loop each second until the 1D rangefinder height D is within 1 cm of D_R.

$$\sigma = (D - D_R) * \frac{\gamma - \lambda}{\max(D)} \tag{3}$$

γ and λ are the maximum and minimum pillar heights recommended, respectively.

Since the RC servo motor was only used as a proof of concept for actuator use in micro-ROS, the same ping node used during testing is used for the prototype. The angle α corresponding to the pillar height H is calculated with equation 4 and then rounded to the nearest integer.

$$\alpha = H * \max(\alpha) \tag{4}$$

8.3.4 Application

A Python application script provided by ABB creates a loop of mYuMi rolling up to the workstation and stretching out one if its hands above the 1D rangefinder, and then uses the height service to adjust the pillar between 25–65 cm generated randomly, while the RC servo motor adjusts accordingly.

While the application is running, Fast DDS Monitor, Tracealyzer, Ozone, and the oscilloscope are live displaying latency data to be compared to each other and the actions of mYuMi. After the application is stopped, the trace recorded by ros2_tracing can be examined alongside the saved data from the live displays.

9 Results and Analysis

The final functionality of the communication, analysis, and prototype implementations, alongside testing of said functionality, are presented and analysed in this section. A test results data set was generated using Tracealyzer, Ozone, ros2_tracing, and Fast DDS Monitor. The data retrieved from the oscilloscope was difficult to prepare due to the file saving structure and the large quantity of data, and was only used visually to validate the system.

9.1 Functionality

The communication between micro-ROS and ROS2 in mYuMi works with F767ZI, 32S, and Due with no noticeable differences in the basic functionality. The 32S Wi-Fi network and the UDP port forwarding with F767ZI and Due establishes a connection to the Agent and can continuously send and receive data when discovered by the ping pong nodes in mYuMi.

The ping pong communication allows mYuMi and the components to interact, displayed visually using the pillar and the motor. ros2_tracing could record ROS2 events for long periods, while Fast DDS Monitor was functional for about 3 hours before becoming unresponsive due to RAM overload. Incoming ROS2 data was recorded by ros2_tracing and ROS2 data sent was monitored by Fast DDS Monitor.

Tracealyzer can trace for more than 24 hours but cannot open full F767ZI and 32S traces longer than 12 and 8 hours, respectively, as the files become too large to open. 32S' is shorter due to additional background tasks active. The only functionality used with Ozone was variable monitoring due to software and hardware limitations with Due and J-Link.

The software[53] used to extract oscilloscope data saved waveforms as separate files with timestamps relative to the waveform, making data processing difficult. The newest version of the software could only save 64 waveforms, with the older more complete version not compatible with Linux, and thus a separate Windows PC had to be used to collect oscilloscope data.

The functionality of the 1D rangefinder had no noticeable differences between each microcontroller, while the RC servo motor could rotate 90° with F767ZI, 180° with 32S, and 135° with Due. A 90° angle was used for testing and the prototype to make it consistent across each MCU.

The prototype successfully adjusted the pillar height relative to the workstation using the 1D rangefinder, taking input from mYuMi when it reached the workstation and stretched out its arm, with the RC servo motor adjusting accordingly to the pillar height. The live analysis of the prototype worked as expected and complied with the test results.

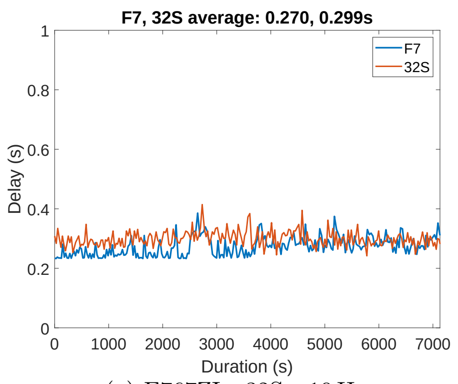
9.2 Test results

The analysis test bench recorded data for nine different scenarios in batches of two hours. The scenarios were F767ZI to 32S, 32S to Due, and Due to F767ZI, tested at 10, 12.5, and 16.7 Hz transmission frequencies. MCU to MCU means the first one transmits data with a ping node and the second one uses a pong node to receive, by communicating with the ping pong nodes in mYuMi.

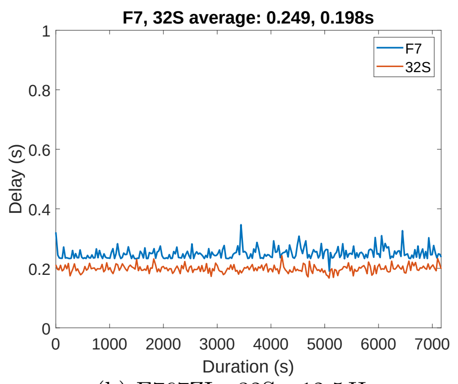
Data is presented in groups for each tool used, with rows representing microcontrollers used and columns being the transmission frequencies. The results for each tool scenario are combined into one graph by using different colours for each MCU or node, with legends indicating which one is which and line averages displayed above each subfigure. The mean of every 0.2% of all values are plotted to make the results more readable, except for Fast DDS Monitor which uses 2% of all values. Graphs were generated with MATLAB[54].

[53]https://www.picotech.com/downloads
[54]https://www.mathworks.com/products/matlab.html

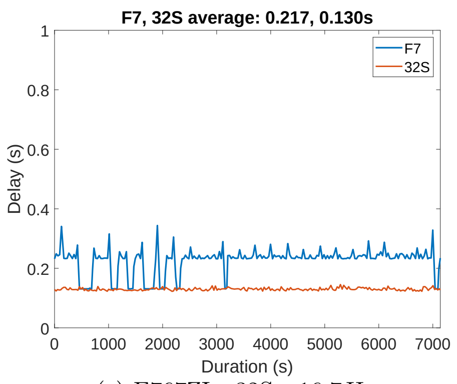
9.2.1 Tracealyzer and Ozone



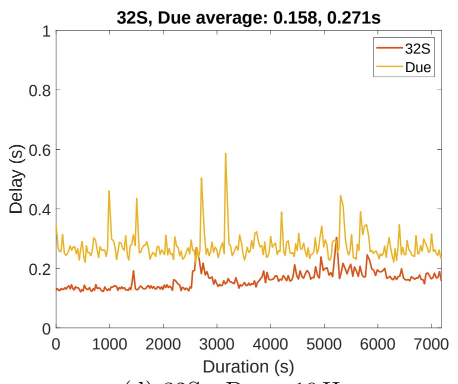
(a) F767ZI - 32S - 10 Hz.



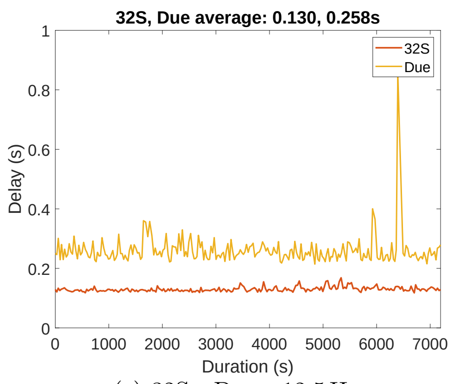
(b) F767ZI - 32S - 12.5 Hz.



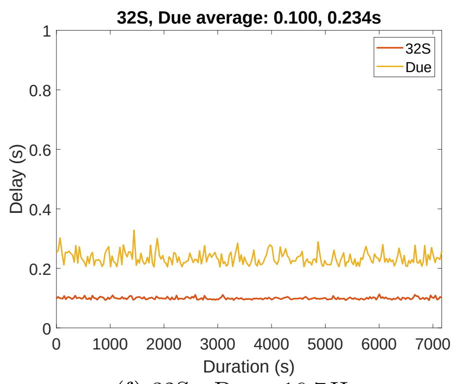
(c) F767ZI - 32S - 16.7 Hz.



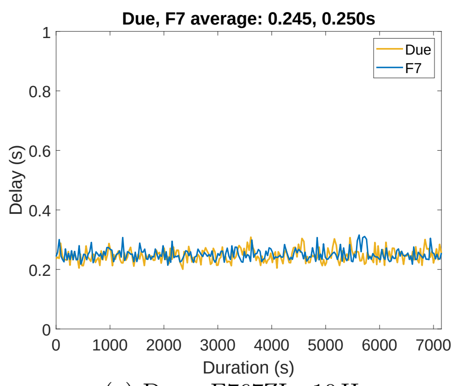
(d) 32S - Due - 10 Hz.



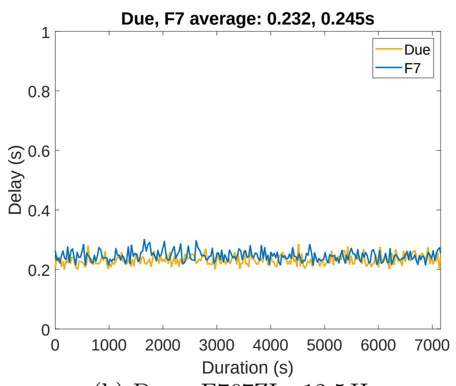
(e) 32S - Due - 12.5 Hz.



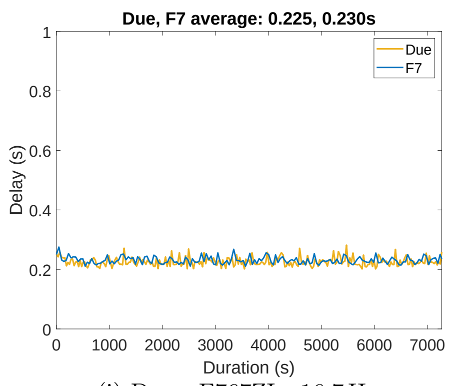
(f) 32S - Due - 16.7 Hz.



(g) Due - F767ZI - 10 Hz.



(h) Due - F767ZI - 12.5 Hz.



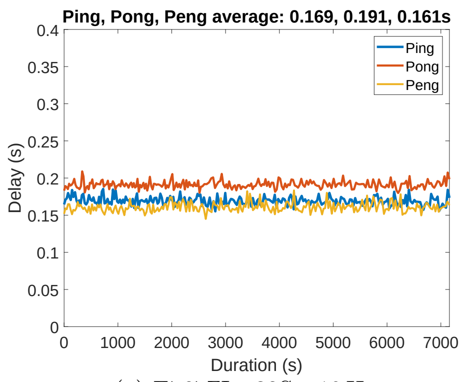
(i) Due - F767ZI - 16.7 Hz.

Figure 5: The two hour RTT results for Tracealyzer with F767ZI and 32S, and Ozone with Due. F767ZI are the blue lines, 32S the red lines, and Due the yellow lines.

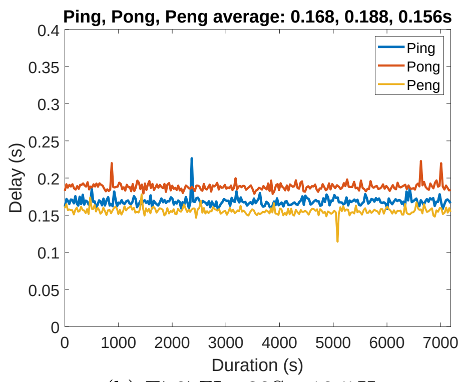
Multiple patterns can be seen in the microcontroller RTT results displayed in figure 5. Across all the MCU results, it is seen that stability increases and the average decreases with higher frequencies. 32S has the lowest average by quite a bit, with F767ZI in second, closely followed by Due.

F767ZI is consistent across each frequency as a transmitter and receiver but bottlenecked for a while at the 16.7 Hz transmission. 32S' reception variance and average substantially decreased as frequency increased, with very low transmission averages for all frequencies. Due shows similar results to F767ZI but with no bottleneck 16.7 Hz and more reception variance.

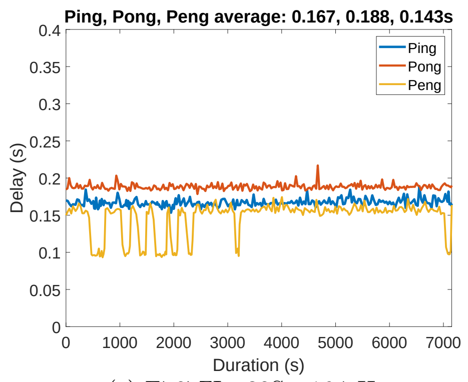
9.2.2 ros2_tracing



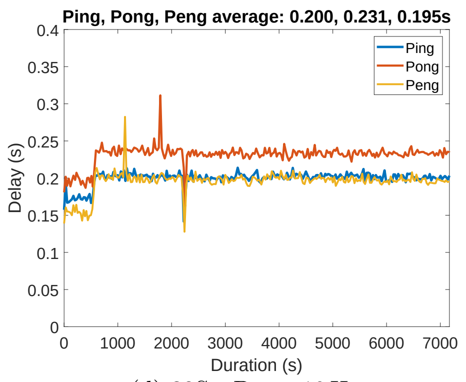
(a) F767ZI - 32S - 10 Hz.



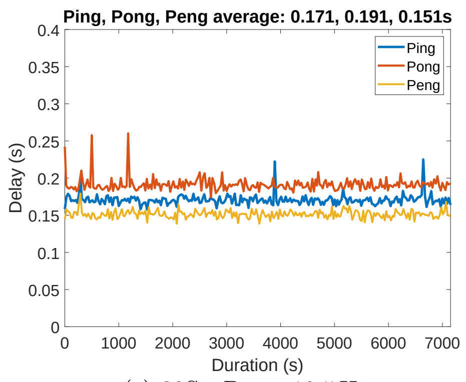
(b) F767ZI - 32S - 12.5 Hz.



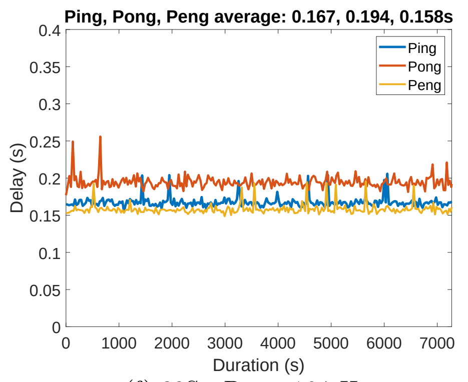
(c) F767ZI - 32S - 16.7 Hz.



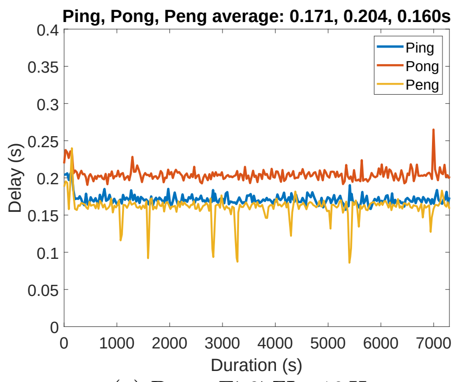
(d) 32S - Due - 10 Hz.



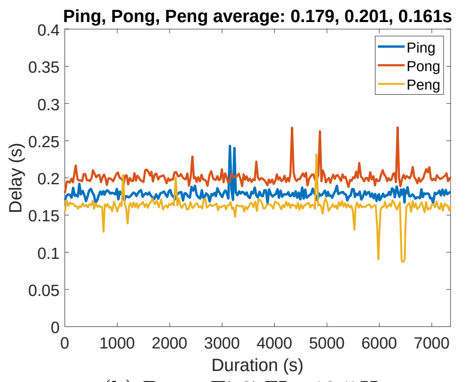
(e) 32S - Due - 12.5 Hz.



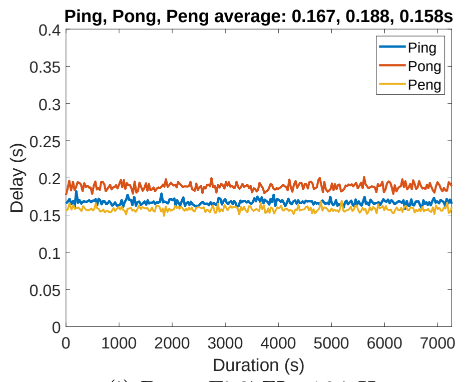
(f) 32S - Due - 16.7 Hz.



(g) Due - F767ZI - 10 Hz.



(h) Due - F767ZI - 12.5 Hz.



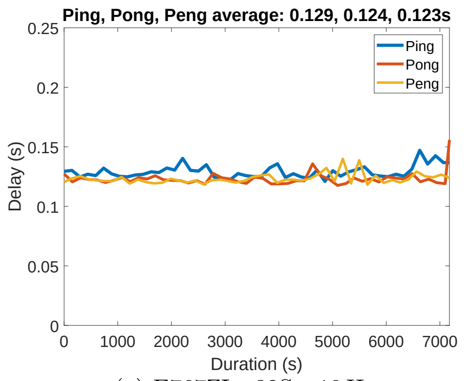
(i) Due - F767ZI - 16.7 Hz.

Figure 6: The two hour reception delay results for the ping, pong, and peng nodes in ROS2. Ping delay are the blue lines, pong delay are the red lines, and peng delay are the yellow lines.

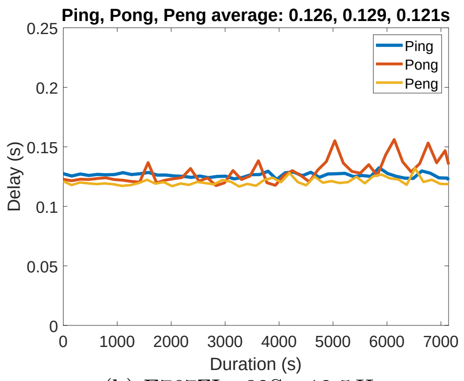
The ROS2 subscription delays calculated by ros2_tracing can be seen in figure 6. The peng delay is consistently the lowest for each result, with the ping delay in second. Several outliers are present in the pong delays, making it hard to determine the relationship between stability and frequency.

The differences between MCU to MCU are many, with no apparent patterns. F767ZI to 32S has the least combined subscription delay and is the most consistent overall, but at 16.7 Hz the peng seems to bottleneck at the same time as 5c, which makes sense as F767ZI publishes to the peng topic. 32S to Due has abnormally large delays at 10 Hz, but otherwise, it is barely behind in averages and is relatively stable. Due to F767ZI has a few spikes at 10 and 12.5 Hz but has the best overall stability at 16.7 Hz, with averages in the middle.

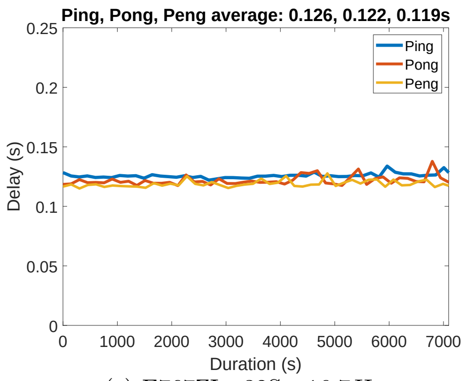
9.2.3 Fast DDS Monitor



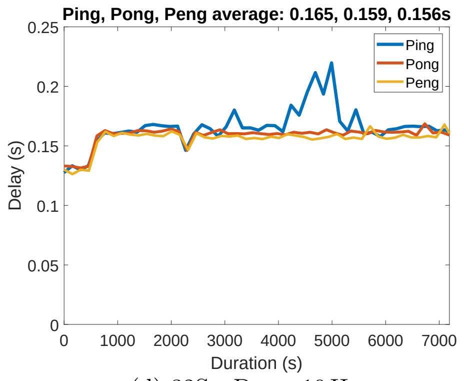
(a) F767ZI - 32S - 10 Hz.



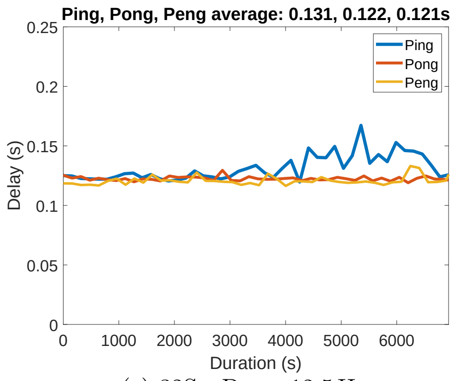
(b) F767ZI - 32S - 12.5 Hz.



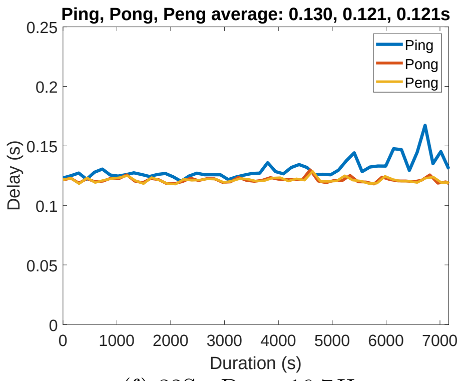
(c) F767ZI - 32S - 16.7 Hz.



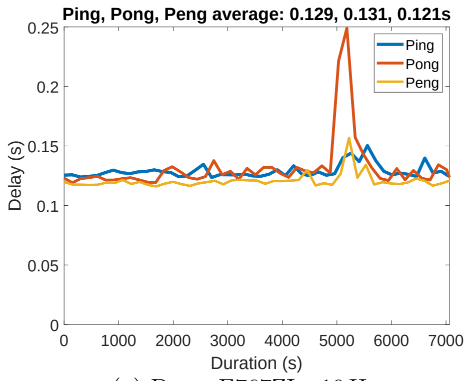
(d) 32S - Due - 10 Hz.



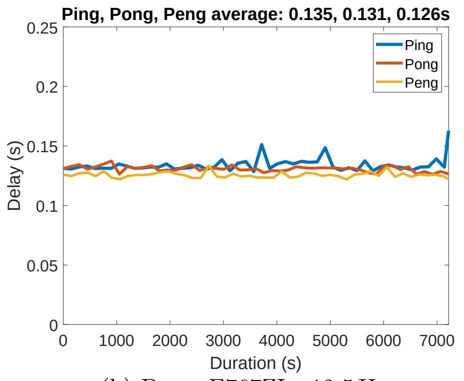
(e) 32S - Due - 12.5 Hz.



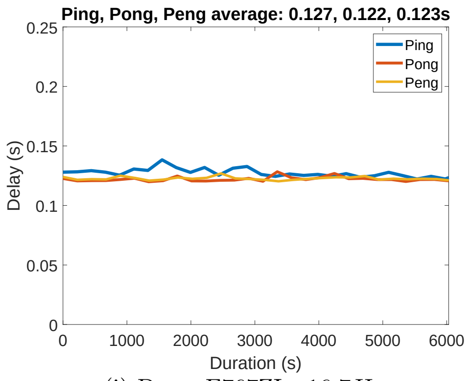
(f) 32S - Due - 16.7 Hz.



(g) Due - F767ZI - 10 Hz.



(h) Due - F767ZI - 12.5 Hz.



(i) Due - F767ZI - 16.7 Hz.

Figure 7: The two hour transmission delay results for the ping, pong, and peng nodes in ROS2. Ping delay are the blue lines, pong delay are the red lines, and peng delay are the yellow lines.

The Fast DDS Monitor results evaluating ROS2 publish delays can be seen in figure 7. The ping, pong, and peng delays are very similar for most results, but there are several outliers. An increase in frequency seems to decrease the average for the most part.

F767ZI to 32S has the lowest average across all results and not many outliers. The 10 Hz result for 32S to Due has a similar extra delay as 6d, indicating an anomaly during this recording. The two other frequencies have sensible results and the second-lowest averages comparatively, with some extra delay on the ping in the second halves of the results. Due to F767ZI has one big pong spike on the 10 Hz result, but otherwise, it is the most stable and barely behind in delay averages.

9.2.4 Summary

	Ping RTT	**Pong RTT**
F7	0.245s	0.242s
32S	0.129s	0.209s
Due	0.234s	0.254s

Table 1: The average RTT results in Tracealyzer and Ozone, across all frequencies for each MCU in figure 5.

	Ping RX	**Pong RX**	**Peng RX**
F7	0.168s	0.198s	0.153s
32S	0.179s	0.189s	0.168s
Due	0.172s	0.205s	0.160s

Table 2: The average reception delay results in ros2_tracing, across all frequencies for each node and MCU in figure 6.

	Ping TX	**Pong TX**	**Peng TX**
F7	0.130s	0.125s	0.123s
32S	0.127s	0.134s	0.121s
Due	0.142s	0.128s	0.133s

Table 3: The average transmission delay results in Fast DDS Monitor, across all frequencies for each node and MCU in figure 7.

In tables 1, 2, and 3 are the MCU averages for each frequency. The Ping and Pong RTTs should be roughly equal to equations 5 and 6, respectively.

$$\text{Ping}_{\text{RTT}} = \text{Ping}_{\text{RX}} + \text{Pong}_{\text{TX}} + \text{MCU}_{\text{TX}} + \text{MCU}_{\text{RX}} + X \tag{5}$$

$$\text{Pong}_{\text{RTT}} = \text{Pong}_{\text{RX}} + \text{Peng}_{\text{TX}} + \text{MCU}_{\text{TX}} + \text{MCU}_{\text{RX}} + X \tag{6}$$

Where MCU_{RX} and MCU_{TX} are the micro-ROS reception and transmission delays, respectively, and X is the sum of other possible delays.

In theory, it should be possible to estimate the sum of the micro-ROS delays using ros2_tracing and Fast DDS Monitor since X can be seen as negligible in this project. However, the sum of the ROS2 delays is greater than the RTT or insignificantly smaller, meaning something is wrong with the system. Due to an unidentified error, nothing conclusive can be said about the delay of micro-ROS.

The exact stability of micro-ROS cannot be deduced either, but by comparing the delay over time for each result, it can be seen that the RTT generally is stable when the ROS2 delays are, meaning the micro-ROS delays also have to be stable. This argument is reinforced by the fact that the stability of most results increases with frequency, and shared patterns between MCU to MCU results can be seen.

10 Discussion

This project implemented micro-ROS with the goals of testing it inside mYuMi and different microcontrollers, developing a prototype, improving upon previous works, and integrating with other methods. With the many different paths available for this work, the selections of methods and their usage in the implementation are justified and reflected on in this section, and then concluded with a discussion regarding the relevance and use cases of the outcome.

10.1 Selections

Starting at the case study of the current State-of-Practice and State-of-the-Art in section 4, the basis of analysis with tracing was formed, shaping the rest of the project. The tools were decided on with this information by testing it with the software and hardware given. The related work could have been more relevant to the act of tracing and the tools chosen, but most of the ideas were thought of during the implementation, as not many scientific papers on these topics exist.

With the hardware and software already set by ABB, excluding components, the tools had to be the ones to conform. The use of ros2_tracing was covered in the related works, which inspired the use of Tracealyzer to trace microcontrollers with FreeRTOS, and in turn, encouraged the use of Ozone for a no OS microcontroller. In addition, the MCUs were externally validated with an oscilloscope, also covered in the related works. Fast DDS Monitor was included as a partial solution to the limitations with ros2_tracing, as Fast DDS Monitor and ros2_tracing could only record ROS2 output and input data, respectively.

The ping pong type communication was a natural choice as it let both sides measure latency and fit the functionality of the tools, with all tools utilising the back-and-forth communication to determine delay. It also fit together well with the prototype since the component data could be put in the ping message. The 1D rangefinder was a suitable sensor for the prototype as it helped with existing functionality and was easy to display in a demonstration. The RC servo motor could have been utilised better or replaced with another actuator, but it served its purpose as a proof of concept.

10.2 Usage

The use of external applications to trace communication was appropriate for this project since the tools used little overhead and could be used on multiple platforms. Unfortunately, these applications could not trace no OS platforms like Due, but Ozone was a sufficient replacement for measurements. Only the QoS parameters in tracing was focused on in this project, but the tracing applications can easily be configured to include other OS traits.

Using port forwarding with F767ZI and Due will likely not yield similar results as either UDP with built-in Ethernet or a direct serial connection, but accomplished its role of connecting these particular microcontrollers remotely. 32S could have also used port forwarding to make it consistent across all boards, but the built-in Wi-Fi was a focal point of 32S and was used instead, utilising the micro-ROS transport layer directly.

The test bench covered all MCU combinations with three different frequencies, testing the performance for 2 hours. More scenarios could have been tested and for extended periods; however, at the current stage of the tools and micro-ROS development, multiple parts cannot be tested legitimately, such as UDP in micro-ROS with F767ZI and Due, and the micro-ROS communication delay with external applications, making this project more of a demonstration of functionality. Integrating the microcontrollers inside mYuMi is another possible improvement when testing, but the remote connection was sufficient to see if the system worked.

10.3 Outcome

The communication between micro-ROS and ROS2 functioned on all three boards with varying results. The results are incomplete due to an unknown error, causing the delays not to add up. The fault could be present in any tool or the system, with insufficient time to locate it. However, the results and the analysis still indicate a ceiling for possible micro-ROS delays and provide compelling information about the stability. The results also show a considerable delay difference between F767ZI, Due, and 32S. This difference might be due to the different transportation layers used, as the port forwarded data travelled with both serial and UDP. These outcomes hint at the capabilities of micro-ROS and lay a foundation for future testing. Since the analysis was conducted with external applications, a similar tool setup should apply to other micro-ROS and ROS2 systems.

The related work in section 4.6 determined the transmission and reception delay of micro-ROS on an MCU level but used a complicated setup with specific hardware and code injection, and only the average results were shown. In comparison, this project displayed the possibility of using different microcontrollers communicating directly with ROS2 while tracing with external applications on different levels, and made a stability analysis by displaying the full traces.

Multiple limitations have been encountered during this project, especially with communication analysis, as the combination of micro-ROS and the individual tools has created these limitations. However, this project has determined that, in theory, the tools together can calculate the combined delay of micro-ROS with equations 5 and 6. ros2_tracing can record the delay of ROS2 subscription but not for publications, as it cannot measure over the RMW. Fast DDS Monitor can monitor the DDS publish delay, but any data incoming from the micro-ROS side is not visible due to micro-ROS limitations with statistical data. Tracealyzer, Ozone, and the oscilloscope measure the RTT, but this is made on a microcontroller level as no explicit micro-ROS analysis is available. Many of these limitations will disappear with time as micro-ROS and the tools continue to evolve, but this project confirmed what currently is possible and that external tools can be used with micro-ROS and ROS2 interchangeably.

By implementing on three different board types, STM32, ESP32, and Arduino, the flexibility of micro-ROS has been tested. The main micro-ROS setup is essentially the same across all boards, with the equivalent publish and subscribe functionality. Differences appear in the transport initialisation, as UART, USB, and Wi-Fi are used as communication layers with F767ZI, Due, and 32S, respectively. With micro-ROS deployed successfully on different boards by utilising the capabilities specific to each MCU, this project has established the microcontroller portability of micro-ROS, meaning most ROS2 systems using microcontrollers should be able to integrate micro-ROS. External solutions like port forwarding have also been implemented to ease integration.

The 1D rangefinder and RC servo motor were implemented on all three MCUs and used during testing and with the prototype. mYuMi successfully adjusted to a workstation using the prototype with all MCU combinations. While the prototype was running, live data could be seen from Fast DDS Monitor, Tracealyzer, Ozone, and the oscilloscope, displaying delay and stability comparable to the results. Unfortunately, ros2_tracing cannot display live data and thereby, the ROS2 publish delay was not visualised. Since the prototype is an extension of the ping pong communication used during testing, the analysis also applies to the prototype. Therefore, no mathematical proof can be made for the delay of micro-ROS, but the same reasoning regarding stability is applicable. The demonstration presented the potential of micro-ROS in mobile robotics systems using ROS2 and the value of early integration into mYuMi, since mYuMi is still in prototype stages and micro-ROS needs more testing.

11 Conclusions

With tracing and other methods utilised in external applications, a QoS analysis of micro-ROS has been presented with test results and validated with a Sense-Plan-Act prototype. Testing and prototype development were conducted with ABB's mobile robot, mYuMi, using ROS2 to control its movement while connected remotely to microcontrollers with micro-ROS. The connection used ping pong communication to transmit sensor and actuator data, controlling mYuMi's pillar height with a 1D rangefinder while mYuMi controlled an RC servo motor indirectly via the microcontrollers.

11.1 Research questions

The RQs of this thesis and the corresponding answers are as follows:

RQ1. What are the limitations to communication analysis between micro-ROS and ROS2?

- There are currently no tools able to analyse micro-ROS directly. External tools are introduced to analyse micro-ROS, each with its limitations. ros2_tracing records the ROS2 reception delay on a ROS2/OS level, but the transmission delay cannot be estimated as ros2_tracing cannot measure the DDS RMW. Fast DDS Monitor does the opposite as it measures on a DDS level, with incoming micro-ROS data not visible as micro-ROS currently has no support for underlying statistics. Tracealyzer, Ozone, and the oscilloscope measure the RTT on a microcontroller software and hardware level.
- Since the tools together cover different delays, it should, in theory, be possible to estimate the combined transmission and reception delay for micro-ROS, but this could not be proven as the calculations did not add up.

RQ2. Can equivalent forms of micro-ROS functionality be deployed on STM32, ESP32, and Arduino development boards?

- The same main micro-ROS setup was put on F767ZI, 32S, and Due development boards to connect to the micro-ROS Agent and initialise publishers and subscribers. However, different transportation layers are used as the micro-ROS integrations have different capabilities.

RQ3. Will micro-ROS be able to match ROS2 in terms of stability and delay when used jointly in a Sense-Plan-Act prototype with the new ABB robot, the mobile YuMi prototype?

- The test results were illegitimate since the sum of the ROS2 delays was greater or insignificantly smaller than the RTTs, meaning micro-ROS delays could not be calculated and the implication of an error present in the setup. The results still indicated a ceiling for possible micro-ROS delays, and it could be seen in the stability analysis that micro-ROS should match ROS2 in terms of stability.
- Since the prototype is an extension of the ping pong communication used during testing and the live analysis displayed results comparable to the test results, the same conclusions from the testing can be made for the prototype.

11.2 Implications

If ABB decides to implement micro-ROS into mYuMi permanently, the seamless integration of additional microcontrollers could drastically reduce costs and increase the system's modularity when implementing new components. This decision could also inspire other corporations to test micro-ROS, advancing the progress of micro-ROS in addition to ROS2, as they are still in the early stages of development.

12 Future Work

This project demonstrated the practical use of external tools with micro-ROS inside a mobile robot, but the results still leave much to be desired. Results would be significantly improved if the oscilloscope results were included, code was optimised, more test bench scenarios were added, and testing time was increased. In order to enable the system to test for longer, the backend[55] of Fast DDS Monitor must be used instead of Fast DDS Monitor when testing, as it can save DDS data directly to memory, and Tracealyzer traces need to either be partially opened or opened with a better PC.

The results for F767ZI and Due can be improved by using Ethernet instead of port forwarding the serial connections. For F767ZI, the transport layer has to be manually implemented in micro-ROS as it does not yet exist for the STM32 integration, while Due needs to use external hardware such as an Ethernet Shield[56]. Tracing and debugging would be enhanced by using a better J-Link debugger[57], as the microcontrollers would be able to use the same debugger, and Due could use instruction tracing in Ozone.

[55] https://www.eprosima.com/index.php/products-all/tools/eprosima-fast-dds-statistics-backend
[56] http://store.arduino.cc/products/arduino-ethernet-shield-2
[57] https://www.segger.com/products/debug-probes/j-trace/

References

[1] M. Quigley, B. Gerkey and W. D. Smart, *Programming Robots with ROS: A Practical Introduction to the Robot Operating System*, 1st. O'Reilly Media, Inc., 2015, ISBN: 1449323898.

[2] H. Cui, J. Zhang and W. R. Norris, 'An Enhanced Safe and Reliable Autonomous Driving Platform using ROS2,' in *2020 IEEE International Conference on Mechatronics and Automation (ICMA)*, 2020, pp. 290–295. DOI: 10.1109/ICMA49215.2020.9233814.

[3] Open Robotics, *Distributions*, http://wiki.ros.org/Distributions, Accessed 29-11-2021.

[4] ——, *ROS: Home*, https://www.ros.org/, Accessed 26-11-2021.

[5] ——, *Changes between ROS 1 and ROS 2*, http://design.ros2.org/articles/changes.html, Accessed 29-11-2021.

[6] micro-ROS, *micro-ROS*, https://micro.ros.org/, Accessed 02-12-2021.

[7] OFERA, *OFERA Project: Open Framework for Embedded Robot Applications*, http://www.ofera.eu/index.php, Accessed 27-12-2021.

[8] T. Kołcon and A. Malki, 'Micro-ROS benchmarks - Extension,' OFERA, Tech. Rep. D5.7, Aug. 2021, Grant agreement no. 780785. [Online]. Available: http://ofera.eu/storage/deliverables/M44/OFERA_93_D57_Micro-ROS_benchmarks_-_Extension.pdf.

[9] A. Malki and T. Kołcon, 'Micro-ROS benchmarking and validation tools Release - Final,' OFERA, Tech. Rep. D5.8, Dec. 2021, Grant agreement no. 780785. [Online]. Available: http://ofera.eu/storage/deliverables/M48/OFERA_94_D58_Micro-ROS_benchmarking_and_validation_tools_Release_-_Final.pdf.

[10] W. Zhao, D. Olshefski and H. Schulzrinne, 'Internet Quality of Service: an Overview,' Mar. 2000.

[11] S. Hasnain and A. Rafi, 'Windows, Linux, Mac Operating System and Decision Making,' *International Journal of Computer Applications*, vol. 177, pp. 11–15, Dec. 2019.

[12] J. A. Stankovic and R. R. Rajkumar, 'Real-Time Operating Systems,' *Real-Time Systems*, vol. 28, pp. 237–253, 2004.

[13] Open Robotics, *Distributions*, https://docs.ros.org/en/galactic/Releases.html, Accessed 25-01-2022.

[14] micro-ROS, *Supported RTOSes*, https://micro.ros.org/docs/overview/rtos/, Accessed 09-01-2022.

[15] ——, *Overview*, https://micro.ros.org/docs/tutorials/core/overview/, Accessed 12-01-2022.

[16] ——, *Supported Hardware*, https://micro.ros.org/docs/overview/hardware/, Accessed 12-01-2022.

[17] ——, *Integration into External Tools*, https://micro.ros.org/docs/overview/ext_tools/, Accessed 05-01-2022.

[18] ——, *Creating custom static micro-ROS library*, https://micro.ros.org/docs/tutorials/advanced/create_custom_static_library/, Accessed 12-01-2022.

[19] Open Robotics, *Understanding ROS 2 nodes*, https://docs.ros.org/en/galactic/Tutorials/Understanding-ROS2-Nodes.html, Accessed 31-12-2021.

[20] ——, *Understanding ROS 2 topics*, https://docs.ros.org/en/galactic/Tutorials/Topics/Understanding-ROS2-Topics.html, Accessed 31-12-2021.

[21] ——, *Understanding ROS 2 services*, https://docs.ros.org/en/galactic/Tutorials/Services/Understanding-ROS2-Services.html, Accessed 18-04-2022.

[22] ——, *Launch tutorials*, https://docs.ros.org/en/galactic/Tutorials/Launch/Launch-Main.html, Accessed 31-12-2021.

[23] ——, *About Quality of Service settings*, https://docs.ros.org/en/galactic/Concepts/About-Quality-of-Service-Settings.html, Accessed 01-02-2022.

[24] ABB, *YuMi - IRB 14000 | Collaborative Robot*, https://new.abb.com/products/robotics/collaborative-robots/irb-14000-yumi, Accessed 15-01-2022.

[25] STMicroelectronics, *STM32 32-bit Arm Cortex MCUs*, https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html, Accessed 02-01-2022.

[26] Espressif, *ESP32*, https://www.espressif.com/en/products/socs/esp32, Accessed 13-01-2022.

[27] Arduino, *What is Arduino?* https://www.arduino.cc/en/Guide/Introduction, Accessed 13-01-2022.

[28] P. Zins and M. R. Dagenais, 'Tracing and Profiling Machine Learning Dataflow Applications on GPU,' *International Journal of Parallel Programming*, pp. 1–41, 2019.

[29] P. Phueakthong and J. Varagul, 'A Development of Mobile Robot Based on ROS2 for Navigation Application,' in *2021 International Electronics Symposium (IES)*, 2021, pp. 517–520. DOI: 10.1109/IES53407.2021.9593984.

[30] S. Dehnavi, D. Goswami, M. Koedam, A. Nelson and K. Goossens, 'Modeling, implementation, and analysis of XRCE-DDS applications in distributed multi-processor real-time embedded systems,' in *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2021, pp. 1148–1151. DOI: 10.23919/DATE51398.2021.9474221.

[31] L. Puck, P. Keller, T. Schnell *et al.*, 'Performance Evaluation of Real-Time ROS2 Robotic Control in a Time-Synchronized Distributed Network,' in *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, 2021, pp. 1670–1676. DOI: 10.1109/CASE49439.2021.9551447.

[32] C. Bédard, I. Lütkebohle and M. Dagenais, *ros2_tracing: Multipurpose Low-Overhead Framework for Real-Time Tracing of ROS 2*, 2022. arXiv: 2201.00393 [cs.RO].

[33] Z. Li, A. Hasegawa and T. Azumi, 'Autoware_Perf: A tracing and performance analysis framework for ROS 2 applications,' *Journal of Systems Architecture*, vol. 123, p. 102 341, 2022, ISSN: 1383-7621. DOI: https://doi.org/10.1016/j.sysarc.2021.102341.

[34] T. Kołcon, A. Malki and M. Maciaś, 'Benchmark deliverables,' OFERA, Tech. Rep. D5.1-D5.8, Grant agreement no. 780785. [Online]. Available: http://ofera.eu/index.php/publications.

[35] micro-ROS, *ROS 2 Feature Comparison*, https://micro.ros.org/docs/overview/ROS_2_feature_comparison/, Accessed 20-12-2021.

[36] W. W. Royce, 'Managing the Development of Large Software Systems: Concepts and Techniques,' in *Proceedings of the 9th International Conference on Software Engineering*, ser. ICSE '87, Monterey, California, USA: IEEE Computer Society Press, 1987, pp. 328–338, ISBN: 0897912160.

[37] Percepio, *Percepio Tracealyzer Documentation*, https://percepio.com/docs/FreeRTOS/manual/, Accessed 27-03-2022.