Latency Overhead of ROS2 for Modular Time-Critical Systems

Tobias Kronauer[*], **Joshwa Pohlmann**[*], **Maximilian Matthé**[*], **Till Smejkal**[†] and **Gerhard Fettweis**[*]

[*]Barkhausen Institute, Dresden, Germany, firstname.lastname@barkhauseninstitut.org
[†]Operating Systems Group, TU Dresden, Dresden, Germany, till.smejkal@tu-dresden.de

***Abstract*—The Robot Operating System 2 (ROS2) targets distributed real-time systems. Especially in tight real-time control loops, latency in data processing and communication can lead to instabilities. As ROS2 encourages splitting of the data-processing pipelines into several modules, it is important to understand the latency implications of such modularization.**

In this paper, we investigate the end-to-end latency of ROS2 data-processing pipeline with different Data Distribution Service (DDS) middlewares. In addition, we profile the ROS2 stack and point out latency bottlenecks. Our findings indicate that end-to-end latency strongly depends on the used DDS middleware. Moreover, we show that ROS2 can lead to 50 % latency overhead compared to using low-level DDS communications. Our results imply guidelines for designing modular ROS2 architectures and indicate possibilities for reducing the ROS2 overhead.

***Index Terms*—distributed systems, mobile robotics, latency, profiling, ROS2**

I. Introduction

Through the advent of robotic applications, such as autonomous driving or household robotics, the robot operating system (*ROS*) emerged as one of the most widely used software development frameworks. With more than tens of thousands of users [1], it is widely used in academia as well as in industry [2].

Shortcomings of ROS1, but also its popularity lead to the development of its successor, ROS2, which is developed by the Open Source Robotics Foundation (*OSRF*) with many industrial contributors [3], [4]. Although being under heavy development, it is already used by notable companies [5] and the Open Source community [6]. Reasons for the development of ROS2 are new use cases, e.g. multiple robots, small embedded platforms, and real-time capability. Further, new technologies are available such as the Data Distribution Service (*DDS*) [7]. [8]

ROS2 and its predecessor share the same core concept. Therefore, we refer to software architectures in ROS1 or ROS2 as *ROS systems*. Among others, a ROS system comprises *nodes*, *messages*, and *topics*. A *node* is a software module performing computation. As ROS features the notion of modularity, a ROS system is usually composed of several nodes. These nodes exchange information by passing *messages*. These messages only contain typed data structures that are standard primitive data types. A message is *published* by a node to a given *topic*, which is described by a string. Nodes that are interested in the data contained in this message *subscribe* to that topic. Each node can have multiple subscribers and publishers. [9]

This research was co-financed by public funding of the state of Saxony, Germany. We thank RTI for the license and support.

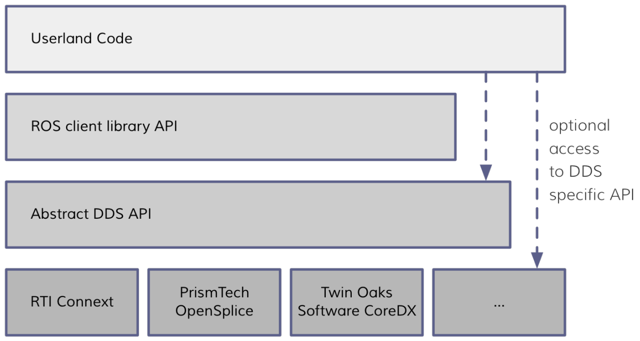

Fig. 1: API layout from ROS2, taken from [10].

Although still using the publish/subscribe mechanism of ROS1, ROS2 builds its transport layer on a new middleware. The middleware is an implementation of the DDS standard [7] that is widely used for distributed, real-time systems. Without changing much of the usercode of ROS1, the goal was to hide the DDS middleware and its API to the ROS2 user as shown in Fig. 1. For that purpose, middleware interface modules (`rmw` in short for *ros middleware*) were introduced. The most recent release of ROS2, Foxy Fitzroy, supports eProsima FastRTPS, Eclipse Cyclone DDS, and RTI Connext as DDS middleware. [10], [11].

Since it is not the main focus of the paper, we will only briefly summarize DDS. The summary is based on the most recent DDS specification 1.4 [7]. It describes a Data-Centric Publish-Subscribe model (*DCPS*) for distributed applications enabling reliable, configurable, and real-time capable information between information points. Similar to ROS1 and ROS2, there are publishers that publish information of different data types. Each publisher has a *DataWriter* that can be regarded as a source of information providing data of predefined type. The subscriber, on the other hand, is responsible for receiving the data sent by the publisher. A *DataReader* is responsible for reading the information obtained by the subscriber. As in ROS, a topic fulfills the task to connect the publisher with one or more subscribers. A distinguishing feature of DDS is the use of *quality of service* (QoS) policies. Each entity has an assigned QoS policy determining the behavior of each entity concerning communication and discovery. There is a huge variety of QoS parameters that can be set.

The described concept advocates the use of a distributed, modularized system. Robotics applications that are based on a ROS system often use a software architecture of many nodes with well-defined interfaces [12]. Complex systems like autonomous cars can benefit from this approach by being easier to evolve and to adapt. However, this entails a certain latency to the system. Since real-time capability is one of the main feature claims of ROS2, the question arises how much latency is entailed by a system of many nodes compared to a system that performs the same computation in a single node. This leads to the central question of this paper: How does a ROS2 system cope with scalability? Latency plays a critical factor for new technologies, e.g. autonomous cars, edge cloud systems, and plenty of other applications enabled by 5G [13]–[15].

The paper is structured as follows: In Sec. II, we will summarize available publications that analyze ROS2 in terms of latency followed by the description of the contribution of our work. In the succeeding section, we will briefly explain our methodology for the latency evaluation. Results are presented in section Sec. IV. This work will be concluded with a future outlook in Sec. V.

II. Related Work

At the early stage of development, ROS2 was evaluated by [16] who used an alpha version and compared it with ROS1. Messages with sizes between 256 B and 4 MB were published at a 10 Hz rate. The authors also vary the DDS middleware used by ROS2, i.e. RTI Connext, OpenSplice, and FastRTPS. The number of nodes that subscribe to a certain topic another node publishes to were varied as well. Latency, throughput via Ethernet, memory consumption, and threads per node were chosen as measurement metrics. Additionally, the overhead entailed by ROS2 to the latency is evaluated. They emphasize the importance of the message size for end-to-end latencies via Ethernet with a constant latency and ROS2 overhead until 64 KB with the QoS policy being the major part. Further, they recommend a fragment size of 64 KB that is used by the DDS middleware UDP protocol.

The results obtained were confirmed by [17] who evaluated ROS2 for autonomous cars on two KIA Niros with respect to real-time capability. They figured out that the results depend on the Linux kernel. Their software architecture is composed of 14 nodes on i7 Intel NUC. Messages were sent at a frequency of 33 Hz and statistics of the time difference between two communicating nodes were calculated. By using a real-time kernel for Linux the jitter of the time difference could be significantly reduced. Using this information, the overall car behavior was analyzed under different scenarios.

The first official ROS2 release in December 2017, Ardent Apalone, was evaluated by [18] using a real-time Linux kernel over Ethernet. By varying system loads, e.g. CPU load and concurrent traffic, round-trip latency was measured. One node was launched on a normal computer, another node was launched on an embedded device. A message was sent from the normal computer to the embedded device that sends the message back. The authors point out how this round-trip latency is influenced by the network and OS stack.

The previous papers emphasize the influence of various QoS parameters and DDS middlewares on the latency and throughput. [19] propose an architecture that dynamically binds different DDS middlewares to ROS2 nodes to exploit performance benefits of the DDS middlewares under varying circumstances.

[20] focused on improving the Inter Process Communication and compared it to the ROS2 implementation obtaining a lower latency and higher throughput. [21] evaluated latency and throughput in combination with encryption.

A. Contribution

To the best of the author's knowledge, [16] did the most comprehensive evaluation from a pure ROS2 perspective. Yet, the authors used a ROS2 alpha version and there have been a couple of releases in the meantime. Although [17] did evaluations on realistic hardware, they only provided a brief, very high-level evaluation of their software architecture. The evaluations were mostly constrained to two or three nodes, i.e. it was not evaluated if the results can be scaled to a node system comprised of many nodes.

In this paper, we answer that question and provide the user with some guidelines to go along if the latency of a ROS system is to be decreased. For that, we perform an intra-layer profiling to concisely evaluate the bottlenecks in the communication in order to show possible implementation improvements for future ROS2 releases. Parameter values, such as QoS parameters, publisher frequency, etc. will be derived from a use case of a modular control loop. We do not focus on the DDS middlewares as there are benchmarks on the homepage of the vendors. In addition, configuration strongly depends on the use case.

III. Methodology

At first, we will define our use case in Sec. III-A, which is followed by the description of the parameter space in Sec. III-B. Afterwards, we will introduce measurement metrics and present our evaluation framework in Sec. III-C and Sec. III-D.

A. Use Case

We focus on the use case of a robot platform that is modular in terms of hardware, i.e. different sensors, and software, i.e. different data processing nodes. As pointed out in [9], the ROS system is meant to be composed of multiple nodes. Therefore, each sensor is represented by a node that accesses the sensor hardware and publishes a message containing the sensor readings. We assume subsequent nodes that post-process the data and afterwards pass it to an estimator, followed by a control loop, and afterwards to the actuator. To generalize such system, we define a setup that passes a message from a starting node, in this case the sensor, through a couple of nodes, to the end node, namely the actuator. Due to the modularity, we assume different sensors, e.g. an IMU, a LiDAR, a camera, etc. with different sampling rates and data sizes. Fig. 2 visualizes the setup.

B. Parameter Space

The sampling frequency of the sensor is a variable parameter. Assuming that the sensor reading node only reads the sensor data and immediately publishes the message, this corresponds to the *publisher frequency*. The data size of the sensor reading is variable as well. In the ROS system, the *payload* of the message represents this quantity. The *number of nodes*, which includes the start and end node in a data-processing pipeline can be modified as well. Moreover, we can change the Quality of Service settings. The aforementioned parameters remain constant throughout the data-processing pipeline.

C. Measurement Metrics

Latency is our main measurement metric. In accordance with the previous papers mentioned in Sec. II, the latency is defined between publishing a message and receiving the message, i.e. the call of the callback of the subscriber.

The focus of the evaluation is the latency entailed by the ROS system and not by the DDS middleware. Therefore, we want to *profile* the call stack between publishing and subscriber callback for evaluating the overhead of ROS2 core and the middleware interfaces. As statistical quantity, we choose median as it is also used by [16] and resilient to outliers.

D. Evaluation Framework

We found two ROS2 evaluation frameworks available that are actively maintained: `ros2-performance` by irobot-ros [22] and `performance_test` by ApexAI [23]. `performance_test` benchmarks only ROS2 Dashing Diademata by default whereas `ros2-performance` supports later ROS2 versions as well. Further, it is based on `performance_test` and was also used for evaluating the intra-process communication of ROS2 [24]. The focus is more on defining a node system and evaluating the ROS2 node system from a ROS2 perspective as opposed to `performance_test` that aims at fine-tuning QoS parameters and the DDS middleware. Therefore, we forked `ros2-performance`. All repositories related to our evaluation can be found on GitHub [25].

Fig. 3 depicts the implementation of an intra-layer profiling for ROS2 messages. The implementation is bundled within a `docker` container containing the custom ROS2 build. The latency overhead of `docker` is negligible [26] provided that it is run in the `host` network. For the implementation, we added a `ros2profiling` package providing functions to generate timestamps since the Epoch in nanoseconds. It patches the ROS2 source tree to include timestamps into processed messages. Therefore, a minimum message size of 100 B needs

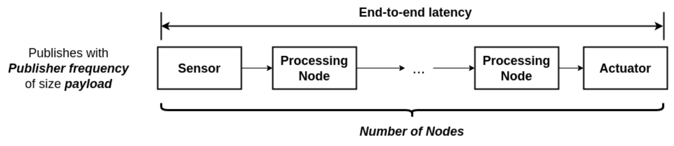

Fig. 2: Visualization of evaluation setup. Parameters are highlighted in italic.

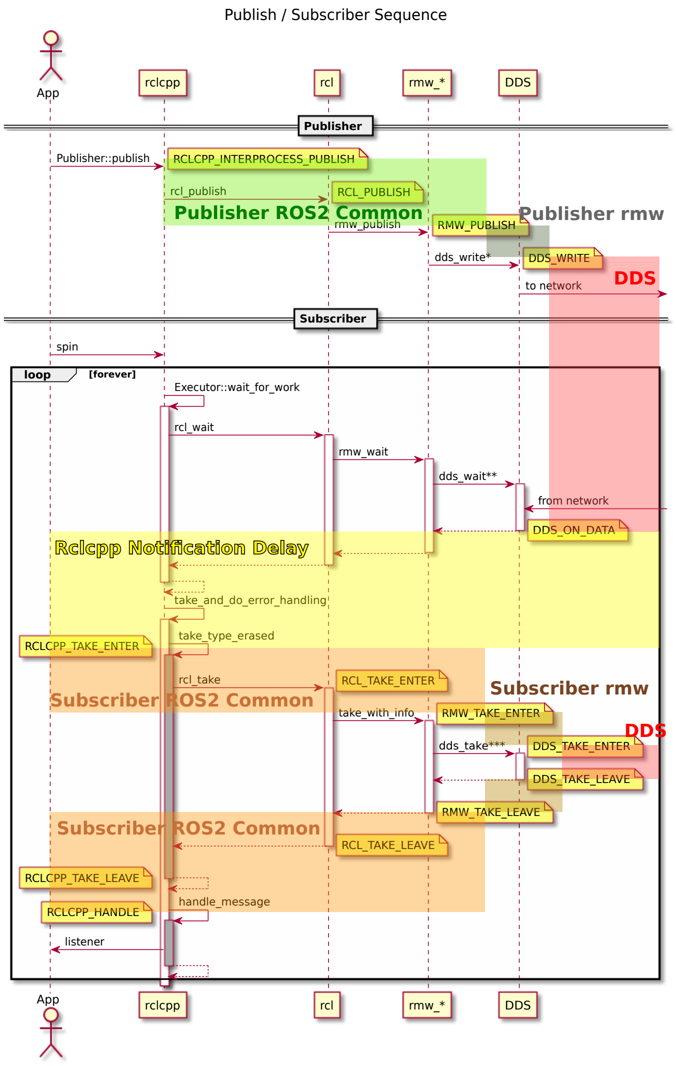

Fig. 3: Sequence diagram of necessary `rmw` adaptions to enable profiling. Notes indicate the layers. Colored boxes highlight the profiling categories.

to be ensured. The timestamps correspond to the position of the notes in Fig. 3 in the ROS2 callstack. Timestamps are recorded when entering and leaving the method of the layer. Arrow annotations denote the called function and asterisks indicate placeholders for the actual function name, as this depends on the middleware. Timestamps were categorized into the following categories:

- **DDS**: This category only contains latency entailed by the DDS transport via network and by the function call to actually receive the message.
- **Subscriber `rmw`** and **Publisher `rmw`**: Latency attributed to the `rmw` layer. Middleware-specific conversions from ROS2 messages to DDS messages that might require DDS utility functions but are not used for the transport of the message itself are contained in this category.
- **Publisher** and **Subscriber ROS2 Common**: Overhead entailed by ROS2 that is independent of the middleware.

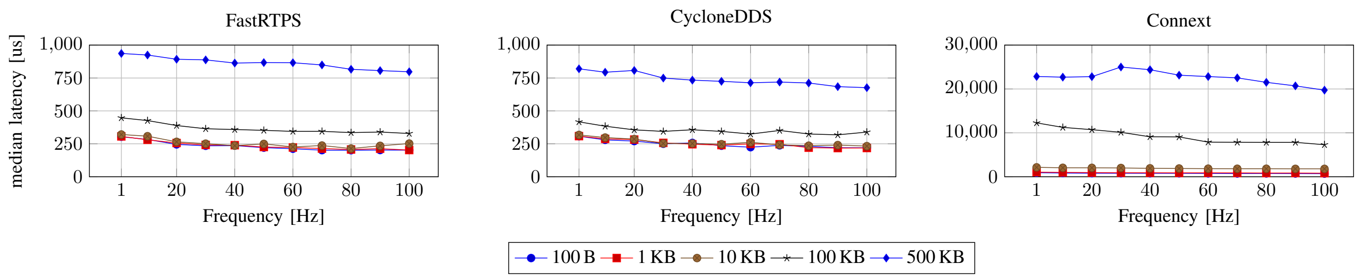

Fig. 4: Investigating the influence of publisher frequency on latency with three nodes. Evaluation is performed on the desktop PC, QoS reliability is set to `BEST_EFFORT`. Note the different scaling of the a y-axis for Connext.

- **Benchmarking**: This latency is entailed by code used for benchmarking the latency.
- **Rclcpp Notification Delay**: The time difference between the notification of DDS to ROS2 that new data is available and triggering of its actual retrieval.

Latency is measured on one machine, as we are interested in the ROS2 overhead and not on the over-the-network performance, which is dictated by DDS itself. Nodes of the data-processing pipeline are created in the same process but in different threads. Each node is associated with its own `StaticSingleThreadedExecutor` spinning in its own thread.

IV. Evaluation

We compare two different hardware settings, namely a desktop PC, with an Intel i7-8700 CPU @ 3.2 GHz x 6 and 32 GB RAM. In addition, we use a Raspberry Pi 4 Model B Rev 1.1 with 4 GB RAM. Kernel versions are 5.4.0-42-generic for the desktop PC and 5.4.0-1015-raspi for Raspberry Pi. Foxy Fitzroy 20200807 is used as ROS2 version for our evaluation. We set the network device as `localhost` with UDP as transport layer. The kernels were updated to the most recent versions. At the time of writing, the Connext middleware was not available for the Raspberry. Therefore, results can only be obtained for FastRTPS and CycloneDDS. Connext version is 5.3.1, FastRTPS is 2.0.1, and CycloneDDS is 0.6.0.

The variable parameters are listed in Tab. I. We assume a publisher frequency of maximum 100 Hz, which is realistic for e.g. time-of-flight sensors. GPS sensors operate at an update rate of 1 Hz up to 10 Hz. We assume an update rate for cameras up to 100 Hz. Therefore, the publisher frequency is in a range from 1 Hz to 100 Hz with a step size of 10 Hz. As variable QoS policies, we modify the reliability from `RELIABLE` to `BEST_EFFORT`, since they are frequently used. The number of nodes in the data-processing pipeline varies between three and 23 nodes with a step size of two. These values are arbitrarily chosen but inspired by [17] who used 14 nodes for their autonomous car setup.

TABLE I: Variable parameter values.

Publisher frequency	1, 10, …, 90, 100
Payload	100 B, 1 KB, 10 KB, 100 KB, 500 KB
Number of Nodes	3, 5, …, 21, 23
DDS Backend	Connext, FastRTPS, CycloneDDS
Reliability	reliable, best effort

In all cases, we use a time duration for one configuration run of 60 seconds. Depending on the publisher frequency, this results in a varying amount of samples, i.e. the higher the publisher frequency the higher the amount of samples. We discard the first ten samples to mitigate initialization effects.

The default settings of the Linux kernel are highly adaptive to the load and other external factors. In order to inhibit idle and energy saving mechanisms, which add additional noise to the latency measured, we change the kernel parameters: the *scaling governor* is set to `userspace` and core frequencies are set to their maximum value. We deactivate the CPU idle by passing `cpuidle.off=1` as a kernel parameter. The option `no_hz` disables the scheduler tick if no work is to be done on that CPU, which is why we deactivate it. For the desktop PC, we turn off hyperthreading.

A. Evaluation of Publisher Frequency

We use three nodes with a publisher frequency between 1 Hz and 100 Hz as depicted in Fig. 4. This corresponds to a ping-pong scenario. While sticking to the desktop PC as hardware and the QoS reliability to `BEST_EFFORT`, we use all possible payload sizes.

For all DDS middlewares, we can see that the median latency decreases with increased frequency. This might be due to (unknown) energy saving features in other hardware devices or reprioritized thread scheduling. Further, we can see that the obtained latencies for 100 B, 1 KB, 10 KB are equal, but increase for 100 KB and 500 KB. This can be explained by the maximum UDP packet size of 64 KB which incurs fragmentation for large payloads. This additional overhead differs with the employed middleware. FastRTPS is slighty slower than CycloneDDS, whereas Connext entails the highest latency. Similar results were obtained for Raspberry Pi.

In discussions with maintainers of the Connext middleware, we found out that the `rmw` is not maintained by RTI itself. We were told that the implementation is highly suboptimal, i.e. lots of calculations and copy instructions need to be performed if a ROS2 message is converted to a DDS message. Further, the node to participant mapping, which results in a high resource consumption if nodes are created in the same process, was only optimized for the `rmw` of FastRTPS and CycloneDDS [27].

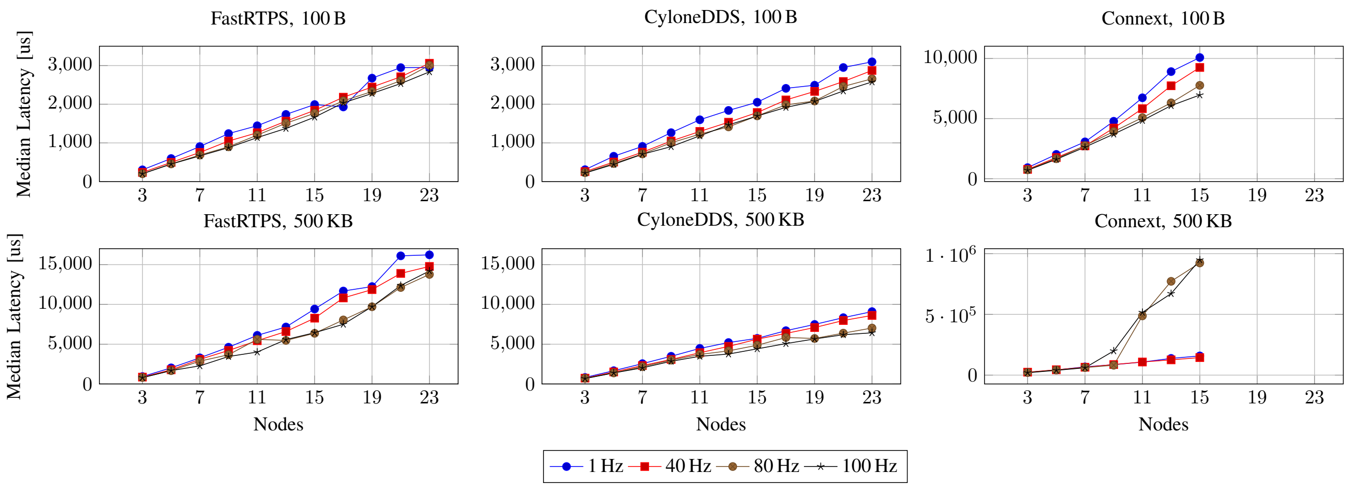

Fig. 5: Investigation of the scalability of a node system. We used a payload of 100 B (upper row) and of 500 KB (lower row). The DDS middleware was varied. For visualization purposes, only a few frequencies were picked. Evaluation was performed on the desktop PC with QoS-reliability `BEST_EFFORT`. Note the different y-axis scaling for Connext.

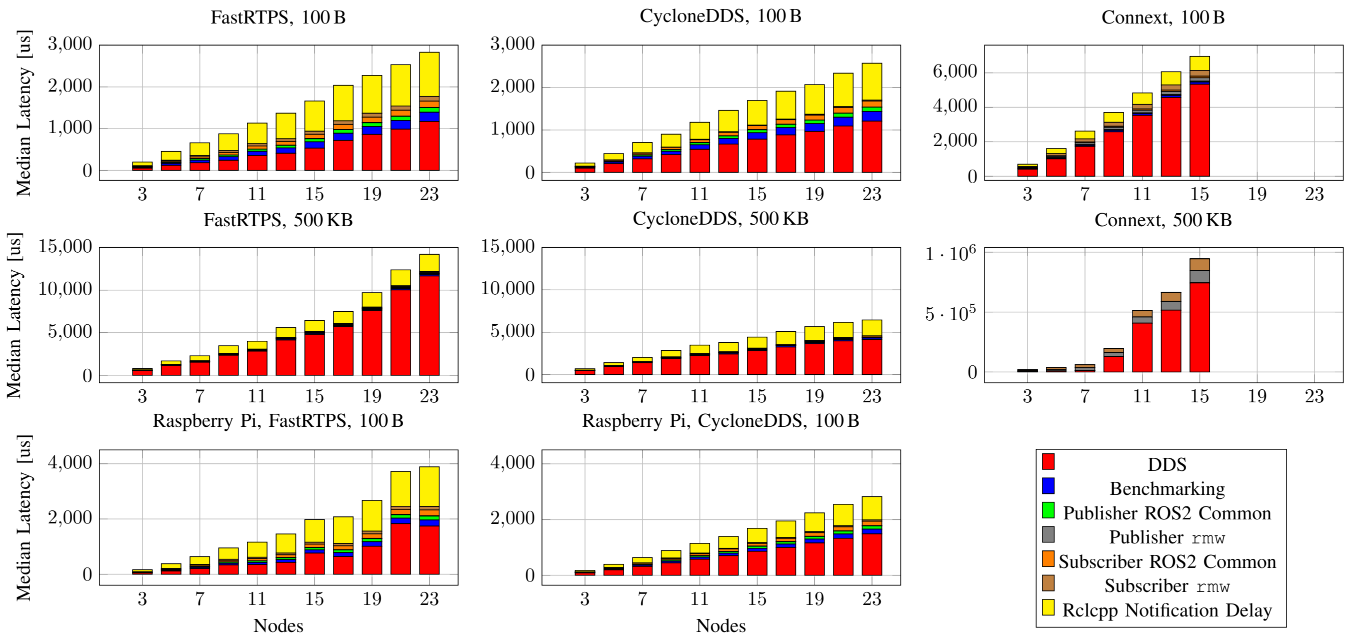

Fig. 6: Categorization of intra-process profiling of different latencies. Evaluation was performed on the desktop PC with QoS reliability `BEST_EFFORT`. Note the different scaling of the y-axis for Connext. The frequency is 100 Hz.

Therefore, higher latencies were already expected prior to the evaluation. We were told that significant improvements are to be expected for the upcoming Connext `rmw` release, which is in early release testing.

B. Evaluation of Scalability

We evaluate the median latency from starting to end node for the data-process pipeline ranging from 3 to 23 nodes. As payload, we use 100 B and 500 KB. For visualization purposes, we restrict ourselves to a subset of the evaluated frequencies. Results are shown in Fig. 5.

We can observe the same pattern as in Fig. 4: the latency decreases if the frequency is increased. This effect is intensified with the length of the data-processing pipeline. For a payload of 100 B, the results indicate a linear relationship between the number of nodes and the median latency for FastRTPS and CycloneDDS. The outliers may occur due to too few samples, which is the case for lower frequencies. It needs to be further investigated if the relationship will be more linear if the number of samples is larger. Similar results were observed for the Raspberry Pi. However, the graph for CycloneDDS is more linear. Reasons can be a simpler hardware, which is not as adaptively controlled to load as the desktop PC.

In the case of Connext, a nonlinear relationship for smaller frequencies is visible. Furthermore, Connext only yields results until 15 nodes, as for higher node numbers the Connext `rmw` raises exceptions. This is independent of the parameter set.

For 500 KB, we can observe a more erratic behavior in the case of FastRTPS. A nonlinear increase of the median latency can be suspected. In order to further investigate this assumption, the number of nodes needs to be increased in future works. In the case of CycloneDDS, there seems to be a saturation for 80 Hz and 100 Hz, which also needs to be verified in the future. Connext yields a strong nonlinear relationship for frequencies of 80 Hz and 100 Hz. Last but not least, the lowest median latency was measured for CycloneDDS.

C. *Profiling*

For a payload of 100 B, the largest amount of latency can be attributed to the categories **DDS** and **Rclcpp Notification Delay** as seen in Fig. 6. Especially for Connext, the largest portion of the latency can be attributed to the DDS middleware itself. However, the overhead of ROS2 compared to raw DDS amounts up to 50 % for small messages.

For a payload of 500 KB, one can clearly see that for all middlewares, the major part of the median latency is due to the DDS middleware. In the case of FastRTPS, the latency seems to increase nonlinearly. In addition, we can observe that the median latency entailed by **Rclcpp Notification Delay** is higher than in the case of 100 B, i.e it is payload-dependent. The erratic behavior of Connext can be mainly attributed to the DDS middleware, but also to the categories **Subscriber `rmw`** and **Publisher `rmw`**.

As we focus on possible overhead reductions of ROS2 in this paper, we assume that the DDS middlewares cannot be changed. Thus, we can observe that major performance improvements can be obtained for the category **Rclcpp Notification Delay**. Similar results could be obtained for the Raspberry Pi with a higher latency, cf. Fig. 6.

D. *Influence of QoS Reliability*

In the last sections, the QoS reliability policy was set to `BEST_EFFORT` and kept as a constant parameter. Because of the use of `localhost` as network device, the network is not lossy, i.e. the influence of the policy will not be immediately visible. Therefore, we pick the highest possible throughput and calculate the relative deviation between the median latencies obtained with the QoS policy `BEST_EFFORT` and `RELIABLE`. As can be seen in Fig. 7, no trend is actually visible, i.e. we cannot simulate a lossy network with the available parameter sets. Similar results were obtained for the Raspberry Pi.

V. Conclusion and Future Outlook

The goal of this paper was to provide the reader with simple guidelines if ROS2 is used for time-critical systems. Given our parameter set, we discovered the following rules of thumb:

- With a payload higher than the fragmentation size of UDP (here, 64 KB), latency increases with the payload size.
- The higher the frequency, the lower the latency.

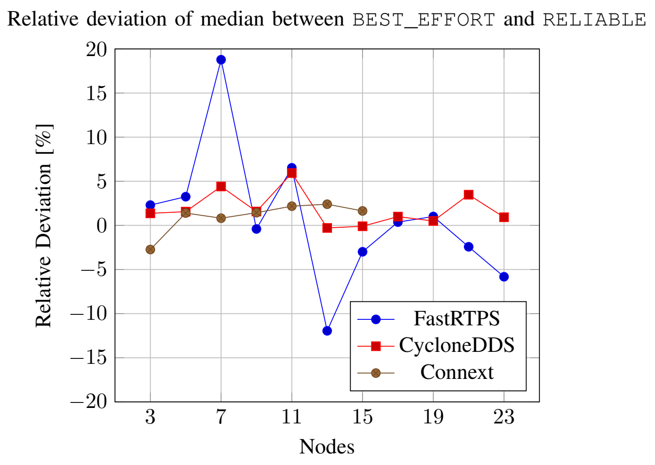

Fig. 7: Evaluation of influence of QoS reliability settings on median latency. For FastRTPS and CycloneDDS, we choose 500 KB and 100 Hz. In the case of Connext, we have a payload of 500 KB and a publisher frequency of 40 Hz.

- CycloneDDS yields the lowest latency.
- The DDS middleware and the delay between message notification and message retrieval by ROS2 contribute the biggest portions to the overall latency.
- The Connext `rmw` is highly suboptimal. In later releases, this will most likely change.
- Latency is larger on Raspberry Pi, however the qualitative results are the same. Fluctuation in latency is less compared to the desktop PC.

During our evaluation, we observe that latency highly depends on energy saving features of the OS and the hardware. Our main focus was on the CPU. However, energy saving features of the NIC, e.g., might play an important role. This needs to be taken carefully into consideration if latency is to be mitigated for real-time critical applications.

For the evaluation of network-independent ROS2 overhead, we created the nodes in one process on the same machine. This use case is unrealistic as Intra-Process Communication would normally be used as this approach is much more efficient. As we use separate executors per node, there should not be much difference between creating the nodes in one process as opposed to creating nodes in separate processes. However, as pointed out by [27], the node to participant mapping is highly suboptimal in the Connext `rmw`. This might be the reason for the bad performance. Additionally, the Connext `rmw` is highly suboptimal in general as thoroughly explained in Sec. IV-A. This will be fixed in future releases as discussed with RTI.

In future work, one might consider an evaluation in a more realistic setup, i.e. with distributed systems. Network effects could be better evaluated. An effect of the QoS reliability possibility should be observable. Aside from the median, one could evaluate other statistical quantities or verify if the messages follow a certain distribution. The obtained information could then be used as an additional uncertainty for state estimation and incorporated into the Kalman Filter.

References

[1] OSRF, Community Metrics Report, 2019 (accessed August 17, 2020). [Online]. Available: http://download.ros.org/downloads/metrics/metrics-report-2019-07.pdf

[2] ——, ROS Robots, 2020 (accessed August 17, 2020). [Online]. Available: https://robots.ros.org/

[3] ——, Project Governance, 2020 (accessed August 17, 2020). [Online]. Available: https://index.ros.org/doc/ros2/Governance/#governance

[4] D. Casini, T. Blaß, I. Lütkebohle, and B. B. Brandenburg, "Response-Time Analysis of ROS 2 Processing Chains Under Reservation-Based Scheduling," in 31st Euromicro Conference on Real-Time Systems (ECRTS 2019), ser. Leibniz International Proceedings in Informatics (LIPIcs), S. Quinton, Ed., vol. 133. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2019, pp. 6:1–6:23. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2019/10743

[5] LGSVL, LGSVL Simulator, 2020 (accessed August 17, 2020). [Online]. Available: https://www.lgsvlsimulator.com/

[6] gazebo_ros2_control, 2020 (accessed August 17, 2020). [Online]. Available: https://github.com/gazebo_ros2_control

[7] OMG, Data Distribution Service, 2015 (accessed August 17, 2020). [Online]. Available: https://www.omg.org/spec/DDS/

[8] OSRF, Why ROS 2?, 2020 (accessed August 17, 2020). [Online]. Available: https://design.ros2.org/articles/why_ros2.html

[9] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in ICRA workshop on open source software, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[10] OSRF, ROS on DDS, 2019 (accessed August 17, 2020). [Online]. Available: https://design.ros2.org/articles/ros_on_dds.html

[11] ——, About different ROS 2 DDS/RTPS vendors, 2020 (accessed August 17, 2020). [Online]. Available: https://index.ros.org/doc/ros2/Concepts/DDS-and-ROS-middleware-implementations/

[12] M. Naumann, F. Poggenhans, M. Lauer, and C. Stiller, "CoInCar-Sim: An Open-Source Simulation Framework for Cooperatively Interacting Automobiles," in 2018 IEEE Intelligent Vehicles Symposium (IV), 2018, pp. 1–6.

[13] M. A. Lema, A. Laya, T. Mahmoodi, M. Cuevas, J. Sachs, J. Markendahl, and M. Dohler, "Business case and technology analysis for 5G low latency applications," IEEE Access, vol. 5, pp. 5917–5935, 2017.

[14] S. Maheshwari, D. Raychaudhuri, I. Seskar, and F. Bronzino, "Scalability and Performance Evaluation of Edge Cloud Systems for Latency Constrained Applications," in 2018 IEEE/ACM Symposium on Edge Computing (SEC), 2018, pp. 286–299.

[15] F. Voigtländer, A. Ramadan, J. Eichinger, C. Lenz, D. Pensky, and A. Knoll, "5G for Robotics: Ultra-Low Latency Control of Distributed Robotic Systems," in 2017 International Symposium on Computer Science and Intelligent Controls (ISCSIC), 2017, pp. 69–72.

[16] Y. Maruyama, S. Kato, and T. Azumi, "Exploring the performance of ROS2," in Proceedings of the 13th International Conference on Embedded Software - EMSOFT '16. Pittsburgh, Pennsylvania: ACM Press, 2016, pp. 1–10.

[17] M. Reke, D. Peter, J. Schulte-Tigges, S. Schiffer, A. Ferrein, T. Walter, and D. Matheis, "A Self-Driving Car Architecture in ROS2," in 2020 International SAUPEC/RobMech/PRASA Conference, Jan. 2020, pp. 1–6.

[18] 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:1809.02595 [cs], Sep. 2018, arXiv: 1809.02595. [Online]. Available: http://arxiv.org/abs/1809.02595

[19] R. Morita and K. Matsubara, "Dynamic Binding a Proper DDS Implementation for Optimizing Inter-Node Communication in ROS2," in 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Aug. 2018, pp. 246–247, iSSN: 2325-1301.

[20] Y.-P. Wang, W. Tan, X.-Q. Hu, D. Manocha, and S.-M. Hu, "TZC: Efficient Inter-Process Communication for Robotics Middleware with Partial Serialization," in 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Nov. 2019, pp. 7805–7812, iSSN: 2153-0866.

[21] J. Kim, J. M. Smereka, C. Cheung, S. Nepal, and M. Grobler, "Security and Performance Considerations in ROS 2: A Balancing Act," arXiv:1809.09566 [cs], Sep. 2018. [Online]. Available: http://arxiv.org/abs/1809.09566

[22] iRobot, ros2-performance, 2020 (accessed October 10, 2020). [Online]. Available: https://github.com/irobot-ros/ros2-performance

[23] ApexAI, performance_test, 2020 (accessed October 10, 2020). [Online]. Available: https://gitlab.com/ApexAI/performance_test/

[24] O. Robotics, Intra-process Communications in ROS 2, 2020 (accessed August 17, 2020). [Online]. Available: http://design.ros2.org/articles/intraprocess_communications.html

[25] Barkhausen-Institut, Benchmarking, 2020 (accessed October 10, 2020). [Online]. Available: https://github.com/orgs/Barkhausen-Institut/projects

[26] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015, pp. 171–172.

[27] OSRF, Node to Participant mapping, 2020 (accessed August 17, 2020). [Online]. Available: http://design.ros2.org/articles/Node_to_Participant_mapping.html