The ring-buffer ROS2 executor: a novel approach for real-time ROS2 Space applications

1st Dr Pablo Ghiglino
R&D Dept
Klepsydra Technologies AG
Zurich, Switzerland
pablo.ghiglino@klepsydra.com

2nd Guillermo Sarabia
R&D Dept
Klepsydra Technologies AG
Zurich, Switzerland
guillermo.sarabia@klepsydra.com

***Abstract*—ROS2 (Robot Operating System Version 2) is an open-source software framework and suite of libraries that facilitates robotics application development and multi-core processing. ROS2 is becoming the de-facto standard for autonomous robotics application and recently. On the other hand, with the support of NASA and Blue Origin, Space-ROS -a Space applications specialized implementation of ROS2- is also starting to be used for Space autonomous applications, like landers and space robotics. ROS was designed to simplified and reduce the development time and effort required to write robotics applications and ROS2 kept this focus plus the main goal of enabling real-time robotics. In the initial design of ROS2, processing performance was given low priority, and, while an effort has been made to increase processing performance, it is not uncommon to face long latency in receiving sensor data or even non-negligible data losses. The research presented here shows a ROS2 executor implemented based on a lock-free ring-buffer that can not only increase by significantly the data processing rate with respect to the built-in ROS2 executors, but also reduce the processing consumption substantially. Successful performance results of this novel ring-buffer ROS2 executor are presented here and validated in several Space computers and the de-facto real-time benchmarking standard: the Raspberry Pi4 with Real-time Linux as operating system.**

***Index Terms*—ROS2, Space onboard computing, high performance computing, parallel processing**

I. Introduction

ROS2 (Robot Operating System 2) represents the evolution of the widely adopted ROS (Robot Operating System) framework, instrumental in the development of software applications for robotics and other autonomous systems [7] [16]. Space-ROS [15] extends this platform specifically for applications in space, addressing the unique challenges presented by the extreme environment including radiation, vacuum, temperature fluctuations, and communication delays. It caters to these challenges by offering specialized components for managing space-specific issues such as radiation-hardened electronics, specialized communication protocols, and fault-tolerant software.

ROS2 data workflow design has been thought out to provide a robust and uniform access to sensor data via different supported middleware including DDS and memory sharing solutions in order to enable real-time processing [11] [13]. While this design has largely simplified the development effort required to write distributed robotics applications, it has somewhat compromised processing performance, resulting in occasional long latencies in receiving sensor data or even data losses. Various studies have addressed this issue [9].

The ROS2 community, specifically the Real-Time Working Group (RTWG), has dedicated significant efforts to improve this situation, leading to the development of the Static Single Thread and the Multi-thread Executors [17]. These executors have demonstrated substantial performance enhancements for certain specific scenarios [18]. However, a universal solution to ROS2's performance bottlenecks remains elusive, leaving developers to determine the optimal combination of executors for their specific use cases.

Other industries, such as finance, have achieved remarkable speeds by parallelizing data processing algorithms, enabling real-time applications through the so-called lock-free programming technique [2]. Although complex to implement efficiently, this technique can yield exceptional performance results when employed correctly [5]. In this paper, the authors have implemented a lock-free ring-buffer and validated it across different on-board computers, yielding impressive performance results for sensor data aggregation, processing, and even artificial intelligence.

The research herein presents a ROS2 executor, built on our lock-free ring-buffer, that significantly enhances data processing rates and reduces processing consumption substantially compared to the previously mentioned executors. This has profound implications for system-level performance. Enhanced processing speeds translate into faster response times, particularly crucial for real-time autonomous navigation. The ability of our executor to make quick, reliable decisions in this context could lead to reductions in accidents and collisions, making for safer and more efficient operations.

Furthermore, our ROS2 multi-node variant of this executor has been implemented, coupled with offline optimization based on genetic algorithms. This combination promises performance enhancements for virtually any scenario involving ROS2 applications. These improvements, both at the micro and macro levels, illustrate the potential of our approach to optimize and enhance the effectiveness of robotic systems in diverse settings.

The paper presents the performance results of this innovative ring-buffer ROS2 executor and its multi-node variation using

the so-called reference system. The performance is validated on several Space computers and the de-facto standard for real-time benchmarking: the Raspberry Pi4 with Real-time Linux as the operating system [3].

The remainder of the paper is structured as follows: Section II elucidates the primary concepts of the ROS2 executor approach and the current state-of-the-art. Section III explains the novel executor and how it utilizes lock-free programming techniques. Finally, Section IV outlines the performance benchmark setup and discusses the obtained results.

II. ROS2 Executor Model

A. ROS2 Executor Explained

The ROS2 Executor facilitates the coordination and scheduling of a ROS2 application. It manages the callbacks associated with subscriptions, messages, services, timers, and nodes. In ROS2, the Executor doesn't maintain its own queue of messages and callbacks. Rather, it retrieves messages from the underlying middleware DDS queues before dispatching them for execution on one of the threads. The DDS system shoulders the responsibility of defining the QoS settings [12].

The interface to the middleware, known as the ROS Middleware Interface (rmw), is displayed in Figure 1. This interface offers an abstraction layer to various DDS implementations, facilitating communication with the ROS 2 Client Library. This package encapsulates the rmw interface for DDS implementation, along with some generally useful functionality for implementers.

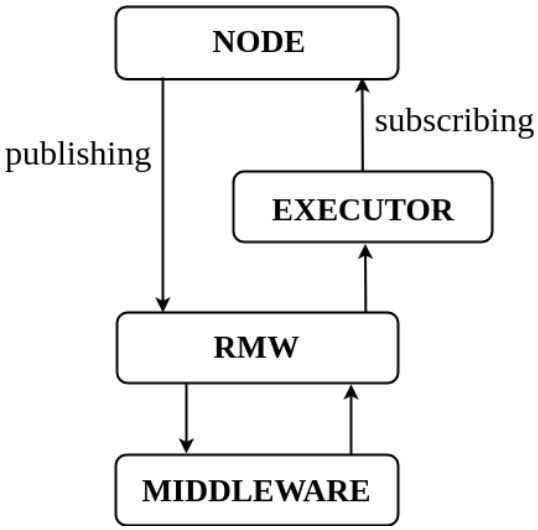


Fig. 1: ROS2 diagram.

To explain the role of the executor in the full ROS2 dataflow, it is important to offer a brief description of the ROS2 software stack, as depicted in Fig 2. This will be discussed from the lowest to the highest level:

- The Middleware layer, utilizing either DDS or RTPS implementation, is responsible for managing publishing/subscribing mechanics, service request/reply mechanics, and message serialization.
- The ROS 2 Middleware Interface (rmw) implements an abstraction layer compatible with various middleware implementations, enabling communication with the ROS2 Client Support Library. Each middleware is associated with a unique rmw adapter, which harnesses the middleware's interfaces and tailors them for its corresponding rwm.
- The ROS Client Library for C++ (comprising rclcpp for C++ and rclpy for Python) is a user-facing, C++ idiomatic interface. This library encapsulates all ROS client functionalities such as node creation, publisher and subscriber setup, etc. It implements sophisticated functionality and is typically the interface users interact with. The executor is housed within this library.
- User code represents the actual application code that deploys the ROS2 stack.

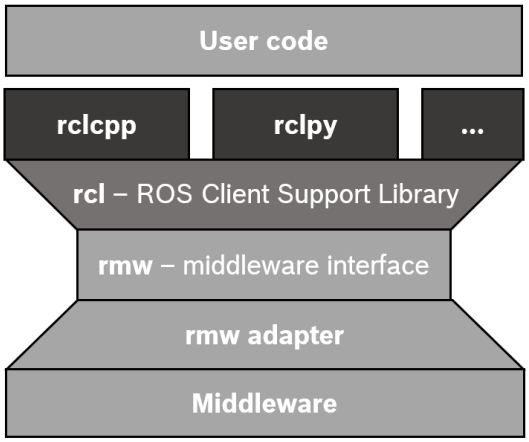


Fig. 2: Description of the ROS2 software stack [12]

The executor operates as a cyclical single process that, when idle, requests a status update from the middleware via the rclcpp and the rmw (including their respective adapters). The responses are not the actual messages from the middleware, but rather a notification of which queues have pending messages within the middleware. These queue status responses are accumulated in a wait-set.

A wait-set is employed to notify the Executor of messages available on the middleware layer, with each queue represented by a binary flag. Additionally, wait-sets serve to detect when timers expire. The executor then proceeds to traverse the Wait-set, gathering a message, triggering the callback, and clearing the corresponding Wait-set queue at each step.

Once the Wait-set is emptied, the executor reverts to an idle state, thereby instigating another request to the middleware cycle.

B. State-of-the-art

ROS2 default C++ executors exhibit different methodologies for assigning nodes to various threads [18]:

- Single-threaded executor: This executor employs a solitary thread to query the middleware and execute callbacks

sequentially. Subsequently, it scrutinizes the structure, updating the number of nodes, subscriptions, services, etc, and regenerates a Wait-set.

- An adaptation of the Single Threaded Executor is the Static Single Threaded Executor. Here, the structure's scan and definition occur solely during initialization. This executor is essentially a static version of the initial single-threaded executor, with "static" referring to the absence of list reconstruction for each iteration. All nodes, callback groups, timers, subscriptions etc. are established before $spin()$ is invoked, and modified only upon adding/removing an entity to/from a node.
- The Multi-threaded executor establishes multiple threads to execute callbacks concurrently. Analogous to the single-threaded executor, it intermittently scans the application's structure and refreshes the problem description. This executor leans on callback groups for parallel execution and initiates a threadpool to execute callbacks. Callbacks within distinct groups may be executed in parallel. Callback groups can be designed to be either:
  - Mutually Exclusive Callback Group: In this case, no two callbacks can be executed concurrently.
  - Reentrant Callback Group: Here, no restrictions are imposed on executing diverse callbacks simultaneously.

III. The Ring-buffer ROS2 Executor

The research discussed herein comprises a high-performance ROS2 Executor implementation. This execution is built on Klepsydra's Lock-free Ring Buffer and capitalizes on its high-performance parallelization features to distribute messages to the ROS2 subscribers in a highly efficient manner.

A. The lock-free ring buffer

1) Lock-free parallel data processing for embedded systems: The foundation of this research is the concept of lock-free ring buffers [10]. These are high-performance concurrency frameworks based on CAS (Compare And Swap). Their efficiency has attracted considerable attention from industry. For instance, corporations such as LMAX have utilized these techniques [2] to develop software capable of processing up to six million orders per second [5].

Lock-free programming, specifically with regard to ring buffers, is traditionally implemented in Java, due to the language's embedded Java Garbage Collector, which simplifies the complex task of lock-free programming, making it more accessible to developers. Furthermore, the popularity of books like "Java Concurrency in Practice" [14] has had a significant influence on the Java development community. The arrival of C++11 and smart pointers [8] paved the way for a partial porting of ring buffers to C++, given the similar coding style to Java Garbage Collector.

2) Lock-free event loop: Relying on this pattern, coupled with the incorporation of the newly introduced smart pointers in C++11 and encapsulating it within a publisher-subscriber pattern [4], we crafted a simplified application public interface (API). This represents a condensed adaptation of the robust LMAX disruptor, customized for embedded systems, as illustrated in figure 3.

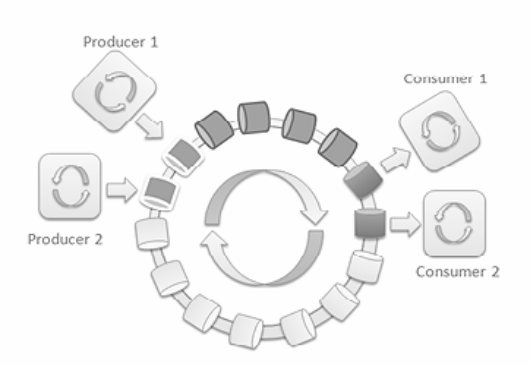


Fig. 3: Developed ring buffer.

The presented ring buffer operates in two modes. The first is the sensor multiplexer as shown in figure 4, which operates as a single producer, multiple consumer solution. Primarily employed for vision navigation robotics and drone applications, this pattern is beyond the scope of this article. The second is an event loop (figure 5) which is based loosely on financial system developments. This event loop is lock-free, offers high performance and a level of determinism that is unprecedented in embedded systems [1].

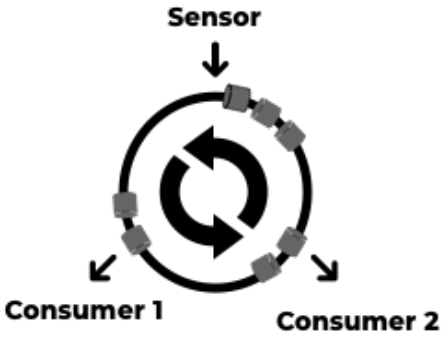


Fig. 4: Sensor multiplexer.

B. The ring-buffer ROS2 executor

The ring-buffer ROS2 Executor utilizes the Klepsydra's event loop to dispatch messages to the subscribers in all nodes, which are coming from the middleware via the rmw, as depicted in Figure 6. The event loop administers these topics using publisher-subscriber pairs. It is crucial to differentiate that the event loop publishers are not the same as the ROS2 node publishers. Henceforth, we will refer to the event loop publisher when we mention "publisher", unless stated otherwise.

In several respects, the ring-buffer executor operates in a way akin to the static single-threaded executor. Firstly, it

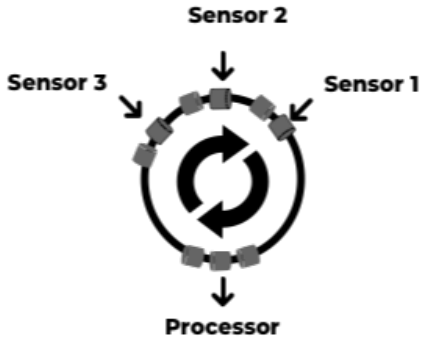


Fig. 5: Lock-free eventloop.

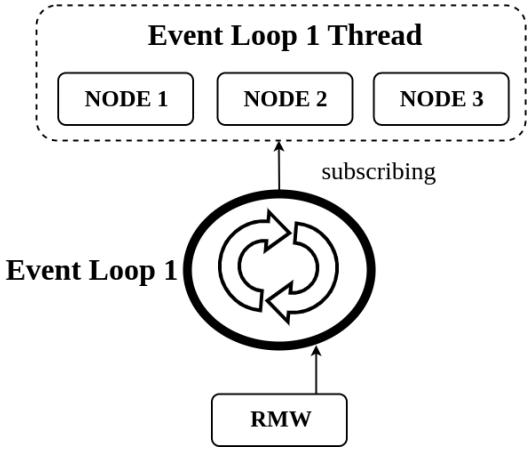


Fig. 6: Single Core ring-buffer Executor.

does not reconstruct the executable list for every iteration. All nodes, callback groups, timers, subscriptions etc. are established at construction time. Secondly, all subscriptions within a node execute on the same thread, regardless of the number of cores allocated for the streaming setup.

The ring-buffer executor relies on the event loop to transmit middleware messages to all nodes in the ROS2 application. Although it borrows inspiration from the static single-threaded executor, its implementation is encapsulated in the following features:

- A publisher-subscriber pair is created for each topic demanded by a specific ROS2 node. Each publisher-subscriber pair is internally distinguished by two parameters: the node name and the topic name. Hence, two distinct nodes publishing to the same topic are managed independently.
- All publisher-subscriber pairs related to topics belonging to the same node are supervised by the same event loop.
- Since a single event loop manages all topics in a node, all subscribers are invoked on the same thread.
- Similar to the static single-threaded executor, the ring-buffer executor establishes the mapping between nodes and event loops before invoking the ROS2 spin API, implying that changes to the nodes (adding or removing topics) are not possible when utilizing the ring-buffer executor.

C. Single and Multicore variations

The ring-buffer executor is available in two variations: single-core and multi-core. The advantage of the ring-buffer executor is that there is no need for multithreading management of the subscribers since all of them are managed by the thread of the associated event loop, which is common to both single-core and multi-core.

The single-core version functions similarly to the static single-threaded executor, as all subscribers in all nodes are invoked by the same thread, as illustrated in Figure 6.

In contrast, the multi-core variant permits mapping of nodes to different cores, where each event loop is entirely dedicated to processing the messages of each node, as demonstrated in Figure 7. The primary advantage of the multi-core is that the system load can be better distributed across different event loops and thus across different cores. Furthermore, the next section elaborates on the usage of genetic algorithms for finding the optimal distribution of nodes to cores.

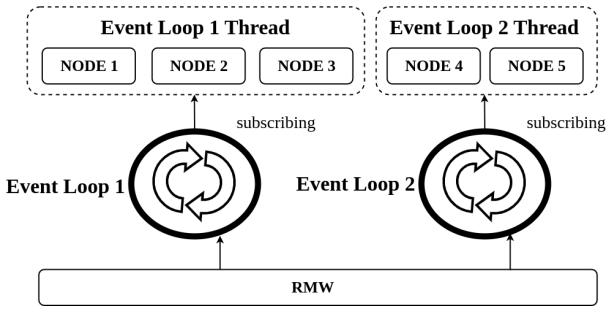


Fig. 7: Multi Core ring-buffer Executor.

D. Multi-core ring-buffer executor optimization

Optimal load distribution of the nodes among cores can significantly enhance the multi-core variant of the ring-buffer executor's performance in terms of latency, power consumption, and data throughput. However, core mapping is not straightforward and demands a systematic method. One potential approach is to define a target function that gauges the system's performance based on the core configuration. A genetic algorithm can optimize the core configuration by testing different configurations iteratively and selecting those that perform well according to the target function. This process continues until an optimal configuration is found, thereby ensuring more efficient utilization of the multi-core system and optimal load distribution.

This approach is generally versatile and proves effective in many scenarios. It makes no presumptions about the general application topology and is applicable to a broad range of systems. However, the genetic algorithm's convergence is not guaranteed and might be influenced by the application topology. Moreover, it presumes the application topology is

static and does not account for potential alterations in the task structure.

IV. The performance benchmark framework

The benchmark performance was evaluated using the Reference System [6], a benchmarking tool devised by APEX.ai to simulate real-world robotics applications constructed with ROS2. The Reference System comprises modular blocks acting as nodes with defined behaviors, a system for connecting and structuring these blocks using callbacks, and a suite of scripts to measure performance.

A. The benchmark setup

All performance data were gathered using a Raspberry Pi 4B with the following specifications:

- ROS galactic
- Ubuntu 20.04
- 4 GB of ram
- Real-time patch.

This configuration aligns with the recommended setup for the reference system. As advised, we set the operating frequency to a constant 1.50 GHz, following the provided setup guidelines. Unlike the repository's suggestion, we will not isolate the CPUs due to our executor's capability to leverage the multi-core configuration.

Our benchmark is based on the Autoware reference system detailed in [6], a realistic driving application example built on the Reference system. Figure 8 presents all executing nodes and their interconnections. In line with the optimization target defined in Section III-D, our genetic algorithm aims to minimize the average latency of the critical path. This is the duration from Lidar Data publication until the Object Collision Estimator completes its task. Figure 9 highlights the critical path we aim to optimize. Each benchmark will run for 90 seconds on all executors.

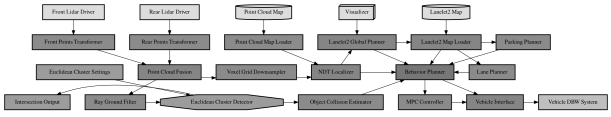


Fig. 8: Node relationship Autoware Reference System.

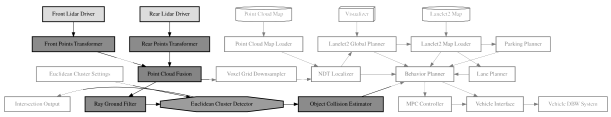


Fig. 9: Critical path reference system

V. Benchmark Outcomes

The Reference System employs a basic prime search algorithm up to a number N, establishing the workload executed by the nodes. To assess our executor's robustness, we executed various N values. Note that the workload does not scale linearly. All units are in milliseconds. Table I and Figure 10 details the results. For comparison, the reference system examples implement an N of 4096. The following conclusions can be drawn:

- Across all test cases, Klepsydra's executor performance matches or surpasses that of the original executors.
- For minor node workloads, the increased complexity does not improve results. The static single-threaded executor, due to its simplicity, outperforms the other executors.
- As expected, as the workload increases, the second-best executor shifts from Static Single Threaded to Multi-threaded. This is anticipated since the complexity added by parallel processing begins to offset its cost.
- It was projected that the Static Single Threaded would consistently outperform the single-threaded executor, given the application does not alter its topology during execution. This is indeed the case, as demonstrated by the results.

VI. Conclusions and future work

This paper introduces a unique strategy to optimize the ROS2 execution model, integrating a lock-free ring-buffer based ROS2 executor implementation with the use of genetic algorithms to optimally distribute the robotic application load among available computer cores. This combination proves highly efficient for systems with a substantial computational load, such as the discussed reference system. A major advantage of this research is its adaptability to different applications: varying ROS2 application topologies can be expedited using the ring-buffer executor coupled with genetic optimization, addressing one of the most debated challenges in ROS2.

Looking ahead, several features are to be incorporated into the ring-buffer executor: timer support, open-source release of the single-core ring-buffer executor, and the employment of the sensor multiplexer along with the event loop for topics with multiple subscribers.

References

[1] Thomas A Henzinger. Two challenges in embedded systems design: Predictability and robustness. *Philosophical transactions. Series A, Mathematical, physical, and engineering sciences*, 366:3727–36, 11 2008.

[2] M. Baker and M. Thompson. Lmax disruptor, 2011.

[3] C. Bedard and A. Kholodnyi. Ros2 real-time working group, 2018.

[4] Patrick Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35:114–131, 06 2003.

[5] Y. Fang, H. Zhu, F. Zeyda, and Y. Fei. Modeling and analysis of the disruptor framework in csp. In *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 803–809, 2018.

[6] Evan Flynn. The reference system.

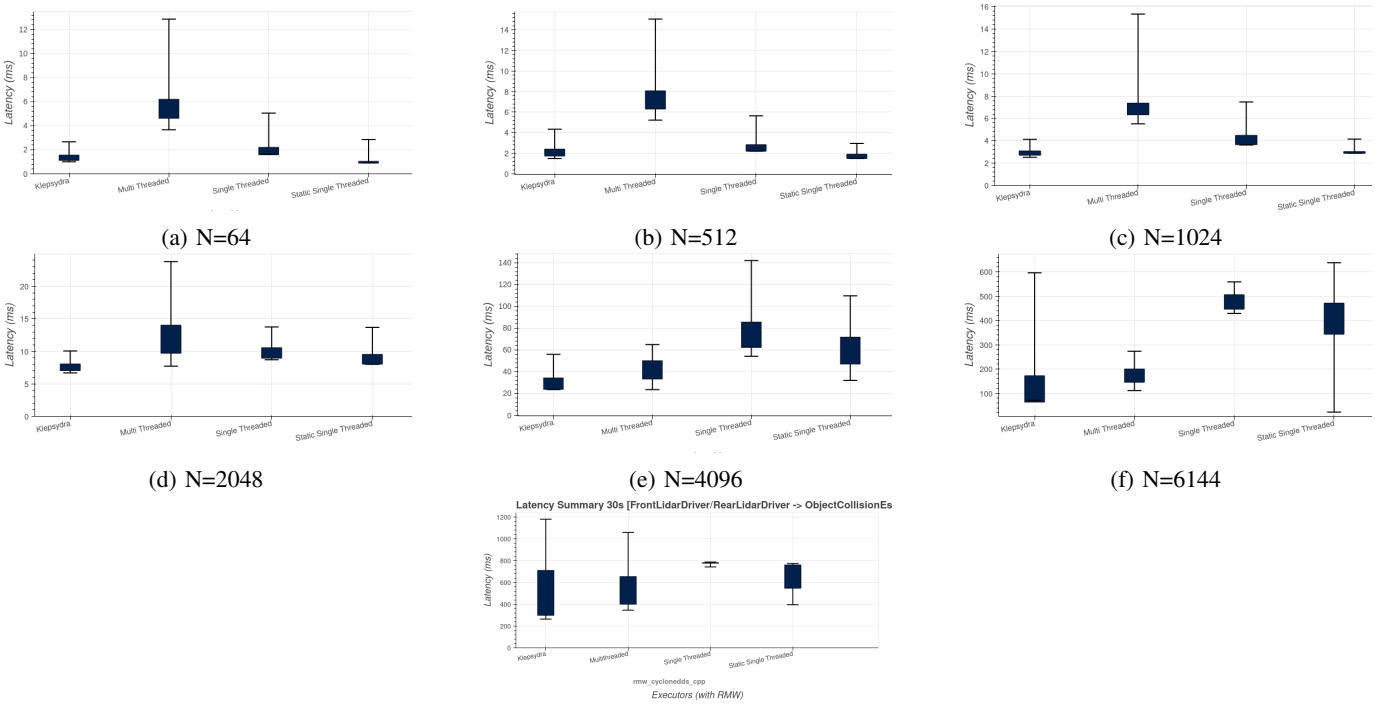
[7] Jiazhen He, Jianwen Zhang, Jiajun Liu, and Xin Fu. A ros2-based framework for industrial automation systems. In *2022 2nd International Conference on Computer, Control and Robotics (ICCCR)*, pages 98–102, 2022.

[8] Richard Kaiser. *C++11 Smart Pointer: shared ptr, unique ptr und weak ptr*, pages 781–799. https://www.rkaiser.de/, 01 2019.

[9] Tobias Kronauer, Joshwa Pohlmann, Maximilian Matthé, Till Smejkal, and Gerhard Fettweis. Latency analysis of ros2 multi-node systems. In *2021 IEEE International Conference on Multisensor Fusion and Integration for Intelligent Systems (MFI)*, pages 1–7, 2021.

TABLE I: Latency results of critical path by executor in ms

N	Executor	Min	Avg.	Max	Std. Dev
64	Klepsydra	1.0	1.3	2.7	0.2
64	Multi Threaded	3.7	5.4	12.9	0.8
64	Static Single Threaded	0.9	1.0	2.9	0.1
64	Single Threaded	1.6	1.9	5.1	0.3
512	Klepsydra	1.5	2.1	4.3	0.3
512	Multi Threaded	5.2	7.2	15.0	0.9
512	Single Threaded	2.2	2.5	5.6	0.3
512	Static Single Threaded	1.5	1.7	3.0	0.2
1024	Klepsydra	2.5	2.9	4.1	0.2
1024	Multi Threaded	5.5	6.9	15.3	0.5
1024	Static Single Threaded	2.9	3.0	4.2	0.1
1024	Single Threaded	3.6	4.1	7.5	0.4
2048	Klepsydra	6.7	7.6	10.1	0.5
2048	Multi Threaded	7.7	11.9	23.8	2.2
2048	Static Single Threaded	8.0	8.8	13.7	0.8
2048	Single Threaded	8.7	9.8	13.8	0.8
4096	Klepsydra	23.6	29.1	56.0	5.2
4096	Multi Threaded	23.6	41.7	65.0	8.4
4096	Static Single Threaded	32.0	59.4	109.7	12.4
4096	Single Threaded	54.2	73.9	142.1	11.7
6144	Klepsydra	70.7	117.8	596.0	54.4
6144	Multi Threaded	111.2	172.6	273.2	27.6
6144	Static Single Threaded	22.4	407.2	637.3	64.3
6144	Single Threaded	429.1	476.1	558.6	30.1
8192	Klepsydra	263.8	504.8	1180.1	207.1
8192	Multi Threaded	345.9	527.1	1058.8	127.7
8192	Static Single Threaded	396.4	654.1	773.9	107.4
8192	Single Threaded	743.4	779.4	787.3	3.0



(a) N=64 (b) N=512 (c) N=1024

(d) N=2048 (e) N=4096 (f) N=6144

(g) N=8192

Fig. 10: Executor Latency Critical Path for different values of N

[10] P. P. C. Lee, T. Bu, and G. Chandranmenon. A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[11] Yanan Liu, Yong Guan, Xiaojuan Li, Rui Wang, and Jie Zhang. Formal analysis and verification of dds in ros2. In *2018 16th ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 1–5, 2018.

[12] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. About executors, 2023.

[13] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. Exploring the performance of ros2. In *2016 International Conference on Embedded Software (EMSOFT)*, pages 1–10, 2016.

[14] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[15] A. B. Probe, S. W. Chambers, A. Oyake, M. Deans, G. Brat, N. Cramer, B. Kempa, B. Roberts, and K. Hambuchen. Space ros: An open-source framework for space robotics and flight software. *AIAA SciTech*, 1 2023.

[16] L. Puck, P. Keller, T. Schnell, C. Plasberg, A. Tanev, G. Heppner, A. Roennau, and R. Dillmann. Performance evaluation of real-time ros2 robotic control in a time-synchronized distributed network. In *2021 IEEE 17th International Conference on Automation Science and Engineering (CASE)*, pages 1670–1676, 2021.

[17] Jan Staschulat, Ingo Lütkebohle, and Ralph Lange. The rclc executor: Domain-specific deterministic scheduling mechanisms for ros applications on microcontrollers: work-in-progress. In *2020 International Conference on Embedded Software (EMSOFT)*, pages 18–19, 2020.

[18] Yuqing Yang and Takuya Azumi. Exploring real-time executor on ros 2. In *2020 IEEE International Conference on Embedded Software and Systems (ICESS)*, pages 1–8, 2020.