Simulation-Based Engineering Lab
University of Wisconsin-Madison
Technical Report TR–2023–03

Performance Analysis of ROS2

Deepak Charan Logavaseekaran, Rakshith Macha Billava

Department of Electrical and Computer Engineering, University of Wisconsin – Madison

April 21, 2023

Contents

1 General Information **2**

2 Introduction **2**

3 Solution Description **2**

3.1 Specifications 2

3.2 rclpy vs rclcpp 3

3.3 ROS2 Humble vs Foxy 4

3.4 Reliable vs Best Effort QoS Policy 5

4 Discussion **5**

5 Conclusion **6**

1 General Information

As part of the Independent study (ECE 699)/ Master's Research (ECE 790), we were tasked to analyze the performance of the communication between the ROS2 entities. This technical report documents the procedure followed to perform the analysis and the results obtained.

2 Introduction

ROS (Robot Operating System) is an open-source framework to facilitate the development of robot applications. ROS aids in managing the complexity and facilitates the rapid prototyping of the application [1]. One of the main advantages of using the ROS framework would be the interoperability and modularity that ROS provides. It supports multiple languages including Python and C++ and offers a plug-and-play node structure even over the network across multiple devices.

ROS1 was initially created in 2007 by Willow Garage and was very famous among enthusiasts. However, it did come with its own flaws and to keep up with the demands ROS2 was developed. ROS2, with a completely different architecture, provides better modularity and platform-independent support. ROS2 promises similar APIs across programming languages and faster deployment of newer features due to its layered implementation wherein the underlying base library remains the same for all programming languages. ROS2 also offers a centralized system with each node capable of discovering other nodes, asynchronous services, and actions and also removes the dependency of the ROS master. ROS2 uses DDS middleware and thereby provides all the associated Quality of Service policies. It also comes with various tools for debugging and visualization and has also extended support for Windows and macOS.

Although ROS2 is extensively used by researchers and enthusiasts alike, there have been some concerns regarding its performance in a real-time environment. In this study, we evaluate the performance of ROS2 and understand its relevance in a real-time ecosystem.

3 Solution Description

3.1 Specifications

All the tests as part of this evaluation were performed on the NVIDIA Jetson Orin Development Kit which is a 12-core ARM Cortex A78 processor with a 64 GB eMMC drive. We have used Jetson JetPack SDK 5.1.1 with Ubuntu 22.04 (Linux Kernel 5.10.X) running on the Jetson Orin device.

We have predominantly used ROS2 Humble for our tests. However, when we did a comparative analysis on both ROS2 Humble and ROS2 Foxy, the results were not significantly different as we will discuss in further sections.

3.2 rclpy vs rclcpp

rclpy and rclcpp are the ROS2 client APIs that need to be used to set up/configure the nodes or interact with the ROS2 core. It is the Python and C++ implementation respectively. Most of the APIs and their usage is similar. However, according to one of the issues we found – ros2_latency repository, rclpy is 30x – 100x slower than rclcpp for publishing large data. The issue has been resolved and merged prior to the ROS2 Foxy release. We were able to clone the repository which demonstrated the issue and were able to run the test with minimal changes and confirmed that the difference in performance is not in the order of 30x.

To evaluate the performance difference between rclcpp and rclpy, we created a simple test of publishing a block of data and measuring the time taken by both these implementations to publish the message. We first generate the data to be published and instantiate a timer. The timer is started just before the publish API call and stopped right after the publish call. The methodology is kept consistent in both the rclpy and rclcpp implementations.

We observed that the time taken by rclpy to publish messages was consistently ∼6x more than rclcpp as shown in Figure 1.

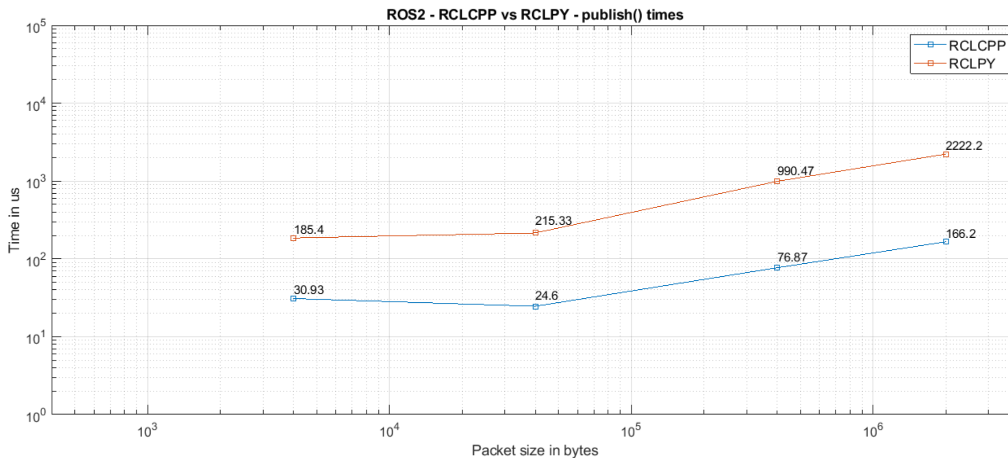

Figure 1: rclpy vs rclcpp publish time

We then performed the tests to compare the latency of both the rclpy and rclcpp implementations. To calculate the latency, we first created the message with the required number of bytes. We then recorded the epoch time and called the publish API. Immediately after, the epoch time is sent to the subscriber. In the subscriber node, we record the epoch time as soon as we receive the message. We then receive the publish time and calculate the difference to obtain the message latency. This test is recreated about 15 times and the average of the 15 results is considered for our evaluation.

We observed that the latency of the rclpy implementation was about 3x that of rclcpp imple-

mentation as shown in Figure 2.

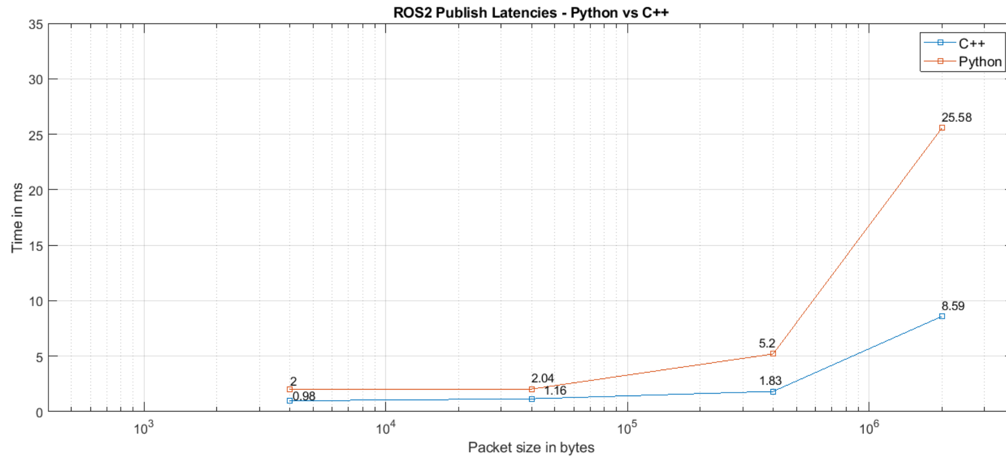

Figure 2: rclpy vs rclcpp latency

3.3 ROS2 Humble vs Foxy

Due to the use of multiple ROS2 versions among the various devices in the lab, we also evaluated the ROS2 versions to see if the different versions have an impact on the results. We ran the same latency test on both ROS2 Humble and Foxy versions. This test was performed using the rclcpp implementation on both Humble and Foxy versions. From our observation, we did not find any conclusive evidence that one version is better than the other as shown in Figure 3, although, for larger messages, Foxy performed slightly better.

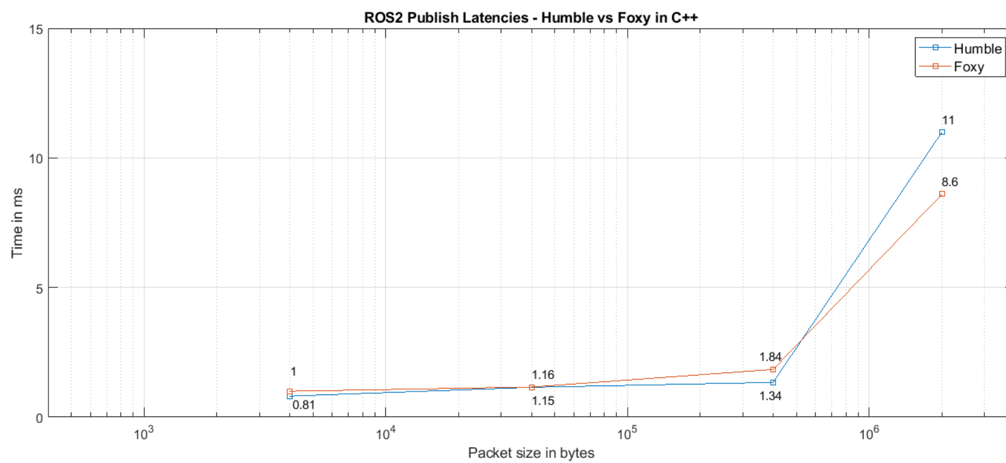

Figure 3: ROS2 Humble vs Foxy latency

3.4 Reliable vs Best Effort QoS Policy

As ROS2 uses the DDS middleware, it also inherently provides various QoS policies. For the purpose of this study, we wanted to evaluate the impact of using reliable and best-effort QoS policies. The latency test used in previous evaluations is used here with the modification of the QoS policy.

Here too, we were not able to find any considerable improvement of one over the other as shown in Figure 4. As the test was performed with just 2 nodes there was no network traffic which implies that there were none or minimal packet losses. Therefore, we reason that both best effort and reliable QoS policy implementation send similar data without repetition which could be the reason for similar results.

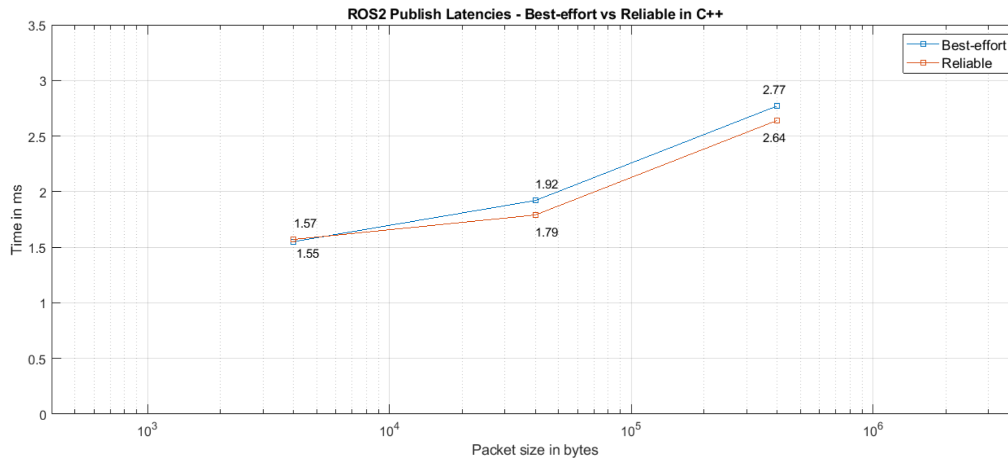

Figure 4: ROS2 Reliable vs Best Effort

4 Discussion

We reached out to a few experts in order to validate our results. A couple of suggestions were provided which we also tried to incorporate into our tests. On obtaining our test results, we reached out to the ROS Developers. However, during the time of writing the report, we did not get any update from the team.

We also reached out to Marc Bestmann, Universität Hamburg, the author of the article Experiences with ROS 2 on our robots. The following were the suggestions made by Marc based on his experience with ROS2.

- Using event executor for the C++ implementation of ROS2
- Deactivating IDLE mode of CPU through Linux OS
- Pinning CPU extensive tasks to individual CPU cores

- Changing Linux scheduler to Round Robin
- Compiling multiple nodes into a single process to avoid IPC and leverage shared memory

Upon implementing these suggestions, we found that it did not give a considerable improvement in the latency for our specific test case.

In addition to this, we got in touch with Prof. Peter Gabriel Adamczyk and his students Yisen Wang and Kieran Nichols. Although they had not pushed ROS2 to send large messages in the order of MB, they mentioned that their experience with ROS2 is somewhat similar. They also recommended using the real-time Linux kernel and the use of UDP instead of TCP for the ROS2 connections. As all of our nodes run on a single device, network protocol choice would not be applicable to our experiments. However, we would want to consider using RT_PREEMPT patch for Linux in the future.

5 Conclusion

Based on the above findings, it is evident that the C++ implementation of ROS2 outperforms its Python version in terms of speed. However, despite the efforts of ROS2 developers to enhance real-time and overall performance, the framework is not yet suitable for time-critical and real-time systems. Our testing results indicate that the message transfer overhead induced by ROS2 is too significant compared to the benefits it offers in a real-time scenario. Therefore, using ROS2 for real-time applications is not recommended at its current stage.

References

[1] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, *et al.*, "Ros: an open-source robot operating system," in *ICRA workshop on open source software*, vol. 3, p. 5, Kobe, Japan, 2009.

[2] T. Kronauer, J. Pohlmann, M. Matthé, T. Smejkal, and G. Fettweis, "Latency analysis of ros2 multi-node systems," in *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pp. 1–7, IEEE, 2021.

[3] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ros2," in *Proceedings of the 13th International Conference on Embedded Software*, pp. 1–10, 2016.

[4] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications," *arXiv preprint arXiv:1809.02595*, 2018.