2016-06-10
EMSOFT 2016 Submission #83
Title: Exploring the Performance of ROS2

Dear Reviewers,

We highly appreciate the insightful suggestions and detailed valuable comments on our paper. The suggestions of the reviewers are very helpful for us and the suggestions are now incorporated in the revised paper as follows. We have attached the last paper review and revised paper in our reply letter. In the revised paper, newly added and updated sentences are written in the red colored font so that the reviewers can easily find them. We hope the reviewers will be satisfied with our replies to the comments and the revised paper.

Yours sincerely,
Authors

1 Response to 1st reviewer

1.1 Premature ROS2 for evaluation

- **Comment:**

  ROS2 is currently still being actively developed (Section 3) and as such, an analysis of the performance at this stage seems a pre-mature. Certain capabilities in ROS2 are still not available and/or do not perform up to expectations. In Section 3.2 the author claims that "DDS is not designed to handle large data", however DDS has API's specifically designed for use with large data packets. While DDS has these API 's for assisting with large packet transfer, they are not yet compatible with ROS2 and therefore the performance drops after the size of packets increases beyond 64KB.

  Our reply:

  Thank you very much for your vital comment. It is true that ROS2 in under heavy development and a very rough draft in this stage. However, this paper aims to conduct proof of concept for DDS approach to ROS and clarifies the needs of abstracting DDS API for large packets. Regardless of development stage of ROS2, we consider that understanding each DDS characteristics are meaningful for ROS2 users, which we provide in this paper. What this paper provides is not simple comparison between ROS1 and ROS2 but DDS characteristics through ROS2. ROS2 is the one of sysytems using DDS. We believe the contributions of this paper are meaningful even when ROS2 is under development. To clarify this contributions, we have highlighted that this paper aims to proof of concept and arranged premature points of current ROS2 for future development.

  Updated contributions about proof of concept in "1. INTRODUCTION" section

  Contribution: In this paper, we provide proof of concept for DDS approach to ROS. We clarify the performance of the data transport for ROS1 and ROS2 in various situations. Performance means latencies characteristics, throughput and distributed capability. Focusing on the DDS capabilities, depending on DDS vendor and configuration, we explore and evaluate the potential and constraints from various aspects: latencies, throughput, the number of threads, and memory consumption. From experimental results, we arrange guidelines and what we can do to solve current constrains. To the best of our knowledge, this is the first study to explore ROS2 performance.

  Added sentences about improvements for ROS2 in "3.7 Lessons Learned" section

  Since ROS2 is under development, we have clarified room for improvement of ROS2 performance and capability to maximize DDS potential. First, *QoS Policies* supposed by ROS2 provide fault tolerance but they are insufficient for real-time processing. ROS2 has to expand the scope of supported *QoS Policies*. Second, for small embedded system, ROS2 needs a minimum DDS implementation and minimum abstraction layer. For example, we need C API library for ROS2 and a small DDS implementation. ROS2 easily supports them because of its abstraction layer. Third, we also clarify a need of alternative API for large *message* to manage divided packets. This is critical to handle large message. Abstraction of this will shorten DDS end-to-end latencies and fulfill deficiency of Table 4. Finally, we must tune DDS configurations for ROS2 because there are numerous vendor specific configuration options.

1.2 Narrow scope of experiments

1. **Comment:**

   The designed experiment only covers latency rates between the two versions of ROS, but there are still other aspects to communications performance. Further research should be conducted to test the throughput, fault tolerance and distributed capabilities of the two.

   Our reply:

   Thank you very much for your vital comments. In the revised paper, we have conducted additional evaluations from various aspects. One is throughput evaluations in "3.4 Throughput of ROS1 and ROS2" section. We have clarified throughput characteristics depending on DDS vendors. Another additional evaluation is measurements of latency for a multiple destinations publisher. This is described in "3.3.4 Multiple Destinations Publisher in local cases" section and will be utilized for fault tolerance and distributed capabilities. From this experiment, we can learn fair latency which DDS brings to ROS. In addition, we prepare *-depth policy and have varied QoS setting by configuring depth option in "3.3.3 Comparison within ROS2" section. The other additional evaluations also expand our performance evaluation. We have conducted measurements for the number of thread in "3.5 Thread of ROS1 and ROS2" and shared library memory consumption in "3.6 Memory consumption of ROS1 and ROS2" section. These experimets provide us insight for embedded systems and distributed capabilities. For precise added sentences, please go on reading following our replies and the revised paper.

2. **Comment:**

   Interacting with multiple devices or experimentation with a real-time application would provide insight into the fault-tolerance and distributed capabilities that DDS brings to ROS.

   Our reply:

   Thank you for your advice. To consider multiple devices, we have conducted evaluations for a multiple destinations publisher. Much of the information is often shared in real applications such as robots. In our evaluation, we prepare five subscribers and measure their end-to-end latencies. The result provides insight for distribute capabilities. Its precise analysis are described in the following paragraph.

   Added sentences in "3.3.4 Multiple Destinations Publisher in `local` cases" section

   In this section, we prepare five *subscriber-nodes* and measure latencies of each *node*. Much of information shared in real applications is destinated to multiple destinations. Hence, this evaluation is practical for user. Figure 17 shows latencies of ROS1. We can observe significant differences between *subscriber-nodes*. This means ROS1 schedules *message* publication in order and is not suitable for real-time systems. For example, in 1 MB, subscriber 5 is about twice as mach as subscriber 1. In contrast, ROS2 has small differences as shown in Figure 18. All subscribers' behavior is fair in ROS2. However, ROS2 latencies significantly depends on the number of packets. This is same characteristic we learned from Figure 8. Figure 19 indicates fair latencies and dependency of packets. Although we cannot say that latencies variance of ROS1 is larger than one of ROS2 due to the difference of the scale, Figures 17, 18, and 19 prove ROS2 *message* publication is more fair to multiple subscribes than ROS1 one.

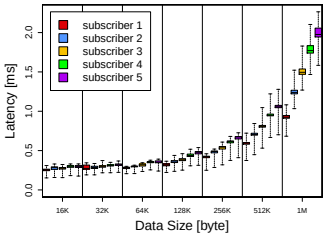


Figure 17: (1-b) ROS1 multiple destinations publisher.

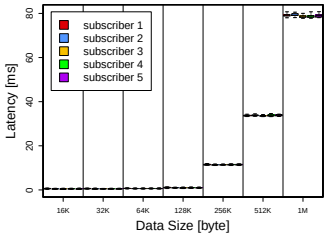


Figure 18: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

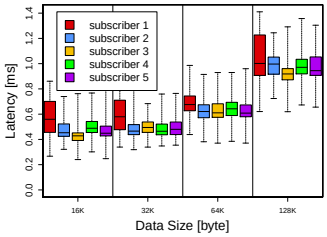


Figure 19: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

3. **Comment:**

   Results from an application would provide insight into how the overhead of DDS scales with increased network load.

   Our reply:

   Thank you very much for your thoughtful suggestion. We have additionally conducted evaluation of a multiple destinations publisher for increased network load in "3.3.4 Multiple Destinations Publisher local cases" section. This evaluation also provides insight for the overhead of DDS scales as shown in 18 and 19. However, we did not conduct further evaluations for real-time guarantees with constrained network because QoS Policies supposed by ROS2 are insufficient for real-time processing. After support of real-time QoS, we will conduct above evaluations with configuring QoS variables for real-time systems. We have explicitly described this fact as future work.

   Updated sentences about future work in "5. CONCLUSION" section

   In future work, we will evaluate real-time applications such as an autonomous driving vehicle as case studies using ROS2. ... Since ROS2 is under development, we must maximize DDS potential by tuning and abstracting more *QoS Policies* for real-time processing and DDS configurations.

4. **Comment:**

   Another possible avenue of research is a more detailed analysis of the effects on performance of varying QoS policy variables such as deadline scheduling and history.

   Our reply:

   Thank you for your advice. At present, ROS2 only supports a few QoS Policies. For example, DEADLINE is only calculated by a node's period and is not utilized for scheduling. In this condition, we prepare *-depth policy and have varied history setting by configuring depth option. This HISTORY QoS Policy provides applications fault tolerance. We have added sentences in "3.3.3 Comparison within ROS2" section and described analysis there.

   Updated sentences in "3.3.3 Comparison within ROS2" section

   In addition, the influence of the *QoS Policy* on end-to-end latencies is evaluated in (2-b) OpenSplice with the `reliable policy`, `best-effort policy`, and `*-depth policy`. `*-depth policy` is prepared for this evaluation and configured by depth as shown in Table 5.

Added sentences about additional QoS in "3.3.3 Comparison within ROS2" section

Figure 15 shows no differences depending on the depth of `*-depth policy`. These *QoS policies* are different in the number *nodes* save `messages`. Although this number influences resources, this does not affect latencies because archiving *messages* is conducted in every publication.

Table 5: Depth Configurable QoS Policies

	`*-depth policy`
DEADLINE	`100 ms`
HISTORY	`LAST`
depth	1, 10, or 100
RELIABILITY	`RELIABLE`
DURABILITY	`TRANSIENT_LOCAL`

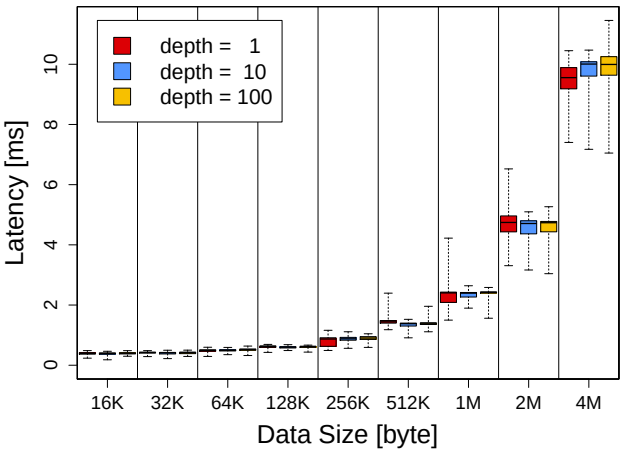


Figure 15: (2-b) Configured `*-depth policy` in ROS2 with OpenSplice

5. **Comment:**

   The experiment in section 3.1 should be kept as it is a good evaluation of overhead and is reproducible since it was performed on the loopback interface of the device, but experiments measuring latency and throughput in an application environment would strengthen the paper.

   Our reply:

   Thank you for your thoughtful comment. Following your advice, we have additionally conducted a throughput evaluation in Section 3.4. Figure 20 clarifies the overhead data for DDS transaction. Figure 21 shows that throughput is limited by the 100 Mbps Ethernet network and not by DDS.

Added "3.4 Throughput of ROS1 and ROS2" section

We also measure each throughput of ROS1 and ROS2 in the `remote` case. In our one-way *message* transport experiment, maximum bandwidth of the network is 12.5 MB/sec because we use 100 Mbps Ethernet (100BASE-TX) and Full-Duplex as shown in Table 2. Nodes repeatedly transport each *message* with 10Hz.
In small data from 256 B to 2 KB, we can observe a constant gap among ROS1, ROS2 with OpenSplice, and ROS2 with Connext from Figure 20. These additional data correspond with RTPS packets for *QoS Policy* and heartbeat. Hence, these gap does not depend on data size. Moreover, Connext throughput is lower than OpenSplice one. This becomes a big impact when users handle many kinds of small data with high Hz and/or network bandwidth is limited.
In large data from 2 KB to 4MB, curves of Figure 21 demonstrate sustainable theoretical throughput. ROS2 and ROS2 is able to utilize all of available bandwidth and similarly behave in this situation. Throughput is limited by the network and not by DDS.

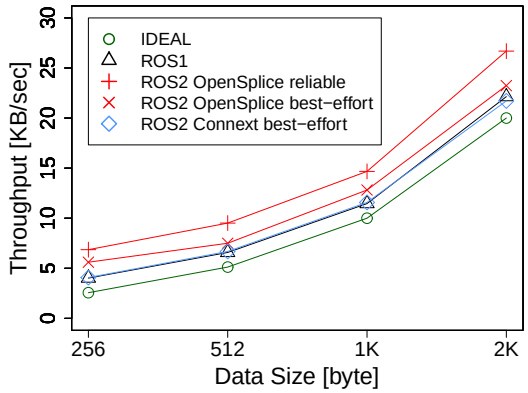


Figure 20: (1-a) and (2-b) `remote` cases throughput with small data

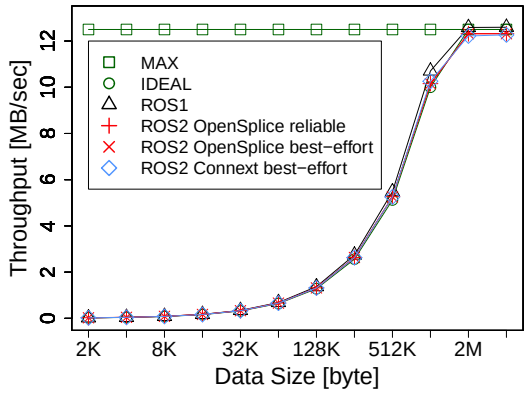


Figure 21: (1-a) and (2-b) `remote` cases throughput with large data

1.3 Unclear contributions

1. **Comment:**

   The contributions of the paper are essentially experimental results, but these results themselves are not sufficient. The authors do not provide any guidelines, lessons learned, or insight that they gained from their experiments. Some examples of interesting questions to be answered from the data collected by the authors include:

   Our reply:

   Thank you very much for pointing out our insufficient contributions and suggesting a lot of hint for lessons. To reply this comment and answer following interesting questions, we create "3.7 Lessons Learned" section. In this section, we show guidelines to highlight our contributions from experiments. For precise sentences, please go on reading following our replies and revised paper. We hope you will be satisfied with our replies.

2. **Comment:**

   Should ROS1 ever be used over ROS2, or do the benefits of ROS2 outweigh the costs?

Our reply:

Thank you for suggesting a good question. ROS1 has relatively small latency, small overhead throughput, various packages, and rich tools. In contrast, ROS2 is under development and does not abstract some DDS APIs and QoS Policies. However, ROS2 supports some *QoS Policies* and does not need master-node. This is important in terms of fault tolerance. Moreover, ROS2 will support RTOS and light DDS implementation for real-time embedded systems. We consider these benefits outweigh the cost and recommend the both with ros_bridge. We have added sentences about this in "3.7 Lessons Learned" section to answer your comment. However, we must note that this paper does not simply compare ROS1 and ROS2. This paper conducts proof of content for DDS approach to ROS and clarify DDS potential.

Added sentences about benefit of ROS2 in "3.7 Lessons Learned" section

DDS brings supports real-time embedded systems to ROS2. We believe ROS2 outweigh its cost for using DDS. Fault tolerance of DDS is superior because it is able to save past data with *QoS Policy* and does not have a master *node*. DDS guarantees fair latencies as shown in Figure 19. In addition, DDS is able to run on multiple platforms include RTOS and switch DDS implementation as needed. Under RTPS protocol, any ROS2 *nodes* communicate with each other without relation to its platform.

3. **Comment:**

What underlying implementation of DDS should people use, and why, or under what circumstances would one be better than the other?

Our reply:

Thank you for suggesting a good question. We recommend each DDS implementation depending on circumstances. In the local cases, OpenSplice is superior to others due to its capability and low latency caused by many threads. In the remote cases, Connext is superior because difference of latency is relatively small and we must consider bandwidth. Connext's throughput is minimum as shown in Figure 20. For embedded systems, regarding memory and thread, we consider that FastRTPS is suitable from Tables 6 and 7. We have added theses guidelines in "3.7 Lessons Learned" section to answer this question.

Added sentences about ROS2 guidelines in "3.7 Lessons Learned" section

DDS supports *QoS Policy* but there is trade-off of end-to-end latencies and throughput. In the `local` case, overhead latencies of ROS2 is not trivial. From Section 3.3, the latencies is caused by two data conversions for DDS and DDS transaction. DDS end-to-end latencies is constant until *message* data size is lower than maximum packet size (64 KB) as shown in Figure 9. On the other hand, as one large *message* is divided into several packets, the latencies sharply increases as show in Figures 10 and 18. Whether *message* data size is over 64 KB or not is important issue especially in DDS because management of divided packets with QoS Policy needs significant processing time and alternative APIs provided by some vendors. We should understand influence of divided packets and keep in mind this issue when using DDS. While DDS and ROS2 abstraction have overhead latencies, OpenSplice utilizes a lot of threads and processes faster than Connext as shown in Figure 13. This is a reason why we currently should use OpenSplice in the underlying implementation of DDS in the `local` case. In the `remote` case, although overhead latencies is trivial, we must consider throughput for bandwidth. As shown in 20, Connext is superior to OpenSplice in terms of throughput. This constant overhead throughput is predictable and exists no matter how small *message* data size is. It influences especially when many kinds of topic are used with high Hz. We recommend Connext to consider minimum necessary throughput in the `remote` case.
DDS brings supports real-time embedded systems to ROS2. We believe ROS2 outweigh its cost for using DDS. Fault tolerance of DDS is superior because it is able to save past data with *QoS Policy* and does not have a master *node*. DDS guarantees fair latencies as shown in Figure 19. In addition, DDS is able to run on multiple platforms include RTOS and switch DDS implementation as needed. Under RTPS protocol, any ROS2 *nodes* communicate with each other without relation to its platform. FastRTPS is currently the best DDS implementation for embedded systems in thread and memory as Table 6 indicates, but it is not suitable for small embedded system.

4. **Comment:**

   What can be done to improve the performance of the interface between ROS2 and DDS, or within the DDS implementations themselves?

   Our reply:

   Thank you for your thoughtful comment. First, ROS2 must expand the scope of supposed QoS Policies for real-time systems. It is a very important issue for a real-time guarantee. Current QoS Policy is insufficient. Second, ROS2 should abstract alternative APIs such as an asynchronous publisher and flow controller for large data transport. This improves DDS performance and reduces overhead latencies. Finally, there are numerous DDS and vendor specific configuration options which might affect its performance. Tuning these configures for ROS2 is to do in order to improve performance within each DDS implementation. We have added some sentences to clarify above things as followed.

Added sentences for improvements of ROS2 in "3.7 Lessons Learned" section

Since ROS2 is under development, we have clarified room for improvement of ROS2 performance and capability to maximize DDS potential. First, *QoS Policies* supposed by ROS2 provide fault tolerance but they are insufficient for real-time processing. ROS2 has to expand the scope of supported *QoS Policies*. Second, for small embedded system, ROS2 needs a minimum DDS implementation and minimum abstraction layer. For example, we need C API library for ROS2 and a small DDS implementation. ROS2 easily supports them because of its abstraction layer. Third, we also clarify a need of alternative API for large *message* to manage divided packets. This is critical to handle large message. Abstraction of this will shorten DDS end-to-end latencies and fulfill deficiency of Table 4. Finally, we must tune DDS configurations for ROS2 because there are numerous vendor specific configuration options.

Added sentences about improvement of ROS2 in "5. CONCLUSION" section

Since ROS2 is under development, we must maximize DDS potential by tuning and abstracting more *QoS Policies* for real-time processing and DDS configurations.

5. **Comment:**

   How does the switch to DDS from pure TCP or UDP protocols affect other performance factors in ROS2 systems?

   Our reply:

   Thank you for your suggestion. In the submitted paper, we only focused on end-to-end latency affected by DDS. In the reviled paper, we show other performance factors by variable aspects. One additional factor is throughput in "3.4 Throughput of ROS1 and ROS2" section. We discuss throughput affected by DDS and clarify overhead throughput of each DDS implementations. Another factor is the number of thread in "3.6 Memory consumption of ROS1 and ROS2" section. The number of used thread are compared depending on DDS implementations.

1.4 Minor points

- **Comment:** Table 4 - ROS1 experiment says 2c, should be 1c

  Our reply: Thank you very much for your careful reading. We have exchanged "2c" and "1c". (in page 5)

- **Comment:** 3.3.1 ROS1 and ROS2 is much less than the difference between remote and local cases

  Our reply: Thank you for pointing out our illegible expression. We have removed "The difference in end-to-end latencies between" and modified the sentence as you proposed. (in Section 3.3.1)

- **Comment:** 3.3.2 with large data, ROS2 has significant overhead depending on the size of data

  Our reply: Thank you for pointing out our unreadable expression. We have changed "Compared to ROS1, with" to "With". (in Section 3.3.2)

- **Comment:** Figure 12 and 13, invert legend

  Our reply: Thank you for your suggestion. We have inverted the legends of Figure 11 and 12 for easy distinguishable difference. (in page 6)

- **Comment:** Table 5 legend missing

  Our reply: Thank you for pointing out our wrong part. We have fulfilled Tabel 5 legend as "Comparison of ROS2 to Related Work". (in page 9)

- **Comment:** The authors use several implementations of DDS but it is unclear from their figures, which data is for OpenSplice, Connext or FastRTPS

  Our reply: I'm sorry to provide unclear figures. We have added explanations for Figures 7, 8, 9, and 10 to clarify what DDS implementation we used. (in page 6)

- **Comment:** Section 3.3.3 analyses performance for the two DDS frameworks with an explicit assumption regarding the performance of Opensplice vortex. The authors assume that the performance of Opensplice vortex professional edition is the same as the performance of Connext professional edition, but conduct their experiments only with the community edition of Opensplice. For a fair analysis, either the professional or community version should be used for both frameworks.

  Our reply: Thank you for pointing out our unfair evaluations. We tried building ROS2 with Vortex OpenSplice, but we could not succeed. Currently, ROS2 does not support OpenSplice Professional Edition. For clear discussion, we added sentence ",but ROS2 does not support this" ("this" means Vortex OpenSplice) in a footnote. (page 4) In addition, after an evaluation of threads, we have changed our view that the performance of Opensplice vortex professional edition is the same as the performance of Connext professional edition. Using OpenSplice, ROS2 has many threads (about 49 threads). Parallelized procession by a lot of threads causes low latency and we assume that Vortex OpenSplice is faster than Connext.

  Added footnote about unspported Vortex in page 4

  Vortex OpenSplice, i.e., OpenSplice commercial edition, supports shared memory transport , but ROS2 does not supports this. In this paper, OpenSplice DDS Community Edition is used because it is open-source.

2 Response to 2nd reviewer

1. **Comment:**

   The experiments provide good data, but unfortunately the data come only from a high-performance, multi-core system. It will be useful to discuss performance in an embedded device used in robotics with some resource constraints.

   Our reply:

   Thank you very much for your thoughtful suggestion. To discuss performance in embedded systems with some resource constrains, we have conducted measurements of thread and memory consumption in newly added Section 3.5 and 3.6. In these sections, we analyze the results of measurements and discuss DDS characteristic. These provide us insight for embedded systems and distributed capabilities.

Added "3.5 Thread of ROS1 and ROS2" section

In this section, we measure the number of threads on each *node*. Table 6 shows the result of measurements. Note that the number described in Table 6 depends on DDS configuration including *QoS Policy*. The number does not be fixed by vendors.
First of all, we can observe that ROS2 *node* with OpenSplice has a lot of threads. This may cause parallelized processing and the fact that OpenSplice is much faster than Connext as shown in Figure 13.
Another interesting point is FastRTPS threads. ROS2 *node* with FastRTPS realizes discovery and serialization, and pub/sub data transport with the same number of ROS1 *node* threads. This result proves improvement of fault tolerance without additional resources because FastRTPS does not need *master-node*.

Table 6: The Number of Thread on ROS1 or ROS2

	ROS1	Connext	OpenSplice	FastRTPS
node	5	8	49	5
master-node	3	-	-	-

Added "3.6 Memory consumption of ROS1 and ROS2" section

We also measure memory size of shared library object (`.so`) in ROS1 and ROS2. Shared libraries are libraries that are dynamically loaded by *nodes* when they start. They are not linked to executable files but they will be vital guidelines for estimation of memory size. We arrange the result in Table 7. In this table, we add up library data size for pub/sub transport. In ROS2, shared libraries are classified into the DDS library and the ROS2 abstraction library. While DDS libraries are provided by each vendor, ROS2 libraries abstract DDS APIs and convert *messages* for DDS. In Table 7, DDS and ROS2 libraries vary depending on vendors. These library data size tends to increase because its QoS capability and abstraction. For small embedded systems, we need a minimal DDS implementation and light abstraction layer.

Table 7: Memory of .so Files for ROS1 and ROS

		DDS [KB]	Abstraction [KB]	Total [MB]
ROS1		2,206		2.26
ROS2	Connext	11,535	9,645	21.18
	OpenSplice	3,837	14,117	17.95
	FastRTPS	1,324	3,953	5.28

2. **Comment:**

   Since the comparison provides evaluations of ROS middleware replacements, considering alternatives (primarily ZMQ + Protobuf) would have made for a compelling case (for or against) the selected DDS approach.

   Our reply:

   Thank you very much for your good insight. Although these alternatives such as ZMQ + Protobuf should be evaluated, ROS2 firstly accepts DDS approach and only supports several DDS implementations. ROS2 does not support ZMQ + Protobuf. To highlight this fact, we have added "Currently ROS2 only supports some DDS implementations.". (in page 1) Since ROS2 currently does not support ZMQ+Protobuf, this paper focuses evaluation of DDS approach to ROS and conducts its proof of concept.

   Added footnote for unsupported ZMQ + Protobuf in page 1

   Currently ROS2 only supports some DDS implementations.

3. **Comment:**

   Additionally, further evaluations of the QoS policies would have been beneficial, since 100 ms deadline is rather meaningless for the message sizes, processor speeds, and network capacities used in the experimental evaluation.

   Our reply:

   Thank you for your thoughtful comment. At present, ROS2 only supports a few QoS Policies. For example, DEADLINE is only calculated by a node's period and is not utilized for scheduling. In this condition, we prepare *-depth policy and have varied history setting by configuring depth option. This HISTORY QoS Policy provides applications fault tolerance. We have added sentences in "3.3.3 Comparison within ROS2" section and described analysis there.

   Updated sentences in "3.3.3 Comparison within ROS2" section

   In addition, the influence of the *QoS Policy* on end-to-end latencies is evaluated in (2-b) OpenSplice with the `reliable policy`, `best-effort policy`, and `*-depth policy`. `*-depth policy` is prepared for this evaluation and configured by depth as shown in Table 5. Figure 14 shows differences in latencies depending on the `reliable policy` and `best-effort policy`.

Added sentences about additional QoS in "3.3.3 Comparison within ROS2" section

Figure 15 shows no differences depending on the depth of `*-depth policy`. These *QoS policies* are different in the number *nodes* save `messages`. Although this number influences resources, this does not affect latencies because archiving *messages* is conducted in every publication.

Table 5: Depth Configurable QoS Policies

	`*-depth policy`
DEADLINE	`100 ms`
HISTORY	`LAST`
depth	1, 10, or 100
RELIABILITY	`RELIABLE`
DURABILITY	`TRANSIENT_LOCAL`

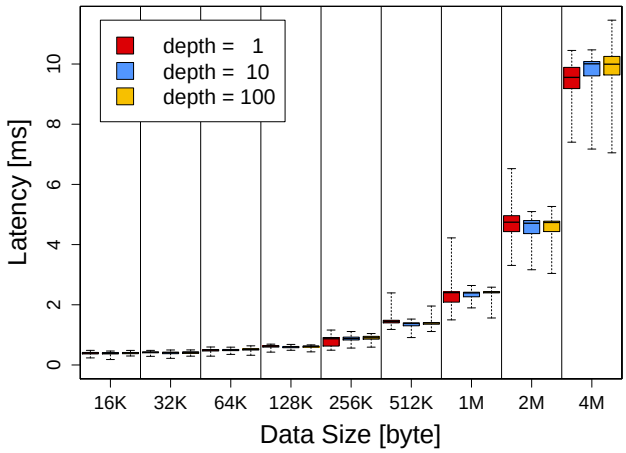


Figure 15: (2-b) Configured **-depth policy* in ROS2 with OpenSplice

4. **Comment:**

   Finally, evaluation of performance of the middleware options under more constrained network resources would have been beneficial as well.

   Our reply:

   Thank you very much for your thoughtful comment. We have additionally conducted evaluation of a multiple destinations publisher for increased network load in "3.3.4 Multiple Destinations Publisher local cases" section. This evaluation also provides insight for the overhead of DDS scales as shown in 18 and 19. However, we did not conduct further evaluations for real-time guarantees with constrained network because QoS Policies supposed by ROS2 are insufficient for real-time processing. After support of real-time QoS, we will conduct above evaluations with configuring QoS variables for real-time systems. We have explicitly described this fact as future work.

Updated sentences about future work in "5. CONCLUSION" section

Since ROS2 is under development, we must maximize DDS potential by tuning and abstracting more *QoS Policies* for real-time processing and DDS configurations.

Added sentences in "3.3.4 Multiple Destinations Publisher in `local` cases" section

In this section, we prepare five *subscriber-nodes* and measure latencies of each *node*. Much of information shared in real applications is destinated to multiple destinations. Hence, this evaluation is practical for user. Figure 17 shows latencies of ROS1. We can observe significant differences between *subscriber-nodes*. This means ROS1 schedules *message* publication in order and is not suitable for real-time systems. For example, in 1 MB, subscriber 5 is about twice as mach as subscriber 1. In contrast, ROS2 has small differences as shown in Figure 18. All subscribers' behavior is fair in ROS2. However, ROS2 latencies significantly depends on the number of packets. This is same characteristic we learned from Figure 8. Figure 19 indicates fair latencies and dependency of packets. Although we cannot say that latencies variance of ROS1 is larger than one of ROS2 due to the difference of the scale, Figures 17, 18, and 19 prove ROS2 *message* publication is more fair to multiple subscribes than ROS1 one.

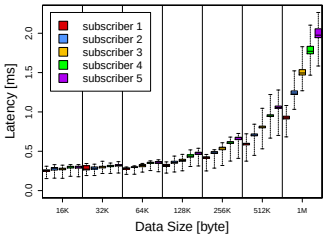


Figure 17: (1-b) ROS1 multiple destinations publisher.

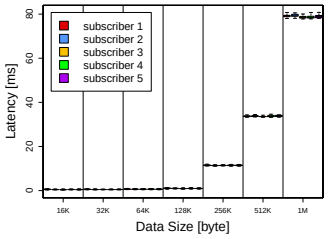


Figure 18: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

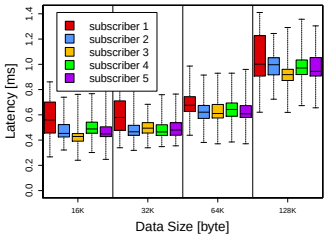


Figure 19: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

3 Response to 3rd reviewer

1. **Comment:**

   Despite this, the title and contributions are not clear, particularly, performance is a broad term and the authors mostly present results on message transmission. The contribution should be stated more explicitly. While it is true that other practical considerations are explained, they are not properly highlighted.

   Our reply:

   Thank you very much for your critical comment. To expand the scope of experiments, we have conducted additional evaluations. In the revised paper, we have conducted additional evaluations from various aspects. One is throughput evaluations in "3.4 Throughput of ROS1 and ROS2" section. We have clarified throughput characteristics depending on DDS vendors. Another additional evaluation is measurements of latency for a multiple destinations publisher. This is described in "3.3.4 Multiple Destinations Publisher in local cases" section and will be utilized for fault tolerance and distributed capabilities. From this experiment, we can learn fair latency which DDS brings to ROS. In addition, we prepare *-depth policy and have varied QoS setting by configuring depth option in "3.3.3 Comparison within ROS2" section. The other additional evaluations also expand our performance evaluation. We have conducted measurements for the number of thread in "3.5 Thread of ROS1 and ROS2" and shared library memory consumption in "3.6 Memory consumption of ROS1 and ROS2" section. These experimets provide us insight for embedded systems and distributed capabilities. We consider that these broad evaluations and analysis represent the performance of ROS2 after revision.

   In addition, we have created new section "3.7 Lessons Learned" to clarify our contributions and practical considerations. In this section, we explicitly show guidelines to highlight our contributions from experiments in this paper. This lessons will be variable and should be shared among ROS users. For precise added sentences and analysis, please go on reading following our replies and the revised paper.

2. **Comment:**

   One of the most promising improvements of ROS2 wrt ROS is the lack of a single point of failure (master). This should probably be highlighted!

   Our reply:

   Thank you very much for pointing out a vital issue. This is important for fault tolerance and significant benefit of using DDS. To highlight this point, we have added several sentences.

   Added sentences in "2.1 Robot Operating System (ROS)" section

   In addition, due to use of DDS, ROS2 does not need a master process. This is a import point in terms of fault tolerance.

   Added sentences in "3.7 Lesson Learned" section

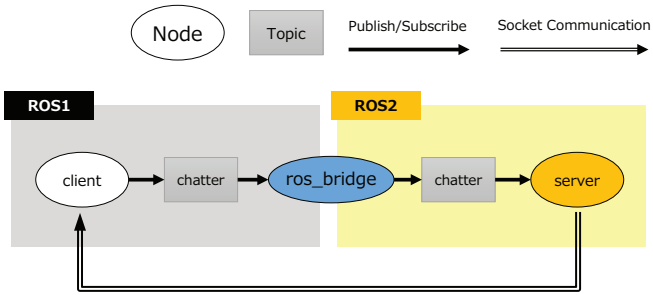
   We believe ROS2 outweigh its cost for using DDS. Fault tolerance of DDS is superior because it is able to save past data with *QoS Policy* and does not have a master *node*.

3. **Comment:**

   How exactly are the end-to-end delays (one way transmission) measured by the same computer? Please explain the process.

   Our reply:

   Thank you for pointing out our lack of explanation. In the remote cases, to avoid time synchronization issues, the experiment adopts simple socket communication that routes through neither ROS1 nor ROS2. Machine1 transmits data through ROS1 or ROS2, and receives short data through socket communication. In the adopted method, evaluation halts when messages do not reach a subscriber-node in the cases with, for example, the best-effort policy, because a publisher-node must wait until a subscriber-node replies during each publish event. We estimate end-to-end latencies by subtracting preliminarily evaluated socket communication time. Using socket communication, the communication latencies between ROS1 and ROS2 can be evaluated respectively. However, dividing round-trip latency in half cannot evaluate them and does not be used for this evaluation. The following figure shows the node-graph for evaluation of communication from ROS1 to ROS2 with socket communication and a ros_bridge in remote cases. For page constraint, we could not have added the above explanation and following figure. Hence, we make source code for evaluation open and have added its url to references of our paper.



Evaluation method for remote cases with ros_bridge.

Added references for source code in "6. REFERENCES" section

[5] Source code using ROS1 evaluations. https://github.com/m-yuya/ros1_evaluation.
[6] Source code using ROS2 evaluations. https://github.com/m-yuya/ros2_evaluation.

4. **Comment:**

   In table 4: Why is there 'none' in the ROS2 best effort policy? Since there is no backlog saved on the nodes, all late joining nodes will suffer from the same problem as ROS nodes.

   Our reply:

   Thank you for pointing out our lack of explanation. In best-effort policy, "none" means there is no inital loss when a subscriber-node is launched before a publisher-node begins to send messages. As you explained, a node with best-effort policy does not have backlog and there will be a lot of message loss for lata-joining nodes. In Table 3, "Inital loss" means whether there is message loss or not for pre-joining nodes. "Inital loss" in not for late-joining nodes. After receiving your comment, we have modified sentences in the beginning of "3.2 Capabilities of ROS1 and ROS2" section to make it easy to understand.

Added sentence in "3.2 Capabilities of ROS1 and ROS2" section

In `best-effort policy`, a `subscriber-node` must be launched before a *publisher-node* begins to send *messages* for "Initial loss" none.

5. **Comment:**

   A major feature of ROS2, is to give Real Time guarantees on message deliveries, however, this is not compared. Specifically, when message loads are increased, a comparison would be welcomed.

   Our reply:

   Thank you very much for your thoughtful suggestion. We have additionally conducted evaluation of a multiple destinations publisher for increased network load in "3.3.4 Multiple Destinations Publisher local cases" section. This evaluation also provides insight for the overhead of DDS scales as shown in 18 and 19. However, we did not conduct further evaluations for real-time guarantees with constrained network because QoS Policies supposed by ROS2 are insufficient for real-time processing. After support of real-time QoS, we will conduct above evaluations with configuring QoS variables for real-time systems. We have explicitly described this fact as future work.

Updated sentences about future work in "5. CONCLUSION" section

In future work, we will evaluate real-time applications such as an autonomous driving vehicle as case studies using ROS2. ... Since ROS2 is under development, we must maximize DDS potential by tuning and abstracting more *QoS Policies* for real-time processing and DDS configurations.

Added sentences in "3.3.4 Multiple Destinations Publisher in `local` cases" section

In this section, we prepare five *subscriber-nodes* and measure latencies of each *node*. Much of information shared in real applications is destinated to multiple destinations. Hence, this evaluation is practical for user. Figure 17 shows latencies of ROS1. We can observe significant differences between *subscriber-nodes*. This means ROS1 schedules *message* publication in order and is not suitable for real-time systems. For example, in 1 MB, subscriber 5 is about twice as mach as subscriber 1. In contrast, ROS2 has small differences as shown in Figure 18. All subscribers' behavior is fair in ROS2. However, ROS2 latencies significantly depends on the number of packets. This is same characteristic we learned from Figure 8. Figure 19 indicates fair latencies and dependency of packets. Although we cannot say that latencies variance of ROS1 is larger than one of ROS2 due to the difference of the scale, Figures 17, 18, and 19 prove ROS2 *message* publication is more fair to multiple subscribes than ROS1 one.

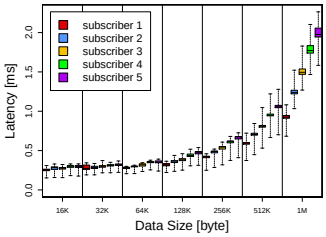


Figure 17: (1-b) ROS1 multiple destinations publisher.

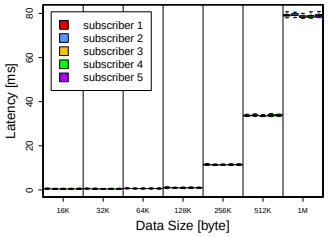


Figure 18: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

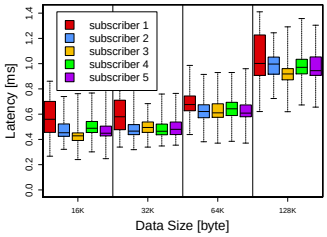


Figure 19: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

6. **Comment:**

   The authors say that Connext and OpenSplice maximum payload size is 64kB, but this is also the maximum payload of both TCP and UDP messages. Why is this an advantage/disadvantages of this?

   Our reply:

   Thank you very much for indicating our unclear explanation. Connext and OpenSplice maximum payload size is officially described as 64KB in [http://www.prismtech.com/vortex/vortex-opensplice/performance] and [http://www.rti.com/products/dds/benchmarks.html]. This is because that 64KB is the maximum payload of IP packet. In DDS, message packets must be sent following QoS. Hence, when large message is divided into several packets, some DDS vendors need users to use alternative APIs and additional processing is needed to manage divided packets with QoS. Whether message is divided or not is critical issue in DDS. In contrast, ROS1 does not handle packets with QoS. ROS1 simply sends packets in order. We have created new section “3.7 Lessons Learned” and explained this issue.

Added sentences about ROS2 guidelines in "3.7 Lessons Learned" section

DDS supports *QoS Policy* but there is trade-off of end-to-end latencies and throughput. In the `local` case, overhead latencies of ROS2 is not trivial. From Section 3.3, the latencies is caused by two data conversions for DDS and DDS transaction. DDS end-to-end latencies is constant until *message* data size is lower than maximum packet size (64 KB) as shown in Figure 9. On the other hand, as one large *message* is divided into several packets, the latencies sharply increases as show in Figures 10 and 18. Whether *message* data size is over 64 KB or not is important issue especially in DDS because management of divided packets with QoS Policy needs significant processing time and alternative APIs provided by some vendors. We should understand influence of divided packets and keep in mind this issue when using DDS. While DDS and ROS2 abstraction have overhead latencies, OpenSplice utilizes a lot of threads and processes faster than Connext as shown in Figure 13. This is a reason why we currently should use OpenSplice in the underlying implementation of DDS in the `local` case. In the `remote` case, although overhead latencies is trivial, we must consider throughput for bandwidth. As shown in 20, Connext is superior to OpenSplice in terms of throughput. This constant overhead throughput is predictable and exists no matter how small *message* data size is. It influences especially when many kinds of topic are used with high Hz. We recommend Connext to consider minimum necessary throughput in the `remote` case.
DDS brings supports real-time embedded systems to ROS2. We believe ROS2 outweigh its cost for using DDS. Fault tolerance of DDS is superior because it is able to save past data with *QoS Policy* and does not have a master *node*. DDS guarantees fair latencies as shown in Figure 19. In addition, DDS is able to run on multiple platforms include RTOS and switch DDS implementation as needed. Under RTPS protocol, any ROS2 *nodes* communicate with each other without relation to its platform. FastRTPS is currently the best DDS implementation for embedded systems in thread and memory as Table 6 indicates, but it is not suitable for small embedded system.

7. **Comment:**

   There is no mention whatsoever on which type of network is used to comunicate between two different machines. Is it a full-duplex ethernet connection, an unreliable WiFi connection, or some other network? This is important, since authors claim "Some failures with the best-effort policy are due to frequent message losses caused by non-reliable communications", which is dependent on the actual protocol.

   Our reply:

   Thank you for pointing out import issue. We have conducted remote case experiments with Full-Duplex 100BASE-TX 100Mbps Ethernet connection. Two machines are physically connected by LAN cable with a switcher. Thus, message loss occurs by UDP and not by network. To clarify this fact, we add network protocol in Table 2: "Evaluation Environment".

Table 2: Evaluation Environment

		Machine1	Machine2
CPU	Model number	Intel Core i5 3470	Intel Core i5 2320
	Frequency	3.2 GHz	3.00 GHz
	Cores	4	4
	Threads	4	4
Memory		16 GB	8 GB
Network		100 Mbps Ethernet / Full-Duplex	
ROS1		Indigo	
ROS2		Cement (alpha3)	
DDS implementations		Connext[1]/ OpenSplice[2]/ FastRTPS	
OS	Distribution	Ubuntu 14.04	
	Kernel	Linux 3.13.0	

8. **Comment:**

   What was the message set in each experiment? Only one message per experiment? Or was there any combination of messages in each experiment? If so, the scheduling algorithm should have been explained and the interference analysed.

   Our reply:

   Thank you for pointing out our lack of explanation. In every data size, a publisher-node sends 100 messages per experimet with 10 Hz. Each simple string message is transported in order. Using 100 measurements in each data size, we prepare medians and boxplots. To clarify this process, we make source code for evaluation open and have added its url to references of our paper.

   Added references for source code in "6. REFERENCES" section

   [5] Source code using ROS1 evaluations. https://github.com/m-yuya/ros1_evaluation.
   [6] Source code using ROS2 evaluations. https://github.com/m-yuya/ros2_evaluation.

9. **Comment:**

   Missing results: Much of the information shared in robots is destined to multiple destinations. A possible improvement (maybe for future work) would be a comparison wrt multi-destination message distribution.

   Our reply:

   Thank you for your advice. In the revised paper, we have conducted evaluations for a multiple destinations publisher. In our evaluation, we prepare five subscribers and measure end-to-end latencies. From this experiment, we can learn fair latency which DDS brings to ROS. However, latencies are significantly depending on the number of packets. We consider that this problems will be improved by abstraction of alternative APIs such as an asynchronous publisher and flow controller and/or vendor specific configuration options. This is effective for not only multi-destination but also single-distination. Challenging above problems is our future work because we did not have enough time to do them.

Added sentences in "3.3.4 Multiple Destinations Publisher in `local` cases" section

In this section, we prepare five *subscriber-nodes* and measure latencies of each *node*. Much of information shared in real applications is destinated to multiple destinations. Hence, this evaluation is practical for user. Figure 17 shows latencies of ROS1. We can observe significant differences between *subscriber-nodes*. This means ROS1 schedules *message* publication in order and is not suitable for real-time systems. For example, in 1 MB, subscriber 5 is about twice as mach as subscriber 1. In contrast, ROS2 has small differences as shown in Figure 18. All subscribers' behavior is fair in ROS2. However, ROS2 latencies significantly depends on the number of packets. This is same characteristic we learned from Figure 8. Figure 19 indicates fair latencies and dependency of packets. Although we cannot say that latencies variance of ROS1 is larger than one of ROS2 due to the difference of the scale, Figures 17, 18, and 19 prove ROS2 *message* publication is more fair to multiple subscribes than ROS1 one.

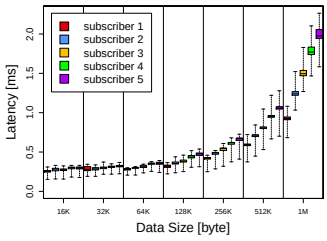


Figure 17: (1-b) ROS1 multiple destinations publisher.

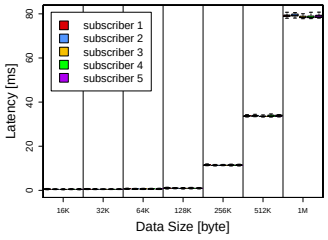


Figure 18: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

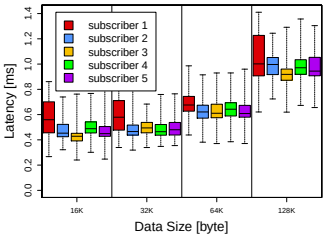


Figure 19: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

Updated sentences about future work in "5. CONCLUSION" section

In future work, we will evaluate real-time applications such as an autonomous driving vehicle as case studies using ROS2. Moreover, we have to breakdown DDS processing time and execute ROS2 on RTOS. We also are interested in ROS2 behavior on embedded devices. Since ROS2 is under development, we must maximize DDS potential by tuning and abstracting more *QoS Policies* for real-time processing and DDS configurations.