SAXION
UNIVERSITY OF
APPLIED SCIENCES

TECHNICAL REPORT

Testing the usability of Robot Operating System (ROS2/ Micro-ROS) in Benchmark

Company: Benchmark Electronics

Place, date: Almelo, 14/06/2022

Student:

Tran Tien Cuong	480048	480048@student.saxion.nl

Supervisors:

Yanin Kasemsinsup, Ph.D.	Saxion	y.kasemsinsup@saxion.nl
Robin Wijnholt, Msc.	Benchmark	Robin.Wijnholt@bench.com
Mathijs van der Werff, Msc.	Benchmark	Mathijs.van.der.Werff@bench.com

FOREWORD

This is my graduation internship for the degree of Bachelor in Electrical and Electronic Engineering program at Saxion University of Applied Sciences. I did my graduation internship at Benchmark, in Almelo, the Netherlands. This report is a research that was composed during the graduation assignment at Benchmark between February and June 2022. It describes the research on the usability of ROS2 and Micro-ROS applications for Benmark projects.

I would like to express my sincere thanks to my Saxion coach Yanin Kasemsinsup, lecturer in Electrical and Electronic Engineering & Applied Computer Science at Saxion University of Applied Sciences for his support during this internship period and advice/ tips so that I can professionally complete this project.

I want to show my gratitude to Mathijs van der Werff as my supervisor and also a senior engineer at the Design team in Benchmark for supporting me whenever I have questions and advising when I am stuck with the projects.

My project would be incomplete without the continuous guidance and support from Robin Wijnholt, my main supervisor from Benchmark. Thanks to him, I had a chance to learn a lot of new things and experience different methods not only for my graduation internship at Benchmark but also for my future career.

Finally, I am profoundly grateful to my colleagues at Benchmark for supporting me when I faced problems.

I wish you would enjoy reading this project report!

Sincerely,

Cuong

Almelo, the Netherlands

June 14th, 2022

SUMMARY

In Benchmark, there is currently a focus on researching and investigating the potential of applying ROS2 and Micro-ROS to future products. This report aimed to research and investigate the usability of ROS2/ Micro-ROS and show its potential benefits as well as limitations for use in Benchmark's products.

Five parameters including latency, memory consumption, reliability, scalability and security were used to evaluate ROS2 and Micro-ROS application's performance.

Overall, the results indicate that objectives are met as there are certain findings for the usability of ROS2 and Micro-ROS, which can lead to potential applications of ROS2 and Micro-ROS to Benchmark products.

The report concludes that ROS2 and Micro-ROS are ready to be used for real applications but need to be considered carefully with hard real-time applications. It is recommended that revising and tuning be applied for latency test setup, especially in dealing with tasks schedule. Besides, ROS2 Security features need to be investigated when applying ROS2 and Micro-ROS to projects.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

B	Byte
DDS	Data Distribution Service
GPIO	General Purpose Input/Output
KB	Kilobyte
MB	Megabyte
MSG	Message
MTU	Maximum Transmission Unit
QoS	Quality of Service
RMW	ROS Miffleware
RSS	Resident set size
RTOS	Real-time Operating System
RTPS	Real Time Publish-Subscribe
RTT	Round-Trip Time
UDP	User Datagram Protocol
XRCE-DDS	eXtremely Resource Constrained Environments - DDS

CHAPTER 1 INTRODUCTION

Benchmark Electronics Inc. which will be referred to as Benchmark for the remainder of this document, is an international electronics manufacturing services company that was founded in 1979 in Clute, Texas. Today Benchmark has twenty-four plants all over the world, of which two are located in Europe. The plant in Almelo was originally a subsidiary of Phillips N.V. After the period of takeovers and partnerships, Benchmark Electronics eventually took over the Almelo plant. The site in Almelo is expanding significantly in the last few years and currently, it has approximately 500 employees, of which there are about 130 design engineers. Benchmark Electronics Inc. provides services to Original Equipment Manufacturers (OEMs). The market sectors that Benchmark Almelo serves include commercial aerospace, complex industrial, defense, semiconductor capital equipment, and medical.

In the contemporary world, collaborative robots are proving their roles in modern industry by becoming part of the automation processes [2]. The Robot Operating system (ROS) is a popular robotics middle-ware framework which is commonly used for building autonomous robot systems. According to [2], due to high latency in the communication of ROS, ROS version 2 (ROS2) by utilizing the Data Distribution Service (DDS) as a transport system was born to improve this issue. ROS2 can provide high flexibility to the users for extensive robotic applications. Robotic systems usually are networks of one or more microprocessors together with several microcontrollers. The functions of microcontrollers are to access sensors and actuators, conduct control, or embed safety mechanisms. A microcontroller which usually has a few kilobytes of RAM cannot operate ROS directly [3]. Therefore, a solution for creating ROS2 nodes into embedded devices was born. It is called Micro-ROS – the robotic framework which bridges the gap between resource-constrained and larger processing units and robotics applications. In Benchmark, there is currently a focus on researching and investigating the potential of applying ROS2/ Micro-ROS to future products. Therefore, this graduation project was created.

The graduation project is about investigating the usability of ROS2/ Micro-ROS in Benchmark products with the help of a demo. Therefore, the company can consider applying them to its products in the future.

The project consists of eight competencies which include analysing, designing, realising, controlling, managing, advising, applied research of a ROS2/ Micro-ROS demo and professionalising. The document will start with the project background of ROS2/ Micro-ROS and the essential parameters in Chapter 2, followed by project objectives and requirements in Chapter 3. Chapter 4 is evaluation framework which defines experiment setup to evaluate ROS2 and Micro-ROS. Following the design phase, the evaluation will be presented in Chapter 5 which will contain details, results and discussions of these evaluations. Finally, Chapter 6 is aimed at the conclusion and recommendation for this project.

CHAPTER 2 BACKGROUND

This chapter introduces the necessary software and background information for understanding this project. Section 2.1 covers information about ROS2 and Micro-ROS. Following is the fundamental of the DDS and DDS-XRCE which are communication methods of ROS2 and Micro-ROS respectively in section 2.2. This chapter concludes with information about QoS policies (Quality of Service) in section 2.3.

2.1 ROS2

ROS2 is an upgraded version of ROS which was born to address the shortcomings of ROS. According to [2], the main design goal of ROS2 is to better the abilities of real-time functioning, which can support the implementation of time-critical control paths inside the framework. The new use-cases of ROS2 are supporting multiple robot systems, bridging the gap between prototypes and products, supporting microcontrollers, supporting real-time control and multi-platform support, so ROS2 not only runs on Linux systems but also can be used in Windows, macOS and Real-Time Operating System (RTOS) [2]. An anonymous publish-subscribe middleware system which is designed and built from the beginning is at the core of ROS2. Communication in ROS2 is based on DDS which is defined as a publish/subscribe data-distribution system by the Object Management Group (OMG). There are various benefits that DDS can offer such as low overhead, high efficiency and loose coupling, which makes it a good solution to exchange data among nodes in robots [8]. Currently, a variety of applications of DDS are available, which allows ROS2 users to choose a suitable one to meet their demands such as FastRTPS of eProsima, OpenSplice of Prismtech, Eclipse Cyclone DDS of Eclipse Foundation, RTI Connext DDS of Real-Time Innovations (RTI) or GurumDDS of GurumNetworks. Since the main focus of this project is ROS2 not DDS, here the project will only briefly summarize DDS.

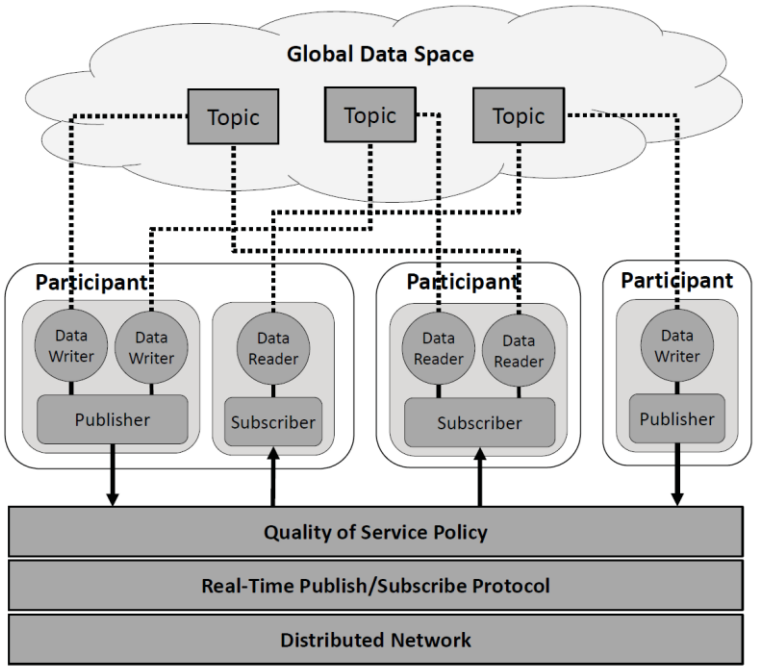

Figure 1: Data-centric publish-subscribe (DCPS) model [9]

The core of DDS is a Data-Centric Publish-Subscribe (DCPS) model which is shown in Figure 1. Each publisher has a Data Writer which is an object to publish data of predefined types. On the other hand, a Data Reader is an object attached to a Subscriber which is responsible for receiving the data sent by the publisher. A topic is used to identify each data object between a Data Writer and a Data Reader. Besides, a publisher and a subscriber can have one or more topics. All DCPS entities have an assigned QoS policy defining the behaviour of each entity concerning communication and discovery. In section 2.3, details of QoS will be described.

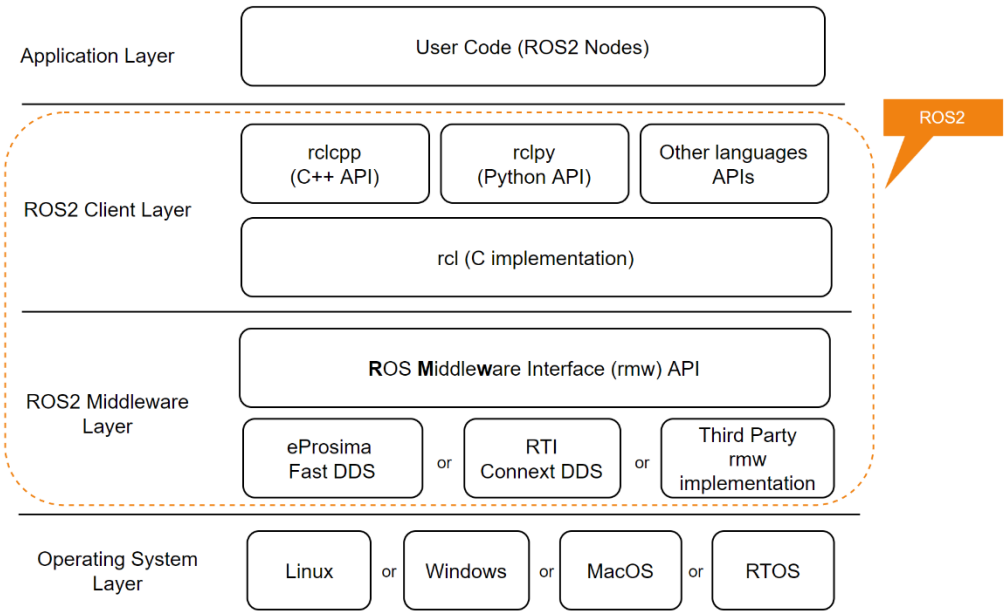

Figure 2: ROS2 architecture overview

Figure 2 illustrates ROS2 layered architecture. The application layer containing user codes is built on top of ROS2 framework which is placed inside the orange rectangle beginning from ROS2 client layer to ROS2 middleware layer. According to [10], for ROS2 to include numerous programming languages capability, there are certain language-specific wrappers, such as Python with rclpy or C++ with rclcpp. The implementation of the upper layer must wrap a C interface called the ROS Client Library (RCL). The implementation of ROS middleware is placed below RCL, which provides communication between ROS2 nodes. The concepts related to communication in ROS2 are presented in Figure 3:

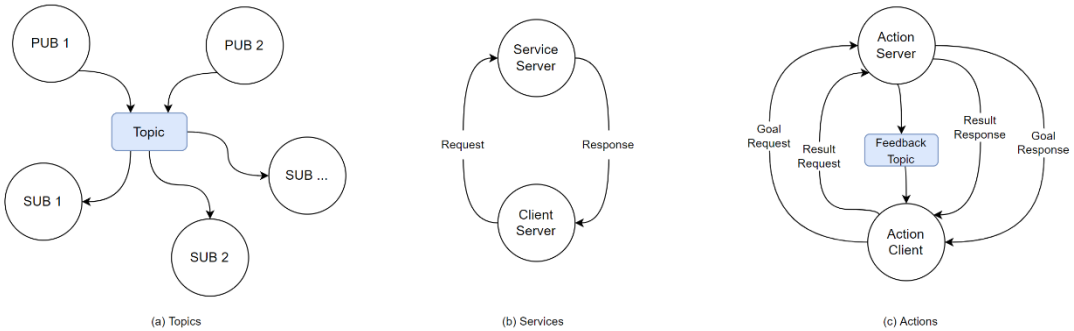

Figure 3: ROS2 communication types

- **Topics** (Figure 3a): Topics act as a bus for publishers and subscribers to exchange messages. It does not have to be only point-to-point communication;

it can be one-to-many, many-to-one, or many-to-many [14]. It is recommended to be used for continuous data streams such as sensor data or robot state [16].

- **Services** (Figure 3b): in service communication type, a service server only responds when a specific client-server requests. Many service clients can use the same service, but only one service server can be used for service [15]. This communication type is recommended for remote procedure calls that terminate quickly [16]

- **Actions** (Figure 3c): Actions are built on topics and services. It includes three parts: a goal, feedback, and a result. First, an action client creates a goal for an action server. When action client receives a response, it will call the result from action server. After that, the action server will steadily provide feedbacks to the action client. Once the task is completed, the action client will receive a response from the action server. Actions can be canceled while executing. This communication type is recommended for long-running tasks.

2.2 Micro-ROS

Micro-ROS is a software solution to develop robotic applications into resource-constrained systems such as microcontrollers. It was developed from the OFERA project, a joint effort of five European partners: eProsima, FIWARE, Bosch, PIAP, and Acutronic Robotics [12]. According to [13], the major objective of Micro-ROS is to fill the gap between resource-constrained microcontrollers and larger processors in robotic applications which are based on the Robot Operating System. The goals of Micro-ROS are:

- Micro-ROS integrates MCUs (Microcontroller Units) with ROS2, which means the standard ROS2 communication tools can access the software on an MCUs via the main processor.
- Micro-ROS brings the main concepts and interfaces of the ROS2 Client Library to MCUs so that software can be ported easily.

Micro-ROS follows the ROS2 architecture and makes use of its middleware pluggability to use DDS-XRCE (DDS standard for Extremely Resource-Constrained Environments), which is an optimized version of DDS for microcontrollers. Figure 4

demonstrates the architecture of Micro-ROS based on ROS2 architecture. In the client library layer, the extension of the ROS Client Support Library rcl by the rclc package makes it a full C API. As [6] mentioned, Micro-XRCE-DDS couples with the standard DDS middleware used in ROS2 by the ROS2 agent. The agent takes over discovery and Quality of Service mechanisms that are too computationally expensive for the microcontroller. Moreover, XRCE-DDS Client library uses POSIX-based RTOS (FreeRTOS, Zephyr, or NuttX) instead of Linux and Windows.

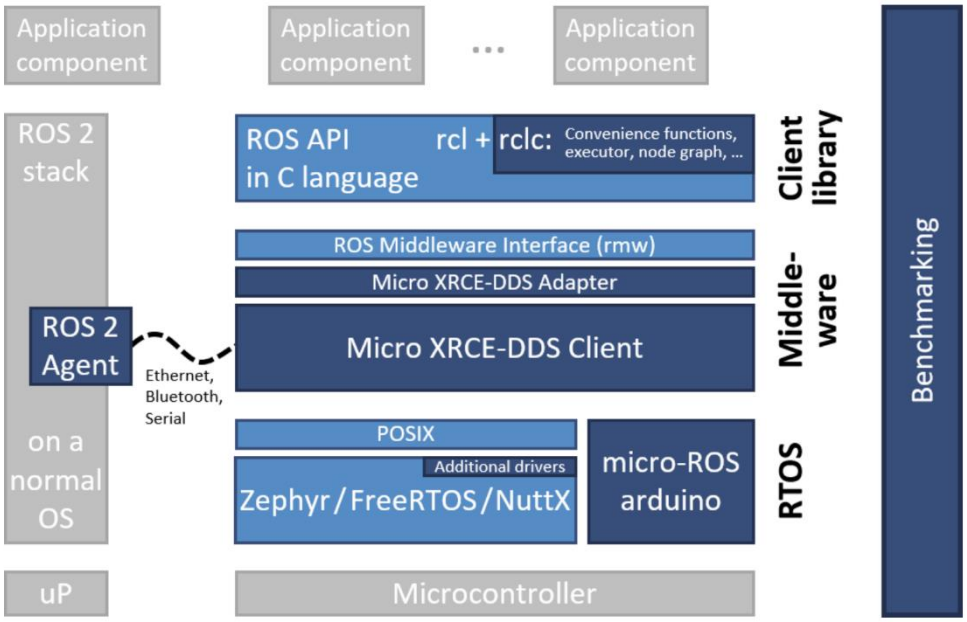

Figure 4: Architecture of Micro-ROS stack. The dark blue layers and components are specifically designed for Micro-ROS [6]

2.3 QoS

DDS provides various sets of Quality of Service (QoS) policies to modify the behaviour of communications between nodes. These QoS policies can be used individually or together to affect a variety of communications aspects, including reliability, performance, the persistence of data, and the number of system resources used [7]. QoS policy options allow for changing the behaviour of communication within a ROS2 network. QoS policies are amended for specific communication objects, such as publishers and subscribers. QoS policies also change the way messages are controlled within the object and transferred among them.

According to [17], durability, reliability, queue depth and sample history storage, Deadline, Lifespan, Liveliness and Lease Duration QoS policies are supported by ROS2. Table 1 provides details of the QoS Policy supported by ROS2

Table 1: QoS Policies of ROS2 [11]

HISTORY	Message caching. Sub-policies for samples storage include: - Keep last: only store up to N samples, configurable via the queue depth option. - Keep all: all samples transmitted by a node are stored in the cache of the data writer
DEPTH	Message queue size: only honored if used together with "keep last"
RELIABILITY	Delivery guarantee of messages - Best effort: attempt to deliver samples without arrival confirmation from the receiver, may lose them if the network is not robust. - Reliable: guarantee that samples are delivered with arrival confirmation, may retry multiple times.
DURABILITY	The behaviour of a node with samples which existed on a topic before the subscriber joined. - Transient local: only applies to DataWriters. All samples stored in the cache of the data writer are sent over to the data reader. - Volatile: no attempt is made to persist samples.
DEADLINE	The determined duration within which messages are sent or received.
LIFESPAN	The duration for which a message is valid.
LIVELINESS	The configuration for Publisher and Subscriber to ensure valid connection.
LEASE DURATION	For the maximum duration, a publisher has to show that it is alive before being indicated to have lost liveliness (losing liveliness could be an indication of a failure) by the system.

The QoS policies must be compatible with any messages transmitted between two communication objects. This is a prerequisite for ROS2 to be able to communicate. Only when publisher and subscriber have compatible QoS policies, a connection between them can be made. The compatibility summary of the different policy settings is shown in

Table 2.

Table 2: Summary of compatibility of Reliability and Durability QoS policies

Policy	Publisher	Subscriber	Compatible
Reliability	BEST_EFFORT	BEST_EFFORT	Yes
	BEST_EFFORT	RELIABLE	No
	RELIABLE	BEST_EFFORT	Yes
	RELIABLE	RELIABLE	Yes
Durability	VOLATILE	VOLATILE	Yes
	VOLATILE	TRANSIENT_LOCAL	No
	TRANSIENT_LOCAL	VOLATILE	Yes
	TRANSIENT_LOCAL	TRANSIENT_LOCAL	Yes

The work in this project will explore the impact of varying QoS policies on the communication performance of ROS2 and Micro-ROS.

CHAPTER 3 OBJECTIVES AND REQUIREMENTS

This chapter will describe in detail what the objectives of the project are and the requirements which are listed in MosCow format.

3.1 Objectives

As introduced in Chapter 1, the goal of this project is to research and investigate the usability of ROS2/ Micro-ROS and show its potential benefits as well as limitations for the use in Benchmark's products.

The market focus of Benchmark Electronics is commercial aerospace, complex industrial, defense and medical, in which the machines or robots used comprise multiple boards, each with its own tasks. Therefore, to ensure those parts work smoothly together to implement functionality, a robust communication system to connect those are needed. As mentioned in section 2.1, ROS2 communications use Data Distribution Service (DDS) as its communication middleware. DDS is suitable for real-time embedded systems thanks to its varied transport configurations (e.g., deadline, reliability, and durability) and scalability [9].

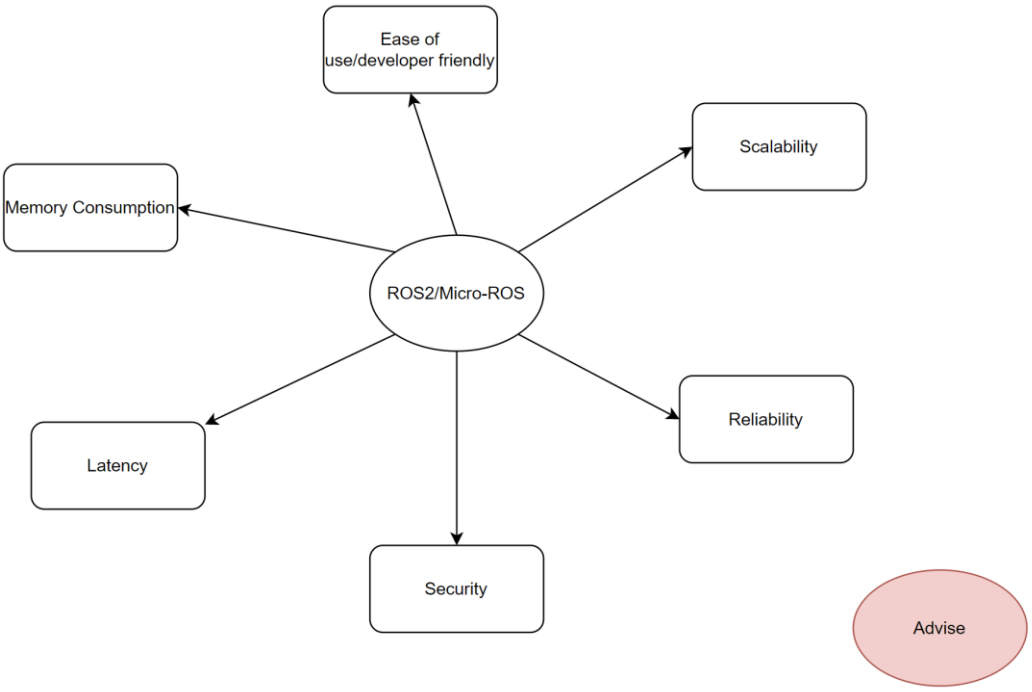

Figure 5: Illustration of the scope of the project

Therefore, this project aims to evaluate the potentials and constraints of ROS2/ Micro-ROS performance in terms of latency, memory consumption, scalability, and reliability shown in Figure 5 and then advise Benchmark when to use it and when not to use it in fields above or more in the future. In order to achieve this goal, an experimental setup which can have multiple boards running ROS2/ Micro-ROS with their own tasks can be set up to support during the research period.

The whole project can be summarised into a research question with sub-questions to support and clarify the research question.

3.1.1 RESEARCH QUESTION

What is the usability of ROS2/ Micro-ROS for products developed by Benchmark?

3.1.2 SUB-QUESTIONS

1. What are the advantages of using ROS2/ Micro-ROS?
2. What are the disadvantages of using ROS2/ Micro -ROS?
3. When to use ROS2/ Micro -ROS?
4. When not to use ROS2/ Micro -ROS?
5. How can ROS2/ Micro -ROS be applied to Benchmark products?

To answer research questions and sub-questions, project requirements are designed based on MoSCow format that includes must, should, could and will not do activities was created in order to set priorities for the project.

3.2 Requirements

In order to begin conceptualizing any project the first step is getting the vision and requirements. Direct and honest communication with supervisor is the best way to know how and in what direction the project needs to proceed. The project is divided into functional and non-functional requirements that need to be met and that are the foundation of what the project will become. The requirements are listed in MosCoW format per part of this project.

3.2.1 FUNCTIONAL REQUIREMENTS

The functional requirements will cover all features that the project should include. The functional requirements are:

1. The project must be able to perform communication between running devices to ROS2 and devices running Micro -ROS.
2. The project must have different latency evaluation scenarios to evaluate ROS2 and Micro-ROS communications.
3. The project must be able to assess the throughput of ROS2 and Micro-ROS.
4. The project must measure the memory size of a shared library and memory footprint objects in ROS2 and Micro-ROS.
5. The project must provide QoS policy settings, which will be used to evaluate communication behaviours.
6. The project must have evaluations of the scalability of ROS2 and Micro-ROS nodes and also make ROS nodes themselves scalable.
7. The project must advise on the security of using ROS2 and Micro-ROS.
8. The project demo must function both via Ethernet cable and Wi-fi connections.
9. The project should have evaluations of the reliability of using ROS2 and Micro-ROS systems.
10. The project could be implemented with different DDS vendors
11. The project could investigate the security aspects of ROS2/ Micro-ROS.
12. The project will not design a new communication protocol.
13. The project will not use Bluetooth

3.2.2 NON-FUNCTIONAL REQUIREMENTS

Besides functional requirements, the project also includes non-functional requirements which are listed below:

1. The project must give advice on the usability of ROS2 and Micro-ROS.
2. The project must be developer-friendly by having a place to store all source codes, software installation as well as developer manuals. Besides, codes must be reusable and expandable.
3. The project should have a demo for Benchmark based on test setups at the end of the project.
4. The project will not implement too complicated or fancy test setups, but it has to have sufficient experiment setups for research.

CHAPTER 4 EVALUATION FRAMEWORK

Knowing the idea and understanding the desires and requirements of the project is not enough to immediately start designing the whole project. This section provides an experimental setup overview which contains the description of the project test setups and all related considerations to judge the outcome.

This part presents the experimental setup which is shown in Figure 6 used to evaluate the performance of ROS2 and Micro-ROS communications over both Ethernet and Wi-Fi. As mentioned in the requirements in section 3.2, the focus of this project will be exploring and researching the usabilities of ROS2 and Micro-ROS. Therefore, the experimental setup has re-used available hardware in the company such as Jetson Nano developer kits.

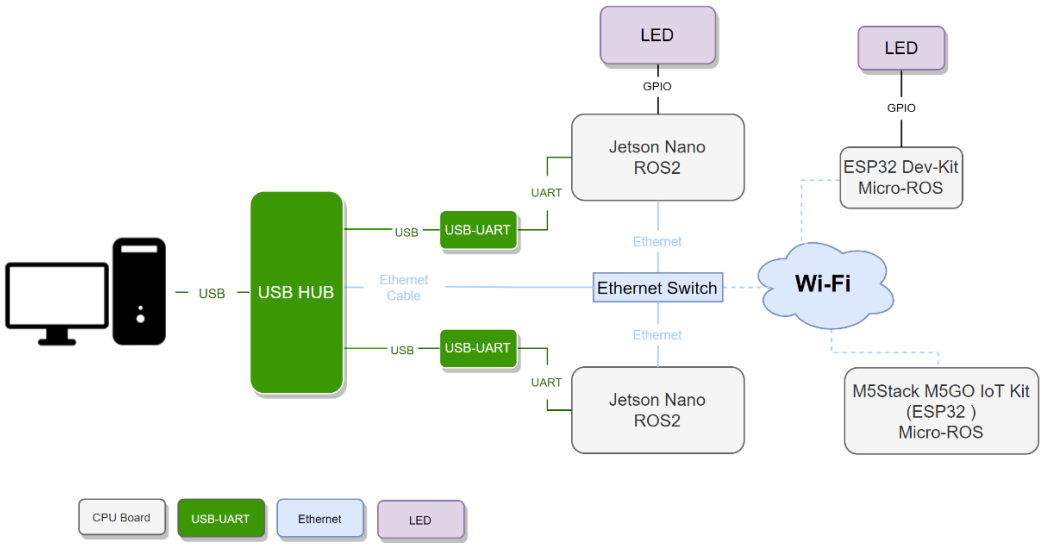

Figure 6: Overview of the experimental setup

Jetson Nano and ESP32 development kits were selected to run ROS2 and Micro -ROS in this experimental setup respectively. The experimental setup used a developer PC to integrate, control or monitor development boards with debugging terminal via Ethernet or USB to UART cables. All of the Ethernet and USB to UART cables are routed via a USB-Hub which allows the developer PC to connect and control all the development boards within just one USB port.

Moreover, each development board was connected with LEDs or sensors to send or receive messages from other nodes via Ethernet and Wi-Fi connection. The technical specifications of the hardware and software environment are listed in Table 3.

Table 3: Evaluation Environment

	Jetson Nano Developer Kit	ESP32 Development Kit	M5GO IoT Starter Kit
Processors	Quad-core ARM A57 @ 1.43 GHz (4 cores)	Dual-core Tensilica LX6 microcontroller with 600 DMIPS @ 240MHz (2 cores)	
Memory	4 GB 64-bit LPDDR4 25.6 GB/s	Integrated 520 KB SRAM	
Network	100 Mbps Ethernet, Full-Duplex	2.4 GHz Wi-Fi	
ROS2 Version	Foxy Fitzroy		
DDS implementation	FastRTPS	XRCE-DDS	
OS	Ubuntu 20.04	FreeRTOS	

There are test setups which show different communication situations. They will be conducted separately inside the experimental setup in Figure 6 to evaluate communication performance between nodes in ROS2 and Micro-ROS. For example, the project can measure the end-to-end latencies between two ROS2 nodes in different machines which are depicted in Figure 7 such as ROS2 and ROS2 nodes in Jetson Nano and Jetson Nano respectively (Figure 7a); ROS2 and Micro-ROS nodes in Jetson Nano and ESP32 development board in turn (Figure 7b); two different Micro-ROS nodes in ESP32 development board and M5STACK M5GO IoT kit which include an ESP32 module (Figure 7c). All of these test setups communicate via a local host using both ethernet cable and Wi-Fi. Moreover, all of these tests will use *topics* styles of interfaces instead of services or actions because the publish-subscribe middleware system is designed and built as the core of ROS2 as mentioned in section 2.1. Besides, by using topics type, ROS2 communication can have more options to communicate such as many-to-many or one-to-many ROS2 communication.

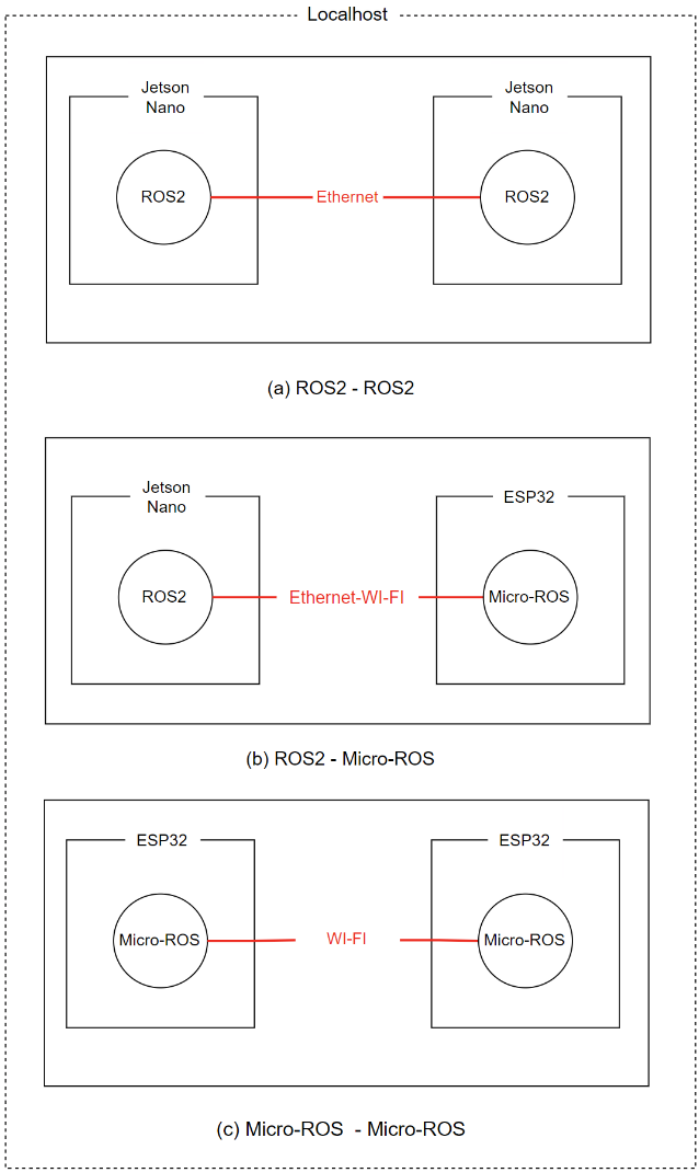

Figure 7: Test setup situations

Besides, the experimental setup also includes complex communication situations to evaluate the performance. For example, in the same network, one ROS2 node in Jetson Nano can send messages to other ROS2 nodes in other Jetson Nano and ESP32 boards.

Parameters that will be evaluated in this project include latency, memory consumption, reliability, scalability and security of ROS2 and Micro-ROS. Chapter 5 will elaborate on the details of each test why and how it was conducted, the results and the conclusion.

CHAPTER 5 EVALUATION

This chapter will present all the setup details and results of parameters which are mentioned in Chapter 4, then discuss these results. The first test is the latency test which is presented in section 5.1. Then section 5.2 will describe the memory consumption test, followed by the reliability test in section 5.3 and scalability in section 5.4. Finally, ROS2 security will be depicted in section 5.5. During the test, Python is the main language used in ROS2 applications and C for Micro-ROS.

5.1 Latency

Latency is one of the most important values used to evaluate the communication performance of ROS2 and Micro-ROS. Latency is defined as the delay between the time that a specific message is published by Publisher and the time that it is received by the Subscriber.

There are a lot of external factors that affect the latency in the communication of the experiment such as publisher frequency, message size/ payload, number of nodes and QoS policies. These parameter values will be used together to evaluate latency in specific test scenarios which are listed below. As mentioned in section 2.1, this project focuses on ROS2 not DDS, so all scenarios will be carried out in FastRTPS and XRCE-DDS for ROS2 and Micro-ROS respectively because they are the default DDS setup of ROS2 Foxy and Micro-ROS. Besides, as [26] proved, FastRTPS is remarkably faster when compared with CycloneDDS and Connext if running in ROS2 applications in Raspberry Pi.

- **Evaluation of Publisher Frequency:** in this test, the test setup uses the Round-Trip test which is depicted in Figure 8, in which the Publisher in Client will use different frequencies to transmit messages starting from 1Hz with a step size of 10Hz. The test defines the QoS reliability in BEST_EFFORT and all possible message payload sizes.

- **Evaluation of test setups under load:** in this test, the project will explore how latency gets affected when the test setup is under heavy load. To generate load,

stress and iPerf tool are chosen. Stress tool is a workload generator which provides CPU, memory and disk I/O stress tests, the CPUs are steadily running different calculations to simulate a heavy computational load. IPerf is a network performance measurement tool which is free and open source. iPerf operates by transferring messages from one host to another and measuring the throughput achieved [4]. In the same network, two ROS2 nodes will communicate with each other under heavy load which is run by stress tool but in the meantime, two different programs also transfer heavy data by using iPerf.

- **Evaluation of scalability of ROS2 nodes:** as shown in the experimental setup in Figure 6, the project will evaluate latency when the number of nodes is increased. It includes many publishers and subscriber nodes communicating with each other on different topics and in the same network.

The latency test is separated into three parts, section 5.1.1 is publisher frequency evaluation. Section 5.1.2 is the system under load test, and finally, section 5.1.3 will analyse the effect of scalability on latency in ROS2 and Micro-ROS.

The details of hardware and software are listed in Table 3 and the summarising of the parameter values which will be used during testing is listed in Table 4.

Table 4: Variable parameter values

Hardware	Jetson Nano, ESP32, M5STACK
Publisher Frequency	1 Hz, 10 Hz, 20 Hz, … 90 Hz, 100 Hz
Message	128B, 1KB, 10 KB and 100KB
Number of Nodes	1, 3, 5, 9, 15, 19
DDS	FastRTPS
Reliability QoS Policy	BEST_EFFORT
Durability QoS Policy	VOLATILE
Depth QoS Policy	10
History QoS Policy	KEEL_LAST

To obtain as many accurate results as possible, in all cases, each result is captured from the average of all samples over the period of 1 minute. They are recorded by using a logic analyzer shown in Figure 9.

5.1.1 IMPACT OF PUBLISHER FREQUENCY ON LATENCY

The purpose of this project is to investigate the usability of ROS2 and Micro-ROS, so this project will not specify a device or sensor. Therefore, the project will try to simulate as many frequency ranges as possible. For example, the GPS sensor operates update rate from 1Hz to 10 Hz [22], while the maximum update frequency of the accelerometer is 100Hz [23] or even higher for other sensors.

5.1.1.1 ROS2 and ROS2 Communication

A Description

This experiment was conducted to investigate the impact of publisher frequency on latency between Jetson Nano (ROS2) and Jetson Nano (ROS2) communication.

B Test setup

The test setup was conducted to evaluate end-to-end latencies between two nodes with different situations using both Ethernet wire and Wi-Fi. The communications between two machines were evaluated by using the Round-Trip Time (RTT) test, also called a ping-pong test as [3] used which is demonstrated in Figure 8:

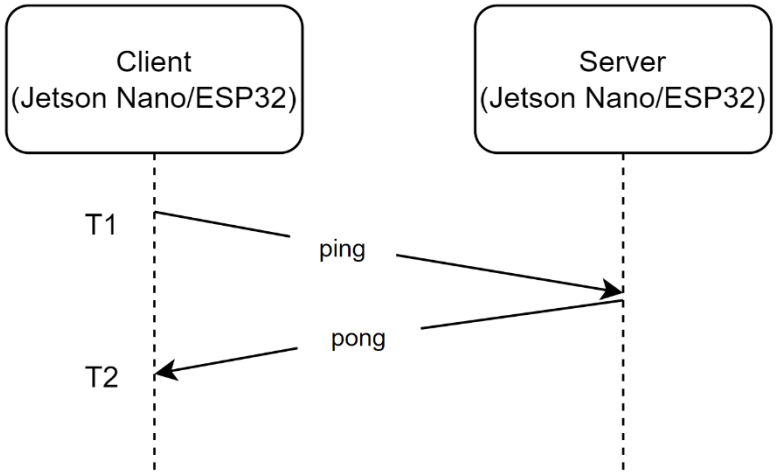

Figure 8: Graphical presentation of measured round-trip latency

The RTT test used Jetson Nano as the client and the server. Each client and server created two publisher and subscriber nodes and they had two different topics called *ping* and *pong*. To be more specific, the client published a message to a server via *ping* topic, the server had a subscriber which received the *ping* topic. On the other

hand, the server also created a publisher which replied to the same received message to the client which already created a subscriber via the *pong* topic.

The RTT latency was measured during the time it took for a message to travel from the client to the server, and from the server back to the client. The RTT was measured as the difference between the time-stamp taken when the message was sent from the client (T1) and the time-stamp taken when the message was received by the client (T2). Therefore, RTT was defined as T2 – T1. Assuming that there was no delay between receiving *ping* message and sending *pong* message, then the latency was defined as RTT/2.

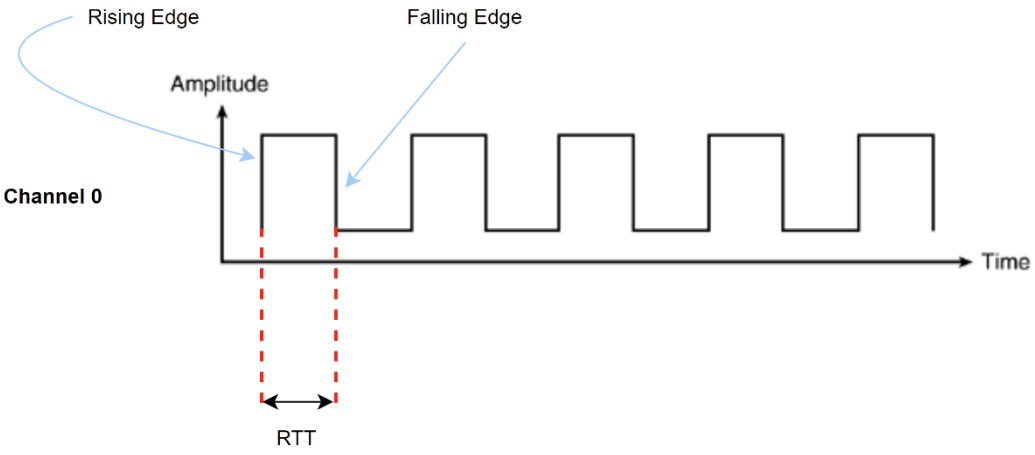

Figure 9: Capturing time-stamp using Logic Analyzer

The project used Logic Analyzer to capture the time stamps. Figure 9 shows how to use Logic Analyzer to capture time-stamp. The client device was attached with a LED, in which the publisher of *ping* topic was connected to the rising edge and the subscriber of pong topic was placed at the falling edge of LED signal. Then Logic Analyzer could attach channel A to pins of LED to measure the signal.

C Test execution and results

The tests were conducted as ping-pong test which is depicted in Figure 8 with a published frequency range from 1 Hz to 100 Hz. The QoS reliability which was used in all the experiments below was set BEST_EFFORT because the project tried to get the maximum publisher frequency and lowest latency of ROS2 communication. Besides, during the test, the project used various message sizes that ROS2 could transfer in this test such as 128 Bytes, 1 Kilobyte, 10 Kilobytes and 100 Kilobytes.

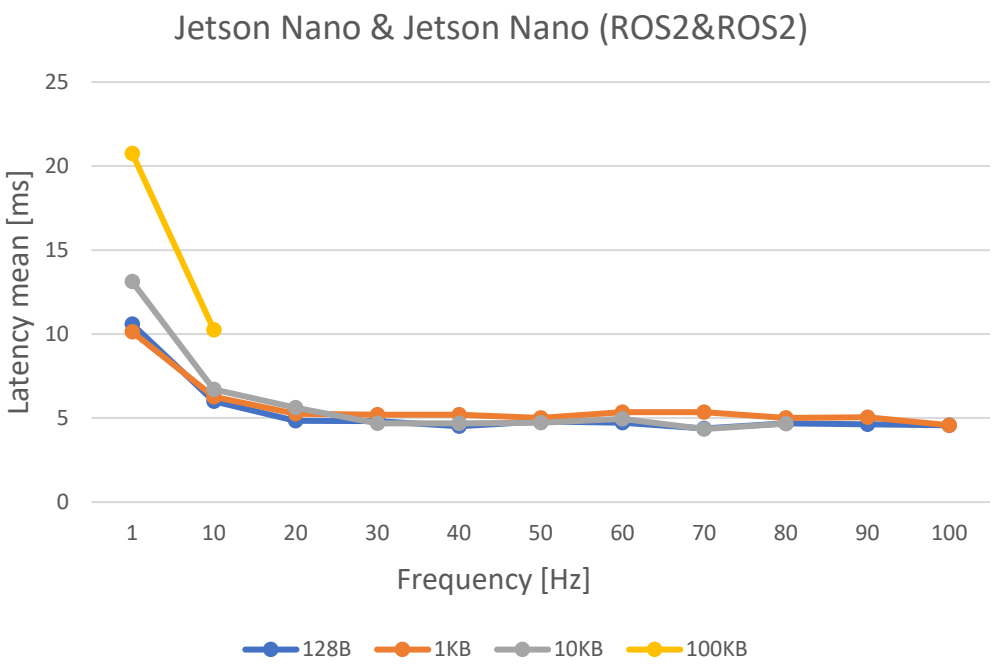

Figure 10: Investigating the influence of publisher frequency on latency between two Jetson Nano running ROS2 application

From Figure 10 (and Table 20), it can be seen that latency dramatically decreases when publisher frequency increases in all message size cases from 1 to 20 Hz. After that, values in the line graph are approximately equal, only slightly fluctuating from 20 Hz to 100 Hz of publisher rate due to different samples recorded in different publisher frequencies. The reason for this might be energy-saving features in hardware devices or reprioritized thread scheduling [26].

Besides, it can also be seen from Figure 10 that the latencies of 128 Bytes, 1 and 10 Kilobytes are almost the same but the latency of 100 Kilobytes payload – 20,75 ms is much higher compared to others which are around 10 ms and 13 ms at 1Hz. This can be explained by the limit of 64 Kilobytes for a UDP datagram [27] which leads to fragmentation for large messages.

D Evaluation and conclusion

To investigate what happened inside the experiment with strange values recorded between the 100 Kilobytes packet and others. The project picked a packet of 128 Bytes which had the largest latency range and a packet of 100 Kilobytes which had the smallest frequency range to calculate the standard deviation and compare it with the mean value. As Figure 11 shows, the standard deviation hovers around the mean

values from 20 Hz to 100 Hz. However, the latency ranges of both 128 Bytes and 100 Kilobytes at 1 and 10 Hz are larger compared to others. It can therefore be concluded that the higher the frequency, the lower the latency.

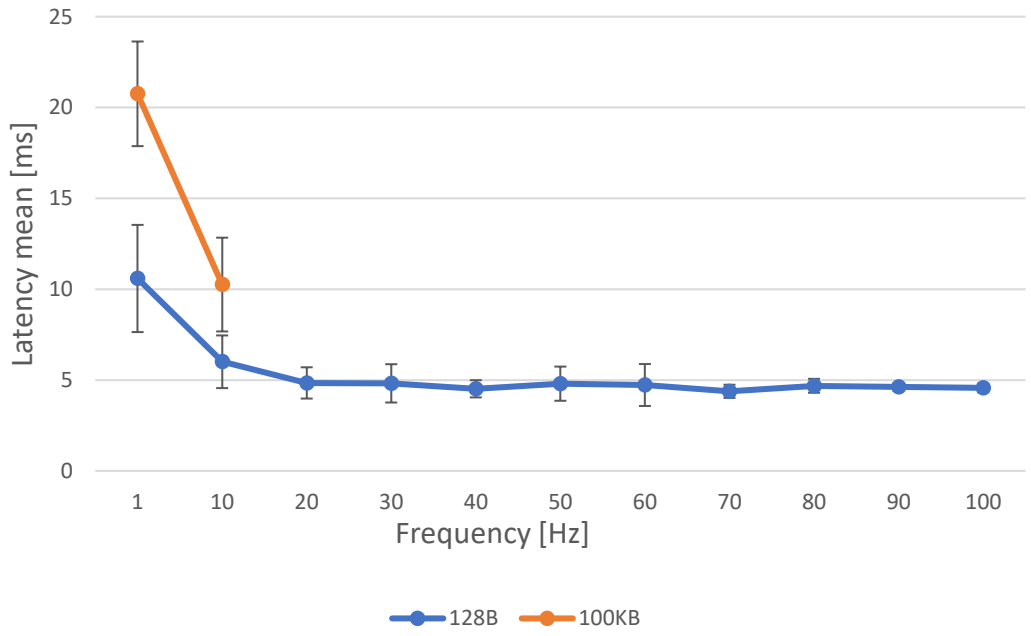

Figure 11: Illustrate of standard deviation with the mean value of 128B and 100KB packets

5.1.1.2 Micro-ROS and Micro-ROS Communication

A Description

This experiment was conducted to investigate the impact of publisher frequency on latency between ESP32 development board (Micro-ROS) and M5 STACK kit (Micro-ROS) communication.

B Test setup

This experiment was run on FreeRTOS and ESP32 boards connected by UDP (via Wi-Fi) to a micro-ROS Agent running on a Linux machine. The tests were conducted by re-using the ping-pong test with the publisher frequency range from 1 Hz to 100 Hz and BEST_EFFORT for reliability QoS policy as mentioned in Table 4. Similarly, a logic analyzer was also re-used to capture the time-stamp of the rising edge and the falling edge of RTT.

C Test execution and results

The project used two message sizes 128 Bytes and 1 Kilobyte. With running Micro-ROS, the test only applied two different payload sizes of messages because it depended on the module of the microcontroller used, which was ESP32 in this test. The payload size of messages was calculated at building time which means the message of Micro-ROS is only fixed-size to avoid dynamic memory allocations [21].

With the small messages below 512 Bytes, the communication can run without any problems, given the default setup of RMW (ROS Middleware Interface) maximum buffer size. However, if the program wants to send messages above 512 Bytes, the RMW maximum buffer size has to be configured by the CMake flags command below:

- With the best-effort stream of RELIABILITY QoS Policy: The total RMW_UXRCE_MAX_BUFFER_SIZE that the best-effort stream can transfer is equal to the value which is set by this command "UCLIENT_<*transport_type*>_MTU".
- With the reliable stream of RELIABILITY QoS Policy: The total RMW_UXRCE_MAX_BUFFER_SIZE that the reliable stream can transfer is equal to the value which is set by this command: "UCLIENT_<*transport_type*>_MTU * RMW_UXRCE_STREAM_HISTORY", this should be at least 2048.

UDP protocol was used for "*transport_type*" in this experiment. Serial communication is also one of the transport types of Micro-ROS but this project only employed wireless connection for Micro-ROS.

As Figure 12 (and Table 21) illustrate, the maximum publisher rate of 1 Kilobytes payload is 60 Hz which is significantly low compared with ROS2 test. Furthermore, the results in Figure 12 had the same pattern as the ROS2 test setup in 5.1.1.1, in which the latency significantly decreased when the publisher rate increased from 1 to 20 Hz and the latency slightly decreased when the publisher rate went up from 20 to 100 Hz.

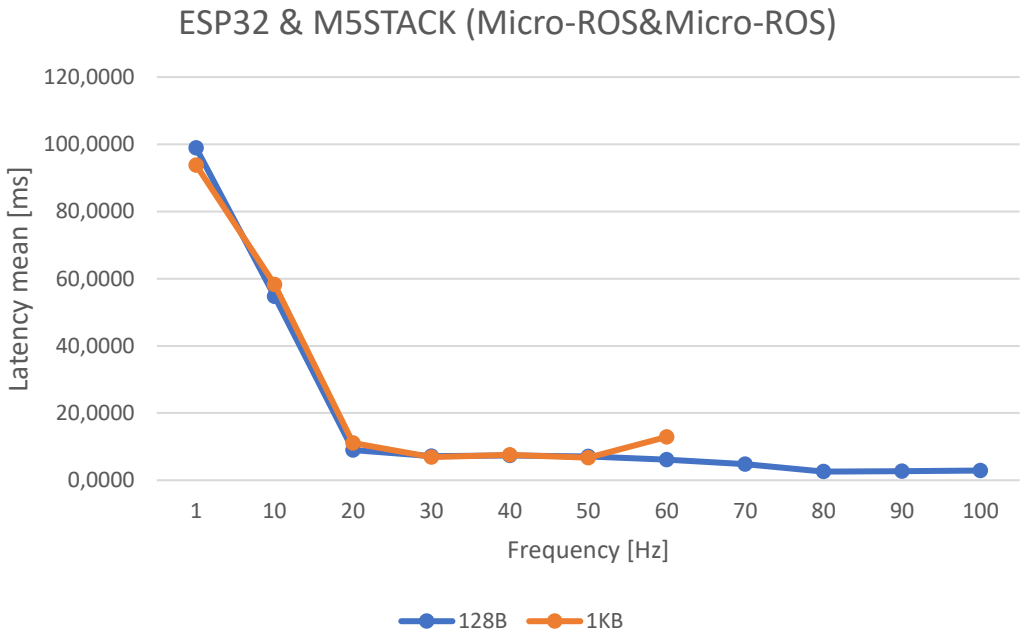

Figure 12: Investigating the influence of publisher frequency on latency between two ESP32 running Micro-ROS applications, using BEST_EFFORT stream

D Evaluation and conclusion

The results show that values in Figure 12 had the same trend as the ROS2 test setup in 5.1.1.1. The higher the publisher rate, the lower the latency, especially from 1 to 20 Hz. However, with the 1 Kilobyte packet, the maximum publisher frequency was only around 60 Hz which is quite low compared to ROS2 communication. This is due to the quality of network speed and different microcontroller modules used.

On the other hand, the difference in result values between 128 Bytes and 1 Kilobyte packets is not noteworthy. It can be concluded that the message size in this experiment does not have an impact on the latency of Micro-ROS communication.

5.1.1.3 ROS2 and Micro-ROS Communication

A Description

In this test, the project investigated the behaviour of the communication latency between ROS2 (Jetson Nano) and Micro-ROS (ESP32).

B Test setup

The Round-Trip test was also re-used for this experiment with two message sizes 128 Bytes and 1 Kilobyte and the same setup in Micro-ROS and Micro-ROS test in section 5.1.1.2 was applied.

C Test execution and results

Figure 13 demonstrates that the results between ROS2 and Micro-ROS experienced the same trend as between Micro-ROS and Micro-ROS tests. However, the communication latency between ROS2 and Micro-ROS was lower than between Micro-ROS and Micro-ROS test, for example, the communication latency of the former is approximately 60 ms (Figure 12) compared with 100 ms (Figure 13) of the latter at 1 Hz of both 128 Bytes and 1 Kilobyte packets

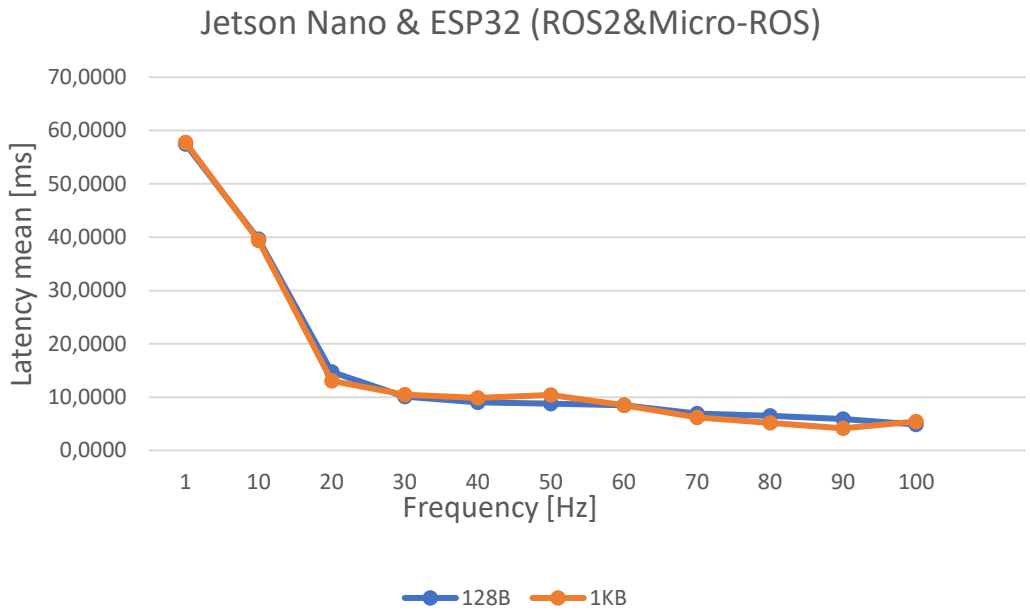

Figure 13: Investigating the influence of publisher frequency on latency between Jetson Nano and ESP32 running ROS2 and Micro-ROS application respectively, using BEST_EFFORT stream

D Evaluation and conclusion

It can be concluded that the communication between ROS2 and Micro-ROS was faster than between Micro-Ros and Micro-ROS, but slower than between ROS2 and ROS2. This is because the time that Jetson Nano (ROS2) which uses ethernet cable to transmit data to the network is shorter than ESP32 (Micro-ROS) which uses UDP to transmit data.

5.1.1.4 Conclusion

To conclude from the results, the latency of ROS2 and Micro-ROS communication is affected when changing the publisher frequency. Moreover, the lowest latency can be achieved by using messages that are below 64 Kilobytes. Besides, messages with a payload below 64 Kilobytes do not have an impact on the communication latency.

It can be extracted from the results that the communication between ROS2 and ROS2 is the fastest, followed by ROS2 and Micro-ROS. Communication between Mico-ROS and Micro-ROS is the slowest.

5.1.2 SYSTEM UNDER LOAD

5.1.2.1 Description

In reality, the system not only runs ROS2 or Micro-ROS applications but they will also be combined to run together with other applications, even with resource-intensive programs. Therefore, in this test, the project explored how communications got affected when the system was under heavy load and with network traffic.

5.1.2.2 Test setup

As introduced from the beginning of section 5.1, iPerf was used to create network traffic between two devices and the stress tool was used to generate load in the system. This test utilized iPerf to send 4, 40 and 80 Megabytes per second from Jetson Nano running iPerf server to another Jetson Nano running iPerf client. In both Jetson Nano client and server, stress tool generated a CPU stress with 2 CPU workers (spinning on sqrt() function), 2 VM workers (spinning on malloc()/free() functions), 2 I/O workers (spinning on sync() function) and 2 HDD workers (spinning on write()/unlink() functions).

The ping-pong test was also re-used in this experiment. For the *ping* and *pong* messages, this test used two payload sizes which are 128 Bytes and 1 Kilobyte. And the latency of captured with publisher frequencies of 1 Hz to 100 Hz.

5.1.2.3 Test execution and results

The test was separated into two different sub-tests including testing under network load using iPerf in Figure 14 (Table 23) and testing under system and network load using iPerf and stress tool in Figure 15 (Table 24).

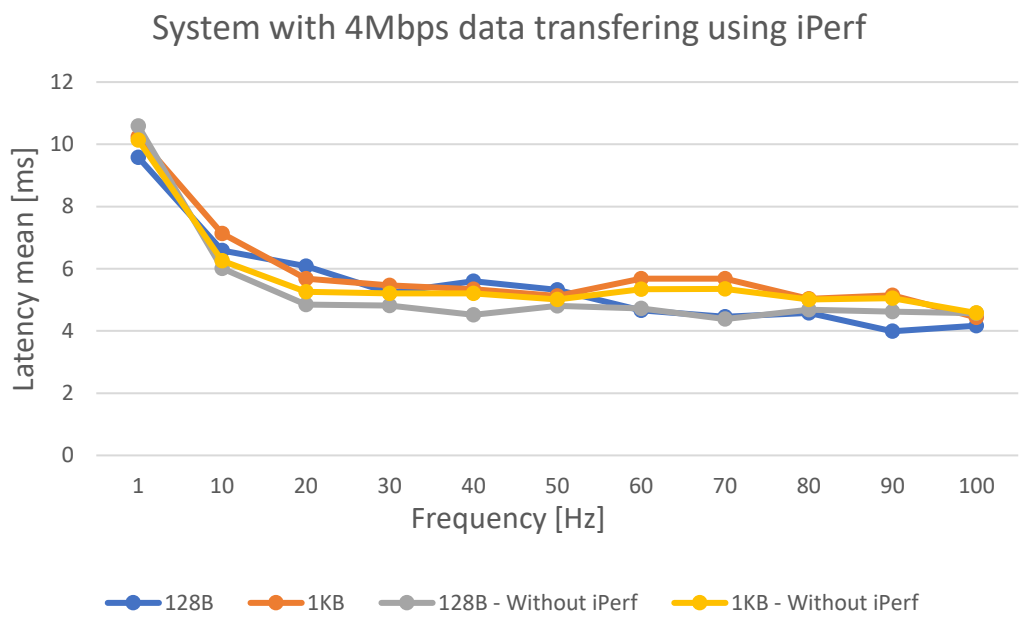

Figure 14: Investigation of the impact of network load on the latency of ROS2 applications

In the former test, iPerf was used to stream 4Mbps which represented IP cloud 1080P 30FPS camera in real-life application. To generate iPerf server, "*iperf -s -u*" command was used in which "*-s*" is server and "*-u*" is created UDP streams. And to generate iPerf client, "*iperf -c <server ip> -u -b 4M -t 43200*" command was run, in which "*-b*" was used to set target bandwidth and "*-t*" to set the duration that iPerf client ran.

Figure 14 shows that there is no difference between the latency of ROS2 communication with 128 Bytes and 1 Kilobyte with and without iPerf respectively. Besides, based on the results from Figure 14, it can be seen that streaming 4Mbps has little effect on the latency of ROS2 communication.

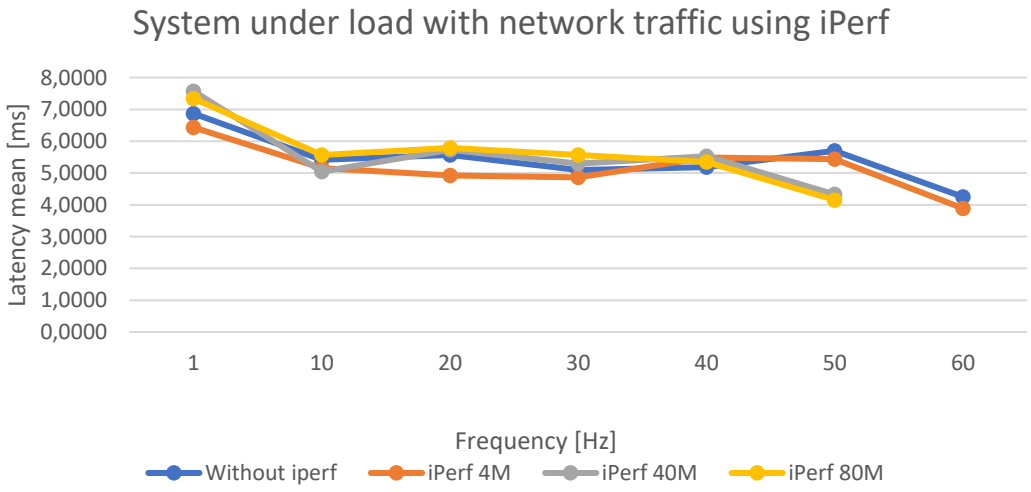

Figure 15: Investigation of the impact of system and network load on the latency of ROS2 applications

The result of the latter test is shown in Figure 15. In this test, a payload of 128 Bytes was used. "*sudo stress --cpu 2 --io 2 --vm2 --hdd 2 --timeout 43200s*" command was used to generate the stress tool. During running stress tool, Jetson Nano consumed a high amount of CPUs which are around 70% to 100% compared with the ideal environment (without running stress tool) which only costs around 5%. Figure 15 shows that latencies fluctuated and were severely affected by the stress tool. The publisher frequency was hardly influenced and the maximum publisher rate that the program could operate also depended on how much CPU was used. This happens mainly because the program threads were contending with the CPU time. Some latencies might also be impacted by memory page faults which led to the memory not being locked [3]. Therefore, getting stable results of latency in this experiment is not possible.

5.1.2.4 Evaluation and conclusion

Based on the results, it can be concluded that there is no difference between the latency of ROS2 communication with 128 Bytes and 1 Kilobyte in the under-network load test. This means network traffic does not have an impact on communication latency. In addition, under heavy loads, the latency communication cannot be measured properly because program threads conflict with the CPU time and memory is not locked caused of memory page faults.

5.1.3 SCALABILITY

5.1.3.1 Description

This test was conducted to investigate the latency performance of ROS2 applications when the number of nodes and topics increased. This test was separated into two common use cases namely data-processing pipeline and running multi topics in parallel.

5.1.3.2 Test setup

The first test which is illustrated in Figure 16 is called the data-processing pipeline with different ROS2 topics. This is quite a commonly seen use case in robotics [20]. The first node sent a message through several nodes with different topics. Each node

contained one publisher and subscriber to receive from the previous node and send to a later node until the node at the end of the pipeline.

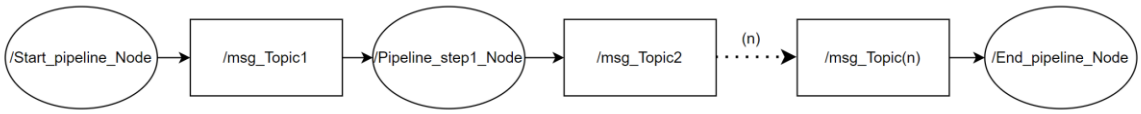

Figure 16: Visualisation of data-processing pipeline setup

The second test depicted in Figure 17 below included multiple nodes with different topics which would run in parallel.

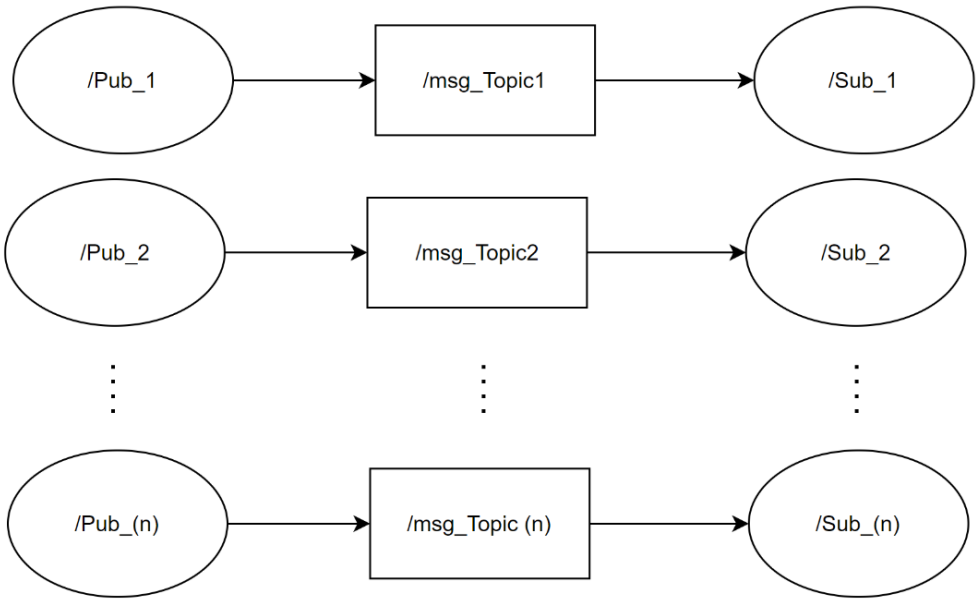

Figure 17: Visualisation of scalability test with running different topics in parallel

5.1.3.3 Test execution and results

The data-processing pipeline running from 3 to 19 nodes was used to measure end-to-end latency between the starting node and the end node of the program. Besides, payload sizes of messages used in this test were 128 Bytes and 1 Kilobyte. This experiment focused on evaluating the latency of scalability, so a few publish frequency steps were ignored. In both Figure 18 and Figure 19, it can be noticed that the higher the frequency, the lower the latency. Besides, different payload sizes did not affect the latency. On the other hand, it is easy to see that when the number of nodes went up, the latency also increased. However, in the case of running 15 and 19 nodes above 80 Hz, the latency dropped compared with previous nodes. Hence, the Jetson Nano seemed to be overloaded in this area.

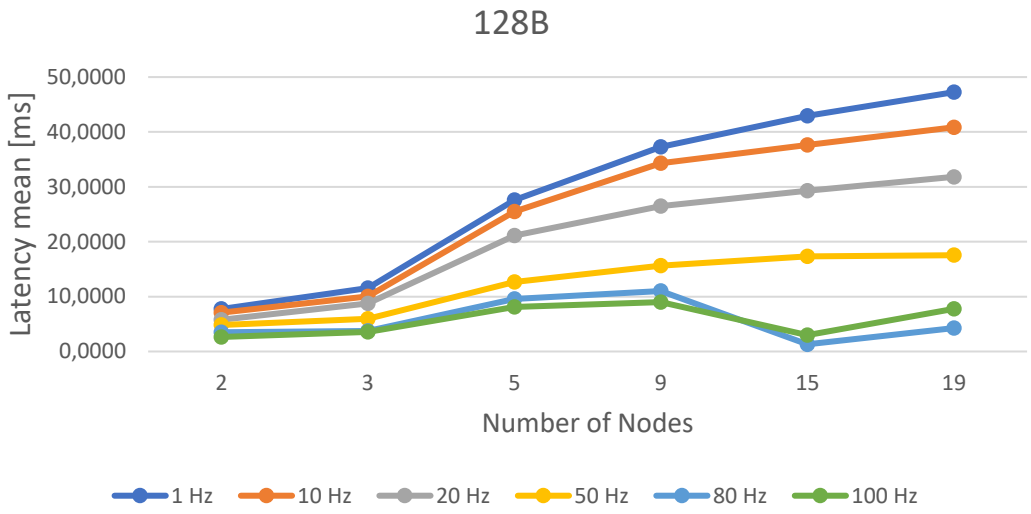

Figure 18: Investigation of the scalability of ROS2 system running in the data-processing pipeline on latency with 128 Bytes payload size

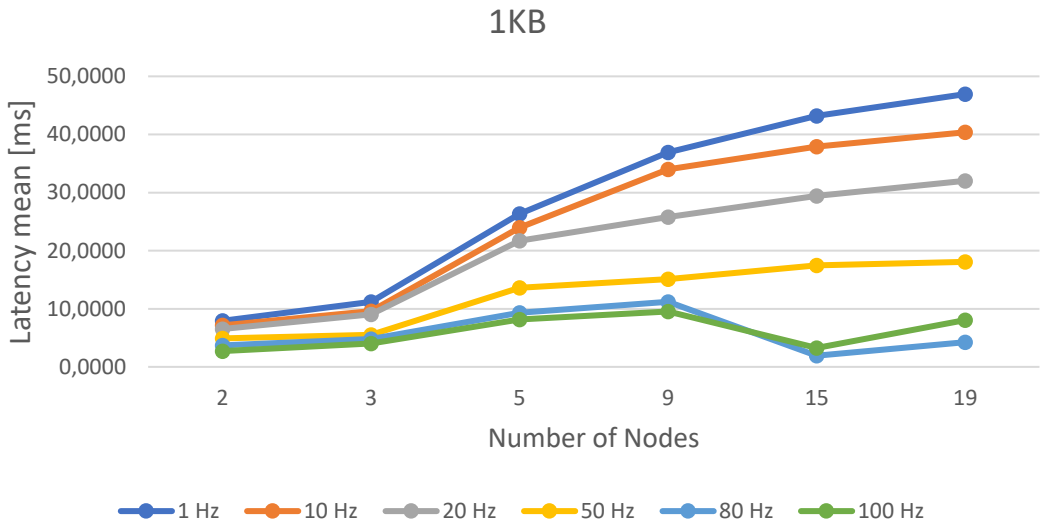

Figure 19: Investigation of the scalability of ROS2 system running in the data-processing pipeline on latency with 1 Kilobyte payload size

Figure 20 shows the results of ROS2 scalability test with different topics running in parallel. The Round-Trip Test was re-used in this experiment. This test was conducted on 4, 6, 12, 18 and 24 different topics and was compared with the publisher frequency test of ROS2 which ran 2 different topics in section 5.1.1. The latency was only measured for the 2 topics, and other topics were running in the same application without measurement.

Figure 20 (and Table 27) show that when the number of topics went up, the latency fluctuated, but insignificantly. This might be due to different samples and jitters during measurement. Therefore, running different topics parallel with each other in the same application did not influence communication latency.

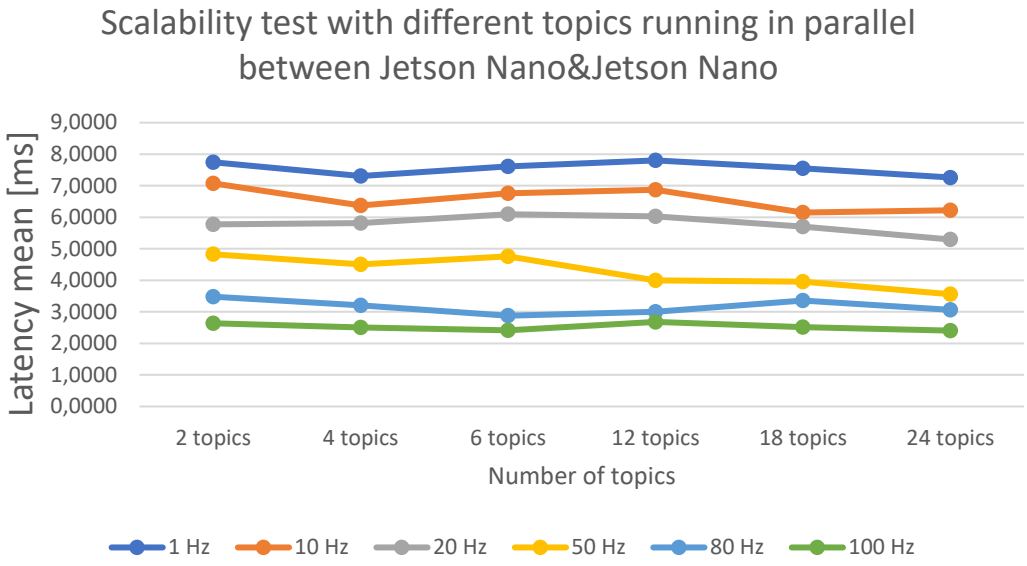

Figure 20: Investigation of the scalability of ROS2 system with different topics running parallel on latency with 128B payload size

5.1.3.4 Evaluation and conclusion

To investigate why when Jetson Nano was overloaded, latency values which were recorded in 15 and 19 nodes at 80 Hz and 90 Hz dropped, the times-tamps were used to evaluate the behaviour of communication which is shown in Figure 21 and the signal captured from Logic Analyser which is shown in Figure 22. It can be seen that the publisher and subscriber did not transmit messages continuously and messages were crossed each other. This happened because the publisher and subscriber in both Jetson Nano were not synchronizing and the duration between the rising edge and the falling edge was measured incorrectly.

The solution for this issue is adding the ability to wait until a new message is received then the Publisher starts sending a new message. However, this feature was not set officially for ROS2 yet. Therefore, this feature needs to be improved in future research.

```
[listener-2] [INFO] [1654366909.841629955] [subscriber]: Sub at 2022-06-04 20:21:49.837093
[listener-2] [INFO] [1654366909.856954226] [subscriber]: Sub at 2022-06-04 20:21:49.850152
[listener-2] [INFO] [1654366909.868015737] [subscriber]: Sub at 2022-06-04 20:21:49.864393
[talker-1] [INFO] [1654366909.862365268] [publisher]: Publishing ... at 2022-06-04 20:21:49.853195
[talker-1] [INFO] [1654366909.877475007] [publisher]: Publishing ... at 2022-06-04 20:21:49.868336
[listener-2] [INFO] [1654366909.882730320] [subscriber]: Sub at 2022-06-04 20:21:49.877596
[talker-1] [INFO] [1654366909.891398757] [publisher]: Publishing ... at 2022-06-04 20:21:49.886141
[talker-1] [INFO] [1654366909.906663653] [publisher]: Publishing ... at 2022-06-04 20:21:49.900752
[listener-2] [INFO] [1654366909.895399591] [subscriber]: Sub at 2022-06-04 20:21:49.890221
[listener-2] [INFO] [1654366909.907761570] [subscriber]: Sub at 2022-06-04 20:21:49.903244
[listener-2] [INFO] [1654366909.923678653] [subscriber]: Sub at 2022-06-04 20:21:49.914810
[talker-1] [INFO] [1654366909.929180320] [publisher]: Publishing ... at 2022-06-04 20:21:49.916477
[listener-2] [INFO] [1654366909.945772976] [subscriber]: Sub at 2022-06-04 20:21:49.934481
[listener-2] [INFO] [1654366909.962892455] [subscriber]: Sub at 2022-06-04 20:21:49.955507
[talker-1] [INFO] [1654366909.956315997] [publisher]: Publishing ... at 2022-06-04 20:21:49.943737
[talker-1] [INFO] [1654366909.972283289] [publisher]: Publishing ... at 2022-06-04 20:21:49.965562
[listener-2] [INFO] [1654366909.974856518] [subscriber]: Sub at 2022-06-04 20:21:49.970204
[listener-2] [INFO] [1654366909.989240476] [subscriber]: Sub at 2022-06-04 20:21:49.982573
```

Figure 21: The capture of time-stamp in terminal of scalability test of 15 nodes at 80 Hz

Figure 22: The capture of time-stamp in logic analyzer of scalability test of 15 nodes at 80 Hz

It can be concluded that in the scalability test, running different topics in parallel does not have an impact on the latency. In contrast, when expanding the number of nodes in the data-processing pipeline experiment, the latency also rises. However, with regards to 15 and 19 nodes running at over 80 Hz, the latency values cannot be measured accurately due to the interruption in message transmission between the Publisher and the Subscriber.

5.2 Memory consumption

5.2.1 DESCRIPTION

The purpose of the memory consumption test is to investigate the memory usage of ROS2 and Micro-ROS applications on the device. Besides, different aspects were also included to see if they had an impact on memory consumption of ROS2 and Micro-ROS applications, for example, publisher frequency, different message size, number of nodes and topics. The project investigated the memory size of the shared library and memory footprint object of ROS2 and Micro-ROS. Shared Libraries are libraries that are dynamically loaded by nodes when they start. The memory footprint of ROS2 entities measured in this test entail nodes, subscribers, publishers and messages.

5.2.2 TEST SETUP

Before investigating the memory used in ROS2 and Micro-ROS, the project took values from a basic program which printed the “Helloworld” string, then the project compared with values taken from the applications running ROS2 such as creating a node, a publisher, a subscriber, many publishers and subscribers or changing message size and publisher rate. Besides, the package size of Micro-ROS application flashed into ESP32 was also investigated. In computing, resident set size (RSS) is the percentage of the memory taken by a process kept in the main memory, so these values were used to evaluate memory consumption.

The *pmap tool* on Linux (Jetson Nano) was used to measure memory usage in this project. This tool provided information about memory metrics like RSS used by each shared library loaded by the application. The main languages used in this project included python and C, so this document focused on analysing the memory consumption of these programs. However, ROS2 C++ application was also mentioned in this test to compare with ROS2 python program.

5.2.3 TEST EXECUTION AND RESULTS

This experiment started with running a non-ROS2 application which printed string value as “Hello world”, then ROS2 shared library was applied to this program starting with creating ROS2 node. After that, the publisher and subscriber were added to the program to see if the memory consumption changed. As Figure 23 demonstrates, the total RSS memory of the “HelloWorld” program only consumed 10000 Kilobytes ≈ 10 Megabytes but when applying ROS2 shared library to the program, the total RSS memory increased to 40000 Kilobytes ≈ 40 Megabytes. So, it can be seen that 30000 Kilobytes ≈ 30 Megabytes were added to the total memory consumption due to ROS2 shared library. The total shared library when running the “HelloWorld” program was only 12 libraries which are shown in Table 28 but when ROS2 node was created, the total shared library went up to 70 libraries (Table 29), so 58 shared libraries were added.

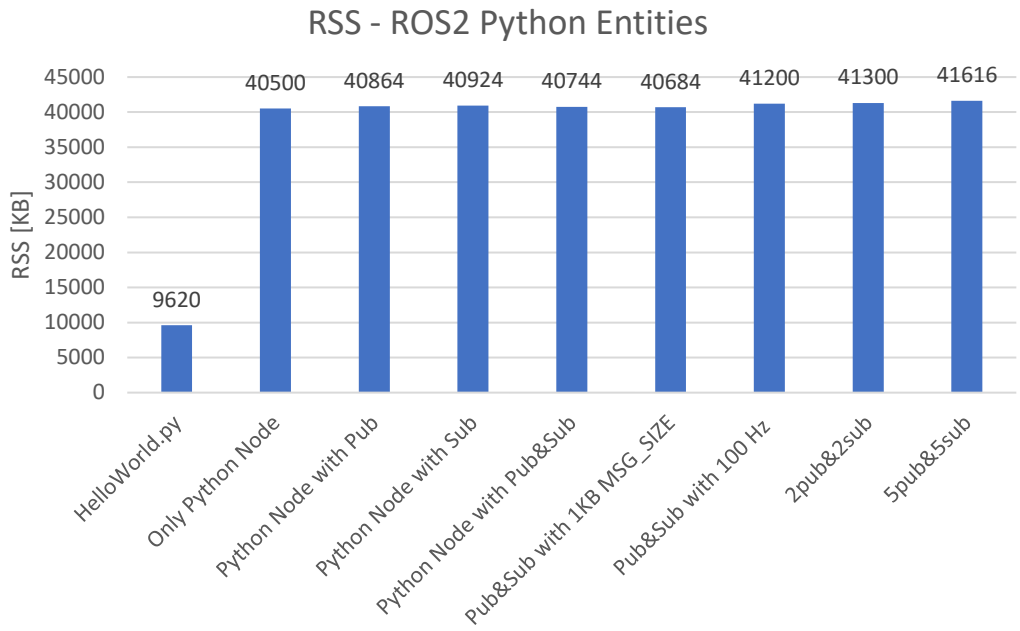

Figure 23: The total RSS memory of ROS2 Python programs

Publisher and subscriber were added to the program using a standard message which was *string* type and publisher rate of 1 Hz. The total RSS memory of ROS2 with publisher and ROS2 with subscriber shown in Figure 23 was 40864 Kilobytes ≈ 40,864 Megabytes and 40924 Kilobytes ≈ 40,924 Megabytes respectively. Besides, 8 and 11 shared libraries were added to shared libraries when running publisher and subscriber in turn. The details of shared libraries were shown in Table 30 and Table 31. It can be easily seen that although shared libraries were added when including publisher and subscriber, the total RSS was only changing slightly.

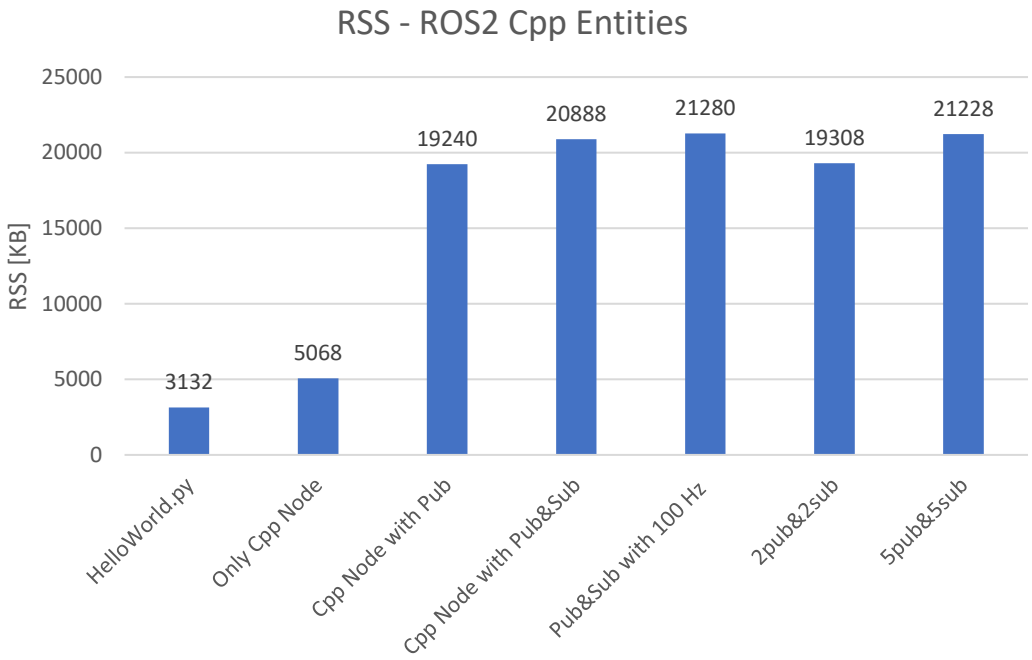

Figure 24: The total RSS memory of ROS2 C++ programs

As mentioned from the beginning of the memory consumption test, ROS2 C++ was used to compare the results of ROS2 Python program. As the bar chart in Figure 24 demonstrates, in general, the total RSS memory of ROS2 C++ application was consumed by far less than ROS2 Python. Besides, ROS2 C++ application only used 2 Kilobytes when ROS2 C++ node was added to the HelloWorld program. Moreover, when adding the publisher into the program, the total RSS memory also increased to around 19240 Kilobytes ≈ 19,240 Megabytes compared with 5068 Kilobytes ≈ 5,068 Megabytes of the program including only ROS2 C++ node. On the other hand, when the program was added more Publisher, Subscriber and publisher rates changed. The total RSS memory did not change significantly, which is the same behaviour as ROS2 Python program.

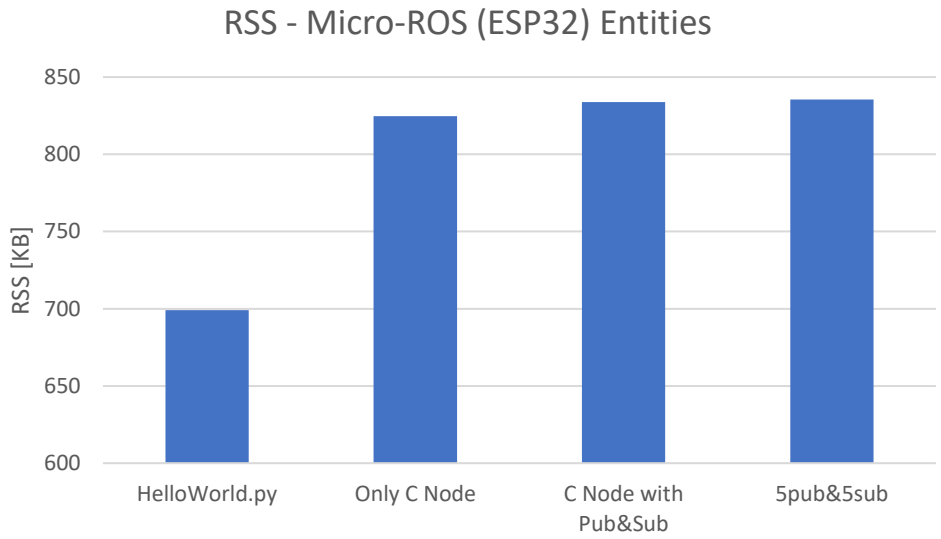

Figure 25: The size of the Micro-ROS library on ESP32

Figure 25 illustrates how Micro-ROS application size was put into ESP32. These values were taken from the terminal when uploading the program into ESP32. The first bar in this graph is the HelloWorld program which consumed 698,976 Kilobytes and the second chart is the HelloWorld program which was added Micro-ROS. It can be seen that Micro-ROS consumed around 130 Kilobytes and it remained roughly the same when adding subscribers and publishers.

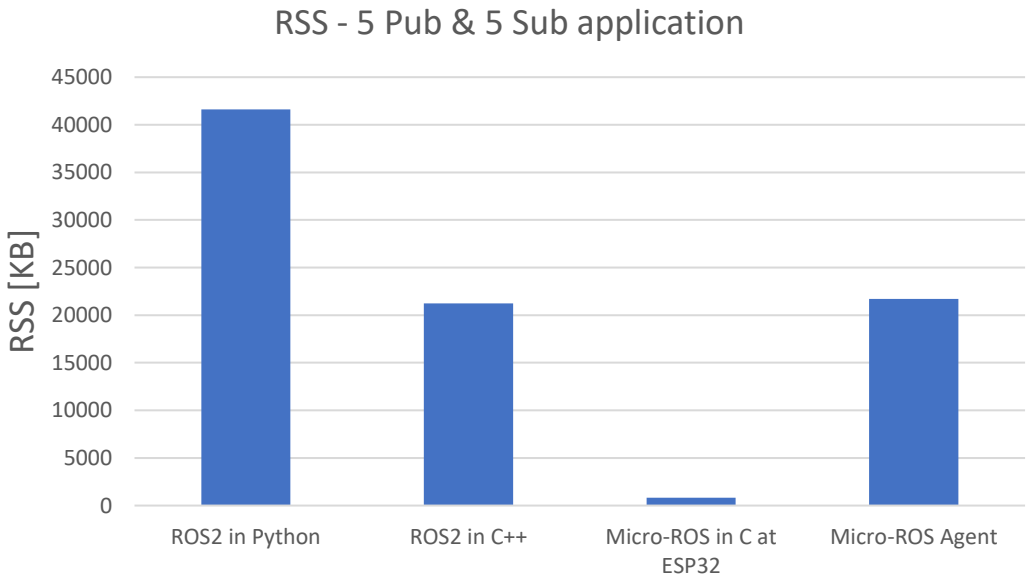

Figure 26: Memory consumption of different ROS2 languages of ROS2 application which run 5 publishers and 5 subscribers in 5 different topics

Figure 26 shows the memory consumption results of ROS2 and Micro-ROS applications that run 5 publishers and 5 subscribers on 5 different topics. To

compare the shared library of Micro-ROS with that of ROS2 in Python and C ++, Micro-ROS Agent had to be included in the graph. It can be seen that running ROS2 using Python consumed much higher memory compared with ROS2 in C++ and Micro-ROS. Besides, running ROS using C++ and running Micro-ROS consumed approximately an equal portion of memory.

5.2.4 EVALUATION AND CONCLUSION

It can be concluded from the test results that is concluded that basic ROS2 using Python shared libraries consume approximately 30 Megabytes because when ROS2 node was created, the total shared library rose from 12 to 70 libraries.

Besides, when changing parameters inside the ROS2 application including the number of nodes, publisher frequency and message sizes, memory consumption was not impacted significantly. This is because the necessary shared libraries were already set and changing parameters only cost a few memories, so it cannot have a large impact on the total memory consumption.

Moreover, it is noticeable that ROS2 Python program consumed more memory compared with ROS2 C++ program, which has rather the same memory usage as Micro-ROS agent. This finding should be taken into account for future projects.

5.3 Reliability Test

5.3.1 DESCRIPTION

The reliability test was conducted to evaluate how reliable the communication between publisher and subscriber in ROS2 and Micro-ROS applications was. The message loss rate is one of the most important factors to evaluate reliability. The rate of message loss is defined as the formula below:

$$MSG\ LOSS = \frac{Total\ MSG\ expected\ to\ receive - Total\ MSG\ actually\ received}{Total\ MSG\ expected\ to\ receive} \times 100\ (\%)$$

To be more specific, each publisher and subscriber message has added a counter. Based on this counter, the program can determine how many messages the subscriber node received in reality compared with publisher messages.

5.3.2 TEST SETUP

This test only ran minimal publisher and subscriber programs based on different QoS policies setup depicted in Table 5. All QoS policies were applied the same in both Publisher and Subscriber to avoid compatibility issues between the two participants. The policies in this table affected the reliability of the delivery of messages sent from Publisher to Subscriber nodes, especially with lost messages. This test focused on the reliability of ROS2 communication, it did not have to require fast publisher rates. Therefore, 1 Kilobyte, 10 Kilobytes, 100 Kilobytes, 1 Megabyte and 2 Megabytes were used for ROS2 application test and 128 Bytes and 1 Kilobyte were used for Micro-ROS application test.

Table 5: QoS profiles summary for the publisher and subscriber

Case	History	Depth	Reliability	Durability
a	KEEP_LAST	10	BEST_EFFORT	VOLATILE
b	KEEP_ALL	N/A	BEST_EFFORT	TRANSIENT_LOCAL
c	KEEP_LAST	10	RELIABLE	VOLATILE
d	KEEP_ALL	N/A	RELIABLE	TRANSIENT_LOCAL

Case a and case b both used BEST_EFFORT reliability but with different history and durability settings. These were designed to determine the impact of the history QoS policy on the ROS2 performance. For case a, history was set to KEEP_LAST with depth set to 10, which means the quantity of samples saved in the queue was set to 10 packets. Besides, the durability was set to VOLATILE which means no old data could be sent to a new subscriber participating during the transmission. In case b, the program tried to keep all history in the cache by using KEEP_ALL history policy while the depth policy was used in this as mentioned in Table 1. The TRANSIENT_LOCAL was selected for durability policy, which let subscribers participate in the topic late and receive previously sent samples.

On the other hand, cases c and d used RELIABLE reliability but other setups were similar to cases a and b. In terms of anticipated performance, cases a and b were expected to have some message loss because they would try to send as many

messages as possible without confirmation. In contrast, cases c and d were expected to receive all packets without any message loss. However, in case c, it might have lost messages if the history depth was not large enough to allow for re-transmissions. Therefore, case c had an extra test whose depth was set to 1000 in the long-term test to compare with the default setup.

5.3.3 TEST EXECUTION AND RESULTS

This test was separated into two sub-tests which included short-term tests in section 5.3.3.1 and long-term tests in section 5.3.3.2. In short-term tests, all cases were run in a few minutes depending on the maximum publisher rate of each message size. In the long-term test, the short-term tests which did not have messages lost were conducted for a longer time, such as one or four hours, even overnight.

5.3.3.1 Reliability – Short-term Test

Results from Table 6, Table 7, Table 8 and Table 9 are the results of case a, case b, case c and case d respectively, which were tested between two ROS2 nodes in two Jetson Nano. And Table 10, Table 11, Table 12 and Table 13 show the results for all cases in Micro-ROS and Micro-ROS applications. Especially, all RELIABLE cases were run various times to guarantee accurate results.

Table 6: Case a of ROS2 application

Reliability = BEST_EFFORT, Durability = VOLATILE, History = KEEP_LAST, Depth = 10

MSG_SIZE	Total packages expect to receive	Total packages actual receive	MSG_LOST rate (%)
1 KB	5986	5986	0
10 KB	6335	6335	0
100 KB	1899	1899	0
1 MB	619	594	4.0387
2 MB	411	379	7.785

Table 7: Case b of ROS2 application

Reliability = BEST_EFFORT, Durability = TRANSIENT_LOCAL, History = KEEP_ALL

MSG_SIZE	Total packages expect to receive	Total packages actual receive	MSG_LOST rate (%)
1 KB	15743	15743	0
10 KB	11896	11896	0
100 KB	8409	8409	0
1 MB	706	671	4.9575
2 MB	783	695	11.239

Table 8: Case c of ROS2 application

Reliability = RELIABLE, Durability = VOLATILE, History = KEEP_LAST, Depth = 10

MSG_SIZE	Total packages expect to receive	Total packages actual receive	MSG_LOST rate (%)
1 KB	10582	10582	0
10 KB	8162	8162	0
100 KB	1832	1832	0
1 MB	799	799	0
2 MB	921	921	0

Table 9: Case d of ROS2 application

Reliability = RELIABLE, Durability = VOLATILE, History = KEEP_ALL

MSG_SIZE	Total packages expect to receive	Total packages actual receive	MSG_LOST rate (%)
1 KB	19474	19474	0
10 KB	10761	10761	0
100 KB	7343	7343	0
1 MB	1501	1501	0
2 MB	766	766	0

Table 10: Case a of Micro-ROS application

Reliability = BEST_EFFORT, Durability = VOLATILE, History = KEEP_LAST, Depth = 10

MSG_SIZE	Total packages expect to receive	Total packages actual receive	MSG_LOST rate (%)
128 B	2572	2563	0.34992
1 KB	2372	2356	0.6745

Table 11: Case b of Micro-ROS application

Reliability = BEST_EFFORT, Durability = TRANSIENT_LOCAL, History = KEEP_ALL

MSG_SIZE	Total packages expect to receive	Total packages actual receive	MSG_LOST rate (%)
128 B	2729	2681	1.7589
1 KB	2886	2819	2.3216

Table 12: Case c of Micro-ROS application

Reliability = RELIABLE, Durability = VOLATILE, History = KEEP_LAST, Depth = 10

MSG_SIZE	Total packages expect to receive	Total packages actual receive	MSG_LOST rate (%)
128 B	2256	2256	0
1 KB	2071	2071	0

Table 13: Case d of Micro-ROS application

Reliability = RELIABLE, Durability = TRANSIENT_LOCAL, History = KEEP_ALL

MSG_SIZE	Total packages expect to receive	Total packages actual receive	MSG_LOST rate (%)
128 B	2366	2366	0
1 KB	2656	2656	0

It can be seen that in the RELIABLE cases which are case c and case d, all message sizes did not have any packet loss (Table 8, Table 9, Table 12 and Table 13).

On the other hand, regarding the BEST_EFFORT cases, in all the Micro-ROS experiment (Table 10 and Table 11) and with 1 Megabyte and 2 Megabytes of ROS2 test (Table 6 and Table 7), some messages loss appeared. This is because the publisher did not receive confirmation from the subscriber and this works as expected behaviour of QoS policies.

5.3.3.2 Reliability – Long-term Test

All short-term experiments were run within the duration of a few minutes. Although this time can be enough to measure the performance and identify certain issues, it is usually not sufficient to point out the worst problems for a given configuration. Therefore, long-term tests were carried out to confirm the reliability of the RELIABLE test in short-term tests. This test was divided into two sub-tests, test A investigated the reliability of ROS2 application using RELIABLE in long-term testing and test B looked at RELIABLE QoS Policy in Micro-ROS application.

A ROS2 APPLICATIONS

Only running ROS2 application in an ideal environment might be not realistic enough for long-term testing, so stress tool and iPerf were re-used in this test. Case c and case d of ROS2 application were repeated under heavy load and 4 Mbps network traffic for eight hours.

It can be seen that ROS2 application still kept zero message loss in case d which is shown in Table 14. However, after four hours, some messages lost appeared in case c which is shown in Table 15, this is because the history depth was not large enough to allow for re-transmissions. Therefore, case c was repeated again but with 1000 history depth. Table 16 shows the results of case c with 1000 history depth, it can be seen that no message loss appeared anymore.

Table 14: Case d of ROS2 application testing using stress tool and 4 Mbps iPerf during 8 hours

MSG_SIZE	QoS Reliability	QoS Durability	QoS Depth	QoS History	Total packages expect received	Total packages actual received	MSG_LOST (%)
100KB	RELIABLE	TRANSIENT_LOCAL	N/A	KEEP_ALL	309393	309393	0

Table 15: Case c with 10 history depth of ROS2 application using stress tool and 4 Mbps iPerf testing during 4 hours

MSG_SIZE	QoS Reliability	QoS Durability	QoS Depth	QoS History	Total packages expect received	Total packages actual received	MSG_LOST (%)
100KB	RELIABLE	VOLATILE	10	KEEP_LAST	9721	9709	0.1234

Table 16: Case c with 1000 history depth of ROS2 application using stress tool and 4 Mbps iPerf testing during 8 hours

MSG_SIZE	QoS Reliability	QoS Durability	QoS Depth	QoS History	Total packages expect received	Total packages actual received	MSG_LOST (%)
100KB	RELIABLE	VOLATILE	1000	KEEP_LAST	19709	19709	0

It is concluded from the results that during long-term tests, in KEEP_ALL History QoS policy, zero message loss was recorded for ROS2 application. On the other hand, when KEEP_LAST History QoS policy was applied, a few messages were lost if the history depth was not large enough. These behaviours were in line with the initial expectations for the QoS setting.

B Micro-ROS APPLICATIONS

The long-term testing of ROS2 application was repeated for Micro-ROS application. However, after one-hour testing, RELIABLE cases of Micro-ROS experiment which are shown in Table 18 and Table 19 had messages lost. However, message loss appeared at a reasonably low rate and depending on how critical the application was, it could be regarded as acceptable.

Table 17: Case d of Micro-ROS application testing during 1 hour

MSG_SIZE	QoS Reliability	QoS Durability	QoS Depth	QoS History	Total packages expect received	Total packages actual received	MSG_LOST (%)
100KB	RELIABLE	TRANSIENT_LOCAL	N/A	KEEP_ALL	41294	41294	0

Table 18: Case c of Micro-ROS with 10 history depth application testing during 1 hour

MSG_SIZE	QoS Reliability	QoS Durability	QoS Depth	QoS History	Total packages expect received	Total packages actual received	MSG_LOST (%)
100KB	RELIABLE	VOLATILE	10	KEEP_LAST	40347	40186	0.3990

Table 19: Case c of Micro-ROS with 1000 history depth application testing during 1 hour

MSG_SIZE	QoS Reliability	QoS Durability	QoS Depth	QoS History	Total packages	Total packages	MSG_LOST (%)

					expect received	actual received	
100KB	RELIABLE	VOLATILE	1000	KEEP_LAST	32131	31827	0.4961

Based on the results, during long-term tests, Micro-ROS application did not meet the pre-test expectations of the QoS setting with all RELIABLE cases having message loss. However, when changing the publisher rate to 1 Hz, the system did not record any lost messages anymore. The reason for this might be the collapse of the internal middleware buffers in Micro-ROS which is taken from the advice of Micro-ROS developers [28].

5.3.4 EVALUATION AND CONCLUSION

The results show that almost all experiments of reliable cases worked as initially expected.

In all RELIABLE of reliability QoS and TRANSIENT_LOCAL of durability QoS with KEEP_ALL of history QoS cases, both ROS2 and Micro-ROS communication recorded zero message loss. On the other hand, in the cases of VOLATILE durability with KEEP_LAST of history QoS policy, history depth had to be considered, this value was set depending on the size and speed of data transmission. With reading and transferring small data such as reading temperature sensors or control motors, history depth can be small, for example, 10 in this test. However, with projects that need to transmit large and fast data such as reading laser sensors, history depth should be large enough to avoid message loss. Besides, with Micro-ROS communication, the speed of data transmission has to be considered because if the buffer is not set properly, it cannot store new data which will lead to message loss. The solution to this issue is to change the “RMW_UXRCE_STREAM_HISTORY” value when increasing the publisher rate.

In the BEST_EFFORT of reliability QoS policy cases, almost all cases had lost messages. This is due to no confirmation of the publisher. However, it can be seen that in some ROS2 with small message size cases such as 1, 10 and 100 Kilobytes, no message loss appeared. This can be explained by the close and speedy network during the time of implementing these experiments, which enabled the messages to be transferred quickly and in full.

5.4 Scalability

The main goal of scalability is to make ROS2 nodes themselves scalable. In the experimental setup in Figure 6, four or even more nodes were running simultaneously. Therefore, to avoid repeating running each node, ROS2 provided launch scripts which started multiple nodes at once. The launch file was written in Python and had to be placed in the same environment of ROS2 application. It had to include at least these parameters as below:

- "*package*": This defines which ROS2 packages need to be launched
- "executable": This defines which programs need to be launched
- "name": This is the name of ROS2 node of ROS2 applications that need to be launched
- "*parameters*" (optional): This will define a configuration value of ROS2 nodes

It is also possible to add parameters to the node, parameters in ROS2 are associated with individual nodes. This allows configuring nodes during startup or changing behaviour during run time, without changing the code. Nodes can also be re-used. For example, when a generic sensor is implemented, the source of the sensor data can be configured by using the parameter configuration. The parameters are defined in the launch file for each ROS2 node. In this project, parameter values were applied for publisher frequency, message sizes, and some QoS policies to make testing easier and faster. For example, during run time, publisher frequency could be changed by "*ros2 param set /publisher_node <publisher_frequency_name> <publisher_frequency_value>*" command. However, if the number of nodes of ROS2 application increased significantly, for example to different nodes in this project, managing each parameter value in the same launch file would cost many lines and make the program not readable. Therefore, defining parameter values in one *YAML* file is a potential solution for the scalability of ROS2. The program only needs to identify "*package*" names and "*executable*" names which are defined in the launch file, then fill in parameter values.

On the other hand, a favourable feature of ROS2 in scalability is that if the program has the same *topics* name, the "ROS_DOMAIN_ID" command can be used. This

command allows the same program to run in the same network but they are not in conflict with each other. By default, ROS_DOMAIN_ID was set to 0.

Moreover, by using Docker, developers can easily install previous work and scale up the setup.

5.5 ROS2 Security

In real-world applications, security features play an important role in protecting the data of any project. Especially at Benchmark, security features are given the highest priority in researching and implementing new projects or applications. Insufficient security implementation might not only lead to security failures in the system but also a dramatic reduction in its performance.

According to [18], DDS-security features are made available for use with ROS2 through a package name Secure ROS2 (SROS2). Through SROS2, ROS2 as the application layer checks for security settings in the application layer and executes the appropriate security plugins in the middleware layer. DDS-security specification is developed on the original DDS specification and adding improvement with Service Plugin Interface (SPI) architecture [19]. SPIs implement the security model as defined or required by the user. Five SPIs are determined for DDS-Security including authentication, access control, cryptographic, logging and data tagging but currently, ROS2's security features only employ the first three.

- **Authentication**: Verify the identity of the Publisher/ Subscriber nodes.
- **Access Control**: Assign which topics the authenticated nodes can publish or subscribe.
- **Cryptography**: Handle all required encryption, signing, and hashing operations.

Security in ROS2 can be turned on and off by configuring the ROS2 environment variables. Three environment variables allow the middleware to locate encryption materials and enable (and possibly enforce) security.

- export ROS_SECURITY_KEYSTORE=~/sros2_demo/demo_keystore: ROS_SECURITY_KEYSTORE states the location where all security policies and keys are stored.
- export ROS_SECURITY_ENABLE=true: This variable switches security on or off
- export ROS_SECURITY_STRATEGY=Enforce: It is set to enforce, and the node checks to ensure that the security keys are present

If it is not set as enforced, it allows the node to fall back to no security when the security keys are absent.

Due to the time limit, this project did not carry out testing for security features of ROS2 and Micro-ROS. However, previous research has been done to evaluate the performance of ROS2 with security added. Therefore, this document will briefly discuss how to set up security features in ROS2, the results and the conclusion based on related literature.

[25] investigated ROS2 security model by using the DDS Security extension. The results found potential security concerns which are not reduced by the DDS Security standard. Besides, turning on security features causes the latency and speed to increase and throughput to decrease compared with when security features are turned off. The research points out that enabling security results in an overhead of 137% in latency performance. [25] recommended that for future research, a vulnerability analysis is needed to identify risks and figured out the solutions to mitigate these security risks.

This finding was also agreed upon by [24] who created a simulation framework which allows multiple ROS2 nodes to be evaluated without the need for multiple hardware to host the ROS2 nodes. The security setting was turned on and off in evaluating the network performance with ROS2 nodes. Based on the result, if security features are turned on, there will be a remarkable overhead in latency, whereas the message drop rate is much higher.

CHAPTER 6 CONCLUSIONS AND RECOMMENDATIONS

In this chapter, the research question which is "What is the usability of ROS2/ Micro-ROS for products developed by Benchmark?" is answered based on the conclusion in section 6.1 and recommendations for improvements and future developments in section 6.2.

6.1 Conclusions

This project evaluated ROS2 and Micro-ROS applications based on 5 parameters including latency, memory consumption, reliability, scalability and security. The objectives stated in the introduction are met as there are certain findings for the usability of ROS2 and Micro-ROS, which might lead to potential applications of ROS2 and Micro-ROS to Benchmark products.

With the support of the test results and their discussions in sections from 5.1 to 5.5, the research questions of this graduation assignment will be answered in this chapter.

Starting with the first sub-question which is "What are the advantages of using ROS2/ Micro -ROS?", the advantages of using ROS2/Micro-ROS are listed below:

1. Owing to ROS2 parameters, launch program and Micro-ROS Agent, ROS2 and Micro-ROS have high scalability. The number of nodes, publishers and subscribers can be easily increased without consuming too much memory of the hardware by using topics. This means for real-world applications, multiple programs from different devices (from computers, embedded devices, and microcontrollers to microprocessors) can be connected in the same network of ROS2. Besides, ROS2 application can be re-used easily on any computers that are installed with Docker.
2. ROS2 works well under heavy network loads comparable to a 1080p camera stream.
3. ROS2 and Micro-ROS take advantage of the QoS of DDS vendors for communication performance and security. This project has shown that ROS2 and Micro-ROS can apply different QoS policies to meet different project

requirements, for example, RELIABLE QoS Reliability policy for projects requiring reliable results or BEST_EFFORT QoS Reliability policy for experiments prioritizing fastest responses.

With regards downsides which are mentioned in the second sub-question “What are the disadvantages of using ROS2/ Micro-ROS?",

1. Publisher timer and subscription callback are not scheduled in precedence relations, which means the system cannot recognize which callback will take place first. This was also pointed out as a drawback by [29] no mechanism is available to enforce precedence relations, synchronization devices, and message passing overhead.
2. Micro-ROS should only be used with a small message size and for low reliable data transfer projects because when running for the long-term experiments, message loss appears even using RELIABLE QoS reliability policy.

Regarding the sub-questions “When to use ROS2/ Micro-ROS?” and “When not to use ROS2/ Micro-ROS?”,

1. For projects that require strong and reliable data transmission and no data loss, ROS2 is more suitable than Micro-ROS.
2. Micro-ROS should be used in projects that require sending and receiving small message sizes such as a few bytes to 1 Kilobyte.
3. With complex and mixed scheduling semantics projects, ROS2 should be restricted because these features are complex and still under development. Micro-ROS can be a good option in this case because they are designed to solve these problems of ROS2.

“How can ROS2/ Micro-ROS be applied to Benchmark products?” to answer this question, it is obvious that depending on the requirement of projects from Benchmark, ROS2 or Micro-ROS are considered to use. To be specific, to enable ROS2 and Micro-ROS in the same project, they have to be set in the same network, so that they can communicate with each other and control or receive from other parties. A demo will also be shown later to other engineers in Benchmark so that other engineers will have more overview of ROS2 and Micro-ROS then the company can know more about how to apply ROS2 and Micro-ROS to company products.

To end this section, the research question is answered. Based on the inputs from the aforementioned sub-questions, it can be clear that ROS2 and Mico-ROS can be used for product development at Benchmark, however careful calculations need to be taken into account when it comes to hard real-time applications.

The next section will cover the recommendations that can be done to improve this project in the future.

6.2 Recommendations

As mentioned in section 6.1, the project has shown that ROS2 and Micro-ROS have advantages that might bring benefits to certain projects. However, they still have disadvantages that need to improve. For experiments of latency testing in section 5.1, although result values were recorded, in some cases the latency could not be measured correctly due to the priority of publisher timer and subscription callbacks.

On the other hand, in the memory consumption experiment in section 5.2, the results have shown that when adding more than 5 publisher and subscriber nodes, the total RSS memory was only slightly changing. However, if the project needs to run too many nodes such as above 100 nodes, the total RSS memory might be impacted harder.

Regarding the reliability experiments, although different cases were tested both for short-term and long-term testing, eight hours is still not enough to guarantee that the ROS2 and Micro-ROS applications can work for a week, a month or even several months without any problems.

Although ROS2 security experimental was not conducted in this project, based on the literature, it can be seen that when turning ROS2 security features on, the latency and speed are affected.

With shortcomings provided, certain recommendations can be made.

1. Add the *wait_set* mechanism for publisher timer and subscription callbacks: With the waiting functionality for data, the system will know when the subscription callback receives a message then the publisher timer will start sending a new message.

2. Investigate memory consumption when increasing the number of nodes such as 100 nodes including publishers and subscribers: right now only 5 publishers and 5 subscribers in 5 different topics are generated and the total RSS is not changing too much, by doing this test, memory consumption of ROS2 and Micro-ROS programs can be explored more.
3. Investigate more different QoS settings for DDS: this project only tested with Reliability, Durability, History and Depth QoS Policies. For future research, Deadline, Lifespan, Liveliness and Lease Duration QoS Policies can be explored to investigate the impact of those QoS on ROS2 and Micro-ROS communication.
4. Conduct more long-term tests for reliability experiments over a longer period: by doing reliability tests in a longer time, some more errors might be figured out and this will be helpful when applying ROS2 and Micro-ROS to a real project.
5. Investigate the impact of ROS2 security on ROS2 and Micro-ROS: as briefly mentioned about ROS2 security, it has effects on latency and speed, so in the later research in Benchmark, ROS2 security needs to be considered before designing ROS2 and Micro-ROS applications.

BIBLIOGRAPHY

[1] A. Cahalan, *pmap(1) – Linux man page*, die.net, 2002. Accessed on: March 29, 2022. [Online]. Available: https://linux.die.net/man/1/pmap

[2] B. Gerkey, *Why ROS2?,* ROS2 Design, June 2014. Accessed on: March 29, 2022. [Online]. Available: http://design.ros2.org/articles/why_ros2.html

[3] C. Gutiérrez, L. Juan, I. Ugarte and V. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ROS 2.0 communications for real-time robotic applications" *Erle Robotics S.L.,* Sep. 7, 2019. [Online]. Available: arXiv:1809.02595v1 [cs.RO]

[4] P. Vouzis, *Linux for Network Engineers: How to Use iPref*, NETBEEZ, July 24, 2019. Accessed on: March 29, 2022. [Online]. Available: https://netbeez.net/blog/how-to-use-iperf/

[5] R. Lange and M.Merlan, *New client library and middleware features*, Micro-ROS, Oct. 2021. Accessed on: March 29, 2022. [Online]. Available: https://www.lange-ralph.de/talks/2021-10-20_micro-ROS_New_client_library_and_middleware_features.pdf

[6] R. Lange, "Micro-ROS – bringing the most popular robotics middleware onto tiny microcontrollers" *Bosch Research Blog*, Jan. 14, 2021. Accessed on: March 29, 2022. [Online]. Available: https://www.bosch.com/stories/bringing-robotics-middleware-onto-tiny-microcontrollers/

[7] *What can DDS do for You?*. Twin Oaks Computing, Inc. USA. Dec. 2011. Accessed on: March 29, 2022. [Online]. Available: https://www.omg.org/hot-topics/documents/dds/CoreDX_DDS_Why_Use_DDS.pdf

[8] Y. Liu, Y. Guan, X. Li, R. W and J. Zhang, "Formal Analysis and Verification of DDS in ROS2" *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, 2018, pp. 1-5, doi: 10.1109/MEMCOD.2018.8556970.

[9] Y. Maruyama, S. Kato and T. Azumi, "Exploring the performance of ROS2," *2016 International Conference on Embedded Software (EMSOFT)*, 2016, pp. 1-10, doi: 10.1145/2968478.2968502.

[10] Y. Yang and T. Azumi, "Exploring Real-Time Executor on ROS 2", *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2020, pp. 1-8, doi: 10.1109/ICESS49830.2020.9301530.

[11] E. Fernandez, *ROS2 Quality of Service policies,* ROS2 Design, Oct. 2015. Accessed on: March 29, 2022. [Online]. Available: https://design.ros2.org/articles/qos.html

[12] F. Finocchiaro, *micro-ROS: A member of the Zephyr Project and integrated into the Zephyr build system as a module!* Zephyr Project, Oct. 22, 2020. Accessed on: March 29, 2022. [Online]. Available: https://zephyrproject.org/micro-ros-a-member-of-the-zephyr-project-and-integrated-into-the-zephyr-build-system-as-a-module/#:~:text=micro%2DROS%20is%20a%20software,%2C%20PIAP%2C%20and%20Acutronic%20Robotics.

[13] Micro-ROS, “Mission”. Accessed on: March 30, 2022. [Online]. Available: https://micro.ros.org/

[14] Github, “Understanding ROS2 topics”. Accessed on: March 30, 2022. [Online]. Available: https://github.com/ros2/ros2_documentation/blob/foxy/source/Tutorials/Topics/Understanding-ROS2-Topics.rst

[15] Github, “Understanding ROS2 services”. Accessed on: March 30, 2022. [Online]. Available: https://github.com/ros2/ros2_documentation/blob/foxy/source/Tutorials/Services/Understanding-ROS2-Services.rst

[16] Github, “Topics vs Services vs Actions”. Accessed on: March 30, 2022. [Online]. Available: https://github.com/ros2/ros2_documentation/blob/foxy/source/How-To-Guides/Topics-Services-Actions.rst

[17] Github, “About Quality of Service settings”. Accessed on: March 30, 2022. [Online]. Available: https://github.com/ros2/ros2_documentation/blob/rolling/source/Concepts/About-Quality-of-Service-Settings.rst

[18] Github, “Introducing ROS2 Security”. Accessed on: Apr. 1, 2022. [Online]. Available: https://github.com/ros2/ros2_documentation/blob/rolling/source/Tutorials/Security/Introducing-ros2-security.rst

[19] K. Fazzari, *ROS2 DDS-Security integration,* ROS2 Design, July 2019. Accessed on: Apr. 1, 2022. [Online]. Available: https://design.ros2.org/articles/ros2_dds_security.html

[20] The Robotics Back-End, “Build a ROS2 Data Pipeline With ROS2 Topics”. Accessed on: May 26, 2022. [Online]. Available: https://roboticsbackend.com/build-a-ros2-data-pipeline-with-ros2-topics/

[21] Micro-ROS, “ROS2 Feature Comparison”. Accessed on: May 29, 2022. [Online]. Available: https://micro.ros.org/docs/overview/ROS_2_feature_comparison/

[22] Sparkfun, “GPS Basics”. Accessed on: June 3th, 2022. [Online]. Available: https://learn.sparkfun.com/tutorials/gps-basics/all#:~:text=Update%20Rate%20%2D%20The%20update%20rate,available%20in%20low%20cost%20modules.

[23] Apple Developer, “Getting Raw Accelerometer Events”. Accessed on: June 3th, 2022. [Online]. Available: https://developer.apple.com/documentation/coremotion/getting_raw_accelerometer_events#overview

[24] Z. Chen, “Performance analysis of ROS 2 networks using variable quality of service and security constraints for autonomous systems” *Calhoun: The NPS Institutional Archive DSpace Repository,* Sep 2019.

[25] V. DiLuoffo, W.R. Michalson and B. Sunar, “Robot Operating System 2: The need for a holistic security approach to robotic architectures”, *International journal of advanced robotics systems,* May-June 2018, 15, doi:10.1177/1729881418770011

[26] T. Kronauer *et al.,* “Latency Analysis of ROS2 Multi-Node Systems” *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)* (2021): 1-7.

[27] G. Fairhurst, “The User Datagram Protocol (UDP)” Nov 2008. Accessed on: June 3th, 2022. [Online]. Available: https://erg.abdn.ac.uk/users/gorry/course/inet-pages/udp.html#:~:text=A%20UDP%20datagram%20is%20carried,packets%20usually%20requires%20IP%20fragmentation.

[28] Github, “Message loss still appears when using RELIABLE QoS #146”. Accessed on: June 13, 2022. [Online]. Available: https://github.com/micro-ROS/micro-ROS-Agent/issues/146

[29] Micro-ROS, “Execution Management”. Accessed on: June 14, 2022. [Online]. Available: https://micro.ros.org/docs/concepts/client_library/execution_management/

APPENDIX A: TABLES DETAIL OF LATENCY MEASUREMENTS

Table 20: Results of the influence of publisher frequency on latency between two Jetson Nano running ROS2 application test in ms

Hz	128B	1KB	10KB	100KB
1	10,5905	10,1357	13,1201	20,7542
10	6,0074	6,2581	6,7032	10,2561
20	4,8425	5,2624	5,6114	NULL
30	4,8196	5,2085	4,6836	NULL
40	4,5195	5,2036	4,6906	NULL
50	4,8015	5,0081	4,7156	NULL
60	4,7281	5,3447	4,9529	NULL
70	4,3818	5,3491	4,3505	NULL
80	4,6847	5,0084	4,6683	NULL
90	4,6241	5,0561	NULL	NULL
100	4,5681	4,5762	NULL	NULL

Table 21: Results of the influence of publisher frequency on latency between two ESP32 running Micro-ROS application test in ms

Hz	128B	1KB
1	98,9291	93,8286
10	54,7569	58,2077
20	8,9453	11,1048
30	7,1450	6,8956
40	7,3405	7,6009
50	7,0414	6,7217
60	6,1199	12,9322
70	4,7858	NULL
80	2,5704	NULL
90	2,7141	NULL
100	2,8689	NULL

Table 22: Results of the influence of publisher frequency on latency between Jetson Nano running ROS2 application and ESP32 running Micro-ROS application test in ms

Hz	128B	1KB
1	57,4462	57,7712
10	39,6250	39,3910
20	14,7161	13,0650
30	10,0853	10,4805
40	9,0237	9,8351
50	8,7823	10,3747
60	8,5088	8,4702

70	6,9295	6,2244
80	6,4928	5,1717
90	5,8855	4,1514
100	4,8725	5,4359

Table 23: Results of the impact of network load using Iperf transmit 4 Mbps on the latency of ROS2 applications test in ms

Hz	128B with iPerf	1KB with iPerf	128B – without iPerf	1KB – without iPerf
1	9,5816	10,2352	10,5905	10,1357
10	6,5894	7,1352	6,0074	6,2581
20	6,0788	5,6854	4,8425	5,2624
30	5,2312	5,4652	4,8196	5,2085
40	5,5947	5,3452	4,5195	5,2036
50	5,3161	5,1256	4,8015	5,0081
60	4,6641	5,6854	4,7281	5,3447
70	4,4552	5,6851	4,3818	5,3491
80	4,5843	5,0354	4,6847	5,0084
90	3,9899	5,1465	4,6241	5,0561
100	4,1658	4,4352	4,5681	4,5762

Table 24: Results of the impact of network and system load on latency of ROS2 applications test in ms

Hz	Without iperf	iPerf 4M	iPerf 40M	iPerf 80M
1	6,8687	6,4324	7,5739	7,3395
10	5,4182	5,1537	5,0443	5,5676
20	5,5675	4,9227	5,7280	5,7912
30	5,0915	4,8641	5,2959	5,5709
40	5,1797	5,4861	5,5280	5,3419
50	5,6943	5,4328	4,3249	4,1508
60	4,2499	3,8877		

Table 25: Results of the scalability of ROS2 system running in data pipeline on latency with 128 Bytes payload size test in ms

Hz	2 nodes	3 nodes	5 nodes	9 nodes	15 nodes	19 nodes
1	7,7398	11,5597	27,5688	37,2687	42,9221	47,2509
10	7,0752	10,0321	25,4730	34,3037	37,6045	40,8470
20	5,7759	8,7642	21,1403	26,4785	29,2887	31,8226
50	4,8270	5,9461	12,6487	15,6460	17,3015	17,5365
80	3,4816	3,7147	9,5850	11,0148	1,3065	4,2398
100	2,6407	3,5677	8,1061	8,9956	2,9497	7,7724

Table 26: Results of the scalability of ROS2 system running in data pipeline on latency with 1 Kilobyte payload size test in ms

Hz	2 nodes	3 nodes	5 nodes	9 nodes	15 nodes	19 nodes

1	7,9285	11,2068	26,3556	36,9132	43,1877	46,9424
10	7,1499	9,5765	23,9741	33,9996	37,8942	40,3767
20	6,5229	9,0194	21,6880	25,7798	29,4466	32,0184
50	4,9045	5,5233	13,6306	15,0870	17,4528	18,0728
80	3,7043	4,7908	9,3289	11,1980	1,9272	4,2647
100	2,7181	4,0225	8,1470	9,5343	3,2711	8,0410

Table 27: Results of the scalability of ROS2 system running in parallel on latency with 1 Kilobyte payload size test in ms

Hz	2 topics	4 topics	6 topics	12 topics	18 topics	24 topics
1	7,7398	7,3089	7,6083	7,8033	7,5499	7,2616
10	7,0752	6,3736	6,7574	6,8668	6,1502	6,2231
20	5,7759	5,8157	6,0955	6,0297	5,7047	5,2953
50	4,8270	4,5056	4,7643	4,0016	3,9543	3,5596
80	3,4816	3,2087	2,8790	3,0037	3,3594	3,0642
100	2,6407	2,5111	2,4136	2,6827	2,5177	2,4062

APPENDIX B: TABLES DETAIL OF MEMORY MEASUREMENTS

Table 28: Result of memory consumption of helloworld.py test

STT	Shared Library name	RSS (KB)
	Total RSS	**9620**
1	[anon]	3056
2	gconv-modules.cache	28
3	[stack]	104
4	libexpat.so.1.6.11	92
5	libm-2.31.so	396
6	libc-2.31.so	1556
7	ld-2.31.so	184
8	python3.8	3572
9	locale-archive	436
10	libz.so.1.2.11	80
11	libutil-2.31.so	16
12	libpthread-2.31.so	100

Table 29: Results of memory consumption of minimal ROS2 node test

STT	Shared Library name	RSS (KB)
	Total RSS	**40500**
1	[anon]	19428
2	[stack]	100
3	_bz2.cpython-38-aarch64-linux-gnu.so	24
4	_json.cpython-38-aarch64-linux-gnu.so	68
5	_lzma.cpython-38-aarch64-linux-gnu.so	36
6	_opcode.cpython-38-aarch64-linux-gnu.so	16
7	_queue.cpython-38-aarch64-linux-gnu.so	20
8	_rclpy.cpython-38-aarch64-linux-gnu.so	116
9	_rclpy_action.cpython-38-aarch64-linux-gnu.so	48
10	_rclpy_handle.cpython-38-aarch64-linux-gnu.so	12
11	_rclpy_logging.cpython-38-aarch64-linux-gnu.so	20
12	_rclpy_pycapsule.cpython-38-aarch64-linux-gnu.so	16
13	_rclpy_signal_handler.cpython-38-aarch64-linux-gnu.so	16
14	builtin_interfaces_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	16
15	fastrtps_67e49a61e3949c1a	528
16	fastrtps_port7412	36
17	fastrtps_port7413	36
18	gconv-modules.cache	28
19	ld-2.31.so	144
20	libbuiltin_interfaces__python.so	16
21	libbuiltin_interfaces__rosidl_generator_c.so	16

22	libbuiltin_interfaces__rosidl_typesupport_c.so	12
23	libbuiltin_interfaces__rosidl_typesupport_fastrtps_c.so	16
24	libbz2.so.1.0.4	68
25	libc-2.31.so	1124
26	libcrypto.so.1.1	776
27	libdl-2.31.so	20
28	libexpat.so.1.6.11	96
29	libfastcdr.so.1.0.13	72
30	libfastrtps.so.2.1.1	5948
31	libgcc_s.so.1	84
32	liblzma.so.5.2.4	72
33	libm-2.31.so	256
34	libpthread-2.31.so	96
35	libpython3.8.so.1.0	820
36	librcl.so	224
37	librcl_action.so	68
38	librcl_interfaces__python.so	92
39	librcl_interfaces__rosidl_generator_c.so	64
40	librcl_interfaces__rosidl_typesupport_c.so	32
41	librcl_interfaces__rosidl_typesupport_fastrtps_c.so	72
42	librcl_logging_spdlog.so	44
43	librcl_yaml_param_parser.so	40
44	librclpy_common.so	24
45	librcpputils.so	28
46	librcutils.so	72
47	librmw.so	32
48	librmw_dds_common.so	72
49	librmw_dds_common__rosidl_typesupport_cpp.so	16
50	librmw_dds_common__rosidl_typesupport_fastrtps_cpp.so	24
51	librmw_fastrtps_cpp.so	208
52	librmw_fastrtps_shared_cpp.so	252
53	librmw_implementation.so	44
54	librosidl_runtime_c.so	32
55	librosidl_typesupport_c.so	20
56	librosidl_typesupport_cpp.so	24
57	librosidl_typesupport_fastrtps_c.so	12
58	librosidl_typesupport_fastrtps_cpp.so	16
59	librt-2.31.so	36
60	libspdlog.so.1.5.0	452
61	libssl.so.1.1	220
62	libstdc++.so.6.0.28	1424
63	libtinyxml2.so.6.2.0	72
64	libtracetools.so	12
65	libutil-2.31.so	16
66	libyaml.so	72
67	libz.so.1.2.11	108

68	locale-archive	380
69	python3.8	3800
70	rcl_interfaces_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	48

Table 30: Results of memory consumption of minimal ROS2 node with publisher test

STT	Shared Library name	RSS (KB)
	Total RSS	**40864**
1	[anon]	21304
2	[stack]	100
3	_bz2.cpython-38-aarch64-linux-gnu.so	24
4	_json.cpython-38-aarch64-linux-gnu.so	68
5	_lzma.cpython-38-aarch64-linux-gnu.so	36
6	_opcode.cpython-38-aarch64-linux-gnu.so	16
7	_queue.cpython-38-aarch64-linux-gnu.so	20
8	_rclpy.cpython-38-aarch64-linux-gnu.so	116
9	_rclpy_action.cpython-38-aarch64-linux-gnu.so	48
10	_rclpy_handle.cpython-38-aarch64-linux-gnu.so	12
11	_rclpy_logging.cpython-38-aarch64-linux-gnu.so	20
12	_rclpy_pycapsule.cpython-38-aarch64-linux-gnu.so	16
13	_rclpy_signal_handler.cpython-38-aarch64-linux-gnu.so	16
14	builtin_interfaces_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	16
15	fastrtps_f752d86c86db12e6	528
16	fastrtps_f7f41659b134a3b2	64
17	fastrtps_port7412	36
18	fastrtps_port7413	36
19	fastrtps_port7414	24
20	fastrtps_port7415	36
21	gconv-modules.cache	28
22	ld-2.31.so	144
23	libbuiltin_interfaces__python.so	16
24	libbuiltin_interfaces__rosidl_generator_c.so	16
25	libbuiltin_interfaces__rosidl_typesupport_c.so	12
26	libbuiltin_interfaces__rosidl_typesupport_fastrtps_c.so	16
27	libbz2.so.1.0.4	68
28	libc-2.31.so	1132
29	libcrypto.so.1.1	784
30	libdl-2.31.so	20
31	libexpat.so.1.6.11	140
32	libfastcdr.so.1.0.13	72
33	libfastrtps.so.2.1.1	5908
34	libgcc_s.so.1	84
35	liblzma.so.5.2.4	72

36	libm-2.31.so	316
37	libpthread-2.31.so	104
38	libpython3.8.so.1.0	828
39	librcl.so	224
40	librcl_action.so	68
41	librcl_interfaces__python.so	84
42	librcl_interfaces__rosidl_generator_c.so	64
43	librcl_interfaces__rosidl_typesupport_c.so	28
44	librcl_interfaces__rosidl_typesupport_fastrtps_c.so	72
45	librcl_logging_spdlog.so	44
46	librcl_yaml_param_parser.so	40
47	librclpy_common.so	24
48	librcpputils.so	28
49	librcutils.so	88
50	librmw.so	32
51	librmw_dds_common.so	72
52	librmw_dds_common__rosidl_typesupport_cpp.so	16
53	librmw_dds_common__rosidl_typesupport_fastrtps_cpp.so	24
54	librmw_fastrtps_cpp.so	208
55	librmw_fastrtps_shared_cpp.so	260
56	librmw_implementation.so	44
57	librosidl_runtime_c.so	32
58	librosidl_typesupport_c.so	20
59	librosidl_typesupport_cpp.so	24
60	librosidl_typesupport_fastrtps_c.so	12
61	librosidl_typesupport_fastrtps_cpp.so	16
62	librt-2.31.so	36
63	libspdlog.so.1.5.0	460
64	libssl.so.1.1	228
65	libstd_msgs__python.so	92
66	libstd_msgs__rosidl_generator_c.so	72
67	libstd_msgs__rosidl_typesupport_c.so	32
68	libstd_msgs__rosidl_typesupport_fastrtps_c.so	64
69	libstdc++.so.6.0.28	1496
70	libtinyxml2.so.6.2.0	72
71	libtracetools.so	12
72	libutil-2.31.so	16
73	libyaml.so	72
74	libz.so.1.2.11	108
75	locale-archive	380
76	python3.8	3800
77	rcl_interfaces_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	48
78	std_msgs_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	52

Table 31: Results of memory consumption of minimal ROS2 node with subscriber test

STT	Shared Library name	RSS (KB)
	Total RSS	**40924**
1	[anon]	23464
2	[stack]	100
3	_bz2.cpython-38-aarch64-linux-gnu.so	24
4	_json.cpython-38-aarch64-linux-gnu.so	68
5	_lzma.cpython-38-aarch64-linux-gnu.so	36
6	_opcode.cpython-38-aarch64-linux-gnu.so	16
7	_queue.cpython-38-aarch64-linux-gnu.so	20
8	_rclpy.cpython-38-aarch64-linux-gnu.so	116
9	_rclpy_action.cpython-38-aarch64-linux-gnu.so	48
10	_rclpy_handle.cpython-38-aarch64-linux-gnu.so	12
11	_rclpy_logging.cpython-38-aarch64-linux-gnu.so	20
12	_rclpy_pycapsule.cpython-38-aarch64-linux-gnu.so	16
13	_rclpy_signal_handler.cpython-38-aarch64-linux-gnu.so	16
14	builtin_interfaces_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	16
15	fastrtps_e9045ec445077ac1	64
16	fastrtps_f7f41659b134a3b2	64
17	fastrtps_528be6c3da952741	528
18	fastrtps_port7415	36
19	fastrtps_port7413	36
20	fastrtps_port7412	36
21	fastrtps_port7414	24
22	fastrtps_port7417	36
23	fastrtps_port7416	36
24	gconv-modules.cache	28
25	ld-2.31.so	144
26	libbuiltin_interfaces__python.so	16
27	libbuiltin_interfaces__rosidl_generator_c.so	16
28	libbuiltin_interfaces__rosidl_typesupport_c.so	12
29	libbuiltin_interfaces__rosidl_typesupport_fastrtps_c.so	16
30	libbz2.so.1.0.4	68
31	libc-2.31.so	1324
32	libcrypto.so.1.1	912
33	libdl-2.31.so	20
34	libexpat.so.1.6.11	140
35	libfastcdr.so.1.0.13	72
36	libfastrtps.so.2.1.1	6332
37	libgcc_s.so.1	84
38	liblzma.so.5.2.4	72
39	libm-2.31.so	316
40	libpthread-2.31.so	104
41	libpython3.8.so.1.0	812

42	librcl.so	224
43	librcl_action.so	68
44	librcl_interfaces__python.so	84
45	librcl_interfaces__rosidl_generator_c.so	64
46	librcl_interfaces__rosidl_typesupport_c.so	28
47	librcl_interfaces__rosidl_typesupport_fastrtps_c.so	72
48	librcl_logging_spdlog.so	44
49	librcl_yaml_param_parser.so	40
50	librclpy_common.so	24
51	librcpputils.so	28
52	librcutils.so	88
53	librmw.so	32
54	librmw_dds_common.so	72
55	librmw_dds_common__rosidl_typesupport_cpp.so	16
56	librmw_dds_common__rosidl_typesupport_fastrtps_cpp.so	24
57	librmw_fastrtps_cpp.so	208
58	librmw_fastrtps_shared_cpp.so	260
59	librmw_implementation.so	44
60	librosidl_runtime_c.so	32
61	librosidl_typesupport_c.so	20
62	librosidl_typesupport_cpp.so	24
63	librosidl_typesupport_fastrtps_c.so	12
64	librosidl_typesupport_fastrtps_cpp.so	16
65	librt-2.31.so	36
66	libspdlog.so.1.5.0	444
67	libssl.so.1.1	212
68	libstd_msgs__python.so	92
69	libstd_msgs__rosidl_generator_c.so	72
70	libstd_msgs__rosidl_typesupport_c.so	32
71	libstd_msgs__rosidl_typesupport_fastrtps_c.so	64
72	libstdc++.so.6.0.28	1480
73	libtinyxml2.so.6.2.0	72
74	libtracetools.so	12
75	libutil-2.31.so	16
76	libyaml.so	72
77	libz.so.1.2.11	108
78	locale-archive	380
79	python3.8	3800
80	rcl_interfaces_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	48
81	std_msgs_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	52

Table 32: Results of memory consumption of minimal ROS2 node with 100Hz publisher frequency test

STT	Shared Library name	RSS (KB)
	Total RSS	**41200**
1	[anon]	21416
2	[stack]	100
3	_bz2.cpython-38-aarch64-linux-gnu.so	24
4	_json.cpython-38-aarch64-linux-gnu.so	68
5	_lzma.cpython-38-aarch64-linux-gnu.so	36
6	_opcode.cpython-38-aarch64-linux-gnu.so	16
7	_queue.cpython-38-aarch64-linux-gnu.so	20
8	_rclpy.cpython-38-aarch64-linux-gnu.so	116
9	_rclpy_action.cpython-38-aarch64-linux-gnu.so	48
10	_rclpy_handle.cpython-38-aarch64-linux-gnu.so	12
11	_rclpy_logging.cpython-38-aarch64-linux-gnu.so	20
12	_rclpy_pycapsule.cpython-38-aarch64-linux-gnu.so	16
13	_rclpy_signal_handler.cpython-38-aarch64-linux-gnu.so	16
14	builtin_interfaces_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	16
15	fastrtps_e9045ec445077ac1	64
16	fastrtps_f7f41659b134a3b2	64
17	fastrtps_528be6c3da952741	528
18	fastrtps_port7415	36
19	fastrtps_port7413	36
20	fastrtps_port7412	36
21	fastrtps_port7414	24
22	fastrtps_port7417	36
23	fastrtps_port7416	36
24	gconv-modules.cache	28
25	ld-2.31.so	144
26	libbuiltin_interfaces__python.so	16
27	libbuiltin_interfaces__rosidl_generator_c.so	16
28	libbuiltin_interfaces__rosidl_typesupport_c.so	12
29	libbuiltin_interfaces__rosidl_typesupport_fastrtps_c.so	16
30	libbz2.so.1.0.4	68
31	libc-2.31.so	1168
32	libcrypto.so.1.1	760
33	libdl-2.31.so	20
34	libexpat.so.1.6.11	140
35	libfastcdr.so.1.0.13	72
36	libfastrtps.so.2.1.1	5880
37	libgcc_s.so.1	84
38	liblzma.so.5.2.4	72
39	libm-2.31.so	316
40	libpthread-2.31.so	104
41	libpython3.8.so.1.0	864
42	librcl.so	224

43	librcl_action.so	68
44	librcl_interfaces__python.so	84
45	librcl_interfaces__rosidl_generator_c.so	64
46	librcl_interfaces__rosidl_typesupport_c.so	28
47	librcl_interfaces__rosidl_typesupport_fastrtps_c.so	72
48	librcl_logging_spdlog.so	44
49	librcl_yaml_param_parser.so	40
50	librclpy_common.so	24
51	librcpputils.so	28
52	librcutils.so	88
53	librmw.so	32
54	librmw_dds_common.so	72
55	librmw_dds_common__rosidl_typesupport_cpp.so	16
56	librmw_dds_common__rosidl_typesupport_fastrtps_cpp.so	24
57	librmw_fastrtps_cpp.so	208
58	librmw_fastrtps_shared_cpp.so	260
59	librmw_implementation.so	44
60	librosidl_runtime_c.so	32
61	librosidl_typesupport_c.so	20
62	librosidl_typesupport_cpp.so	24
63	librosidl_typesupport_fastrtps_c.so	12
64	librosidl_typesupport_fastrtps_cpp.so	16
65	librt-2.31.so	36
66	libspdlog.so.1.5.0	444
67	libssl.so.1.1	212
68	libstd_msgs__python.so	92
69	libstd_msgs__rosidl_generator_c.so	72
70	libstd_msgs__rosidl_typesupport_c.so	32
71	libstd_msgs__rosidl_typesupport_fastrtps_c.so	64
72	libstdc++.so.6.0.28	1596
73	libtinyxml2.so.6.2.0	72
74	libtracetools.so	12
75	libutil-2.31.so	16
76	libyaml.so	72
77	libz.so.1.2.11	108
78	locale-archive	380
79	python3.8	3800
80	rcl_interfaces_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	48
81	std_msgs_s__rosidl_typesupport_c.cpython-38-aarch64-linux-gnu.so	52