Comparing ROS and ROS2 For Robot Communication

Hrushikesh Sanap

School of Computing Science

Sir Alwyn Williams Building

University of Glasgow

G12 8RZ

A dissertation presented in part fulfillment of the requirements of the Degree of Master of Science at the University of Glasgow

1st September 2023

Abstract

Robotic Middleware emerged as a solution to systems integration challenges in robotics. It provides a common intermediary layer that connects hardware and software modules, enabling seamless communication and coordination. ROS (Robot Operating System) and ROS2 (Robot Operating System 2) are examples of two industry leading robotic middlewares. ROS is the most widely adopted open-source Robotic Middleware which is about to reach its end-of-life. Its successor ROS2 modernizes ROS capabilities required for emerging robotics like security, real-time control, and multi-robot coordination. The objective of this research is to conduct benchmark tests on the ROS and ROS 2 frameworks in order to compare their performance across various parameters. This study systematically benchmarks the intra-board and inter-board communication performance between ROS and ROS2 using various metrics like latency, throughput and bandwidth. The results revealed that for intra-board communication, ROS2 consistently outperformed ROS in terms of latency. However, ROS maintained 100% throughput up to 30Hz whereas ROS2 plateaued at slightly lower levels. Beyond 30Hz, ROS experienced a throughput drop in range 10-30% compared to only 5-10% for ROS2. A similar trend was seen for bandwidth, with ROS performing better at lower frequencies but dropping off rapidly beyond 30Hz. For inter-board communication, ROS2 again showed lower latency compared to ROS. Throughput followed a similar pattern with ROS leading up to 20Hz and then deteriorating rapidly, while ROS2 declined gradually beyond 20Hz. In conclusion, for both intra-board and inter-board communication, ROS2 demonstrated lower latency attributed to efficient DDS middleware. However, ROS achieved higher throughput and bandwidth at lower frequencies. The abrupt performance drop of ROS beyond frequency thresholds suggests scalability limitations compared to the graceful degradation of ROS2. Considering the superior latency and gradual decline in throughput and bandwidth, ROS2 emerges as the preferred platform for high frequency robotic applications requiring intra-board or inter-board communication. It is safe to say that ROS2 delivers substantial performance improvements, especially in real-time communication critical for robotics systems.

Education Use Consent

I hereby give my permission for this project to be shown to other University of Glasgow students and to be distributed in an electronic form.

Name: Hrushikesh Sanap Signature: Hrushikesh Sanap

Acknowledgements

Foremost, I would like to express my sincere gratitude to my advisor Professor Phil Trinder for the continuous support of my MSc study and research, for his patience, motivation, enthusiasm, and immense knowledge. His guidance helped me in all the time of research and writing of this dissertation.

Besides my Advisor, I would like to thank the University of Glasgow for providing necessary tools and facilities for my research.

Also, I thank my family and friends, who supported me throughout the entire process, both by keeping me harmonious and helping me put pieces together. I will be grateful forever for your love.

Finally, I express my profound gratitude to everyone who supported me throughout the year of my MSc study. This dissertation would not have been possible without them.

Thank you.

Contents

Chapter 1 Introduction

1.1 Context

Prior to the emergence of Robotic Middleware, assembling intricate robotic systems was an extremely challenging process that necessitated substantial customization and tight integration of hardware and software components. Vendors developed proprietary elements designed for particular architectures, interfaces and protocols. Integrating a novel module necessitated redesigning the whole system architecture for compatibility. These constraints resulted in inflexible frameworks where exchanging a sensor or actuator mandated reworking the overall communications and workflows. Upgrading to new algorithms also entailed extensive recording. Moreover, component reusability across projects was low due to custom interfaces. Enabling distributed or collaborative robotics was exceptionally difficult without standardized messaging protocols (Li et al., 2012). In summary, developing complex robotic systems from a diverse set of components was an exceedingly time-consuming and error-prone undertaking.

Robotic Middleware materialized as a solution to these systems integration challenges. It functions as an intermediary layer that connects the various hardware and software modules in modern robotic systems. Middleware abstracts the technical details of each component and provides a common set of interfaces and protocols. This facilitates seamless communication and coordination between components, enabling them to operate in unison as an integrated robotic system. Middleware also promotes modularity and code reuse. Novel hardware and software modules can be readily incorporated into existing robotic architectures via standardized middleware interfaces. The emergence of Robotic Middleware enhanced flexibility and adaptability in robotics (Li et al., 2012).

ROS (Robot Operating System) (Open Robotics, n.d.) and ROS2 (Robot Operating System 2) (ROS 2 Documentation, n.d.) are the most widely adopted Robotic Middleware solutions. ROS is an open-source framework that offers standardized abstraction above hardware and streamlines robotic software development through tools, libraries and conventions. ROS enables modular code organization, straightforward integration of new capabilities, and distributed computing across multiple systems. A key capability is the ROS messaging system that establishes communication between processes using topics, services and actions. ROS also provides an ecosystem of open-source packages contributed by the community for functions like hardware drivers, visualizations, simulations and perception. Its vibrant community support cements ROS as a leading middleware choice for prototyping and deploying robotic systems ranging from hobby projects to industrial systems (Quigley et al., 2009).

While ROS pioneered Robotic Middleware and remains extensively utilized today, it has limitations that led to the creation of ROS2. ROS2 modernizes ROS with new capabilities required for emerging robotics applications. It bolsters

security via encrypted communications, makes ROS real-time capable for time-critical control, and expands cross-platform support including embedded systems. A major innovation in ROS2 is substituting the ROS messaging framework with the DDS (Data Distribution Service) standard. DDS enables decentralized peer-to-peer networking between robots versus ROS's centralized model. This facilitates coordination between multiple robots. Although ROS still sees wider adoption presently, ROS2 uptake is accelerating as it matures. Its redesigned architecture tailors ROS for upcoming robotics challenges that demand distributed intelligence, real-time performance, security and runtime flexibility. ROS2 also combines the strengths of ROS with DDS to deliver the best of both platforms (Macenski et al., 2022).

1.2 Contributions

a. **Literature and Technology Review**: (Chapter 2) An extensive literature review was undertaken, encompassing a thorough exploration of Robotic Middleware, a detailed analysis of the communication architectural nuances of ROS and ROS2, and an evaluation of the inherent limitations of ROS. This provided the necessary background for a comprehensive comparison between ROS and ROS2 in terms of communication.

b. **Communication Benchmarks to compare ROS and ROS2**: (Chapter 3) The research involved designing and conducting benchmarking experiments to evaluate the performance of ROS and ROS2 communication under different conditions. For ROS, tests were devised to measure the impact of varying message sizes and frequencies on latency, throughput, and bandwidth for communication within the same Raspberry Pi device and between two devices. Similar benchmarking tests were designed for ROS2 to assess how message size and frequency affect key performance parameters. A key component was evaluating two different DDS implementations, Fast DDS (eProsima, n.d.) and Cyclone DDS (Eclipse Foundation, n.d.), to determine the influence of the DDS selection on ROS2 communication efficiency. As with ROS, experiments were executed for both intra-board and inter-board communication scenarios.

c. **Comparative Analysis of Intra-Board and Inter-Board Communication utilizing ROS and ROS2 with different DDS:** (Chapter 4.1) A comparative analysis was undertaken of intra-board and inter-board communication for ROS and ROS2. By incrementally increasing message sizes and frequencies, the impact on crucial performance metrics, namely latency, throughput, and bandwidth, was assessed. Also, a comparative analysis was also conducted for ROS2 communication, with a specific focus on the influence of DDS selection (Chapter 4.2). By analyzing the effects of varying message sizes and frequencies on benchmarking parameters, insights were gained into the performance discrepancies between Fast DDS and Cyclone DDS in the context of ROS2.

d. **Comparative Analysis of ROS and ROS2 Communication:** (Chapter 4.3) A comprehensive comparative analysis was undertaken, directly contrasting ROS against ROS2. By examining the impact of message sizes and frequencies on latency, throughput, and bandwidth, this analysis illuminated the relative strengths and weaknesses of ROS and ROS2. For intra-board communication, at all frequencies ROS shows higher latency compared to ROS2 irrespective of DDS. ROS shows an advantage in terms of Throughput and Bandwidth at low frequencies (Below 30 Hz). But as frequency increases, ROS2 irrespective of DDS substantially outperforms ROS in latency, throughput and bandwidth. For inter-board communication below 20 Hz, ROS provides maximum throughput and bandwidth; however, at higher frequencies, ROS2 is preferable for its substantially lower latency and more gradual declines in throughput and bandwidth.

Chapter 2 Literature and Technology Review

2.1 Robotic Middleware

Robotic middleware refers to a software layer that is situated between the operating system and the application layer within a robotic system. It serves as an intermediary, furnishing a set of tools, libraries, and services that facilitate communication, coordination, and integration between the various software components and hardware devices in a robotics ecosystem. (Li et al., 2012)

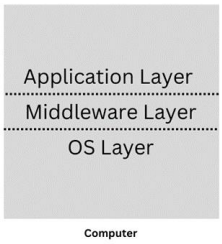

Figure 2-1 Robotic Ecosystem (Li et al., 2012)

Robotic middleware plays a crucial role in enabling the development and operation of intricate robotic systems. It provides a standardized and modular framework for communication and coordination between different components, such as sensors, actuators, control algorithms, and higher-level decision-making modules (Li et al., 2012).
Examples of popular robotic middleware frameworks include ROS (Open Robotics, n.d.), ROS2 (ROS 2 Documentation, n.d.) and Orocos (Orocos, n.d.). These middleware platforms furnish a rich set of functionalities and tools that streamline the development, deployment, and maintenance of robotic systems, promoting code reusability, modularity, and interoperability.

2.2 ROS: Architecture and Features

ROS is a flexible and modular framework designed to support the development of robot software. It provides a collection of tools, libraries, and conventions that facilitate the creation of robot applications. The primary goal of ROS is to enable collaboration and code reuse among researchers and developers in the field of robotics. ROS was developed considering following philosophical objectives :

a. *Peer-to-peer*: ROS enables efficient inter-process communication through its peer-to-peer publish/subscribe model. (Quigley et al., 2009)
b. *Tool-based*: ROS offers a comprehensive set of tools for development, debugging, and management of robotics applications. (Quigley et al., 2009)
c. *Multilingual*: ROS supports multiple programming languages, primarily Python and C++, for flexible development. (Quigley et al., 2009)
d. *Thin*: ROS serves as a lightweight middleware layer, providing communication and hardware abstraction while relying on external libraries for specific functionality. (Quigley et al., 2009)

e. *Free and open source*: ROS is freely available and distributed under an open-source license, encouraging collaboration and community-driven innovation. (Quigley et al., 2009)

2.2.1 ROS: Architecture

Major Parts of ROS architecture are discussed below.

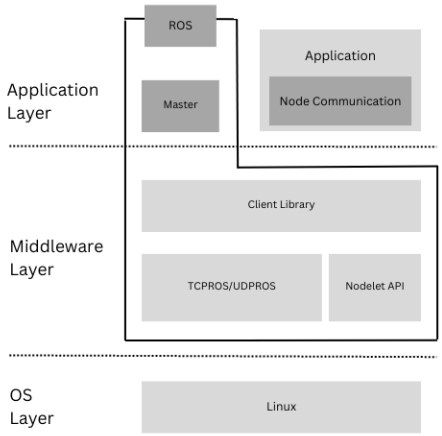

Figure 2-2 ROS Architecture (Kronauer et al., 2021)

The underlying OS (Operating System) layer provides essential functionalities and services to the ROS system. Linux-based operating systems furnish the foundation for ROS. ROS applications and components operate on top of Linux, harnessing its memory allocation and hardware interaction capabilities. (Kronauer et al., 2021)

The middleware layer in ROS offers the communication infrastructure enabling interaction between various nodes in a distributed system. This layer encompasses libraries, protocols, and mechanisms that facilitate inter-process communication within the ROS ecosystem. The client libraries are software packages providing programming interfaces for developing ROS nodes. These libraries offer abstractions and APIs for creating, configuring, and managing ROS nodes and their communication. TCPROS (TCP-based ROS) (ROS Wiki, n.d.) and UDPROS (UDP-based ROS) (ROS Wiki, n.d.) are communication protocols used by ROS nodes to exchange messages, determining how messages are transmitted between nodes across the network. TCPROS utilizes a reliable TCP connection for message delivery while UDPROS uses a more lightweight, less reliable UDP connection. The Node API is a set of programming interfaces ROS nodes employ to communicate with each other, providing methods for sending/receiving messages, managing subscriptions/publications, and handling other node-related tasks (Kronauer et al., 2021).

The application layer of ROS involves the higher-level functionalities defining the behavior of the robotic system. The ROS Master is a central registry assisting nodes in discovering each other and facilitating communication coordination. It manages node registration, topic name resolution, and service discovery, crucial for enabling nodes to efficiently find and communicate with each other. Node communication occurs through publishing messages on specific topics and subscribing to topics of interest, enabling data exchange, coordination, and

cooperation among the robotic system's different components (Kronauer et al., 2021).

ROS Communicational Architecture consists of following parts:

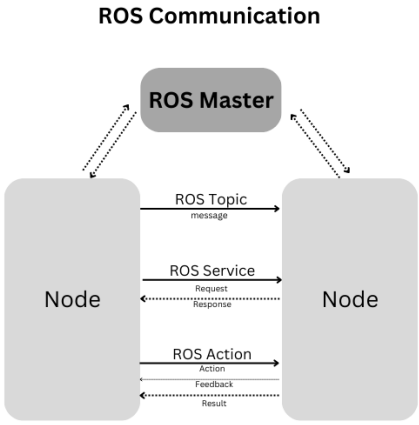

Figure 2-3 ROS communicational Architecture

a. Nodes: ROS architecture is based on a distributed system of nodes. A node is a modular component that performs a specific task. Consider them as interconnected software component. Nodes can communicate with each other by publishing and subscribing to topics or by using services. (Quigley et al., 2009)
b. Master: The master node is responsible for facilitating communication between different nodes. It keeps track of the available nodes, manages the ROS parameter server, and enables nodes to find each other. (Quigley et al., 2009)
c. Topics: Nodes communicate with each other by publishing messages on topics. The topic is a named bus where data is exchanged. Multiple nodes can subscribe to a topic to receive the messages. (Quigley et al., 2009)
d. Messages: Messages define the data structure and format of the information exchanged between nodes. ROS provides a variety of predefined message types, and users can define their custom messages. (Quigley et al., 2009)
e. Services: Nodes can also provide and consume services. Services allow nodes to make requests and receive responses. (Quigley et al., 2009)
f. Action: provide a mechanism for executing long-running, goal-oriented tasks that require feedback and cancellation capabilities, enhancing the coordination and execution of complex robotic behaviours. (Quigley et al., 2009)
g. Parameter Server: The parameter server is a global storage system for sharing configuration data across nodes. It allows nodes to retrieve and set parameters dynamically. (Quigley et al., 2009)

2.2.2 ROS: Features

ROS employs a modular architecture that promotes code reusability and accelerated development by enabling developers to construct nodes that can be effortlessly reused and combined with other nodes. Efficient communication between nodes is facilitated through message passing, with nodes able to publish and subscribe to topics to ensure loose coupling of components. Powerful visualization tools such as RViz and rqt_plot are provided, empowering

developers to visualize robot states, sensor data, and other information critical for debugging and analysis during development. A package-based system is utilized to organize and share code, with packages containing nodes, libraries, launch files, and configuration files. ROS package management tools streamline the installation and dependency management of software components. Robust logging capabilities allow developers to record and analyze robot system behavior while debugging tools like roscpp and rospy offer features to inspect and debug code. Seamless integration with simulation environments like Gazebo and Stage enables developers to validate algorithms in virtual environments prior to deployment on physical robots (Quigley et al., 2009). An active community of developers and researchers contributes to ROS development, providing support, knowledge sharing, and numerous open-source packages and libraries to enhance robot development. Platform independence enables portability of ROS applications across hardware architectures and operating systems. A rich ecosystem of libraries and tools extends the capabilities of ROS through contributions like MoveIt! for motion planning, ROS Control for robot control, and perception libraries such as PCL (PCL, n.d.).

2.3 ROS: Limitations

While ROS provides numerous benefits and has become a popular choice for robotics development, it also has certain limitations. These are discussed below.

a. *Limited Real-time Capabilities*: ROS is not designed as a real-time operating system. While it provides timing and synchronization mechanisms, it lacks the determinism and precise timing control required for certain applications, such as control systems in critical environments.
b. *Lack of Security Measures*: ROS was initially designed for research and development purposes, where security concerns were not the primary focus. As a result, there are limited built-in security measures in the core ROS framework. When deploying ROS in production environments, developers must implement additional security measures to protect against unauthorized access and data breaches.
c. *Limited Compatibility (Operating System*): Historically, ROS has primarily been developed and used on Linux systems. Even now it is primarily developed for operating systems such as ubuntu and Debian. Though some experimental versions ROS had been launched for windows, there are still some limitations and certain functionalities that are not fully supported on Windows.
d. *Limited Compatibility (Programming Languages):* One of the limitations of ROS is its limited language support compared to other frameworks. While ROS provides support for multiple programming languages, the core libraries and tools are primarily developed and optimized for two main languages: C++ and Python. The majority of ROS libraries and packages are written in C++ and Python (initially it supported Lisp as well), which means that developers using other languages may have limited access to existing functionality. While efforts have been made to provide bindings or wrappers for other languages, the coverage and community support for these languages is comparatively lower.
e. *Master Node Failure*: One limitation of ROS is the vulnerability to master node failure. In ROS, the master node acts as a central coordination point,

facilitating communication between different nodes in the system. If the master node fails or experiences an issue, it can disrupt the entire ROS network, leading to communication failures and the inability to coordinate nodes effectively.

To overcome the above issues, ROS2 was developed.

2.4 ROS2: Architecture and Features

ROS2 is the second generation of the Robot Operating System, designed to address the challenges faced in the commercialization of robotics technology. It introduces new node types and executor models that provide control over where, how, and when information is processed in the computational graph. with robots being deployed for various commercial applications. To meet the challenges of modern robotic systems, ROS2 was developed as a redesigned version of ROS. ROS2 incorporates necessary production-grade features and algorithms, addressing limitations in security, performance, reliability in non-traditional communicational environments, and support for large scale. (Macenski et al., 2022)

2.4.1 ROS2: Architecture

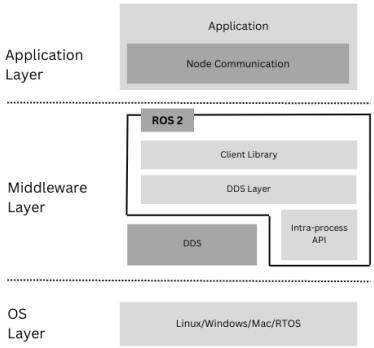

Figure 2-4 ROS2 Architecture (Kronauer et al., 2021)

The operating system layer for ROS2 consists of various operating systems including Linux, Windows, macOS and RTOS (real-time operating system) (FreeRTOS, n.d.). The ROS2 middleware layer can be constructed on all these operating systems, conferring cross-platform capabilities (Kronauer et al., 2021).

The middleware layer in ROS2 is built around DDS. It handles communication between nodes and furnishes a distributed publish-subscribe mechanism enabling data exchange. Similar to ROS, ROS2 also provides client libraries offering programming interfaces for creating and managing ROS2 nodes. The DDS represents a standardized middleware protocol that facilitates real-time communication and data sharing between distributed systems. In ROS2, DDS is utilized as the communication middleware, enabling nodes to publish and subscribe to topics while handling message delivery and synchronization. An Inter-Process API is provided, furnishing a set of standardized methods and interfaces for ROS2 nodes to communicate with each other. This API abstracts the complexities of DDS and provides a unified means for nodes to send/receive

messages, establish service requests/responses, and manage other interactions. (Kronauer et al., 2021)

As with ROS, communication between ROS2 nodes transpires in the application layer. However, unlike ROS, ROS2 does not necessitate a master node to facilitate node discovery and communication. This allows a more decentralized architecture. Both ROS and ROS2 are based on same philosophies, leading to both having mostly similar communicational architecture (Kronauer et al., 2021). These similarities are as follows.

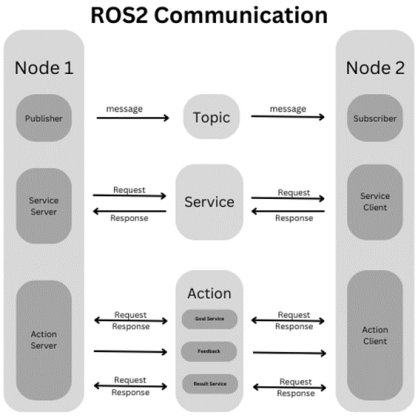

Figure 2-5 ROS2 Communicational Architecture

ROS and ROS2 share fundamental architectural elements including a node-based architecture that enables modularity, facilitating the design of separate nodes implementing different functionalities that can communicate. Both employ message-passing for data exchange between nodes. The notion of topics is present in ROS and ROS2, incorporating a publish-subscribe messaging pattern for communication between nodes. Nodes can publish messages to topics while other nodes can subscribe to those topics to receive the messages, enabling loose coupling as publishers need not be aware of specific subscribers. Services are also supported in both systems, furnishing request-response communication between nodes. While ROS introduced the concept of actions for managing asynchronous tasks, ROS2 actions build upon this by offering improved reliability and finer control over robotic behaviors through enhanced feedback and cancellation capabilities. The shared architectural elements provide continuity while ROS2 aims to furnish critical improvements. (Macenski et al., 2023)

2.4.2 ROS2: Features

Unlike ROS, ROS2 uses the DDS as its default communication middleware. DDS allows for fast, real-time communication between nodes and makes sure data is reliably exchanged. This boosts ROS2's performance and scalability, making it suitable for mission-critical applications where timing is important. ROS2 puts a big focus on real-time capabilities by integrating the Real-Time Working Group stuff. This lets developers build systems that need precise timing control, which is useful for time-sensitive applications. (Macenski et al., 2022)

Security is also improved in ROS2. It has secure communication protocols, access control, data encryption - things that provide confidentiality, integrity, authenticity for communication. This allows ROS2 to be used in secure, sensitive

settings. More languages like Java, C#, and Rust are supported beyond C++ and Python. This lets developers use their preferred language while still taking advantage of ROS2. ROS2 has built-in time synchronization to coordinate nodes in distributed systems. Nodes can share a common sense of time, which helps coordinate actions that need to happen at specific times. There's also a Quality-of-Service(QoS) framework that gives users a lot of control over reliability, durability, deadlines, bandwidth, and other parameters for data transmission. This lets people customize communication to match what their application requires. (Macenski et al., 2023)

2.5 ROS and ROS2: Comparison

Parameter	**ROS**	**ROS2**
Communication Model	Master-Slave communication model. where nodes communicate through a central master node.	Decentralized communication model. where nodes communicate directly with each other without Master node
Communication Protocols	TCPROS and UDPROS are primary communication protocols	Communication via DDS. It is more reliable faster and secure. Fast DDS and Cyclone DDS are the examples of available options.
Language Support	Primarily supports C++ and Python programming languages.	Support languages such as C++, Python, Java, C#, and Rust.
Operating System Support	Primarily developed for Ubuntu Linux	Supports Ubuntu Linux, MacOS, Windows etc.
Real-time Capabilities	ROS does not provide built-in real-time capabilities and is not suitable for applications with strict timing requirements.	ROS2 introduces real-time extensions and features, making it more suitable for real-time and time-critical applications.
Robustness and Reliability	ROS has limitations in terms of robustness and reliability due to its centralized architecture and limited error handling mechanisms. Failure in Master node can crash the whole system.	ROS2 addresses the limitations of ROS by adopting a decentralized architecture, introducing better error handling, and providing mechanisms for fault tolerance and recovery.
Security	ROS has limited built-in security measures, as it was primarily designed for research and development	ROS2 focuses on security and incorporates features such as secure communication protocols, access control, and

	purposes.	data encryption.
Quality of Service (QoS)	ROS has limited support for specifying Quality of Service (QoS) parameters.	ROS2 incorporates a comprehensive QoS framework that allows users to define QoS settings for reliable, time-sensitive communication.

Table 2-1 Comparison Between ROS and ROS2 (Quigley et al., 2009) (Macenski et al., 2022) (Kronauer et al., 2021), (Gurjot, 2021), (Electronic Design, 2022) (Robotics Backend, 2022)

2.6 Benchmarking

Benchmarking is a critical process in evaluating and comparing the performance of systems, software, or hardware components. It involves running a set of predefined tests and measurements to assess various metrics such as speed, throughput, latency, memory usage, and scalability (ASQ, 2023).

2.6.1 Benchmarking: Overview

To conduct effective benchmarking, it is important to define clear goals, select appropriate benchmarks that represent real-world scenarios, ensure reproducibility, and consider the hardware and software environments. Additionally, benchmarking should be performed on a variety of test cases to account for different usage scenarios and workloads. The benchmarking process typically involves several steps to ensure a systematic and reliable evaluation of system performance. Step-by-step explanation of the benchmarking process is as follows:

Define Goals and Metrics	Clearly state the objectives and identify the specific performance metrics to be measured.
Select Benchmarks	Choose representative tests that simulate real-world scenarios and cover a range of system operations.
Prepare Test Environment	Set up a controlled and consistent environment, ensuring stable system conditions for benchmark execution.
Execute Benchmarks	Run the selected benchmarks multiple times, collecting relevant data on execution times and resource usage.
Analyze Results	Interpret benchmark data, compare metrics, identify performance bottlenecks, and draw meaningful conclusions.
Optimize and Iterate	Make necessary system optimizations based on benchmark analysis, repeating the process after each iteration.
Document and Report	Create a detailed report documenting the benchmarking process, setup, parameters, and results
Benchmark Maintenance	Regularly repeat the benchmarking process to monitor performance changes over time and ensure continued optimization

(ASQ, 2023) (KnowledgeHut, 2023)

Benchmarking plays a crucial role in optimizing and validating the performance of ROS and ROS2 robotics frameworks. By simulating real-world conditions, benchmarks identify performance bottlenecks, allow objective comparisons between systems, and ensure communication requirements are met in terms of latency, reliability and high-volume handling. Benchmarking establishes industry standards, best practices and baselines for measuring future improvements. The insights gained help fine-tune configurations for optimal efficiency. Overall, benchmarking enables continuous performance enhancement, informed decision-making and the development of robust and responsive robotics communication protocols and frameworks.

2.6.2 ROS and ROS2: Communication Benchmarking

It's important to note that while ROS2 introduces significant improvements, ROS remains a valuable framework with extensive resources and support. The choice between ROS and ROS2 depends on specific project requirements, real-time constraints, language preferences, and the maturity of the associated libraries and tools (Kronauer et al., 2021).

Communication benchmarking plays a crucial role in evaluating the performance and efficiency of the communication infrastructure in ROS and ROS2. It involves assessing the messaging capabilities, latency, throughput, and reliability of the communication mechanisms used within these frameworks. Performance evaluation of ROS and RPS 2 can be done using following parameters.

a. *Latency*: Latency is the time delay between sending and receiving data, measuring the responsiveness and speed of communication in a system. Latency measures the time it takes for a message to travel from a sender to a receiver. It is an essential metric for assessing the responsiveness of the communication system. Lower latency indicates faster message propagation and reduced delays in data exchange. (Kronauer et al., 2021)
b. *Throughput*: Throughput measures the rate at which messages can be transmitted or processed. It is a measure of the communication system's capacity and efficiency in handling a high volume of messages. Higher throughput implies better scalability and the ability to handle increased data traffic. (Kronauer et al., 2021)
c. *Bandwidth*: Bandwidth is the data transfer rate of a communication network. It signifies how much data can be sent within a certain time. In ROS and ROS2 communication benchmarking, bandwidth is crucial for assessing the systems' capacity to handle data. A higher bandwidth enables smoother data exchange between components, vital for tasks like sensor-heavy robotics or real-time control. Measuring bandwidth helps gauge the frameworks' efficiency in managing data-intensive operations. (Kronauer et al., 2021)

Chapter 3 Benchmark Design and Methodology

This chapter outlines the methodology utilized to conduct comprehensive benchmarking of the ROS and ROS2 middleware for intra-board and wireless inter-board communication. Rigorous benchmarking provides quantifiable performance data to compare these two middleware options for real-time distributed robotics applications using embedded computing platform like Raspberry pi. The benchmarking focuses on evaluating three key communication parameters - latency, throughput, and bandwidth. Four main test categories were defined based on conducting benchmarks for ROS and ROS2, considering both internal intra-board and wireless inter-board communication scenarios. For each category, separate experiments were designed to assess latency, throughput and bandwidth performance.

The test platform consists of Raspberry Pi 3B+ single board computers running Ubuntu 20.04 with the ROS Noetic and ROS2 Foxy distributions. This hardware and software configuration was selected to allow benchmarking the latest long-term support versions of both ROS and ROS2 on the same embedded hardware. Custom ROS nodes were developed to publish and subscribe to messages for measuring latency and throughput. Data was collected on the publishing and subscribing endpoints to quantify the message delay and message rate. The subsequent sections detail the experimental methodology, test configurations, and data collection process for each of the defined benchmark categories and experiments. The approach provides a comprehensive assessment of ROS and ROS2 communication performance critical for real-time distributed robotics applications. The results establish baseline metrics that can inform middleware selection based on application requirements and hardware constraints.

3.1 Experimental Setup

A robot's software stack is made up of three main layers that work together with the hardware to create the robotic system. At the base is the operating system, which manages the hardware resources and low-level controls. Built on top of that is middleware like ROS and ROS2, which allows the different software components to communicate. And at the highest level is the application layer, which contains the specific programs and behaviours for that particular robot. The hardware provides the physical computing platform, while the software enables all the functionality. Together, the hardware and this layered software architecture make up the robotic ecosystem, combining physical and digital capabilities to accomplish complex tasks. These individual components of robotic ecosystem are discussed below.

3.1.1 Hardware: Raspberry Pi model 3B+

The Raspberry Pi 3 Model B+ (Raspberry Pi Foundation, 2018) is an enhanced iteration of the popular Raspberry Pi single-board computer line, incorporating upgrades in processing power, connectivity, and capabilities. It comes equipped with a faster 64-bit quad-core processor clocked at 1.4GHz, delivering superior performance in multitasking and computations compared to previous Pi models. Connectivity is expanded through built-in dual-band WiFi, Bluetooth 4.2, Ethernet, and multiple USB ports for wireless and wired options. This upgraded

Pi retains support for existing enclosures and add-ons, as well as compatibility with a diverse range of operating systems. It provides the same flexible GPIO pins for connecting external electronics and hardware components. With its improved hardware and versatility, the Raspberry Pi 3 Model B+ is widely applied in education, home automation, robotics, entertainment systems, and IoT products. A key advantage that continues from earlier Pis is the ability to run various Linux distributions or other OS options beyond the official Raspberry Pi OS. The GPIO interface enables projects involving sensors, motors, and other DIY circuits. Overall, the Raspberry Pi 3 Model B+ offers enhanced performance and connectivity while maintaining the benefits of accessibility, customization, and community support from the Raspberry Pi ecosystem.

3.1.2 Software: Operating System and Middleware

a. Operating System:
Ubuntu Server 20.04 (Focal Fossa) + *Lubuntu desktop*: In order to obtain consistent and valid benchmark results for both ROS and ROS2 middleware, it was important to choose an operating system that supports both. Since most ROS distributions have reached end-of-life, ROS Noetic Ninjemys on Ubuntu 20.04 Focal Fossa (Ubuntu Documentation, 2020) was the only option. This is the last ROS release with support until May 2025. Ubuntu 20.04 is also recommended for ROS2 Foxy Fitzroy, which was still supported during this research. Therefore, to run benchmarks using the latest available ROS and a supported ROS2, Ubuntu 20.04 Focal Fossa was selected as the operating system. This ensured compatibility with both middleware versions for consistent benchmarking. The Ubuntu 20.04 Focal Fossa server image was installed on the Raspberry Pi 3B+ hardware. The server image was chosen due to the limited RAM availability on the Raspberry Pi. The server edition has no GUI installed by default for improved security and lower resource usage. To provide a desktop environment on top of the Ubuntu server, Lubuntu was installed. Lubuntu is a lightweight desktop version of Ubuntu, making it suitable for resource-constrained devices like the Raspberry Pi. Installing Lubuntu over the Ubuntu server image allowed for a desktop environment while still conserving RAM on the Raspberry Pi 3B+.

b. Middleware Versions:
- *ROS Noetic Ninjemys*: (Robot Operating System, 2020) It is released in May 2020, is the most recent long-term support (LTS) version of the ROS middleware. It is expected to reach end-of-life status in May 2025, providing five years of support. It also supports Ubuntu 20.04 Focal Fossa. Key thing to note that it is going to be the last ROS distribution.
- *ROS2 Foxy Fitzroy*: (Robot Operating System, 2020) It first launched in June 2020. This major ROS2 version targeted the Ubuntu 20.04 LTS (Focal Fossa) and Ubuntu 18.04 LTS (Bionic Beaver) Linux distributions as its officially supported platforms. After its initial release, Foxy Fitzroy received three years of support and maintenance as per the ROS 2 LTS policy, before reaching its designated end-of-life in June 2023. It is important to mention that in ROS2 message communication is done primarily

via DDS. For benchmarking two DDS are utilized and they are Fast DDS and Cyclone DDS.

3.2 Benchmark Classification

The literature review (Chapter 2.6.2) discussed that benchmarking ROS and ROS2 communication would involve evaluating three key parameters: latency, throughput, and bandwidth. The benchmarking would assess both intra-board and inter-board wireless communication scenarios for each middleware. For intra-board communication, the publishing and subscribing nodes would be on the same Raspberry Pi device, testing internal communication latency and throughput. For inter-board communication, the nodes would be on separate Pis communicating wirelessly, evaluating external wireless latency, throughput, and derived bandwidth. Individual experiments would be performed to benchmark latency and throughput independently for both ROS and ROS2. Latency provides the message delay, while throughput reveals the message rate. Bandwidth, or data transfer rate, could be calculated from throughput by multiplying by message size.

This benchmarking would allow evaluating how ROS and ROS2 perform for real-time distributed robotics applications using low-powered embedded devices like Raspberry Pis. It would show their communication effectiveness on small boards and wireless networks. The benchmarking tests are classified into four main categories: ROS intra-board, ROS inter-board, ROS2 intra-board, and ROS2 inter-board. Within each category, separate experiments will be conducted to evaluate latency and throughput/bandwidth performance.

1. ROS Intra-Board Communication Benchmarking
    a. ROS Intra-Board Latency Benchmark
    b. ROS Intra-Board Throughput and Bandwidth Benchmark
2. ROS Inter-Board Wireless Communication Benchmarking
    a. ROS Inter-Board Latency Benchmark
    b. ROS Inter-Board Throughput and Bandwidth Benchmark
3. ROS2 Intra-Board Communication Benchmarking
    a. ROS2 Intra-Board Latency Benchmark with Fast DDS and Cyclone DDS
    b. ROS2 Intra-Board Throughput and Bandwidth Benchmark with Fast DDS and Cyclone DDS
4. ROS2 Inter-Board Wireless Communication Benchmarking
    a. ROS2 Inter-Board Latency Benchmark with Fast DDS and Cyclone DDS
    b. ROS2 Inter-Board Throughput and Bandwidth Benchmark with Fast DDS and Cyclone DDS

This comprehensive benchmarking approach across both ROS and ROS2 and for both intra-board and wireless inter-board communication will provide a detailed performance comparison between the two middleware options. The results will quantify the latency, throughput, and bandwidth that can be expected for real-time robotic applications utilizing ROS versus ROS2 on embedded hardware platforms. These Benchmarking tests are discussed individually and in details below.

3.3 Communication Benchmarks

3.3.1 ROS Intra-Board Communication Benchmarking

a. ROS Intra-Board Latency Benchmark

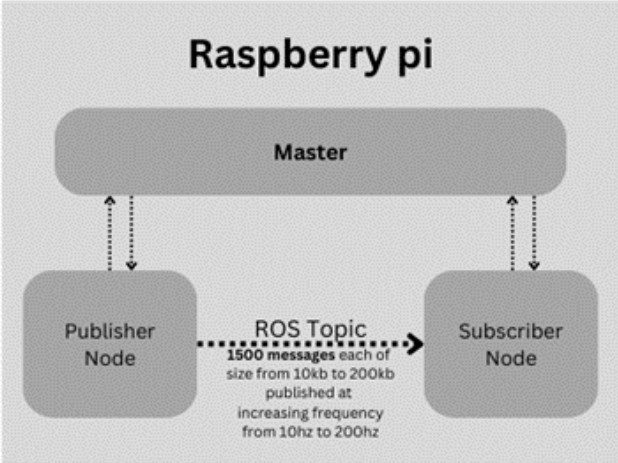

Figure 3-1 ROS intra-board latency benchmark

This benchmark test was designed to calculate the median latency for ROS communication within a single Raspberry Pi board. It involved a Publisher node and a Subscriber node running on the same Pi. The Publisher node published messages ranging in size from 10kb to 200kb, incrementing by 10kb. It published at frequencies from 10Hz to 200Hz, incrementing by 10Hz. The queue size was set to 10000 and 1500 messages were published per test run. After publishing 1500 messages, a 30 second cool down period was implemented before the next test. This ensured all messages were fully published before moving on, avoiding data loss.

The Subscriber node was subscribed to the same topic as the Publisher to receive the messages. It logged the time each message was received. ROS provides the ability to timestamp when a message is published or received by a subscriber. The Publisher node logged a timestamp when each test message was published along with message size, frequency and message number. The Subscriber logged a timestamp when that same message was received from the topic. By comparing the publish timestamp to the receive timestamp for a given message, the latency could be calculated for that specific message. This was done for all 1500 messages in each test run. Since there were 1500 latency data points per test, a median latency could be determined that represented the central tendency of the results. Taking the median helped account for any outlier values that could skew the mean.Sweeping across the different message sizes and frequencies revealed how those parameters impacted the median latency. Doing 1500 iterations per test provided a large sample set that gave an accurate median value reflecting typical performance for that condition. This approach leveraged ROS's timestamp functionality to empirically measure the end-to-end latency for communication within a single Raspberry Pi board. The median latency metric calculated from the 1500 message timings quantified the intra-board performance.

b. ROS Intra-Board Throughput and Bandwidth Benchmark

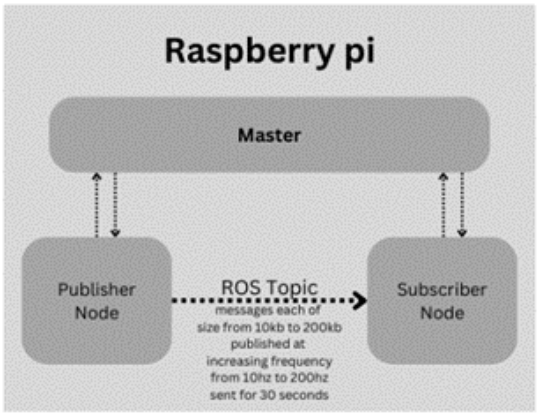

Figure 3-2 ROS intra-board throughput and bandwidth benchmark

This benchmark Studies the throughput and bandwidth of ROS communication within a Raspberry Pi. It uses a Publisher node and Subscriber node on the same Pi. The Publisher node is configured to publish messages at incremental sizes from 10kb to 200kb, increasing by 10kb. It transmits these message sizes at frequencies sweeping from 10Hz up to 200Hz, incrementing by 10Hz. This allows characterizing how message size and frequency impact performance. The queue size is set large at 10000 to avoid limiting throughput. For each message size and frequency combination, the Publisher transmits for a fixed 30 second interval. This provides a consistent window for comparison. After 30 seconds, a 15 second cool down period is implemented before moving to the next message size and frequency to allow the system to reset.

The Subscriber node runs on the same Pi and subscribes to the topic that the Publisher is transmitting on to receive the messages. ROS provides timestamping functionality when publishing and subscribing messages. The Publisher logs the message size, publishing frequency, and timestamp when each message is transmitted. The Subscriber logs the timestamp when each message is received. By comparing the total messages received within the 30 second window to the expected number based on the publishing frequency, the throughput percentage and bandwidth can be calculated. Bandwidth can then be derived by multiplying the throughput percentage by the message size.This benchmark across varying sizes and frequencies characterizes how these parameters impact the intra-board throughput and bandwidth capacity of ROS communication.

3.3.2 ROS Inter-Board Communication Benchmarking

a. ROS Inter-Board Latency Benchmark

This benchmark Studies the inter-board communication latency of ROS over wireless connections between two Raspberry Pi devices. In this benchmark, two distinct Raspberry Pi devices are used. The setup involves wireless ROS communication established between two Raspberry Pi's, with the ROS master node running on just one of them, while the other works as a ROS node. Overall three nodes are utilized. Node1 assumes the role of a publishing node, responsible for transmitting data on Topic1. Node2 operates on the second

Raspberry Pi and serves the dual role of publisher and subscriber. Specifically, it subscribes to Topic1, processes messages received from Node1, and subsequently publishes confirmation messages on Topic2, signalling the successful reception of data from Node1. Node3, residing on the first Raspberry Pi, subscribes to Topic2, where it receives the confirmation messages transmitted by Node2.

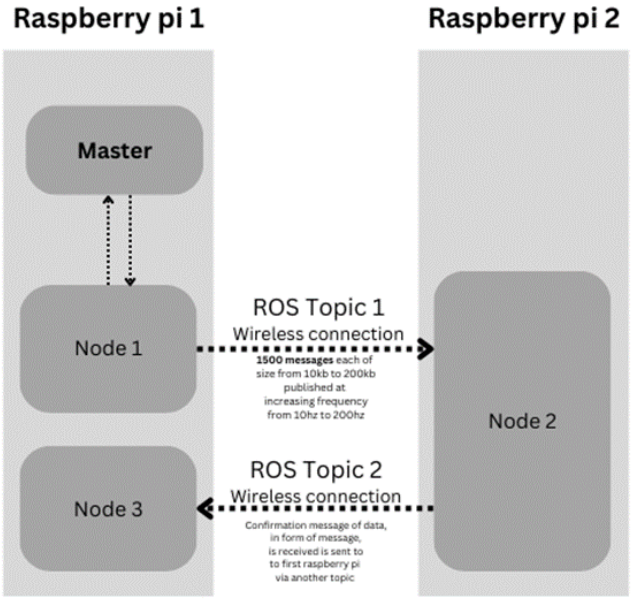

Figure 3-3 ROS inter-board latency benchmark

The Node1 published messages ranging in size from 10kb to 200kb, incrementing by 10kb. It published at frequencies from 10Hz to 200Hz, incrementing by 10Hz. The queue size was set to 10000 and 1500 messages were published per test run. After publishing 1500 messages, a 30 second cool down period was implemented before the next test. This ensured all messages were fully published before moving on, avoiding data loss.

Node1 logs a timestamp along with message size, frequency, and message number for each test message upon publication to Topic1. Concurrently, Node3 captures a timestamp upon receiving the same message from Topic2. By computing the time difference between the Node3 timestamp and the Node1 timestamp for given message, divided by two, the latency for that specific message can be approximated. This division by two is performed to derive an estimate of the time taken to transmit a message from Node1 to Node2. This latency calculation is conducted for all 1,500 messages within each test run. Since there were 1500 latency data points per test, a median latency is calculated to represent central tendencies of the results.

Node1 and Node3 timestamps were used, rather than Node1 and Node2, because Node1 and Node3 were on the same Raspberry Pi. Their synchronized clocks enabled precise measurement of the end-to-end elapsed time. In contrast, Node1 and Node2 were on separate devices with likely clock skew. Using just their timestamps would have introduced inaccuracy. Leveraging the synchronized Node1 and Node3 clocks provided an accurate round-trip communication time. Dividing by two estimated the one-way latency between boards, removing the impact of clock skew.

b. ROS Inter-Board Throughput and Bandwidth Benchmark

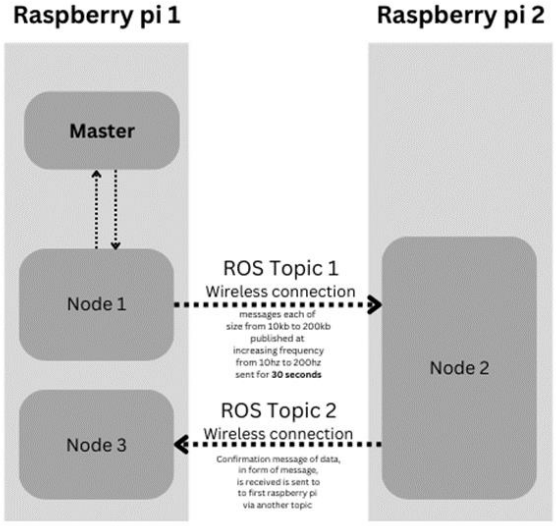

Figure 3-4 ROS inter-board throughput and bandwidth benchmark

This Benchmark test investigates the inter-board communication throughput and bandwidth of the ROS over wireless connections between two Raspberry Pi devices. The experimental setup utilizes ROS for wireless communication between two separate Raspberry Pis, with one device hosting the ROS master node and the other operating as a ROS node. In total, three ROS nodes are employed. Node 1 serves as the publishing node, responsible for transmitting data on Topic 1. Node 2 runs on the second Raspberry Pi, acting as both a publisher and subscriber - it subscribes to Topic 1 to receive data from Node 1, processes the messages, and subsequently publishes confirmation messages on Topic 2 to indicate successful reception of Node 1's data. Node 3, on the same device as Node 1, subscribes to Topic 2 to receive the confirmation messages from Node 2.

To evaluate throughput and bandwidth, Node 1 publishes messages ranging in size from 10kB to 200kB in 10kB increments, at frequencies from 10Hz to 200Hz in 10Hz steps. The queue size is set to 10,000 and messages are published for 30 seconds during each test run, followed by a 15 second cool down period before the next test. Node 1 logs a timestamp and the message size and frequency for each published message. Concurrently, Node 3 records a timestamp upon receiving the corresponding message on Topic 2. By comparing the total messages received in the 30 second window to the expected number based on the publishing frequency, the throughput percentage and bandwidth can be determined. The bandwidth is calculated by multiplying the throughput percentage by the message size.

3.3.3 ROS2 Intra-Board Communication Benchmarking

a. ROS2 intra-board Latency Benchmark with Fast DDS and Cyclone DDS

This benchmark characterizes the intra-board latency of ROS 2 communications on a Raspberry Pi system using Fast DDS and Cyclone DDS middleware implementations. The experimental configuration consists of a Publisher node

transmitting messages to a Subscriber node executing on the same Pi device. This benchmark characterizes the intra-board latency of ROS 2 communications on a Raspberry Pi system using Fast DDS and Cyclone DDS middleware implementations.

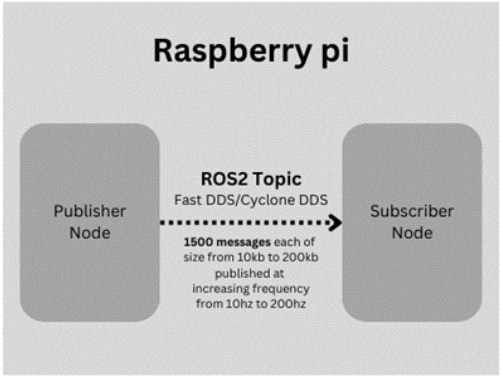

Figure 3-5 ROS2 intra-board Latency Benchmark with Fast DDS and Cyclone DDS

The experimental configuration consists of a Publisher node transmitting messages to a Subscriber node executing on the same Pi device. The Publisher node transmits messages of sizes ranging from 10kb to 200kb in 10kb increments. It also sweeps across frequencies from 10Hz to 200Hz in 10Hz steps. For each test run, 1500 messages are published before enforcing a 30 second cool-down period. The Publisher node logs the message size, publishing frequency, message number, and timestamp at each message publish event. The Subscriber node timestamps the receipt of each message. Latency is calculated by comparing the publish and receipt timestamps for each of the 1500 messages. Median latencies are then computed for each message size and frequency pair. Sweeping across the message sizes and frequencies reveals their impact on median latency for both Fast DDS and Cyclone DDS. Transmitting 1500 messages per test condition provides an accurate median value. This median latency quantifies the typical intra-board communication performance. Thoroughly varying the test parameters profiles how message attributes influence latency when using the two DDS implementations on the Raspberry Pi platform. This benchmark systematically evaluates intra-board ROS 2 communication latency on a Raspberry Pi for a comprehensive range of message sizes and frequencies using Fast DDS and Cyclone DDS.

b. ROS2 Intra-Board Throughput and Bandwidth Benchmark with Fast DDS and Cyclone DDS

This benchmark evaluates the throughput and bandwidth of ROS 2 communications within a Raspberry Pi system using Fast DDS and Cyclone DDS middleware. It utilizes a Publisher node transmitting messages to a Subscriber node on the same Pi. The Publisher transmits varying message sizes from 10kb to 200kb in 10kb increments, across frequencies sweeping 10Hz to 200Hz in 10Hz steps. This allows characterizing performance impact of size and frequency. The Publisher transmits for 30 seconds per test, followed by a 15 second cool-down period.

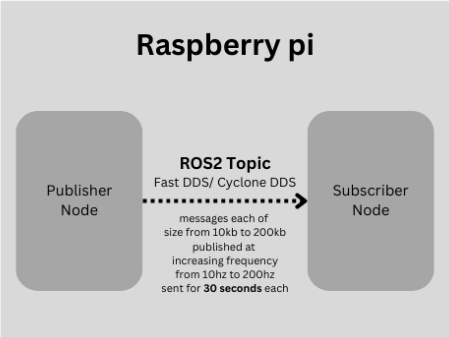

Figure 3-6 ROS2 Intra-Board Throughput and Bandwidth Benchmark with Fast DDS and Cyclone DDS

The Publisher logs message attributes and timestamps on each publish event. The Subscriber timestamps receipt of each message. By comparing received messages in the 30 second window to the expected number based on frequency, throughput percentage is calculated. Bandwidth equals throughput percentage multiplied by message size.

This benchmark analyzes how message size and frequency affect intra-board throughput and bandwidth capacity for ROS 2 using Fast DDS and Cyclone DDS on a Raspberry Pi. It profiles the performance differences between the two DDS implementations.

3.3.4 ROS2 Inter-Board Communication Benchmarking

a. ROS2 Inter-Board Latency Benchmark with Fast DDS and Cyclone DDS

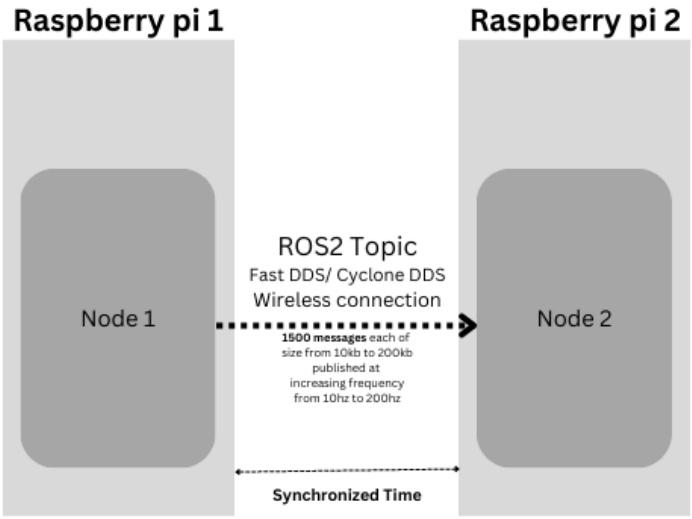

Figure 3-7 ROS2 Inter-Board Latency Benchmark with Fast DDS and Cyclone DDS

This benchmark evaluates the inter-board communication latency of ROS 2 over wireless connections between two Raspberry Pi devices. The experimental setup utilizes wireless ROS 2 communication via Fast DDS or Cyclone DDS between two distinct Raspberry Pis. As time synchronization across multiple devices is

inherent in ROS 2, only two nodes are required - Node 1 on one Pi publishing messages, and Node 2 on the second Pi receiving messages.

Node 1 publishes message payloads ranging from 10kb to 200kb in 10kb increments, with frequencies sweeping from 10Hz to 200Hz in 10Hz steps. For each combination of message size and frequency, Node 1 publishes 1500 messages while logging timestamps, message size, frequency, and message number on each publish event. Node 2 timestamps each message receipt. The publish and receipt timestamps are compared to calculate latency for each of the 1500 messages per test condition. Median latencies are then computed for each message size and frequency pair.

Sweeping across message sizes and frequencies reveals their impact on median latency for both Fast DDS and Cyclone DDS middleware. The transmission of 1500 messages per condition provides an accurate median value. This median latency quantifies the typical inter-board communication performance. Thoroughly varying the test parameters profiles how message attributes influence latency for the two DDS implementations.

b. ROS2 Inter-Board Throughput and Bandwidth Benchmark with Fast DDS and Cyclone DDS

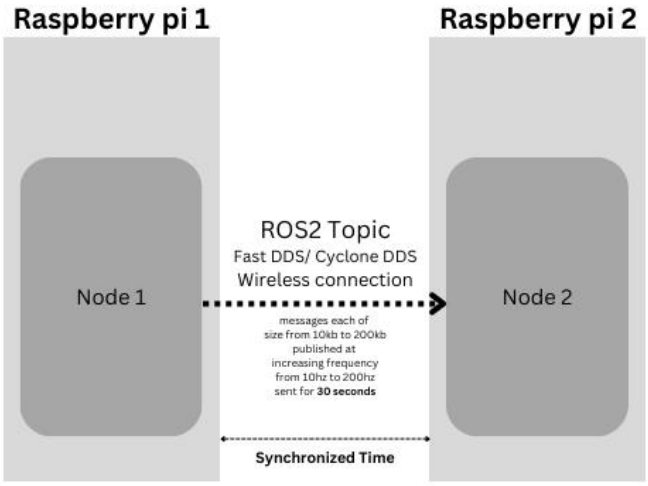

Figure 3-8 ROS2 Inter-Board Throughput and Bandwidth Benchmark with Fast DDS and Cyclone DDS

This benchmark evaluates the throughput and bandwidth capacity of ROS 2 communication over wireless connections between two distinct Raspberry Pi devices. The experimental setup utilizes wireless ROS 2 communication via Fast DDS or Cyclone DDS middleware between the two Pis. Since ROS 2 enables time synchronization across multiple devices, only two nodes are necessary - Node 1 publishing messages on one Pi, and Node 2 receiving messages on the second Pi.

Node 1 publishes message payloads ranging from 10kb to 200kb in 10kb increments, with frequencies sweeping from 10Hz to 200Hz in 10Hz steps. This range of message sizes and frequencies allows characterizing the performance impact of both parameters. For each combination of message size and frequency, Node 1 transmits continuously for 30 seconds, followed by a 15 second cool-down

period before proceeding to the next test condition. This allows the system to reset between variable changes. The publisher Node 1 logs the message size, publishing frequency, and timestamp at each message publish event. The subscriber Node 2 logs the timestamp upon receipt of each message. By comparing the number of messages received within the 30 second transmission window to the number expected based on the set frequency, the throughput percentage is calculated. The bandwidth is then derived by multiplying this throughput percentage by the message size.

This benchmark provides in-depth analysis of how message size and frequency impact the inter-board throughput and bandwidth capacity of ROS 2 communications over wireless links between Raspberry Pis using Fast DDS and Cyclone DDS implementations. The results reveal performance differences between the two DDS middleware options when operated on low-cost hardware over wireless connections. Sweeping across a range of substantive message sizes and frequencies while transmitting for an extended duration elicits the throughput and bandwidth limitations of the system configurations.

Chapter 4 Results and Observations

This chapter presents the findings from systematic benchmarking of ROS and ROS2 communications infrastructure. The aim of the communication benchmarking is to quantify the performance differences between ROS and ROS2 in areas like latency, throughput and bandwidth. Rigorous testing was conducted using standardized procedures to evaluate the communications performance of the two robotic middleware platforms under varied conditions.

As per the previous chapter, the experiment involved a thorough analysis of message sizes ranging from 10 kb to 200 kb, using a systematic 10 kb increment. Similarly, message frequencies are spanning from 10 Hz to 200 Hz, with 10 Hz increments. To enhance clarity and facilitate a more intuitive understanding of the data, it is decided to present results specifically for a 50 kb message size. Though, the patterns and trends observed were consistent across all message sizes, Message size of 50 kb was chosen because it exhibited neat and recognizable variation in performance metrics. Another reason being this choice plays crucial role while comparing ROS and ROS2 communication. To avoid complexity and maintain clarity, results are shown on a specific frequency range within result dataset, specifically, frequencies ranging from 10 Hz to 90 Hz, with 20 Hz increments. This decision was made to emphasize key data points while ensuring that graphical representations remain manageable and informative.

In this chapter, examination the performance of ROS and ROS2 communications for both intra-board and inter-board scenarios is done. It is structured into three main sections and they, along with their sub-sections, are as follows:

1. Benchmark Analysis of ROS and ROS2 communication
    a. Benchmark analysis for ROS's intra-board and inter-board communication
    b. Benchmark analysis for ROS2's intra-board and inter-board communication with Fast DDS
    c. Benchmark analysis for ROS's intra-board and inter-board communication with Cyclone DDS
2. Comparative Benchmark Analysis of ROS2 communication with different DDS implementation
    a. Comparative Benchmark analysis for Intra-board communication
    b. Comparative Benchmark analysis for Inter-board communication
3. Comparative analysis of ROS and ROS2 communication
    a. Comparative Benchmark analysis for Intra-board communication
    b. Comparative Benchmark analysis for Inter-board communication

Classification is done to structure and organize the key insights derived from the experiment effectively. This structured approach allows the performance of default ROS messaging to be established first, followed by evaluation of different ROS2 DDS options. The side-by-side comparisons then clearly demonstrate the relative performance between ROS and ROS2. The results are categorized by communication type for easy digestion and mapping to the research objectives.

It's crucial to mention that the comprehensive dataset, including results for all message sizes and frequencies, can be found in the appendices.

4.1 Benchmark Analysis of ROS and ROS2 Communication

4.1.1 Benchmark analysis for ROS's intra-board and inter-board communication

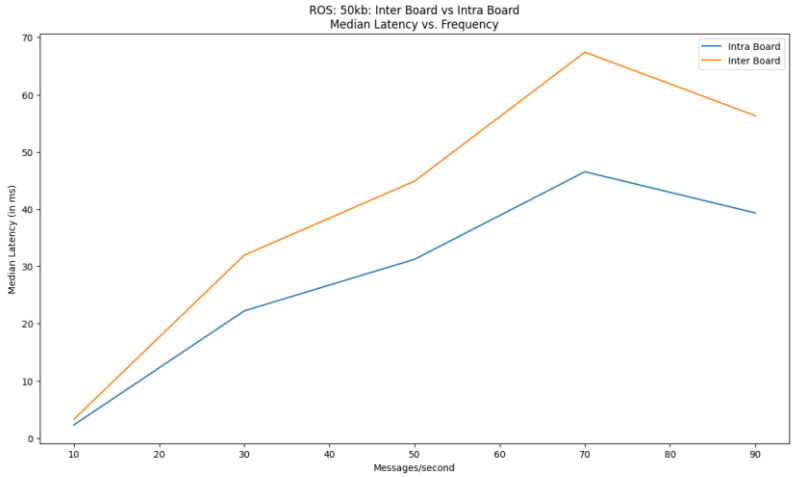

Figure 4-1 Median Latency vs Frequency graph for ROS intra-board and inter-board communication at message size of 50kb

An examination of ROS communication indicates noticeable distinctions between intra-board and inter-board configurations. Specifically, inter-board communication requires 1.4-1.5 times greater duration for wireless message transmission compared to intra-board networks (Figure 4-1). Throughput analysis demonstrates that while intra-board communication sustains 100% throughput up to 30 Hz, inter-board networks exhibit throughput decline beyond 20 Hz. At 90 Hz, intra-board and inter-board communication experienced 55% and 60% reductions respectively. On average, inter-board throughput is 20% lower than intra-board after 20 Hz, with a range of 10-30% (Figure A-1). Similarly, bandwidth comparisons reveal comparable capabilities up to 20 Hz, beyond which intra-board communication consistently showcases superior bandwidth. Significantly, at 90 Hz, the intra-board configuration exhibits a 242.55 kb/sec advantage over the inter-board system (Figure B-1). Taken together, these findings highlight appreciable differences in latency, throughput and bandwidth between intra-board and inter-board ROS communication, with inter-board configurations demonstrating inferior performance across key metrics.

Conclusion:

This Benchmark reveals significant performance gaps between ROS intra-board and inter-board communication in terms of higher latency, lower throughput, and reduced bandwidth for inter-board configurations. Quantitatively, inter-board performs 40-50% worse in latency, 10-30% worse in throughput, and has over 200 kb/sec less bandwidth at high frequencies compared to intra-board.

4.1.2 Benchmark analysis for ROS2's intra-board and inter-board communication with Fast DDS

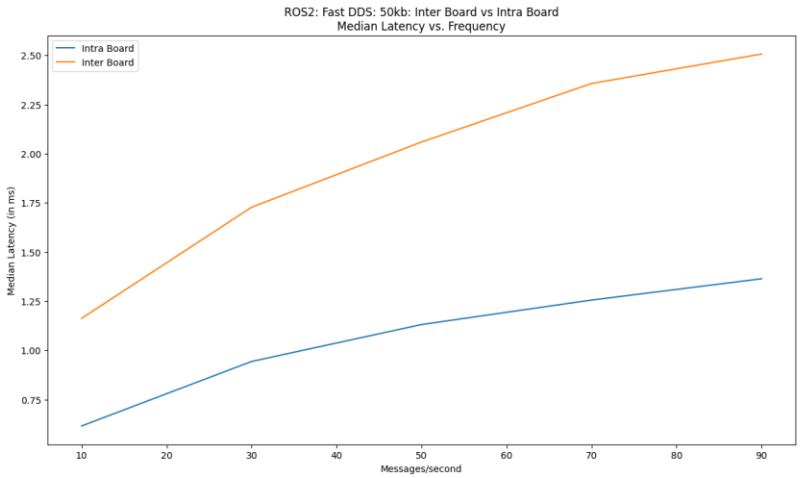

Figure 4-2 Median Latency vs Frequency graph for ROS2 (Fast DDS) intra-board and inter-board communication at message size of 50kb

An examination of ROS2 communication utilizing Fast DDS reveals salient distinctions between intra-board and inter-board configurations. Specifically, inter-board communication requires 1.8-1.9 times greater duration for wireless message transmission compared to intra-board networks, indicating substantially higher latency (Figure 4-2). Unlike ROS, achieving full 100% throughput was unattainable in either intra-board or inter-board ROS2 communication. The maximum throughputs observed were 95% at 10 Hz for intra-board, and 87.65% at 10 Hz for inter-board. From 10 to 90 Hz, both configurations displayed progressive throughput decline, with total reductions of 15.89% and 15.21% for intra-board and inter-board respectively. On average, inter-board throughput was 5-10% lower than intra-board across the frequency range, quantifying the gap between the two configurations (Figure A-2). Similarly, bandwidth comparisons demonstrate consistently lower bandwidth for inter-board communication, which exhibited 5-10% less bandwidth than its intra-board counterpart (Figure B-2). Taken together, these findings showcase appreciable distinctions between intra-board and inter-board ROS2 communication using Fast DDS, with inter-board configurations demonstrating markedly higher latency, moderately lower throughput on average, and consistently reduced bandwidth relative to intra-board networks.

Conclusion:

Compared to intra-board, inter-board ROS2 communication using Fast DDS experiences 80-90% higher latency, up to 7.35% lower peak throughput, a 5-10% average throughput gap, and a 5-10% lower bandwidth. This quantifies the significant performance limitations of inter-board configurations.

4.1.3 Benchmark analysis for ROS's intra-board and inter-board communication with Cyclone DDS

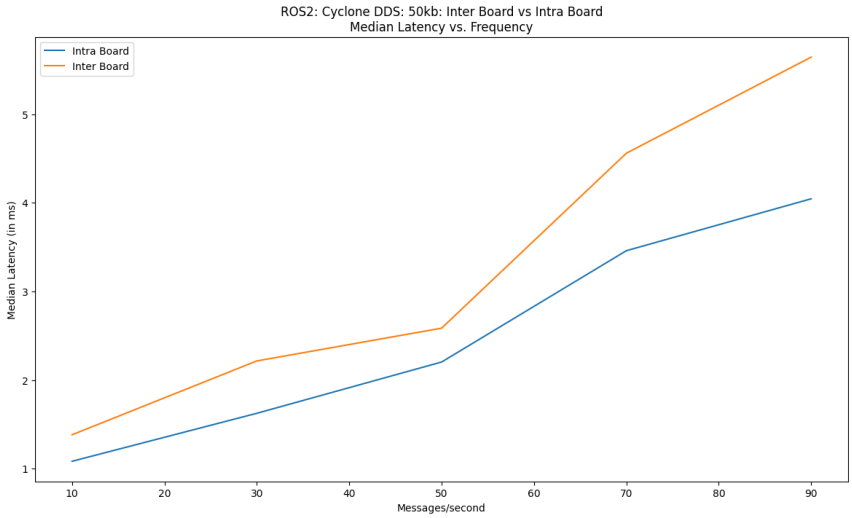

Figure 4-3 Median Latency vs Frequency graph for ROS2 (Cyclone DDS) intra-board and inter-board communication at message size of 50kb

An analysis of ROS2 communication utilizing Cyclone DDS indicates notable performance differences between intra-board and inter-board configurations. Specifically, inter-board communication exhibits 1.2-1.4 times higher latency, taking longer duration for wireless message transmission versus intra-board networks (Figure 4-3). Achieving full 100% throughput was unattainable for both arrangements, with peak throughputs of 93.60% (intra-board) and 86.37% (inter-board) observed at 10 Hz. From 10 to 90 Hz, progressive throughput decline was evident in both configurations, with total reductions of 15.73% and 14.92% for intra-board and inter-board respectively. On average, shifting from intra-board to inter-board incurred a throughput decrease of 5-10%, quantifying the gap (Figure A-3). Similarly, bandwidth comparisons demonstrate consistently lower bandwidth for inter-board communication, which displayed 5-10% less bandwidth than its intra-board counterpart (Figure B-3). Taken together, these findings reveal appreciable distinctions between intra-board and inter-board ROS2 communication leveraging Cyclone DDS. The inter-board configuration exhibits moderately higher latency, marginally lower peak throughput, moderately reduced average throughput, and consistently lower bandwidth relative to the intra-board system. The results showcase the limitations of wireless inter-board communication for ROS2 using Cyclone DDS.

Conclusion:

Compared to intra-board ROS2 communication using Cyclone DDS, inter-board configurations exhibit 20-40% higher latency, up to 7.23% lower peak throughput, a 5-10% lower average throughput, and a 5-10% reduction in bandwidth. This highlights the quantitative performance limitations of inter-board networks.

4.2 Comparative Benchmark Analysis of ROS2 communication with different DDS implementation

4.2.1 Comparative Benchmark analysis for ROS2 Intra-board communication

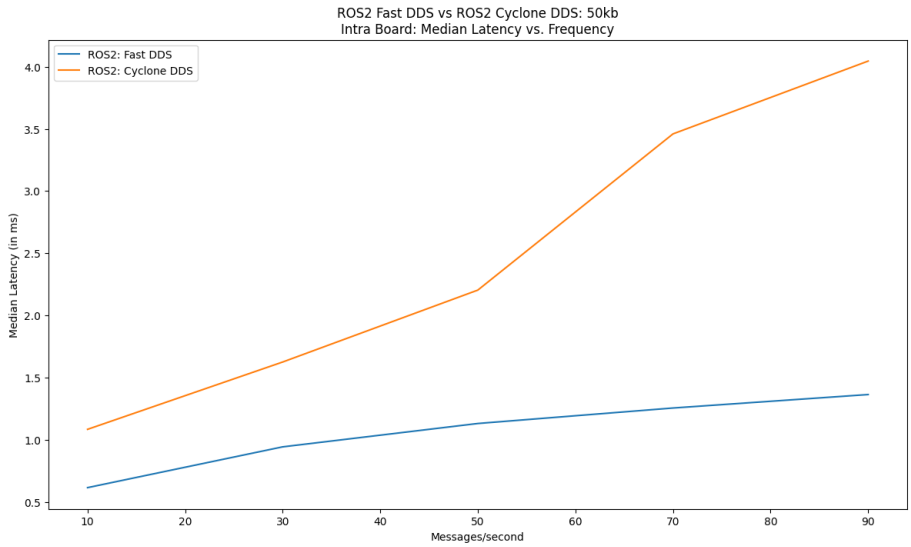

Figure 4-4 Median Latency vs Frequency graph for ROS2 intra-board communication utilizing Fast DDS and Cyclone DDS (message size 50kb)

An examination of ROS2 communication performance using Fast DDS and Cyclone DDS middleware reveals nuanced distinctions between the two implementations. Regarding latency, Fast DDS consistently demonstrated reduced duration for message transmission compared to Cyclone DDS, although no consistent numerical pattern emerged. Notably however, Cyclone DDS required over 1.7 times the latency of Fast DDS, with nearly a threefold increase observed for a 50kb message at 90Hz specifically (Figure 4-4). In terms of throughput, both DDS solutions exhibited broadly similar performance, with Cyclone DDS showing a slight 1-2% reduction versus Fast DDS. For a 50kb message, peak throughputs were 95% and 93.60% for Fast DDS and Cyclone DDS respectively at 10Hz. From 10 to 90Hz, total throughput declines were comparable at 15.89% and 15.73% for each DDS implementation (Figure A-4). Lastly, Cyclone DDS displayed marginally lower bandwidth, approximately 1-2% less than Fast DDS, indicating another area of similarity between the two middleware solutions (Figure B-5). In summary, while nuanced distinctions exist, particularly regarding latency, the performance analysis highlights broad similarities between Fast DDS and Cyclone DDS in managing ROS2 communications, with only minor differences in throughput and bandwidth evident between the two implementations.

Conclusion:

Cyclone DDS exhibited at least 70% higher latency compared to Fast DDS, with over 200% increase at high frequencies. Meanwhile, throughput and bandwidth were broadly similar, with Cyclone showing only 1-2% lower performance than Fast DDS. This highlights the major latency differences but otherwise comparable capabilities between the two ROS2 middleware implementations.

4.2.2 Comparative Benchmark analysis for ROS2 Inter-board communication

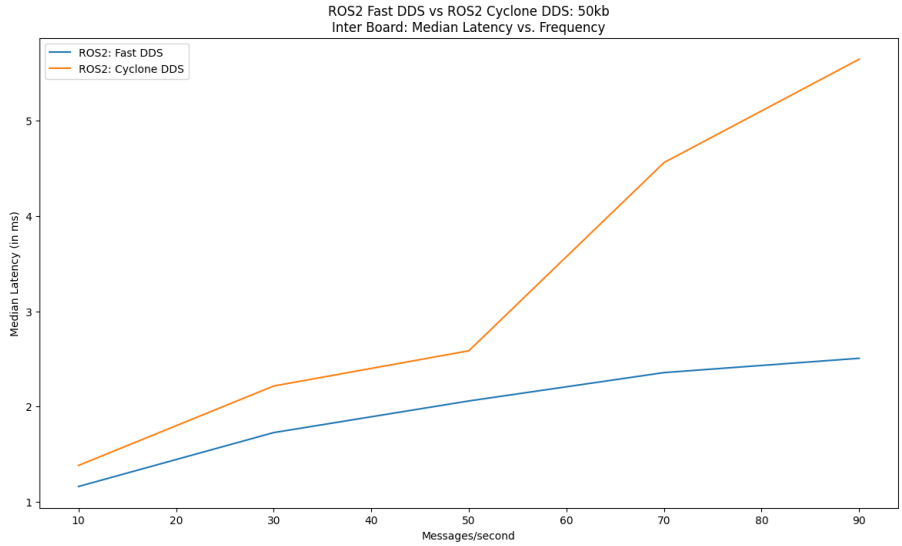

Figure 4-5 Median Latency vs Frequency graph for ROS2 inter-board communication utilizing Fast DDS and Cyclone DDS (message size 50kb)

An analysis of inter-board ROS2 communication performance using Fast DDS and Cyclone DDS middleware reveals nuanced differences between the two implementations. Regarding latency, Fast DDS consistently demonstrated quicker message transmission than Cyclone DDS, taking 1.15 times less duration versus the 1.7 times difference observed in intra-board configurations. While no consistent numerical pattern emerged, transmission time incrementally increased with higher frequencies overall (Figure 4-5). In terms of throughput, both DDS solutions again showcased broadly similar performance, with Cyclone DDS exhibiting a minor 1-2% reduction compared to Fast DDS. For a 50kb message, maximum throughputs were 87.65% and 86.37% for Fast DDS and Cyclone DDS respectively at 10Hz. From 10 to 90Hz, total throughput declines were comparable at 15.21% for Fast DDS and 14.92% for Cyclone DDS (Figure A-5). Finally, Cyclone DDS displayed slightly lower bandwidth, around 1-2% less than Fast DDS, highlighting another area of similarity (Figure B-5). In summary, while moderate latency differences exist favoring Fast DDS, the analysis indicates broad parallels between Fast DDS and Cyclone DDS in facilitating inter-board ROS2 communication, with only negligible distinctions in throughput and bandwidth between the two middleware platforms. However, the latency gap appears less pronounced compared to intra-board configurations.

Conclusion:

The latency gap between Cyclone DDS and Fast DDS is 55% smaller for inter-board versus intra-board networks. Meanwhile, throughput and bandwidth differences remain negligible at around 1-2% lower for Cyclone DDS. This quantifies the convergence in performance between the two middlewares for inter-board communication.

4.3 Comparative Benchmark analysis of ROS and ROS2 communication

4.3.1 Comparative Benchmark analysis for Intra-board communication

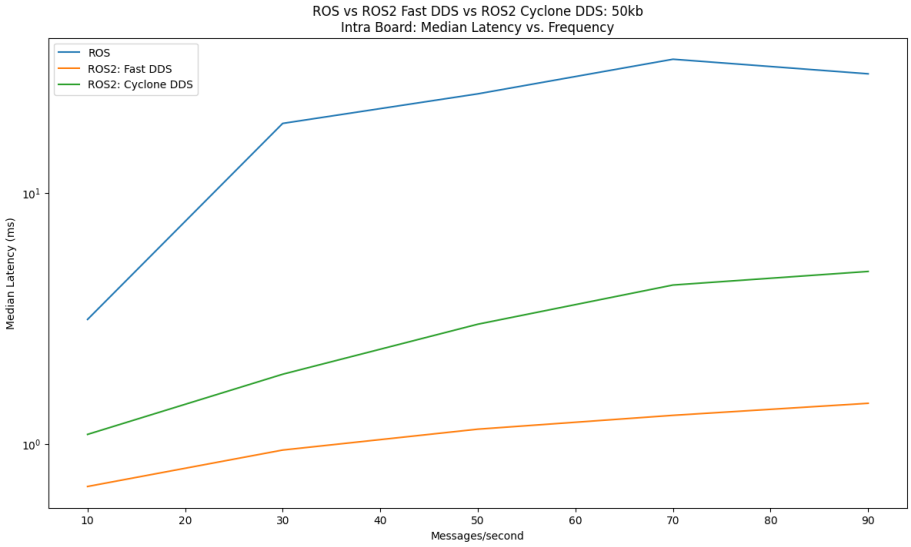

Figure 4-6 Median Latency vs Frequency graph for intra-board ROS and ROS2 (Fast DDS and Cyclone DDS) communication (message size 50kb)

Regarding latency, it is decided to represent the data using a logarithmic y-axis scale intentionally. This decision was made due to the fact that, with a 50 kb message size, as the frequency approaches 90 Hz, the latency in intra-board ROS2 employing Fast DDS becomes nearly 28.83 times greater compared to intra-board ROS communication. This observation emphasizes that when it comes to intra-board communication, and when considering different DDS implementations, ROS consistently lags significantly behind ROS2 in terms of latency. Comparing intra-board communication within ROS and ROS2 using Cyclone DDS, a distinct numerical pattern doesn't emerge. Nevertheless, one observation is that at 10 Hz, ROS takes 114% more time than ROS2 with Cyclone DDS, while at 90 Hz, ROS takes 872% more time than ROS2 with Cyclone DDS, reflecting an approximately tenfold speed advantage for ROS2. The most substantial latency difference is evident at 50 Hz, where ROS2 with Cyclone DDS operates 14.17 times faster than ROS. In the comparison of intra-board ROS communication with intra-board ROS2 communication employing Fast DDS, there's no consistent numerical pattern. However, a common observation is that at 10 Hz, ROS takes 278% more time than ROS2 with Fast DDS, whereas at 90 Hz, ROS takes 2783% more time than ROS2 with Fast DDS, indicating a nearly 28-fold slower performance for ROS. The most significant latency variation is observed at 70 Hz, where ROS2 with Fast DDS operates 37 times faster than ROS (Figure 4.6).

An interesting pattern came to light in the observations regarding throughput. In case of intra-board communication, ROS consistently maintained a perfect 100% throughput until reaching the 30 Hz mark, whereas ROS2, whether using Cyclone DDS or Fast DDS, showed slightly lower values at 93.60% and 95%,

respectively. Beyond the 30 Hz threshold, there was a sudden decline in throughput for ROS, whereas ROS2 experienced a gradual decrease with less magnitude. At the 90 Hz point, intra-board ROS communication achieved a throughput of 44.67%, whereas intra-board ROS2 communication reached figures of 77.87% and 79.11% for Cyclone DDS and Fast DDS, respectively. Over the range of 10 Hz to 90 Hz for intra-board communication, ROS exhibited a net throughput decrease of 55.33%, whereas ROS2 with Cyclone DDS showed a decrease of 15.73%, and Fast DDS showed a decrease of 15.89%. These observations lead to the conclusion that ROS provides better throughput at lower frequencies, while ROS2 excels at higher frequencies (Figure A-6). A similar trend was observed in terms of bandwidth. Up to the 30 Hz point, ROS demonstrated superior bandwidth compared to ROS2 utilizing various DDS solutions (Figure B-6). However, beyond this 30 Hz threshold, ROS2 began to outperform ROS in terms of bandwidth performance. The same conclusion holds for bandwidth as for throughput, indicating that for intra-board communication at lower frequencies, ROS proves to be the superior choice for achieving enhanced bandwidth. However, as frequencies increase, ROS2 takes the lead.

Conclusion:

At all frequencies ROS shows higher latency compared to ROS2 irrespective of DDS. ROS shows an advantage in terms of Throughput and Bandwidth at low frequencies. But as frequency increases (beyond 30 Hz), ROS2 irrespective of DDS substantially outperforms ROS in latency, throughput and bandwidth for intra-board communication.

4.3.2 Comparative Benchmark analysis for Inter-board communication

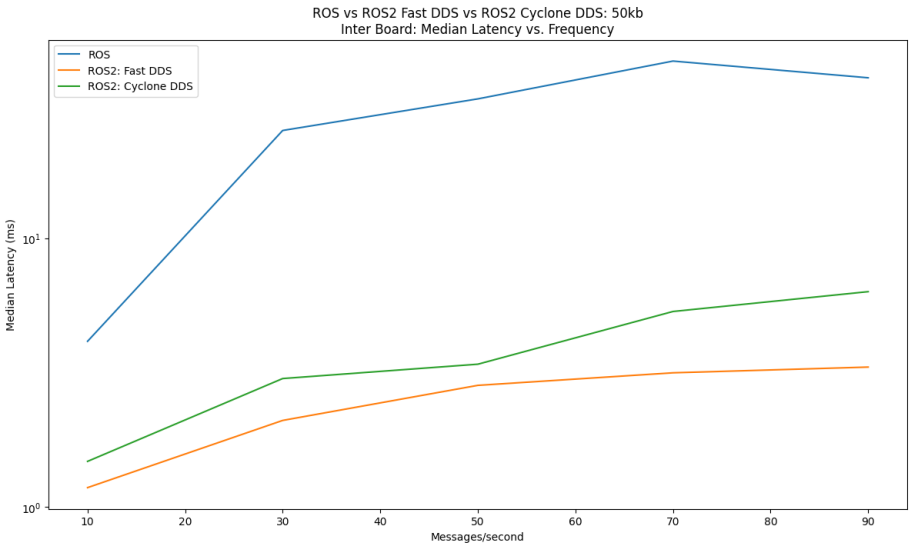

Figure 4-7 Median Latency vs Frequency graph for inter-board ROS and ROS2 (Fast DDS and Cyclone DDS) communication (message size 50kb)

In terms of latency, graphs were generated with an exponential y-axis scale. This choice was made because, for a 50 kb message size, as the frequency approaches 90 Hz, the latency in inter-board ROS2 with Fast DDS becomes nearly 20 times greater compared to inter-board ROS communication. Concerning latency, it's evident that when it comes to inter-board communication, ROS lags significantly behind ROS2 when various DDS implementations are considered. On comparing inter-board communication in ROS to inter-board communication in ROS2 using Cyclone DDS, no distinct numerical pattern emerges. However, a consistent observation is that at 10 Hz, ROS takes 140% more time than ROS2 with Cyclone DDS, while at 90 Hz, ROS takes 898% more time than ROS2 with Cyclone DDS, which is approximately 10 times faster than ROS. The most substantial latency difference is observed at 50 Hz, where ROS2 with Cyclone DDS operates 17.35 times faster than ROS. In the comparison of inter-board ROS communication to inter-board ROS2 communication using Fast DDS, no consistent numerical pattern is observed. A common observation is that at 10 Hz, ROS takes 180% more time than ROS2 with Fast DDS, whereas at 90 Hz, ROS takes 2146% more time than ROS2 with Fast DDS, which is roughly 22 times faster than ROS. The most significant latency variation is noted at 70 Hz, where ROS2 with Fast DDS operates 28.6 times faster than ROS (Figure 4-7).

An intriguing pattern emerged in the observations of throughput. In the context of inter-board communication, ROS consistently maintained a full 100% throughput up to the 20 Hz mark, while ROS2, whether employing Cyclone DDS or Fast DDS, exhibited slightly lower figures at 86.37% and 87.65%, respectively. Beyond the 20 Hz threshold, there was a sudden drop in throughput for ROS, whereas ROS2 experienced a gradual and less pronounced decline. At the 90 Hz point, inter-board ROS communication displayed a throughput of 39.28%, whereas for inter-board ROS2 communication, figures stood at 71.45% and 72.44% for Cyclone DDS and Fast DDS, respectively. For inter-board communication from 10 Hz to 90 Hz, net throughput drop for ROS was 60.28%. For ROS2 Cyclone DDS it was 14.92% and for Fast DDS it was 15.21%. Based on these observations, it can be concluded that for lower frequencies, ROS offers better throughput, while for higher frequencies, ROS2 emerges as the superior choice (Figure A-7). A similar trend was noted in terms of bandwidth. Up to the 20 Hz point, ROS exhibited superior bandwidth when compared to ROS2 with various DDS solutions. However, beyond this 20 Hz threshold, ROS2 began to surpass ROS in terms of bandwidth performance (Figure B-7). The same conclusion can be drawn for bandwidth as for throughput, highlighting that at lower frequencies, ROS proves to be the better option for achieving enhanced bandwidth. However, as frequencies increase, ROS2 takes the lead.

Conclusion:

For inter-board communication below 20 Hz, ROS provides maximum throughput and bandwidth; however, at higher frequencies, ROS2 is preferable for its substantially lower latency and more gradual declines in throughput and bandwidth.

Chapter 5 Conclusion

5.1 Summary

It is evident that there are notable differences in communication performance between intra-board and inter-board communication within the context of ROS and ROS2, with varying implementations such as Fast DDS and Cyclone DDS. Specifically, inter-board communication consistently requires more time than intra-board communication across these systems. When examining the specific ratios, ROS inter-board communication took 1.8-1.9 times longer than intra-board communication. For ROS2 with Fast DDS, this ratio increased to 1.85 times, whereas with Cyclone DDS, it was 1.2-1.4 times. Intra-board communication in ROS maintained 100% throughput up to 30 Hz, whereas inter-board communication plateaued at 20 Hz. Beyond these thresholds, ROS experienced a throughput drop of 10-30% when transitioning from intra-board to inter-board communication. On the other hand, ROS2, regardless of DDS, failed to achieve 100% throughput in both scenarios, and the overall throughput reduction when shifting from intra-board to inter-board communication ranged from 5-10% for both DDS implementations. The bandwidth exhibited a similar trend as throughput. In ROS, the comparison between intra-board and inter-board communication revealed a bandwidth reduction ranging from 10-30%. Conversely, in ROS2, regardless of DDS, inter-board communication showed 5-10% less bandwidth compared to intra-board communication.

When comparing Fast DDS and Cyclone DDS for ROS2, latency was a key differentiator, with Fast DDS exhibiting lower latency. However, throughput and bandwidth were similar between the two, with Cyclone trailing by only 1-2%. For intra-board communication, Cyclone required over 1.7 times the duration of Fast DDS, and for inter-board it was over 1.15 times longer.

Now, turning our attention to the primary benchmark results between ROS and ROS2, focusing first on intra-board communication. It is evident that ROS2 outperforms ROS in terms of latency. For a 50kb message size at 10 Hz, ROS displayed latency more than three times that of ROS2 using Fast DDS and twice as much as Cyclone DDS. As the frequency increased to 90 Hz, ROS's latency was 28.83 times that of Fast DDS and 9.72 times that of Cyclone DDS. Therefore, when considering latency, ROS2 is the preferred choice. In terms of throughput and bandwidth, a unique pattern emerged. ROS maintained 100% throughput until 30 Hz, while ROS2, regardless of DDS, exhibited slightly lower values. Beyond the 30 Hz threshold, ROS experienced a sudden decline in throughput, whereas ROS2 experienced a gradual decrease with less magnitude. A similar trend was observed in bandwidth. ROS had higher bandwidth than ROS2 with different DDS implementations up to 30 Hz, after which ROS experienced a sudden decline. In contrast, ROS2, irrespective of DDS, maintained higher bandwidth. This trend was consistent across various message sizes. From these observations, it can be concluded that, in terms of throughput and bandwidth, ROS takes the lead at low frequencies, while ROS2 emerges as the winner at high frequencies.

In the context of inter-board communication, it is noteworthy that ROS2 outperforms ROS in terms of latency. For instance, when considering a 50 kb message size at a 10 Hz frequency, ROS exhibits latency that is nearly three times greater than that of ROS2 when employing Fast DDS. Furthermore, it surpasses ROS2 by more than twice the latency when employing Cyclone DDS. As the communication frequency is increased to 90 Hz, ROS's latency becomes notably elevated, reaching 22.46 times that of Fast DDS and 9.98 times that of Cyclone DDS. Consequently, when evaluating latency, ROS2 emerges as the preferable option. Turning our attention to throughput and bandwidth, a distinct pattern reoccurs. ROS maintains a continuous 100% throughput rate until reaching a 20 Hz frequency threshold, whereas ROS2, irrespective of the DDS, exhibits slightly lower throughput values. However, beyond the 20 Hz threshold, ROS experiences a sudden and marked reduction in throughput, while ROS2 experiences a gradual decline with less severity. A similar trend is observed when analyzing bandwidth. ROS outperforms ROS2 in bandwidth, regardless of the DDS used, up to a 20 Hz frequency. Subsequently, ROS experiences an abrupt decrease in bandwidth, whereas ROS2 consistently maintains higher bandwidth levels. This consistent trend holds true across various message sizes. To sum up, with respect to throughput and bandwidth in the context of inter-board communication, ROS takes the lead at lower frequencies, while ROS2 emerges as the superior choice at higher frequencies.

In summary, ROS2 outperforms ROS in terms of lower latency and more graceful degradation of throughput and bandwidth at higher frequencies for both intra-board and inter-board communication. This is likely due to optimizations in the ROS2 architecture and middleware. However, ROS maintains an advantage in throughput and bandwidth at lower frequencies. Therefore, the optimal choice depends on the specific communication requirements. If low latency or high frequency communication is critical, ROS2 is preferable. But if maximizing throughput and bandwidth at lower frequencies is the priority, ROS remains a strong contender.

5.2 Limitation: Hardware

The benchmarks utilized a Raspberry Pi model 3B+ which contains 1GB of RAM. In general, running Ubuntu 20.04 Focal Fossa, a relatively heavyweight operating system, is not recommended on this model, as it requires a minimum of 4GB of RAM for proper performance. However, limitations with ROS2 constrained the operating system selection, as ROS2 only officially supports Tier 1 operating systems such as Ubuntu. Although, Raspberry Pi OS (64 bit) could potentially be used with ROS2 via Docker containers, this lacks GUI support which complicates coding efforts. Consequently, Ubuntu 20.04 was necessitated despite being suboptimal for the 3B+ hardware. The use of this high-tier OS on the RAM-constrained Raspberry Pi leads to diminished performance. While ROS can run on low-resource hardware like 1GB RAM Raspberry Pi's, ROS2 2 needs more memory. Experts recommend utilizing a Raspberry Pi 4 with at least 4GB of RAM for ROS2 applications, with 2GB RAM considered a minimum for basic ROS2 operation. However, as all benchmarks were executed on the 3B+ model, it is plausible that the overall decreased performance in ROS2 compared to ROS, particularly the reduced throughput and bandwidth relative to ROS up to 30Hz intra-board and 20Hz inter-board, stems partially from hardware constraints. Specifically, the heavy memory demands of Ubuntu 20.04 may have contributed

to throttled capabilities on the 3B+ that were exposed in the ROS2 results but not in ROS.

5.3 Future Scope

5.3.1 Evaluating ROS and ROS2 Benchmarks on High-Efficiency Hardware

If the benchmarks were re-run on more capable Raspberry Pi 4 hardware with sufficient RAM to handle Ubuntu 20.04, ROS2 may potentially demonstrate higher throughput and bandwidth on par with ROS at lower frequencies. Overall, while some performance differentials can be attributed to contrasting software architectures, the choice to use the minimally outfitted 3B+ model for all benchmarking likely exacerbated performance gaps and prevented ROS2 from fully showcasing its strengths, especially for metrics like throughput and bandwidth that are heavily reliant on available computing resources. Upgrading to more robust Raspberry Pi versions could allow for a more equitable comparison between ROS and ROS2 in future analyses.

5.3.2 Benchmarking real-world performance

The current analysis relies on synthetic microbenchmarks. An extension could involve testing ROS and ROS2 performance in real-world robotics applications. This provides additional practical insights beyond artificial tests. Real-world metrics may differ due to effects like sensor data streams, actuator control loops, motion planning, and simulation. Benchmarking with representative workloads can improve understanding of how performance translates to end-user robotics systems.

5.3.3 Benchmarking CPU and Memory tradeoffs between ROS and ROS2

The current analysis focuses solely on latency, throughput, and bandwidth metrics. An extension could involve running the same benchmarks while also monitoring CPU and memory usage. This would provide insights into the computational resource utilization tradeoffs between ROS and ROS2 implementations. Factors like memory leaks or inefficient CPU usage may be uncovered under load, which could impact suitability for long-running robotics systems. Overall, profiling CPU and memory complements network performance data to provide a more complete picture of real-world system overhead.

5.3.4 Benchmarking impact of security mechanism on ROS2

ROS2 offers security mechanisms like encryption and authentication that can impact runtime performance. An analysis extension could benchmark ROS2 with these features enabled versus disabled. This would reveal the performance tradeoffs when using security features critical for sensitive robotics applications. The overhead of encryption versus the importance of security must be weighed based on use case sensitivity. Formal benchmarking can guide developers in configuring appropriately secure ROS2 systems without sacrificing needed performance.

Appendix A Throughput Results

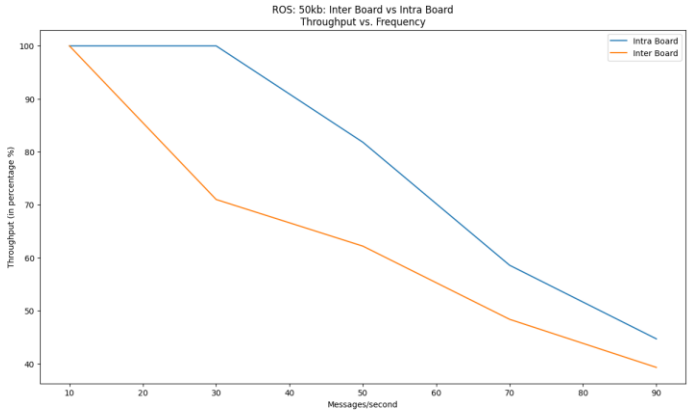

Figure A-1 Throughput vs Frequency graph for ROS intra-board and inter-board communication at message size of 50kb

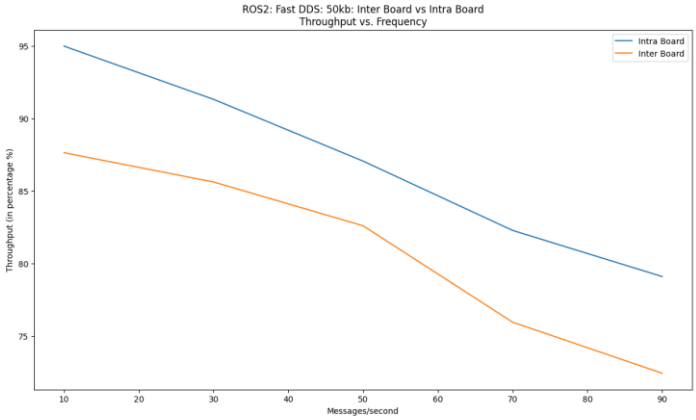

Figure A-2 Throughput vs Frequency graph for ROS2 (Fast DDS) intra-board and inter-board communication at message size of 50kb

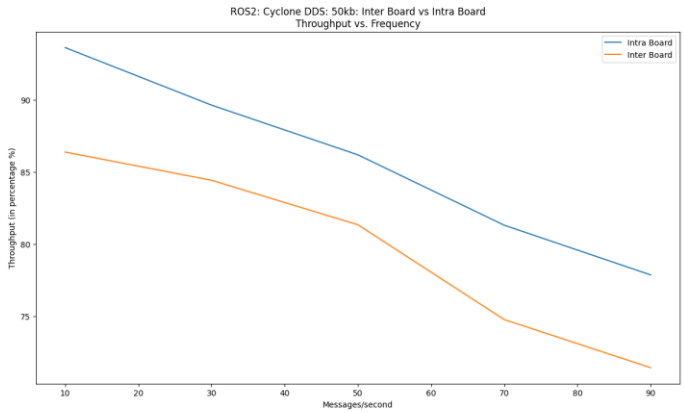

Figure A-3 Throughput vs Frequency graph for ROS2 (Cyclone DDS) intra-board and inter-board communication at message size of 50kb

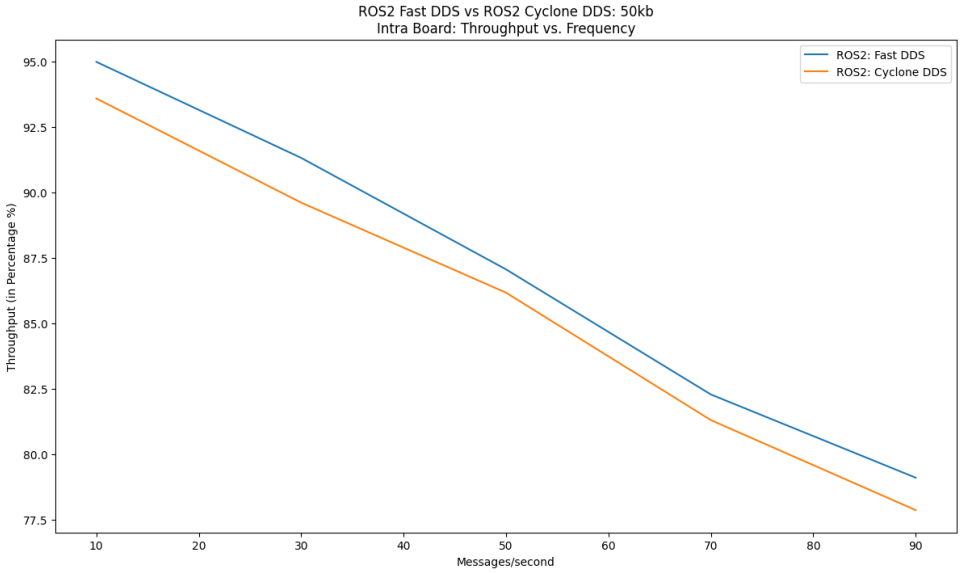

Figure A-4 Throughout vs Frequency graph for ROS2 intra-board communication utilizing Fast DDS and Cyclone DDS (message size 50kb)

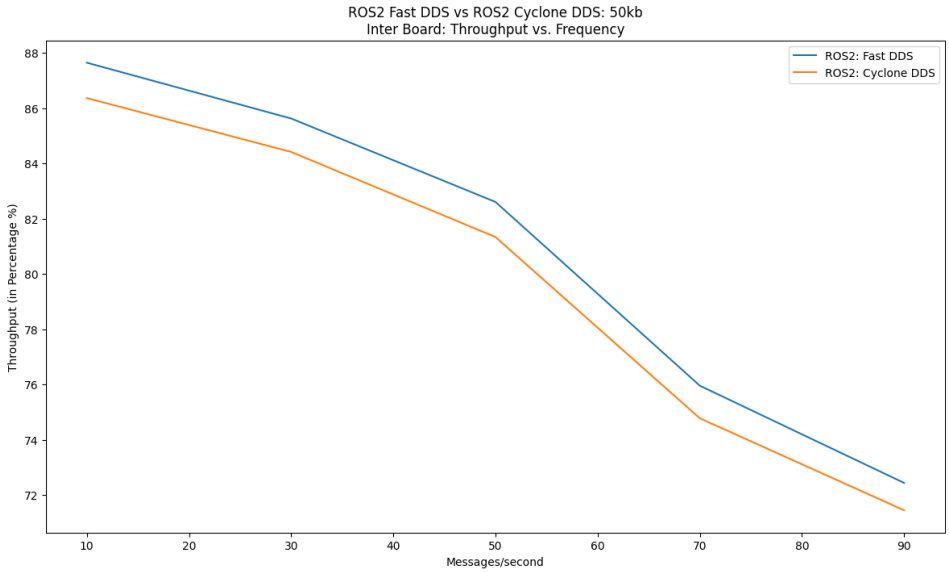

Figure A-5 Throughput vs Frequency graph for ROS2 inter-board communication utilizing Fast DDS and Cyclone DDS (message size 50kb)

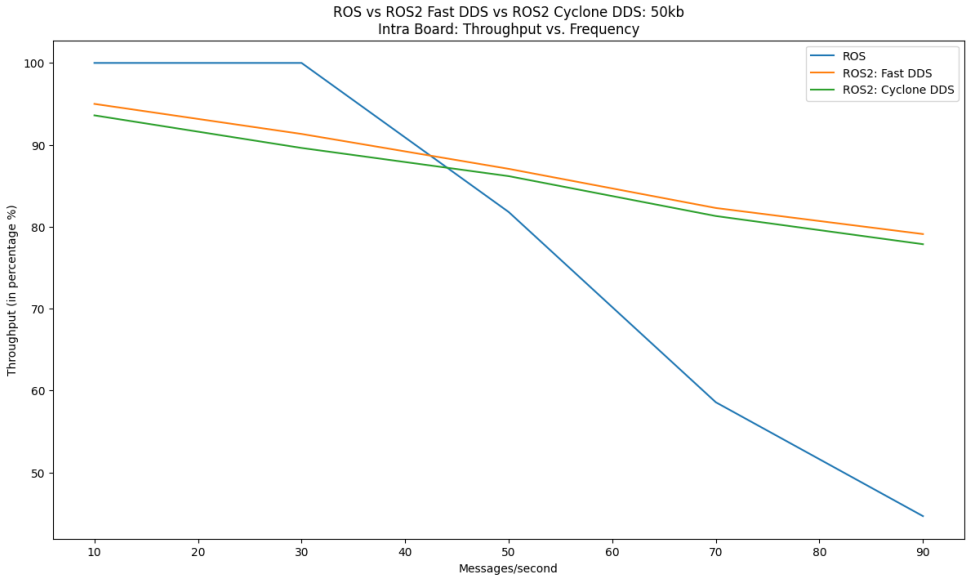

Figure A-6 Throughput vs Frequency graph for intra-board ROS and ROS2 (Fast DDS and Cyclone DDS) communication (message size 50kb)

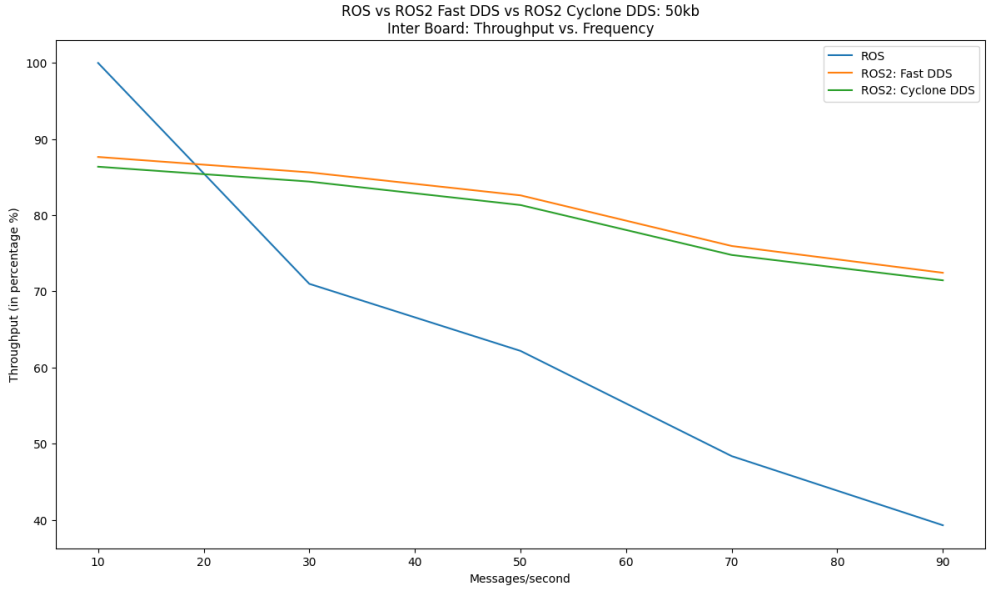

Figure A-7 Throughput vs Frequency graph for inter-board ROS and ROS2 (Fast DDS and Cyclone DDS) communication (message size 50kb)

Appendix B Bandwidth Results

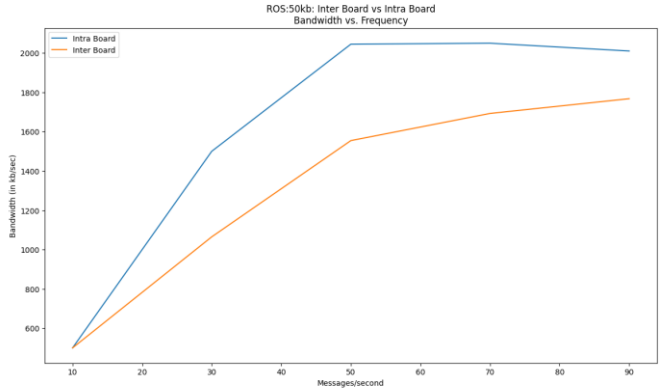

Figure B-1 Bandwidth vs Frequency graph for ROS intra-board and inter-board communication at message size of 50kb

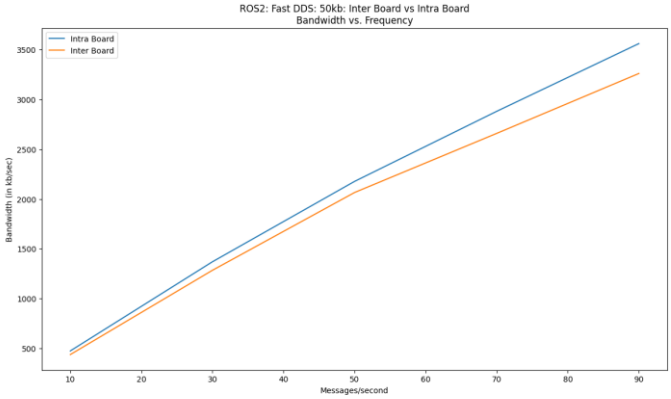

Figure B-2 Bandwidth vs Frequency graph for ROS2 (Fast DDS) intra-board and inter-board communication at message size of 50kb

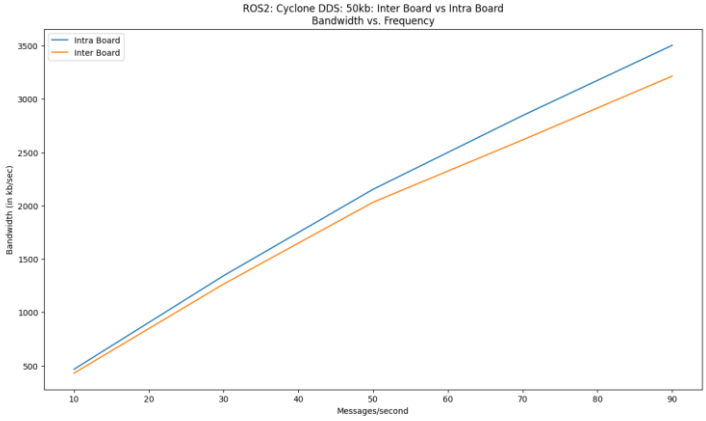

Figure B-3Bandwidth vs Frequency graph for ROS2 (Cyclone DDS) intra-board and inter-board communication at message size of 50kb

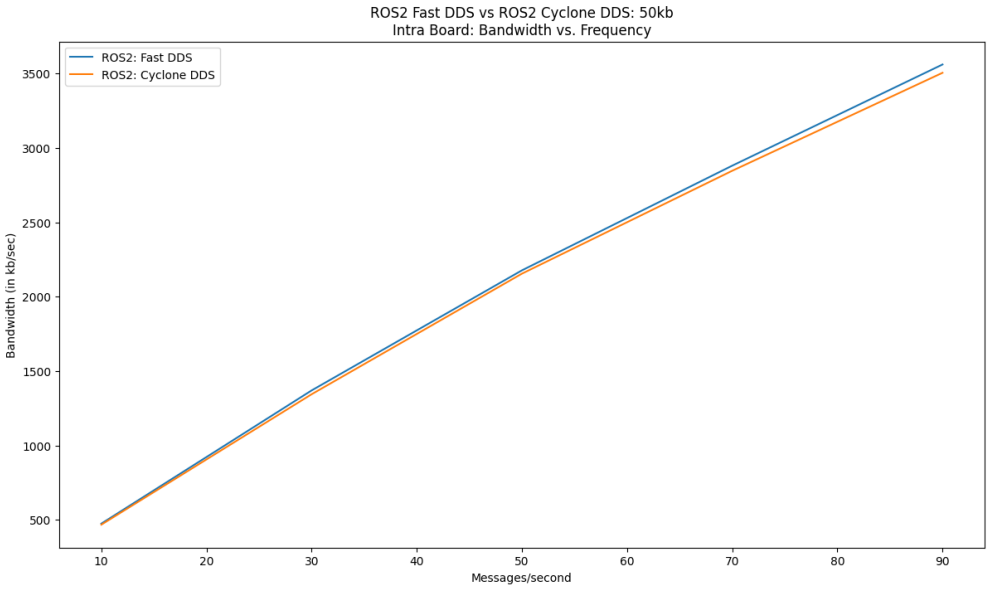

Figure B-4 Bandwidth vs Frequency graph for ROS2 intra-board communication utilizing Fast DDS and Cyclone DDS (message size 50kb)

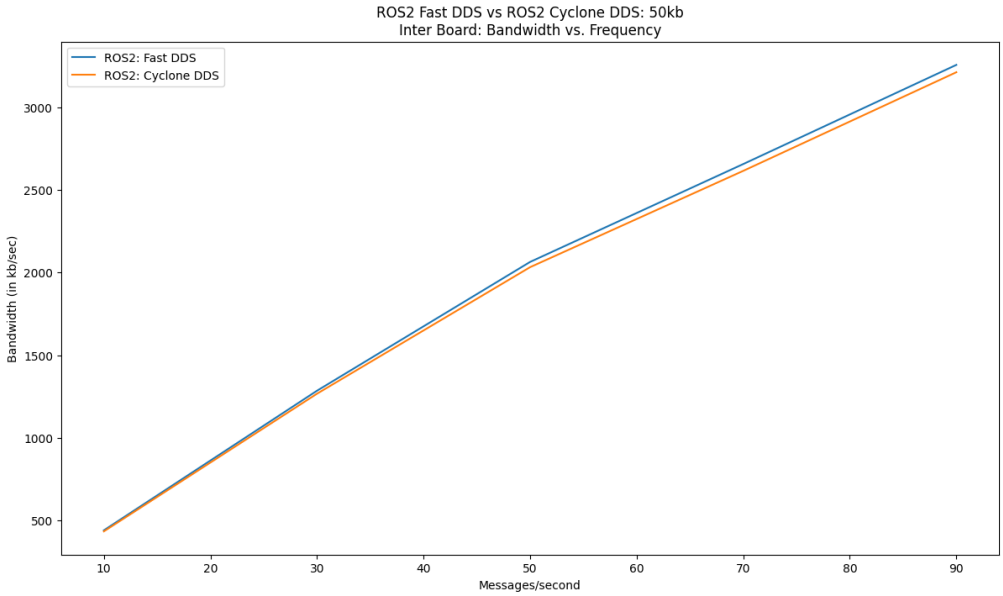

Figure B-5 Bandwidth vs Frequency graph for ROS2 inter-board communication utilizing Fast DDS and Cyclone DDS (message size 50kb)

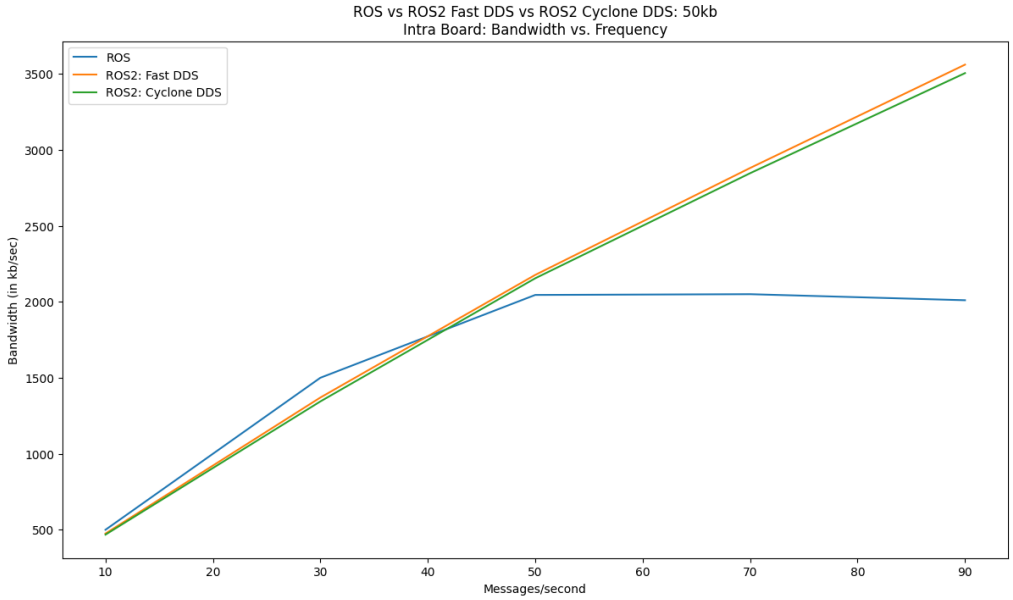

Figure B-6 Bandwidth vs Frequency graph for intra-board ROS and ROS2 (Fast DDS and Cyclone DDS) communication (message size 50kb)

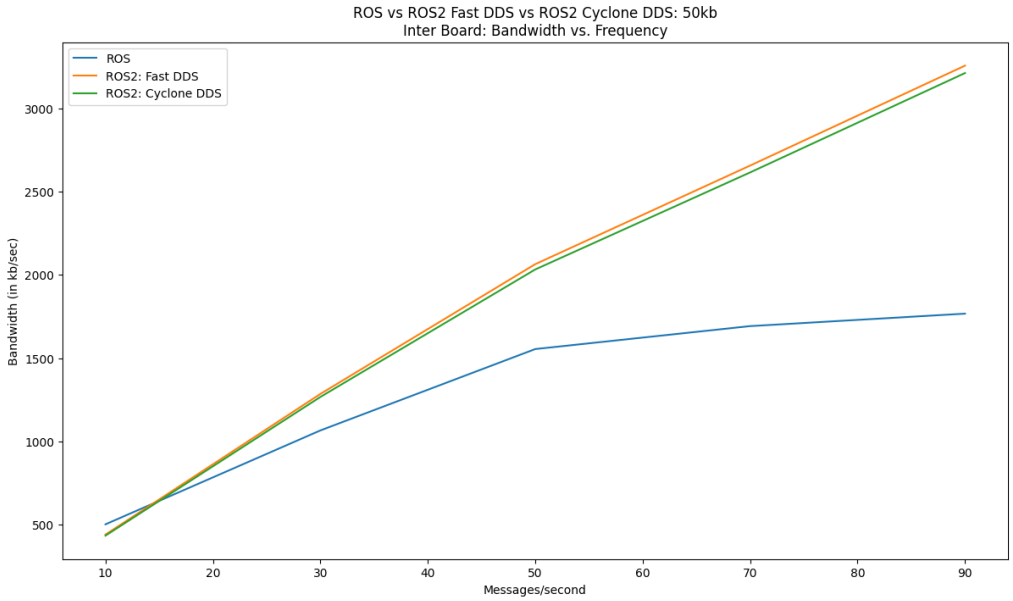

Figure B-7 Bandwidth vs Frequency graph for inter-board ROS and ROS2 (Fast DDS and Cyclone DDS) communication (message size 50kb)

Appendix C Code and Benchmarking Results

The GitHub repository referenced contains code implemented in both the ROS and ROS 2 frameworks to facilitate communication within a computer as well as between two boards. To analyze the performance of various message sizes, the codebase includes graphs benchmarking round trip latency for messages of 10kB, 50kB, 100kB, 150kB and 200kB.

https://github.com/hrushisanap/ROS_ROS2_benchmarking

Appendix D List of Figures and Tables

List of Tables

List of Figures

Bibliography

1. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. and Ng, A.Y. (2009) 'ROS: an open-source Robot Operating System', ICRA workshop on open source software, Vol. 3, No. 3.2, p. 5.
2. Macenski, S., Foote, T., Gerkey, B., Lalancette, C. and Woodall, W. (2022) 'Robot Operating System 2: Design, Architecture, and Uses In The Wild', arXiv preprint arXiv:2211.07752.
3. Macenski, S., Soragna, A., Carroll, M. and Ge, Z. (2023) 'Impact of ROS2 Node Composition in Robotic Systems', IEEE Robotics and Automation Letters, 8(7), pp. 3996-4003.
4. N. Mohamed, J. Al-Jaroodi and I. Jawhar, "Middleware for Robotics: A Survey," 2008 IEEE Conference on Robotics, Automation and Mechatronics, Chengdu, China, 2008, pp. 736-742, doi: 10.1109/RAMECH.2008.4681485.
5. Li, Y., Elkady, A. and Sobh, T. (2012) 'Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography', Journal of Robotics, 2012, pp. 959013. doi: 10.1155/2012/959013.
6. Kronauer, T., Pohlmann, J., Matthé, M., Smejkal, T. and Fettweis, G. (2021) 'Latency Analysis of ROS2 Multi-Node Systems', 2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI), Karlsruhe, Germany, pp. 1-7. doi: 10.1109/MFI52462.2021.9591166.
7. ASQ (American Society for Quality), 2023. What is Benchmarking? Technical & Competitive Benchmarking Process. [online] Asq.org. Available at: https://asq.org/quality-resources/benchmarking .
8. KnowledgeHut, 2023. Benchmarking Process - What is Benchmarking in Project Management. [online] KnowledgeHut. Available at: https://www.knowledgehut.com/tutorials/project-management/benchmarking-process-
9. Gurjot, 2021. Advantages and disadvantages of using ROS. ROS Answers. Available at: https://answers.ros.org/question/200423/advantages-and-disadvantages-of-using-ros/
10. Open Robotics, 2022. Why ROS 2. ROS 2 Design Articles. Available at: https://design.ros2.org/articles/why_ros2.html
11. Electronic Design, 2022. ROS 2 Explained: Overview and Features. Electronic Design. Available at: https://www.electronicdesign.com/markets/automation/article/21214053/electronic-design-ros-2-explained-overview-and-features
12. HyperSpec.AI, 2021. ROS 1 vs ROS 2 Tradeoffs and Advantages. HyperSpec.AI. Available at: https://hyperspec.ai/ros-1-vs-ros-2-tradeoffs-and-advantages/
13. Robotics Backend, 2022. ROS1 vs ROS2 Practical Overview. Robotics Backend. Available at: https://roboticsbackend.com/ros1-vs-ros2-practical-overview/
14. Raspberry Pi Foundation (2018) Raspberry Pi 3 Model B+. [online] Available at: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/
15. Ubuntu Documentation. (2020). Ubuntu 20.04 LTS (Focal Fossa) Documentation. https://releases.ubuntu.com/20.04/

16. Robot Operating System. (2020). ROS Noetic Ninjemys. http://wiki.ros.org/noetic
17. Robot Operating System. (2020). ROS Foxy Fitzroy. http://wiki.ros.org/foxy
18. Open Robotics. (n.d.). ROS: Robot Operating System. http://www.ros.org/
19. ROS 2 Documentation. (n.d.). ROS 2 Documentation. https://docs.ros.org/en/foxy/index.html
20. eProsima. (n.d.). Fast DDS. https://fast-dds.docs.eprosima.com/
21. Eclipse Foundation. (n.d.). Eclipse Cyclone DDS. https://www.eclipse.org/cyclonedds/
22. Orocos. (n.d.). Orocos Real-Time Toolkit. https://www.orocos.org/
23. ROS Wiki. (n.d.). TCPROS. http://wiki.ros.org/ROS/TCPROS
24. ROS Wiki. (n.d.). UDPROS. http://wiki.ros.org/ROS/UDPROS
25. PCL. (n.d.). Point Cloud Library (PCL). http://www.pointclouds.org/
26. FreeRTOS. (n.d.). FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. https://www.freertos.org/