Comparative Analysis of ROS and ROS2 Performance in Industrial Robotics Applications

MARTIN Erwan, DUVAL Fabrice

CESI LINEACT, 80 rue Edmund Halley, St Etienne du Rouvray, 76800, France

ARTICLE INFO

Keywords:
ROS
ROS2
Benchmark
Performances
CPU

ABSTRACT

The Robot Operating System (ROS) and its successor, ROS2, are widely used robotics technologies for industrial applications. In this article, we present a benchmark study comparing the performance of ROS and ROS2 using the robot Open-Manipulator-X from Robotis. We will compare with this robot for the sames packages (with the minimal traduction required) on each technologies, the performances with indicators for resource utilization. The study will be done on a first hand on a straight line trajectory, followed by the end effector of the robot following, in a second part it will follow a more complex shape : A 3D Bernoulli's lemniscate. As ROS2 brings new features, it seems normal that it will have a bigger resource consumption than ROS. The objective is to figure out if ROS2 can actually replace ROS in every robotics application, or if ROS isn't completely obsolete yet. After the experiments we can conclude that while ROS2 offers enticing new features, our research highlights that ROS remains a viable choice for certain applications, particularly when resource consumption and compatibility with less powerful devices are critical factors. Researchers and practitioners can make informed decisions based on their specific requirements and the trade-offs between the two frameworks.

1. Introduction

In recent years, robotics systems, autonomous vehicles, path control, process systems and other real-time demanding systems became various, revolutionizing the manufacturing and automation industries.

ROS [11] (Robot Operating System) is an open-source middleware. It has emerged as a dominant framework and has been really used for every kind of robotics systems[1].

ROS/ROS2 has been maintained by Willo George[5] and the Open-Source Robotics Foundation (OSRF)[4] ROS improves productivity [2] providing users a publish/subscribe transport, diverse packages and libraries that helps developers to create their robotics softwares. The actual problems with ROS is that it only runs on a few OSs and does not satisfy real-time run requirement [9] [14]. It is also not made to control several robots (e.g a robot swarm) Also, it cannot guarantee fault-tolerance, deadlines, or process synchronization. This means that the ROS is not adapted for real-time embedded systems. To satisfy those new requirements, ROS has been reconstructed from it's core and upgraded to ROS2[7]. ROS2 has been made for improving the User-interfaces API and incorporate new technologies such as Data Distribution Service (DDS)[10], an industry-standard real-time communication system and end-to-end middleware it provide a reliable publish/subscribe transport.

ROS2 brings several new features[7]. In ROS2 you don't need a Rosmaster as in ROS(1), the rosmaster was a central name server that was handling the publications and the subscriptions. ROS2 works with a peer to peer discovery method. ROS2 also provides multiple nodes per process, a node state management (lifecycle), micro-ROS for embedded systems, a multi-platform support (Linux, Windows,macOS), and security that was clearly lacking on ROS(1) [13].

Despite those improvements, several issues have been observed with ROS2. One of the main problems is the latency with larges messages due to DDS. Scientists have given guidelines to the DDS utilization [9] and that Cyclone DDS[3] has better results in term of latency [14]. As ROS2 is a recent technology, for the moment there is not the same amount of package than in ROS(1) all the implementation has not been implemented yet.

The objective of this study is to make a benchmark between ROS and ROS2 with industrial indicators such as CPU time and usage/ RAM used / CPU frequency using moveit and ROS/ROS2 packages. ROS2 is, as we saw previously seen as unreliable or still not well supported, however, it is still under heavy development and recent updates are making it more and more reliable. This study will define if it is time for all ROS developers to switch on ROS2 or if ROS(1) is still more reliable in some specific applications.

2. BACKGROUND

In this section, we provide the prerequisite knowledge about ROS/ROS2 in order to explain our experiments in the next section. Firstly, we will describe the global functioning of ROS, and then we will review the new features that bring ROS2 to compare the core's differences.

2.1. Robot Operating System

A program running on ROS is called a node. Nodes are the fundamental components of ROS applications, operating as separate computational processes. They offer benefits such as fault isolation, accelerated development, modularity, and code reusability. Communication between nodes relies

This Paper is to be submitted for the ROSCon2023 at New Orleans.
emartin@cesi.fr (M. Erwan)
ORCID(s):

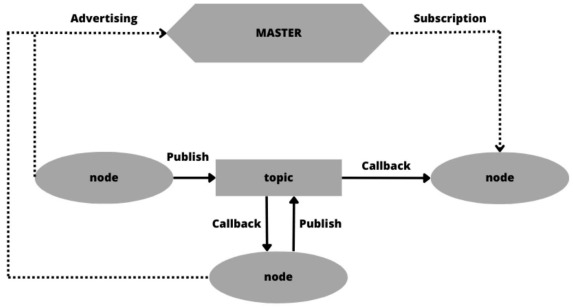

Figure 1: publish/subscribe architecture of ROS(1)

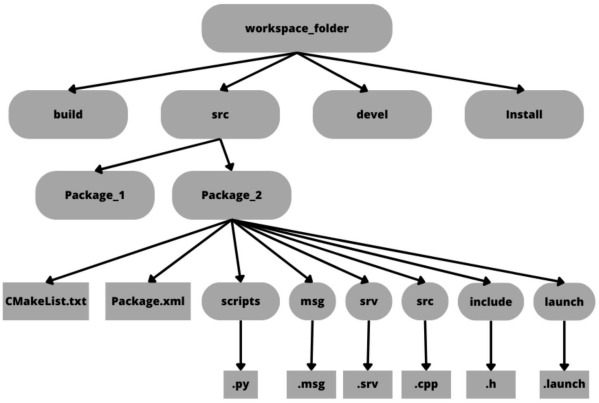

Figure 2: general organization of a ROS workspace

on the publish/subscribe mechanism as you can see on the Figure 1, that represents the publish/subscribe architecture of ROS. Nodes are communicating by publishing a message on a topic that the other nodes are listening. We have a central element called the rosmaster that is handling the messages transmission. A node has to advertise the rosmaster it will publish messages on a topic, and when a node subscribe on a topic, the rosmaster will link those two nodes and data will be transported directly between those nodes. The actual data does not pass through the master-node and ROS realize peer-to-peer data transport between the nodes. Messages are simple data structures which are similar to C structs. Those messages are defined in .msg files. Both of the nodes publishing and subscribing on a topic needs the imported message type from the .msg file to be able to format it or interpret it. This system is based on TCPROS and UDPROS that are using TCP and UDP sockets. A node can handle a service, this is a process similar to a client/server communication as it waits for a requests and return a response to the node calling it. The services structures are defined in .srv files. Nodes written in Python or C++, srv, msg, are regrouped in packages that are built on the machine in a ros workspace. The Figure 2 represents the workspace general organization. The links made on the Package 2 represents the different folders regrouping the different files that can be in this kind of package. A ROS package require a package.xml and a CMakeList.txt files to be built. It is possible once the package is built, to run the nodes, or run several at once with a .launch file.

2.2. Robot Operating System 2

The Robot Operating System 2 (ROS2) regroup the same concepts related to communications than ROS(1) such as nodes, topic, and services. ROS2 also provide actions servers, that are intended for long running tasks. They consist of three parts: a goal, feedback, and a result. There is a goal service called to perform the action, then a results service is called to get a result of the action and the result service will publish results to a feedback topic. ROS2 enhances its capabilities by leveraging the Data Distribution Service (DDS) and includes a DDS abstraction layer. This abstraction layer eliminates the need for users to directly interact with DDS APIs. By incorporating this layer, ROS2 enables high-level configurations and optimizes the utilization of DDS. Furthermore, the utilization of DDS eliminates the requirement for a master process in ROS2, which is significant for enhancing fault tolerance. The figure 3 resume the differences in term of architecture between ROS(1) and ROS2. To resume, this is a summary of what brings ROS2 compared to ROS(1):

- Network Transport : The current standard, DDS, provides support for abstraction to facilitate the integration of additional standards.
- Network Architecture : Peer-to-peer discovery
- Platform Support: Multiple (Windows, Linux, MacOS)
- Client Libraries: Sharing a common underlying C library (rcl)
- Node vs. Process : Multiple node per process (not possible in ROS(1))
- Threading Model : Swappable executor
- Node State Management : Lifecycle Node (Not available in ROS(1))
- Embedded Systems : Micro-ROS [6]
- Parameter Access: Implemented using service calls
- Parameter Types: Type declared and enforced

In addition to those features, ROS 2 incorporates a comprehensive security system that encompasses authentication, encryption, and access control mechanisms. Designers have the flexibility to customize ROS 2 according to their requirements by defining access control policies. These policies determine which individuals or entities can engage in communication and the specific topics they can interact with.

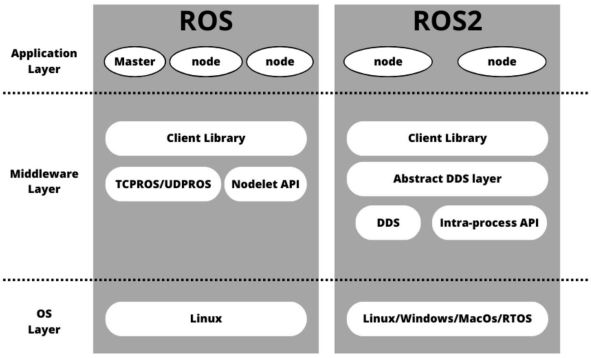

Figure 3: ROS and ROS2 Architecture

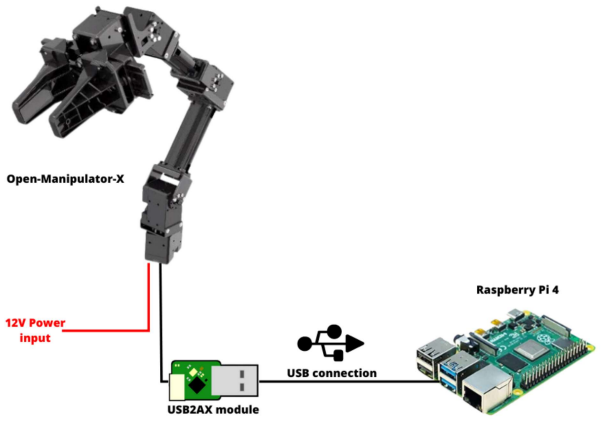

Figure 4: Physical experiments architecture

3. Experiment explanation

In this section, we clarify the environment and methods used for the experiment. The aim is to figure out if the ROS2 new implementations and features are not consuming to much for the features it provides. note that ROS2 is still on an heavy development. The following experiments were conduct to measure the resources taken by ROS and ROS2 in order to compare the two technologies.

3.1. Experiment Environments

Table 1
This table describes the hardware and software environment list.

	ROS	ROS2
Computer	Raspberry Pi 4	Raspberry Pi 4
CPU	Broadcom BCM2711	Broadcom BCM2711
RAM	4GB	4GB
OS	Ubuntu20.04 Server	Ubuntu22.04 Server
ROS distro	ROS Noetic	ROS2 Humble
Robot	Open-Manipulator-X	Open-Manipulator-X

As we can see in the Table 1, we used the same computer for the two versions of ROS and we took the last release of ROS and ROS 2 (Noetic for ROS and Humble for ROS2). ROS Noetic is primarily targeted at the Ubuntu 20.04 and ROS2 Humble is primarily targeted at the Ubuntu 22.04 so we took those different OS to have our ROS distros at their bests capacities.

The Robot we used for the comparison was the Open-Manipulator-X from Robotis[12], a 4 DOF robot powered with Dynamixel xm430-w350-t motors. The Figure 4 represents the physical installation we had for performing our tests. the USB2ax module permit us to perform the TTL communication from the Raspberry to the dynamixels motors of the robot arm. To be certain that we have the exact same packages in ROS and in ROS2 with the minimal changes, we did our own minimal packages for the robot's control [8]. Here is the list of the ROS/ROS2 packages and their role.

- dynamixel_arm_control : regroup the control nodes of the robot, the 4 nodes we use are "hardware_manager" that handle the communication with the Dynamixels motors from the robot, "control_node" that is handling the requests and returns given informations (e.g the joints_states), "position_node" that publish in the /joint_state topic the actual angles of the different joints and "moveit_controller_bridge" that is translating the moveit planned paths to control order for the control node.
- dynamixel_arm_msgs : regroup the .msg file that we use for publish/subscribe communication
- dynamixel_arm_srv : regroup the .srv files that we use for services server and client nodes.
- dynamixel_arm_moveit : regroup the moveit configuration files and the differents .launch files to launch the moveit nodes. This package was made with the moveit_setup_assistant package.
- dynamixel_arm_description : regroup the .urdf files, this is a description of the robot that we took directly from Robotis, it will permit us to have a view of the robot and all joints limits and dimensions are defined in this file.

For the different operations, we chose the moveit package that is an industrial standard for ROS applications. We used Moveit2 with ROS2 as it is the evolution of moveit. The important thing is that we used the sames algorithms and the minimal changes on our nodes to convert them from ROS to ROS2, then the differences of used resources will only be the result of ROS/ROS2 functionalities. To measure the used resources of the raspberry we used a python program [8] with the psutil module. It writes into a CSV file and we can then exploit data from it.

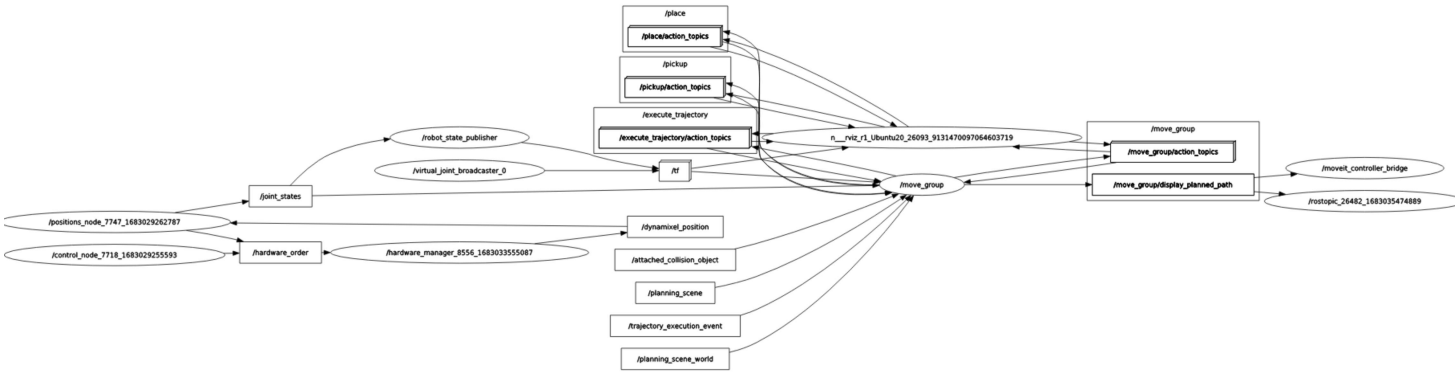

Figure 5: ROS(1) rqt graph of running nodes/topics

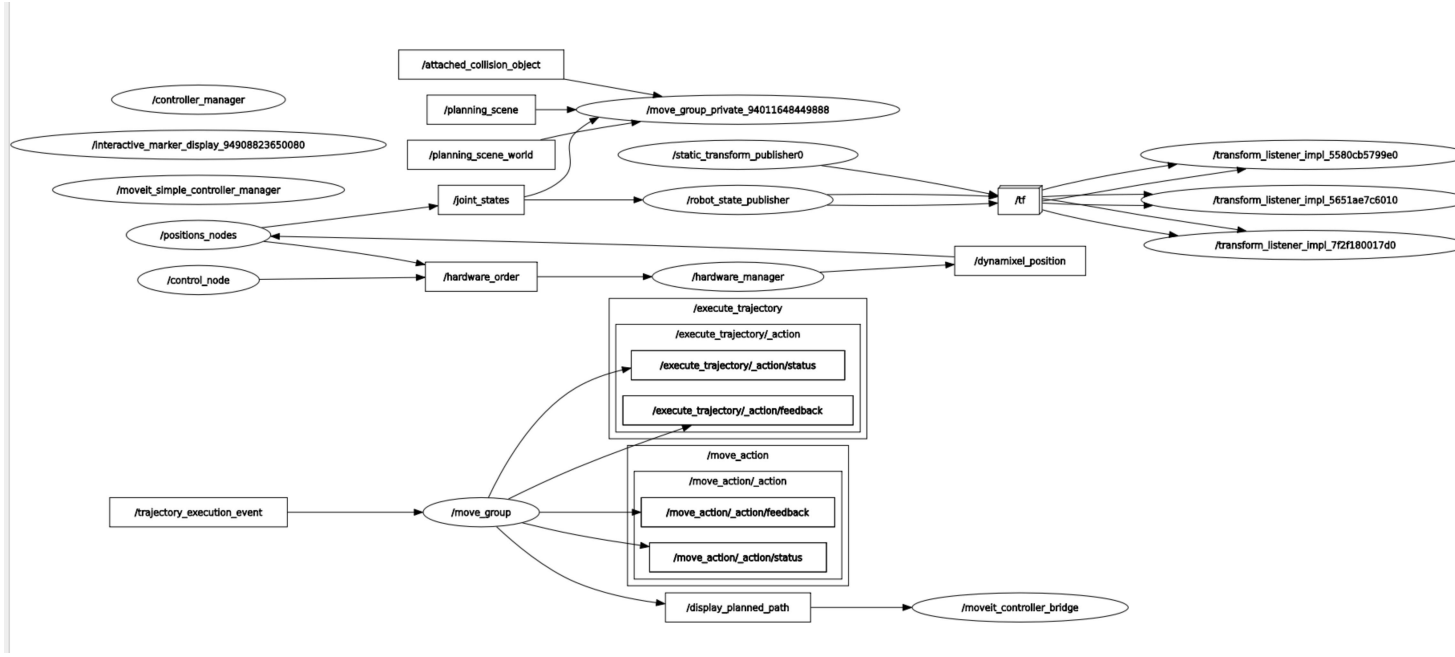

Figure 6: ROS2 rqt graph of running nodes/topics

3.2. Running nodes

In the ROS(1) environment, a maser-node is running. the necessary nodes used for robot control, running in both ROS/ROS2 environments, are listed in the Table 2. Figures 5 and 6 shows the running nodes and topics, those graphs are generated by the rqt package. Note that an other node will be added for the second part of the experiment with the Bernoulli's lemniscate trajectory..

Table 2
This table describes running nodes lists.

Package	Node	Device
._control	hardware_manager_node	Raspberry Pi 4
._control	control_node	Raspberry Pi 4
._control	position_node	Raspberry Pi 4
._control	moveit_controller_bridge	Raspberry Pi 4
._moveit	move_group launch	remote Computer
rviz	rviz	remote computer

3.3. Performed operation

Moveit can perform planning operation. it results a serie of goal messages from the robot. for each message moveit will send, the moveit_controller_bridge message will react and send messages to the other nodes in order to make the robot realize the moveit planned scene.

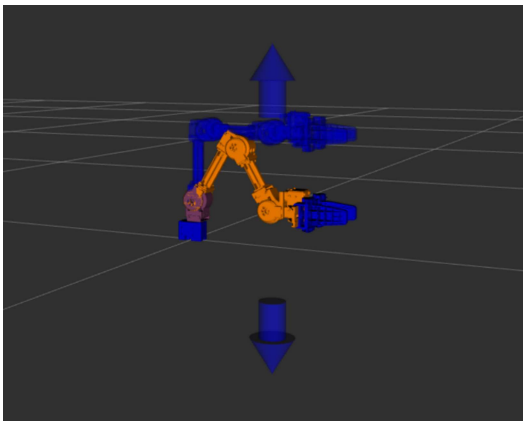

Figure 7: rviz view of real state and goal of the robot

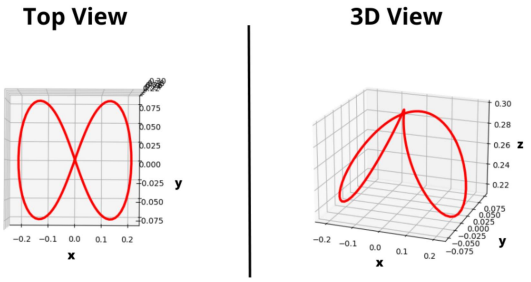

Figure 8: Graphical view of the 3D Lemniscate of Bernoulli

In a first part, we are just making the robot performing a straight line with his end effector. On the figure 7, we can visualize two robots position. The blue one is the real position of the robot with the joint states listened on the /joint_state topic. the orange one is the goal we are about to plan. A planning is necessary because we want a linear trajectory of the end-effector, this means that every steps and speed has to be calculated for each joints and this is what moveit do.

This test has a duration of 50 seconds during which we gives 3 goal position to plan to the robot. we performed those tests for ROS Noetic, for ROS2 humble with default DDS Vendor (eProsima's Fast DDS) and for ROS2 Humble with Cyclone DDS Vendor.

In a second part, we tried the experiment with a more complex trajectory. As we needed a complex trajectory using all the robot motors. this will increase the calculation part and more importantly increase the amount of messages and services calls. This is why we chose the lemniscate of Bernoulli, a shape widely used in many kind of experiments, and corresponding to our criteria. To make it even more complex, we made it 3D, here is the parametric equation of our trajectory:

$$\begin{cases} x = 2r\sqrt{2} * \frac{sin(t)}{1+cos^2(t)} \\ y = 2r\sqrt{2} * \frac{sin(t)*cos(t)}{1+cos^2(t)}) \\ z = \sqrt{4 * r^2 - x^2 - y^2} \end{cases}$$

On the Figure 8, you can see a graphical view of the theoretical trajectory the robot's end effector has to follow. The second test is measuring performances while the robot is making 3 iterations of the lemniscate.

4. Experiment Results

4.1. Linear trajectory performances graphs interpretation

Our Python benchmark program records:

- The CPU usage in % of each logical CPU (4 in our case)
- The CPU Frequency
- The CPU time for system and user
- the ram used in % and in GB

The Figure 9 shows us that we have a barely constant frequency at 1,5GHz. we have some falling points but nothing exceptional, except at the seconds 39-40 of the test for the ROS(1). We needed to take care of the frequency because the Raspberry Pi tends to slow it down when the CPU temperature gets higher, it could have been compromise our datas.

On the figure 10, for each tests, we've made the average of each CPU percentages to get only one curve (we would had 4*3 = 12 curves if we'd let the 4 CPU percentages). What is obvious is that ROS require less CPU resources than ROS2 with and without Cyclone DDS. we could expect this results because ROS2 brings many news features. In the average according to the graph, ROS stays between 10 and 20% of CPU usage, and ROS2 stays between 20 and 30%. We can also observe that using cyclone DDS reduces the amount of high peaks of CPU usage. Our final observation about this graph is that globally ROS2 use 50% more CPU resources than ROS.

The Figure 11 represents the comparison of CPU Time in % occupation over time of the user, we used this graph to verify that the results of the figure 9 were not due to process other than user's. The results are accurate with the previous curve.

On the figure 12, we can see the comparison of RAM usage over time. even if the curves seems spaced, the actual usage is pretty close with an average of 0.77GB for ROS, 0.79GB for ROS2 and 0.8GB for ROS2 using Cyclone DDS.

4.2. 3D Lemniscate trajectory performances graphs interpretation

The figures 13, 14, 15, 16 are the result of the second experiment, where the robot's end effector is following the lemniscate described in section 3.3. we used the same performances indicators. Instead of getting just the performances during the trajectory, it was interesting to record the full process from the nodes start until the killing. The process steps according to the time are described in the figure 17. It is interesting to notice that the path calculation is longer

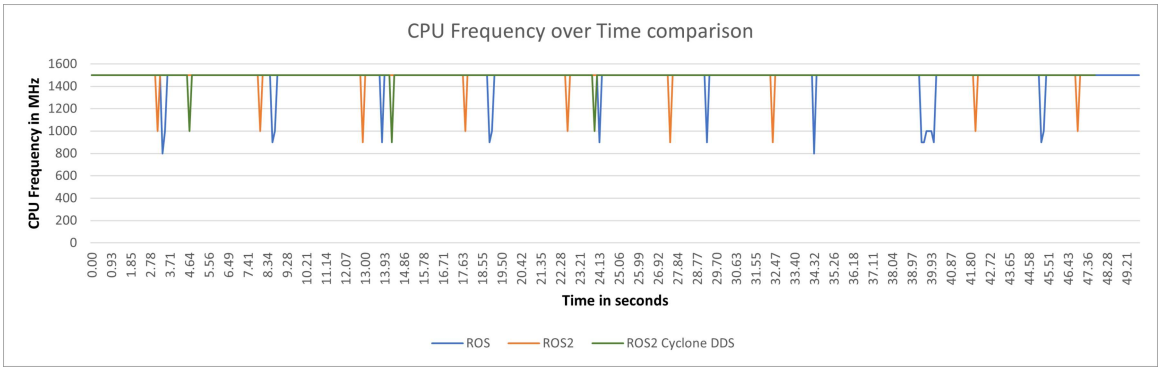

Figure 9: comparison of CPU Frequency over time

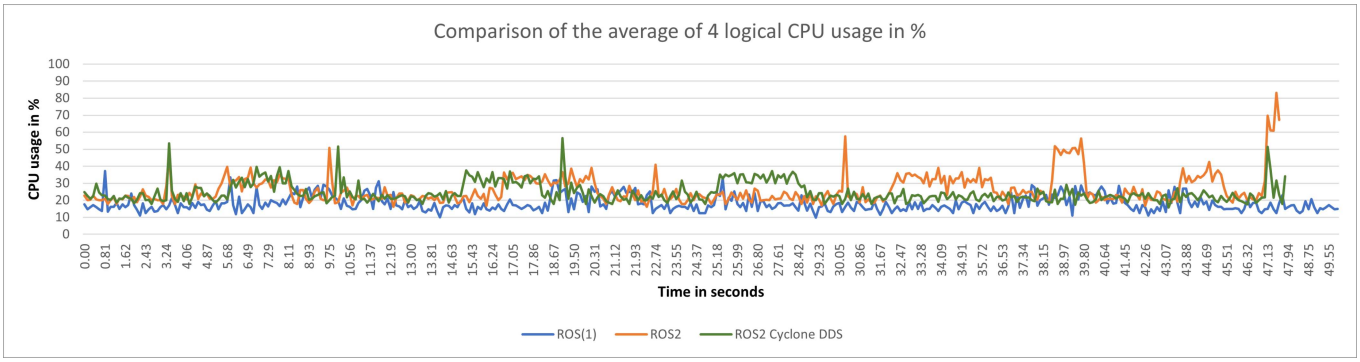

Figure 10: Comparison of the average of 4 logical CPU usage in % over time

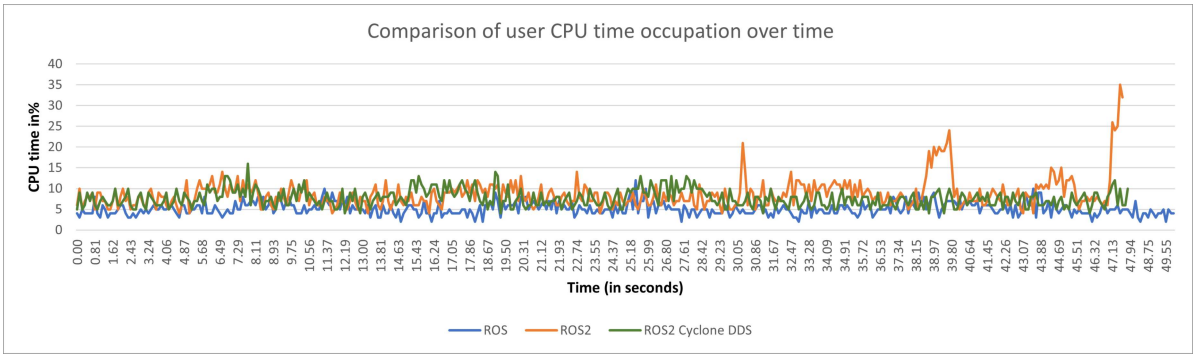

Figure 11: Comparison of user CPU time occupation in % over time

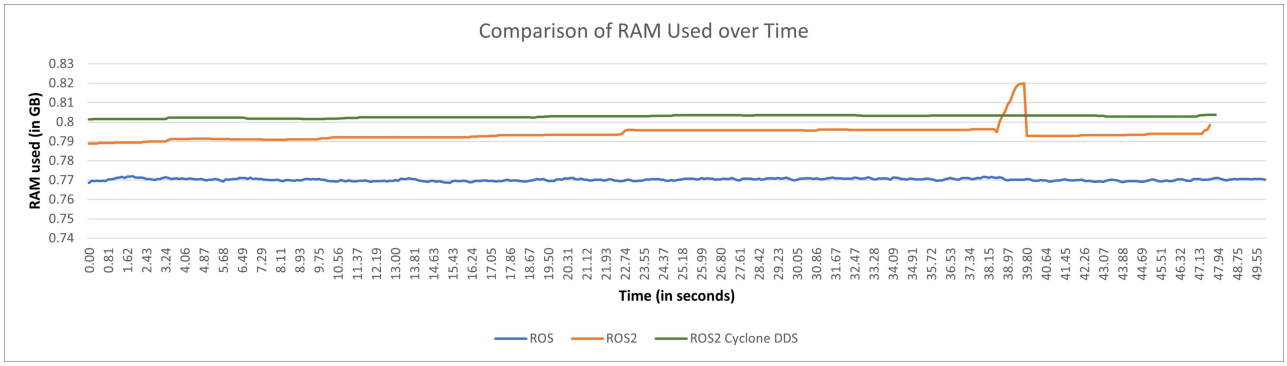

Figure 12: Comparison of RAM used over Time

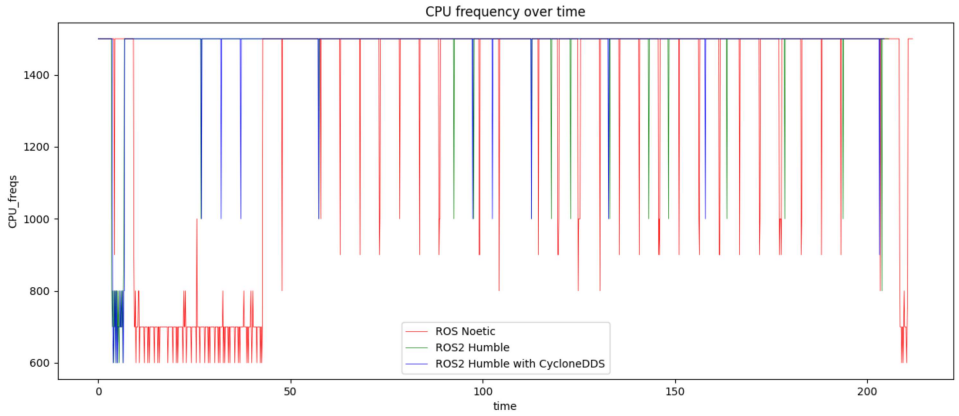

Figure 13: comparison of CPU Frequency over time

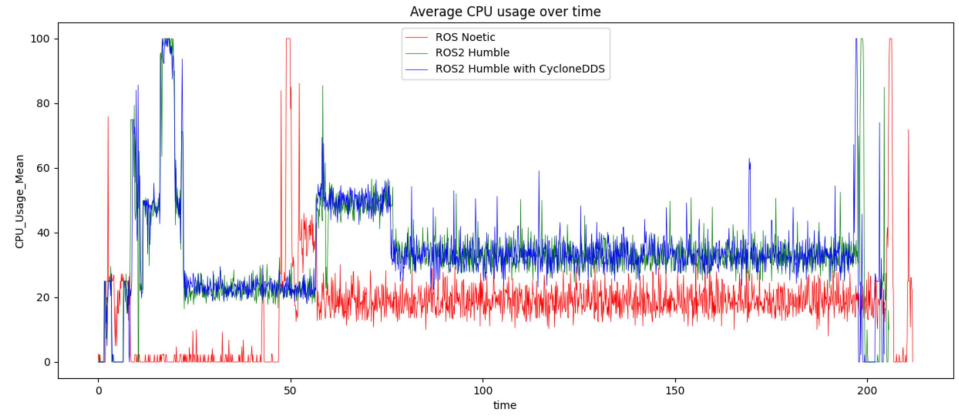

Figure 14: Comparison of the average of 4 logical CPU usage in % over time

on ROS2 with moveit2 than on moveit with ROS, but the trajectory takes less time on ROS2 compared to ROS, the processes are nearly equivalent in term of time.

On the figure 13, we can see that for ROS(1) that during the moveit launch process, the frequency is going down. The frequency is also going down on ROS2 when the nodes are launched. The frequency is going slower on the raspberry pi if the temperature is increasing, as a security measure. The frequency after this incident is stable with few insignificant lower points. We needed to assure the frequency is stable because for a given percentage of CPU usage, the real power used by the raspberry pi won't be the same if we are at the maximum frequency or if we are not.

On the figure 14, we can see high resources consumption during the node launching for both ROS/ROS2. During the trajectory itself, we can see that ROS2 processes (about 35%) is taking more CPU resources than ROS (about 20%). on the global on the trajectory step, ROS2 is taking 75% more resources than ROS.

The figure 15 represents the comparison of CPU Time in % occupation over time of the user. like in the 4.1 section, this graph is made to verify that the resources consumption were not dues to process other that user's. as the graph is accurate with the one on the figure 14, it confirms our results.

The RAM as shown on the figure 16, is more consumed during the trajectory by ROS than by ROS2 (the difference is not huge, it is about 50-100 Mo)

4.3. Overall

According on those two experiments, we can observe that ROS2 has a global resource requirements higher than ROS in term of CPU. The RAM depends of the amount of services calls and messages sent (there are more in complex shapes as in 4.2 than on a single goal in 4.1). These findings provide insights into the performance characteristics and resource utilization of ROS and ROS2, indicating that ROS generally requires fewer CPU resources compared to ROS2, while the RAM usage difference between the two frameworks is minimal. ROS can be preferred in certain applications such as the utilization of small devices such as a Raspberry pi where we could need to save some resources in order to run some heavy algorithms.

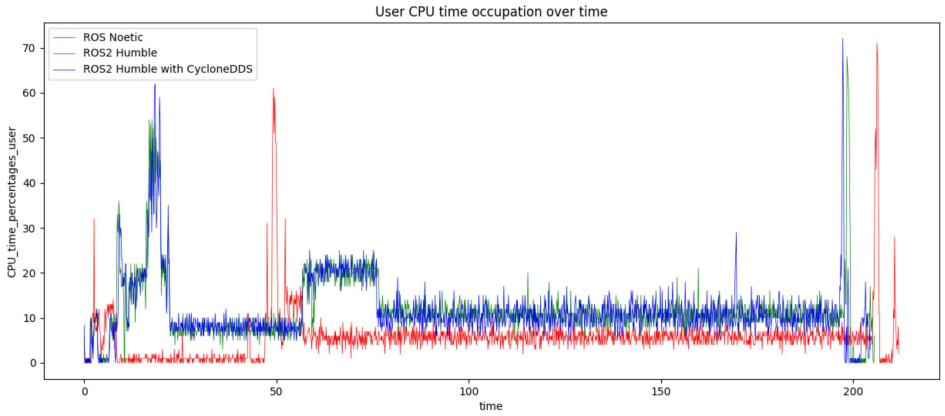

Figure 15: Comparison of user CPU time occupation in % over time

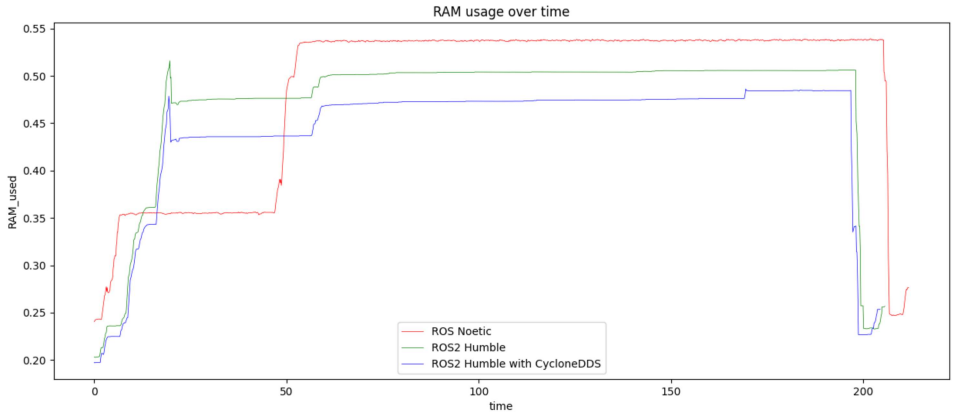

Figure 16: Comparison of RAM used over Time

5. Conclusion

In this paper, we have presented an in-depth analysis of the current state of the art in ROS and ROS2, highlighting the key advancements introduced in ROS2. Furthermore, a comprehensive comparison between ROS and ROS2 has been conducted and the findings indicate that ROS2 exhibits higher global resource requirements compared to ROS. However, for scenarios involving resource-constrained devices such as the Raspberry Pi, employing computationally intensive algorithms, and where the new features of ROS2 are not essential, ROS may be better suited due to its lower resource consumption.

While micro-ROS remains a viable option as it works with ROS2, but it is specifically designed for significantly less powerful devices than the Raspberry Pi, it should be noted that ROS is far from obsolete. Despite the increasing appeal and growing potential of ROS2 and its innovative

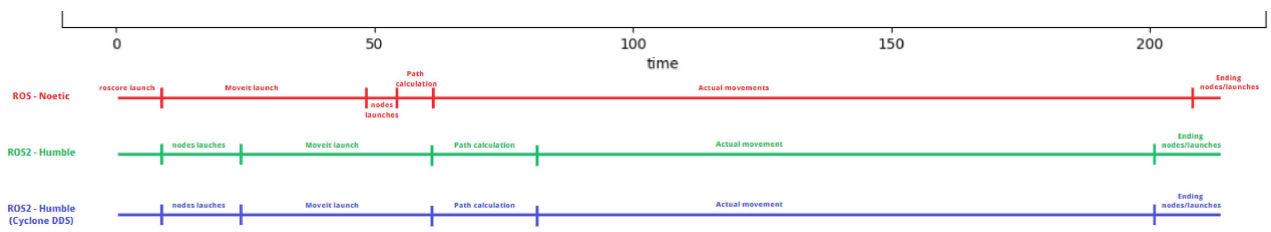

Figure 17: Comparison of RAM used over Time

features, ROS continues to serve its purpose effectively in various application domains.

In conclusion, our study emphasizes that researchers and practitioners have the flexibility to choose between ROS and ROS2 based on the specific requirements of their applications. The decision should take into account factors such as resource limitations, algorithm complexity, and the necessity of ROS2's new functionalities.

6. Further work

In future work we will compare the performances in term of position precision. The robot will perform the same lemniscate and we will register the path poses over time with a motion capture technology that permit us to get a real time 3D position of markers we will equip the robot. As the motion capture is really precise (inferior than 1mm), we will take it as real pose for the robot, then we will compare this real pose with the theoretical curve directly from the parametric equation and with the "inner" robot pose in ROS/ROS2. This way will permit us to compare the position performances in term of precision.

To perform those tests, we will need to use a 6axis robot in order to control the orientation in addition of the position, because the motion capture markers has to be oriented in the room for better results. we are planning to use a IRB-1200 from ABB.

References

[1] S. Cousins. "Exponential growth of ROS". In: *IEEE Robotics Automation Magazine* (2011).

[2] S. Cousins et al. "Sharing software with ROS". In: *EMSOFT'16* (2016).

[3] Eclipse Cyclone DDS. "https://projects.eclipse.org/projects/iot.cyclonedds". In: ().

[4] Open Source Robotics Foundation. "http://www.osrfoundation.org/". In: ().

[5] Willow Garage. "https://www.willowgarage.com/". In: ().

[6] I. Lutkebohle et al. "micro-ROS: ROS 2 on microcontrollers". In: *ROSCon. Open Robotics* (2019).

[7] S. Macenski et al. "Robot Operating System 2: Design, Architecture, and Uses In The Wild". In: *Science Robotics* (2022).

[8] Erwan MARTIN. "Github Profile". In: *https://github.com/R1leMargoulin* (2023).

[9] Y. Maruyama, S. Kato, and T. Azumi. "Exploring the Performance of ROS2". In: *EMSOFT'16* (2016).

[10] G. Pardo-Castellote. "OMG Data-Distribution Service: Architectural Overview". In: *IEEE International Conference on Distributed Computing Systems Workshops* (2003).

[11] M. Quigley et al. "ROS: an open-source Robot Operating System". In: *Proc. of IEEE International Conference on Robotics and Automation Workshop on Open Source Software* (2009).

[12] Robotis. "https://robotis.com/". In: ().

[13] Rafael R. Teixeira, Igor P. Maurell, and Paulo L. J. Drews-Jr. "Security on ROS: analyzing and exploiting vulnerabilities of ROS-based systems". In: *IEEE* (2020).

[14] Y. Yang and T. Azumi. "Exploring Real-Time Executor on ROS 2". In: *2020 IEEE* (2020).