Exploring the Performance of ROS2

Yuya Maruyama
Graduate School of Engineering Science
Osaka University

Shinpei Kato
Graduate School of Information Science and Technology
The University of Tokyo

Takuya Azumi
Graduate School of Engineering Science
Osaka University

ABSTRACT

Middleware for robotics development must meet demanding requirements in real-time distributed embedded systems. The Robot Operating System (ROS), open-source middleware, has been widely used for robotics applications. However, the ROS is not suitable for real-time embedded systems because it does not satisfy real-time requirements and only runs on a few OSs. To address this problem, ROS1 will undergo a significant upgrade to ROS2 by utilizing the Data Distribution Service (DDS). DDS is suitable for real-time distributed embedded systems due to its various transport configurations (e.g., deadline and fault-tolerance) and scalability. ROS2 must convert data for DDS and abstract DDS from its users; however, this incurs additional overhead, which is examined in this study. Transport latencies between ROS2 nodes vary depending on the use cases, data size, configurations, and DDS vendors. We conduct proof of concept for DDS approach to ROS and arrange DDS characteristic and guidelines from various evaluations. By highlighting the DDS capabilities, we explore and evaluate the potential and constraints of DDS and ROS2.

Keywords

robot operating system; data distribution service; quality of service; real-time; embedded; publish/subscribe

1. INTRODUCTION

In recent years, real-time distributed embedded systems, such as autonomous driving vehicles, have become increasingly complicated and diverse. Autonomous driving has attracted attention since the November 3, 2007 DARPA Urban Challenge [34]. The Robot Operating System (ROS) [28] is open-source middleware that has undergone rapid development [11] and has been widely used for robotics applications (e.g., autonomous driving systems). The ROS is built almost entirely from scratch and has been maintained by Willow Garage [7] and Open Source Robotics Foundation (OSRF) [2] since 2007. The ROS enhances productivity [12], providing publish/subscribe transport, multiple libraries (e.g., OpenCV and the Point Cloud Library (PCL) [3]), and tools to help software developers create robotics applications.

However, the ROS does not satisfy real-time run requirements and only runs on a few OSs. In addition, the ROS cannot guarantee fault-tolerance, deadlines, or process synchronization. Moreover, the ROS requires significant resources (e.g, CPU, memory, network bandwidth, threads, and cores) and can not manage these resources to meet time constraints. Thus, the ROS is not suitable for real-time embedded systems. This critical problem has been considered by many research communities, including ROS developers, and various solutions have been proposed and evaluated [13], [19], [36]. However, these solutions are insufficient[1] to address the ROS's limitations for real-time embedded systems.

To satisfy the needs of the now-broader ROS community, the ROS will undergo a significant upgrade to ROS2 [23]. ROS2 will consider the following new use cases: real-time systems, small embedded platforms (e.g., sensor nodes), non-ideal networks, and cross-platform (e.g., Linux, Windows, Mac, Real-Time OS (RTOS), and no OS). To satisfy the requirements of these new use cases, the existing version of ROS (hereinafter ROS1) will be reconstructed to improve user-interface APIs and incorporate new technologies, such as Data Distribution Service (DDS) [24], [30], Zeroconf, Protocol Buffers, ZeroMQ, Redis, and WebSockets.[2] The ROS1 transport system will be replaced by DDS, an industry-standard real-time communication system and end-to-end middleware. The DDS can provide reliable publish/subscribe transport similar to that of ROS1.

DDS is suitable for real-time embedded systems because of its various transport configurations (e.g., deadline, reliability, and durability) and scalability. DDS meets the requirements of distributed systems for safety, resilience, scalability, fault-tolerance and security. DDS can provide solutions for some real-time environments and some small/embedded systems by reducing library sizes and memory footprints. Developed by different DDS vendors, several implementations of this communication system have been used in mission-critical environments (e.g., trains, aircrafts, ships, dams, and financial systems) and have been verified by NASA and the United States Department of Defense. Several DDS implementations have been evaluated and validated by researchers [37], [32] and DDS vendors. These evaluations indicate that DDS is both reliable and flexible.

Contribution: In this paper, we provide proof of concept

[1]Reasons why prior work is insufficient are discussed in Section 4.
[2]Currently ROS2 only supports some DDS implementations.

EMSOFT'16, October 01-07, 2016, Pittsburgh, PA, USA
© 2016 ACM. ISBN 978-1-4503-4485-2/16/10…$15.00
DOI: http://dx.doi.org/10.1145/2968478.2968502

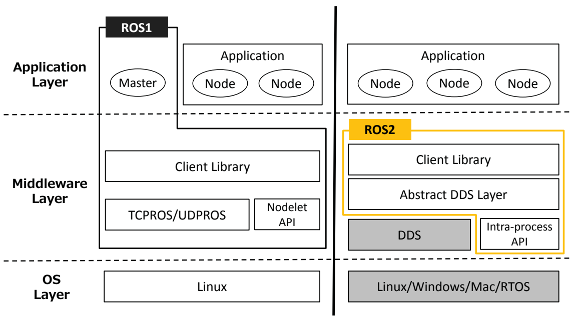

Figure 1: ROS1/ROS2 architecture.

for DDS approach to ROS. We clarify the performance of the data transport for ROS1 and ROS2 in various situations. Performance means latency characteristics, throughput and distributed capability. Focusing on the DDS capabilities, depending on DDS vendors and configurations, 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.

Organization: The remainder of this paper is organized as follows. Section 2 provides background information and describes the ROS and DDS system models. Section 3 validates experimental situations and evaluates the performance of ROS1 and ROS2 with various configurations. Section 4 discusses related work. Finally, Section 5 concludes the paper and offers suggestions for future work.

2. BACKGROUND

In this section, we provide background knowledge. First, we describe the ROS2 system model compared to ROS1, focusing on its communication system. We then review aspects of the ROS, such as the publish/subscribe model. Finally, we describe DDS, which is used as the communication system for real-time systems in ROS2.

2.1 Robot Operating System (ROS)

Figure 1 briefly illustrates the system models of ROS1 and ROS2. In the left side of Figure 1, ROS1's implementation includes the communication system, TCPROS/UDPROS. This communication requires a master process (unique in the distributed system) because of the implementation of ROS1. In contrast, as shown in the right side of Figure 1, ROS2 builds upon DDS and contains a DDS abstraction layer. Users do not need to be aware of the DDS APIs due to this abstraction layer. This layer allows ROS2 to have high-level configurations and optimizes the utilization of DDS. In addition, due to use of DDS, ROS2 does not need a master process. This is a import point in terms of fault tolerance.

ROS applications consist of independent computing processes called *nodes*, which promote fault isolation, faster development, modularity, and code reusability. Communication among *nodes* is based on a publish/subscribe model. In this model, *nodes* communicate by passing *messages* via a *topic*. A *message* has a simple data structure (much like C structs) defined by .msg files. *Nodes* identify the content of the *message* by the *topic* name. As a *node* publishes a *message* to a *topic*, another *node* subscribes to the *topic* and utilizes the *message*. For example, as shown in Figure 2, the "Camera" *node* sends *messages* to the "Images" *topic*. The *messages* in the *topic* are received by the "Car Detection" *node* and "Pedestrian Detection" *node*. The publish/subscribe model is designed to be modular at a fine-grained scale and is suitable for distributed systems.

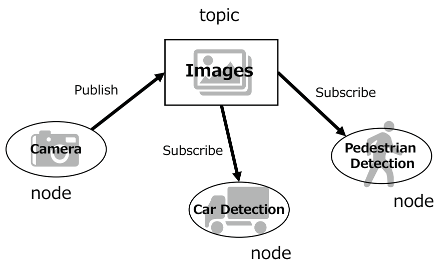

Figure 2: Example of ROS publish/subscribe model.

In ROS1, the above communication system is implemented as middleware based on TCPROS and UDPROS using TCP and UDP sockets. When *subscriber-nodes* and *publisher-nodes* are launched, they interact with a *master-node* that collects information and manages *topics*, similar to a server. After an XML/Remote Procedure Call (RPC) transaction with the *master-node*, *subscriber-nodes* request a connection to *publisher-nodes*, using an agreed upon connection protocol. Actual data (i.e., a *message*) is transported directly between *nodes*. This data does not route through the master. ROS1 realizes a peer-to-peer data transport between *nodes*.

Optionally, ROS1 provides *nodelets*, which provide efficient *node* composition for optimized data transport without TCPROS and UDPROS. A *nodelet* realizes non-serialized data transport between *nodes* in the same process by passing a pointer. ROS2 inherits this option as *intra-process communication*, which addresses some of the fundamental problems with *nodelets* (e.g., safe memory access).

ROS2 adopts DDS as its communication system. However, as an exception, *intra-process communication* is executed without DDS. DDS is provided by many vendors and has several implementation types. Developers can select an appropriate DDS implementation from a variety of DDS vendors.

2.2 Data Distribution Service (DDS)

The DDS specification [21] is defined for a publish/subscribe data-distribution system by the Object Management Group (OMG) [1]. The OMG manages the definitions and standardized APIs; however the OMG hides the details of implementation. Several implementations have been developed by different vendors (e.g., RTI [29] and PRISMTECH [25]). DDS supports a wide range of applications, from small embedded systems to large scale systems, such as infrastructures. Note that distributed real-time embedded systems are also supported.

The core of DDS is a Data-Centric Publish-Subscribe (DCPS) model designed to provide efficient data transport between processes even in distributed heterogeneous platforms. The DCPS model creates a "global data space" that can be accessed by any independent applications. DCPS facilitates efficient data distribution. In DDS, each process that publishes or subscribes to data is called a *participant*, which corresponds to a *node* in the ROS. *Participants* can read and write from/to the global data space using a typed interface.

As shown in Figure 3, the DCPS model is constructed of *DCPS Entities*: *DomainParticipant*, *Publisher*, *Subscriber*,

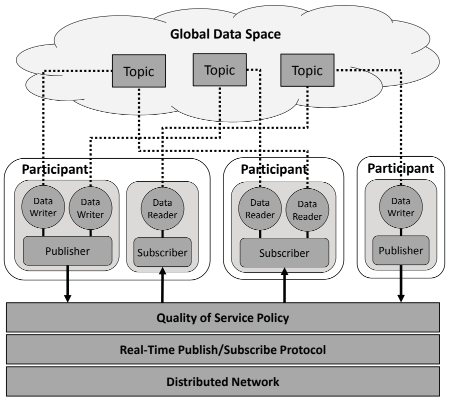

Figure 3: Data-centric publish-subscribe (DCPS) model.

DataWriter, *DataReader*, and *Topic*. Each data transport between processes is executed according to a *Quality of Service (QoS) Policy*.

***DomainParticipant*:** A *DomainParticipant* is a container for following other entities and the entry-point for the service. In DDS, all applications communicate with each other within a *Domain*, which promotes isolation and communication optimization.

***Publisher*:** A *Publisher* is the object responsible for data issuance. Managing one or several *DataWriters*, the *Publisher* sends data to one or more *Topics*.

***Subscriber*:** A *Subscriber* is responsible for receiving published data and making the data available. The *Subscriber* acts on behalf of one or more *DataReaders*. According to a *Subscriber*, a *DomainParticipant* can receive and dispatch data of different specified types.

***DataWriter*:** A *DataWriter* is an object that must be used by a *DomainParticipant* to publish data through a *Publisher*. The *DataWriter* publishes data of a given type.

***DataReader*:** A *DataReader* is an object that is attached to a *Subscriber*. Using the *DataReader*, a *DomainParticipant* can receive and access data whose type must correspond to that of the *DataWriter*.

***Topic*:** A *Topic* is used to identify each data-object between a *DataWriter* and a *DataReader*. Each *Topic* is defined by a name and a data type.

***QoS Policy*:** All *DCPS Entities* have a *QoS Policy*, which represents their data transport behavior. Each data transaction is configurable at various levels of granularity via many *QoS Policy* options. In Figure 4, we show an example of DDS data transport following a *QoS Policy*. The deadline period, depth of history, and communication reliability are configured by a *QoS Policy*. Table 1 shows the details of the *QoS Policy* supported by ROS2. In DDS, there are many other *QoS Policies* [21], which ROS2 should support to extend its capabilities.

In the DCPS model, data of a given type is published from one or several *DataWriters* to a *topic* (its name is unique in the *Domain*). One or more *DataReaders* identify a data-object by *topic* name in order to subscribe to the *topic*. After this transaction, a *DataWriter* connects to a *DataReader* using the Real-Time Publish/Subscribe (RTPS) protocol [20] in distributed systems. The RTPS protocol, the DDS standard protocol, allows DDS implementations from multiple vendors to inter-operate by abstracting and optimizing transport, such as TCP/UDP/IP. The RTPS protocol is flexible and is defined to take advantage of a *QoS Policy*. Several vendors use UDP and shared memory transport to communicate. However, in several circumstances, the TCP protocol might be required for discovery and data exchange.

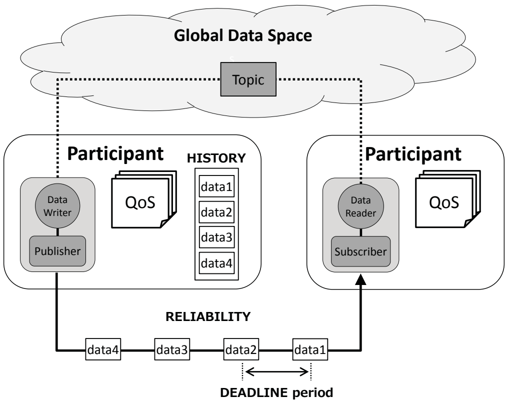

Figure 4: DDS QoS Policy.

Table 1: All QoS Policies of ROS2

DEADLINE	A *DataWriter* and a *DataReader* must update data at least once every deadline period.
HISTORY	This controls whether the data transport should deliver only the most recent value, attempt to deliver all intermediate values, or attempt to deliver something in between (configurable via the `depth` option).
RELIABILITY	In `BEST_EFFORT`, data transport is executed as soon as possible. However, some data may be lost if the network is not robust. In `RELIABLE`, missed samples are retransmitted. Therefore, data delivery is guaranteed.
DURABILITY	With this policy, the service attempts to keep several samples so that they can be delivered to any potential late-joining *DataReader*. The number of saved samples depends on HISTORY. This option has several values, such as `VOLATILE` and `TRANSIENT_LOCAL`.

Data transport between a *DataWriter* and a *DataReader* is executed in the RTPS protocol according to a *QoS Policy*. Each *DCPS Entity* manages data samples according to a unique user-specified *QoS Policy*. The DCPS middleware is responsible for data transport in distributed systems based on the *QoS Policy*. Without considering detailed transport implementations, DDS users generate code as a *DomainParticipant*, including *QoS Policies* using the DDS APIs. Thus, users can focus solely on their purpose and determine ways to satisfy real-time constraints easily.

3. EVALUATIONS

This section clarifies the capabilities and latency characteristics of ROS1 and ROS2. At present, ROS2 has been released as an alpha version whose major features are a C++ client library, a build-system and abstraction to a part of the DDS middleware from several vendors. Note that ROS2 is a very rough draft and is currently under heavy development. Therefore, this evaluation attempts to clarify the currently achievable capabilities and latency characteristics of ROS2.

The following experiments were conducted to evaluate end-to-end latencies for publish/subscribe messaging. The latencies are measured from a publish function on a single *node* until the callback function of another *node* using the hardware and software environment listed in Table 2. The

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	

[1] RTI Connext DDS Professional [29]
[2] OpenSplice DDS Community Edition [25]

Table 3: QoS Policies for Evaluations

	reliable policy	best-effort policy
DEADLINE	100 ms	100 ms
HISTORY	ALL	LAST
depth	-	1
RELIABILITY	RELIABLE	BEST_EFFORT
DURABILITY	TRANSIENT_LOCAL	VOLATILE

range of the transferred data size is 256 B to 4 MB because large image data (e.g., 2 MB) and point cloud data (.pcd) are frequently used in ROS applications, such as an autonomous driving system [18]. A string type *message* is used for this evaluation. In the following experiments, we use two QoS settings, i.e., **reliable policy** and **best-effort policy**, as shown in Table 3. In the **reliable policy**, TRANSIENT_LOCAL allows a *node* to keep all *messages* for late-joining *subscriber-nodes*, and RELIABLE facilitates reliable communication. In the **best-effort policy**, *nodes* do not keep *messages* and communicate unreliably. While each *node* is executed at 10 Hz, the experiments are repeated up to 4 MB. Boxplots and the medians obtained from 100 measurements for each data size are presented. For precise evaluation methods, we make the source code open in [5] and [6]. We compare three DDS implementations, i.e., Connext [29], OpenSplice [25], and FastRTPS [14]. Connext and OpenSplice are well-known commercial license DDS implementations. Note that Connext also has a research license. Several implementations of OpenSplice and FastRTPS have been released under the LGPL license. By default, Connext uses UDPv4 and shared memory to exchange data. Note that OpenSplice[3] and FastRTPS do not support shared memory data transport. For precise evaluations and real-time requirements, *nodes* follow *SCHED_FIFO* [15] and the *mlockall* system call. A *SCHED_FIFO* process preempts any non-*SCHED_FIFO* processes, i.e., processes that use the default Linux scheduling. Using *mlockall*, a process's virtual address space is fixed in physical RAM, thereby preventing that memory from being paged to the swap area.

3.1 Experimental Situations and Methods

As shown in Figure 5, various communication situations between *nodes* in ROS1 and/or ROS2 are evaluated in the following experiments. Whereas ROS1 is used in (1-a), (1-b), and (1-c), ROS2 is used in (2-a), (2-b), and (2-c). In (3-a) and (3-b), ROS1 and ROS2 *nodes* coexist. Note that the case of (2-c) does not require DDS due to *intra-process communication*, i.e., shared memory transport. Shared memory transport is used in the (1-c) *nodelet* and (2-c) *intra-process* cases. In the experiments, Machine1 is only used in (1-b), (1-c), (2-b), (2-c), and (3-b). End-to-end latencies are measured on the same machine by sending *messages* between *nodes*. *Messages* pass over a local loopback in local cases, i.e., (1-b), (2-b), and (3-b). Otherwise, for communication across the network, Machine1 and Machine2 are used in **remote** cases, i.e., (1-a), (2-a), and (3-a). They are connected by a local IP network without any other network.

[3]Vortex OpenSplice [26], i.e., OpenSplice commercial edition, supports shared memory transport, but ROS2 does not support Vortex OpenSplice. In this paper, OpenSplice DDS Community Edition is used because it is open-source.

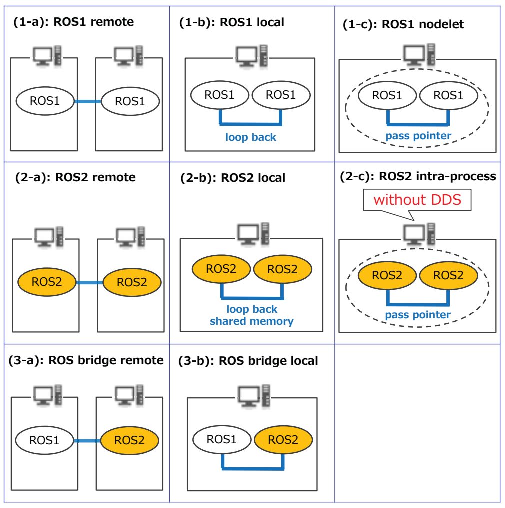

Figure 5: Experimental situations.

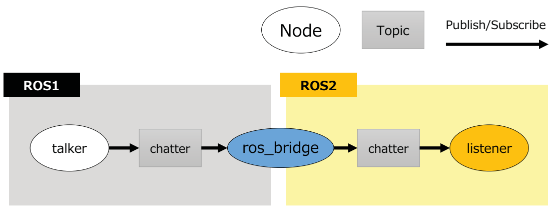

Figure 6: **ros_bridge** evaluation in (3-a) and (3-b).

Communication between ROS1 and ROS2 *nodes* requires a **ros_bridge** [33], a *bridge-node* that converts *topics* for DDS. The **ros_bridge** program has been released by the Open Source Robotics Foundation (OSRF) [2]. A **ros_bridge** dynamically marshals several *topics* for *nodes* in ROS2. Thus, in (3-a) and (3-b), a **ros_bridge** is launched on which ROS2 *nodes* run. Figure 6 shows the *node*-graph for evaluation of communication from ROS1 to ROS2. Note that a **best-effort policy** is the only one used when using a **ros_bridge** because a **ros_bridge** does not support the RELIABLE policy in the *QoS Policy*.

3.2 Capabilities of ROS1 and ROS2

Table 4 shows whether end-to-end latencies can be measured for each data size with a comment about the causal factors of the experimental results. Table 4 summarizes ROS2's capabilities, and several interesting observations can be made. In the "Initial loss" column, ROS1 fails to obtain initial *messages* when a *node* sends *messages* for the first time even though ROS1 uses TCPROS with small data such as 256 B and a *subscriber-node* is launched before a *publisher-node* begins to send *messages*. Although TCPROS is reliable for delivering intermediate *messages*, it does not

Table 4: Capabilities of ROS1 and/or ROS2 for each Data Transport

				Initial loss	256 [byte]	512	1K	$\cdots$ *	64K	128K	256K	512K	1M	2M	4M
ROS1	(1-a) remote			any	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\triangle^1$	$\triangle^1$
	(1-b) local			any	✓	✓	✓	$\cdots$		✓	✓	✓	✓	✓	✓
	(1-c) nodelet			none	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	✓	✓
ROS2	(2-a) remote	Connext	reliable	none	✓	✓	✓	$\cdots$	✓	$\blacktriangle^2$	$\blacktriangle^2$	$\blacktriangle^2$	$\blacktriangle^2$	$\blacktriangle^2$	$\blacktriangle^2$
			best-effort	none	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\triangle^1$	$\triangle^1$
		OpenSplice	reliable	none	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	✓	✓
			best-effort	none	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\triangle^1$	$\triangle^1$
		FastRTPS		none	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\cdots$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$
	(2-b) local	Connext	reliable	none	✓	✓	✓	$\cdots$	✓	$\blacktriangle^2$	$\blacktriangle^2$	$\blacktriangle^2$	$\blacktriangle^2$	$\blacktriangle^2$	$\blacktriangle^2$
			best-effort	none	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\triangle^2$	$\blacktriangle^1$
		OpenSplice	reliable	none	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	✓	✓
			best-effort	none	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\triangle^2$	$\triangle^2$
		FastRTPS		none	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\cdots$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$	$\blacktriangle^3$
	(2-c) intra-process			none	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	✓	✓
ROS1 to 2	(3-a) remote	Connext		any	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\blacktriangle^1$	$\blacktriangle^1$
		OpenSplice		any	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\triangle^1$	$\triangle^1$
	(3-b) local	Connext		any	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\blacktriangle^1$	$\blacktriangle^1$
		OpenSplice		any	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	✓	✓
ROS2 to 1	(3-a) remote	Connext		any	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\blacktriangle^1$	$\blacktriangle^1$
		OpenSplice		any	✓	✓	✓	$\cdots$	✓	✓	$\blacktriangle^1$	$\blacktriangle^1$	$\blacktriangle^1$	$\blacktriangle^1$	$\blacktriangle^1$
	(3-b) local	Connext		any	✓	✓	✓	$\cdots$	✓	✓	✓	✓	✓	$\blacktriangle^1$	$\blacktriangle^1$
		OpenSplice		any	✓	✓	✓	$\cdots$	✓	✓	$\triangle^1$	$\blacktriangle^1$	$\blacktriangle^1$	$\blacktriangle^1$	$\blacktriangle^1$

*: same behavior as 1 and 64 KB; ✓: data transport possible; $\triangle^1$: possible but missing the deadline; $\triangle^2$: data loss possible;
$\blacktriangle^1$: impossible due to a halt of process or too much data loss;
$\blacktriangle^2$: impossible with an error message (deficiency of additional configurations for large data);
$\blacktriangle^3$: impossible with an error message (unsupported large data for the DDS implementation)

support reliable transport of initial *messages*. This influences ROS2 when using a `ros_bridge`. In contrast, ROS2 does not lose initial *messages*, even when using large data such as 4 MB. This proves the reliability of DDS. In `best-effort policy`, a *subscriber-node* must be launched before a *publisher-node* begins to send *messages* for no "Initial loss". On the other hand, with ROS2 `reliable policy`, a *subscriber-node* does not have to be launched before a *publisher-node* starts sending *messages*. This is attributed to `TRANSIENT_LOCAL` in DURABILITY of the *QoS Policy*. The `reliable policy` is tuned to provide resilience against late-joining *subscriber-nodes*. In ROS1, published *messages* are lost and never recovered. This *QoS Policy* accelerates fault-tolerance.

Another interesting observation from Table 4 is that ROS2 has many problems when transporting large data. Many experiments fail in various situations with ROS2; however, we can observe differences in performance between Connext and OpenSplice. These constraints on large data originate from the fact that the maximum payload of Connext and OpenSplice is 64 KB. This is the maximum packet size of IP protocol. It is hard to maintain divided packets with *QoS Policy* by default API. Therefore, we consider that DDS is not designed to handle large data. This is important for the analysis of ROS2 performance. For example, FastRTPS does not support large data because it is designed as a lightweight implementation for embedded systems. Even a string of 256 B exceeds the maximum length in FastRTPS. Many DDS vendors do not support publishing large data with reliable connections and common APIs. To send and manage divided packets, such DDS vendors provide an alternate API such as an asynchronous publisher and flow controller, which has not been abstracted from ROS2. In our experiments, Connext with `reliable policy` yields errors when data are greater than 64 KB. Some failures with the `best-effort policy` are due to frequent *message* losses caused by non-reliable communication. When a *publisher-node* fails to transfer data to a *subscriber-node* frequently, we cannot collect sufficient samples and conduct evaluations. Several evaluations fail in (3-b) and `remote` cases, as shown in Table 4. Currently, the above results indicate that ROS2 is not suitable for handling large *messages*.

3.3 Latency Characteristics of ROS1 and ROS2

As shown in Figures 7, 8, 9, 10, a tendency of end-to-end latencies characteristics is clarified in each situation shown in Figure 5. In (2-a) and (2-b), ROS2 uses OpenSplice with the `reliable policy` because ROS1 uses TCPROS, i.e., reliable communication. In (3-a) and (3-b), to evaluate latencies with large data (e.g., 512 KB and 1 MB), Connext with the `best-effort policy` is used. First, we analyze ROS2 performance compared to ROS1. We then evaluate ROS2 with different DDS implementations and configurations, such as the *QoS Policy*.

3.3.1 Comparison between `remote` and `local` cases

ROS1 and ROS2 is much less than the difference between `remote` and `local` cases. Figures 7 and 8 show the medians of the latencies for the `remote` and `local` cases. Since the conversion influences from ROS1 to ROS2 and from ROS2 to ROS1 are similar, Figures 7 and 8 contain one-way data. In Figure 7, the behavior of all latencies is constant up to 4 KB. In contrast, the latencies in the `remote` cases grow sharply from 16 KB, as shown in Figures 7 and 8. This is because ROS1 and ROS2 divide a *message* into 15 KB packets to transmit data through Ethernet. This difference between the `remote` and `local` cases corresponds to the data transmission time between `Machine1` and `Machine2`, which was measured in a preliminary experiment. The preliminary experiment measured transmission time for each data size using ftp or http. This correspondence indicates that the RTPS protocol and data about the *QoS Policy* have little influence on data transmission time in the network. In addition, all latencies are predictable by measuring the data transmission time.

3.3.2 Comparison among `local`, nodelet, and intra-process cases

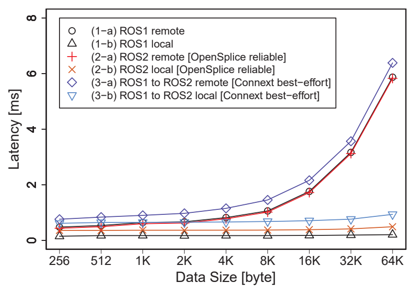

Figure 7: Medians of end-to-end latencies with small data in `remote` and `local` cases.

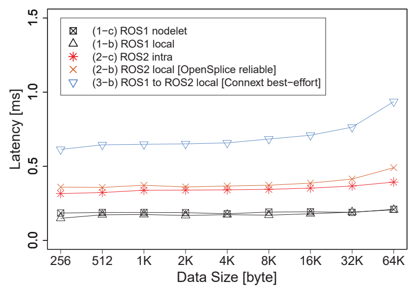

Figure 9: Medians of end-to-end latencies with small data in `local`, *nodelet*, and *intra-process* cases.

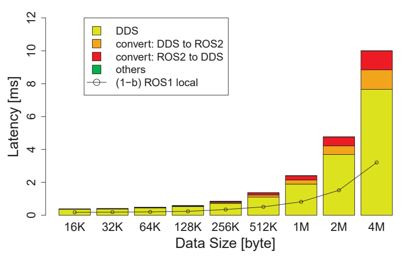

Figure 11: (2-b) `reliable policy` breakdown of ROS2 latencies with the OpenSplice.

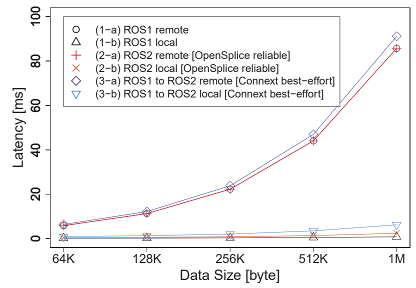

Figure 8: Medians of end-to-end latencies with large data in `remote` and `local` cases.

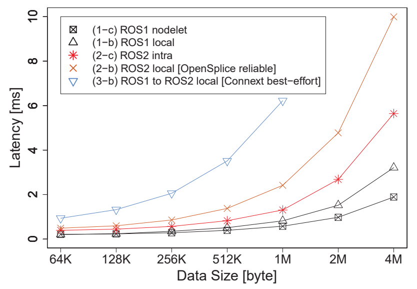

Figure 10: Medians of end-to-end latencies with large data in `local`, *nodelet*, and *intra-process* cases.

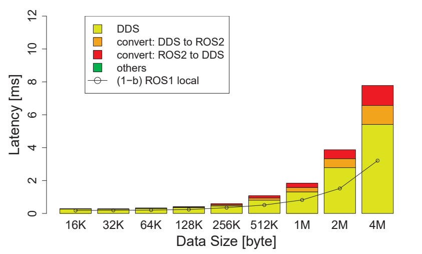

Figure 12: (2-b) `best-effort policy` breakdown of ROS2 latencies with the OpenSplice.

The latenciy characteristics differ in the cases of small and large data. For discussion, we divide the graph into Figures 9 and 10, which show the medians of the end-to-end latencies for local loopback and shared memory transport. This is because whether a *message* is divided into several packets or not is an import issue to consider end-to-end latencies.

For data size less than 64 KB, a constant overhead with ROS2 is observed, as shown in Figure 9, because DDS requires marshaling various configurations and decisions for the *QoS Policy*. We observe a trade-off between latencies and the *QoS Policy* regardless of data size. Although the *QoS Policy* produces inevitable overhead, the latencies are predictable and small. (3-b) has significant overhead due to the `ros_bridge` transaction. In the (3-b) case, a `ros_bridge` incurs more overhead to communicate with ROS1 and ROS2.

With large data, ROS2 has significant overhead depending on the size of data, as shown in Figure 10. The overhead of ROS2 in (2-b) is attributed to two factors, i.e., data conversion for DDS and processing DDS. Note that ROS2 in (2-a) and (2-b) must convert *messages* between ROS2 and DDS twice. One conversion is from ROS2 to DDS, and the other conversion is from DDS to ROS2. Between these conversions, ROS2 calls DDS APIs and passes *messages* to DDS. Figures 11 and 12 show a breakdown of the end-to-end latencies in the (2-b) OpenSplice `reliable policy` and `best-effort policy`. We observe that ROS2 requires only conversions and processing of DDS. As shown in Figures 11 and 12, there are nearly no transactions for “others”. In addition, note that data size influences both conversions and the DDS processing. Compared to ROS1, the DDS overhead is not constant, and the impact of DDS is notable with large data. As a result, ROS2 has significant overhead with large data, while the impact of DDS depends on the *QoS Policy*.

Furthermore, the influence of shared memory with large data is observed in Figure 10. As data becomes large, notable differences can be observed. However, the influence appears small in Figure 9 because small data hides the impact of shared memory.

Another interesting observation is that the latencies in the (2-c) *intra-process* are greater than the latencies in (1-b) despite using shared memory. This result is not due to conversions for DDS and processing of DDS, because *intra-process communication* does not route through DDS. As ROS2 is in development, that gaps will be closed. *Intra-process communication* needs to be improved.

3.3.3 Comparison within ROS2

End-to-end latencies with data less than 16 KB exhibit similar performance in (2-b). We discuss performance for data of 16 KB to 4 MB.

A comparison of different DDS implementations in (2-b) is shown in Figure 13. We evaluate OpenSplice and Connext with and without shared memory in (2-b) with the `best-effort policy`. Despite shared memory, the performance is not significantly better than that of local loopback. This is caused by marshaling of various tools (e.g., logger and observer), even when using shared memory transport. Moreover, OpenSplice is superior to Connext in terms of latency, as shown in Figure 13, because we use Connext DDS Professional, which has much richer features than the OpenSplice DDS Community Edition. We assume that the performance of Vortex OpenSplice is similar to that of OpenSplice DDS Community Edition. However, Vortex OpenSplice needs a commercial license and is not supported by ROS2.

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`

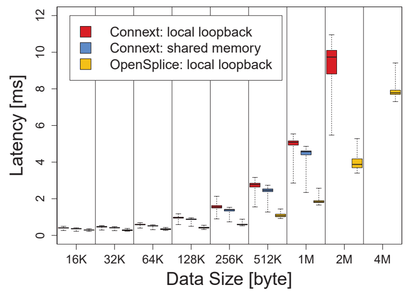

Figure 13: (2-b) Different DDS in ROS2 with `best-effort policy`.

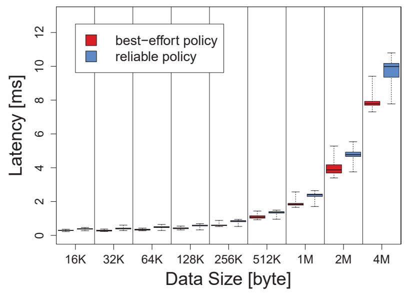

Figure 14: (2-b) Two *QoS policies* in ROS2 with OpenSplice.

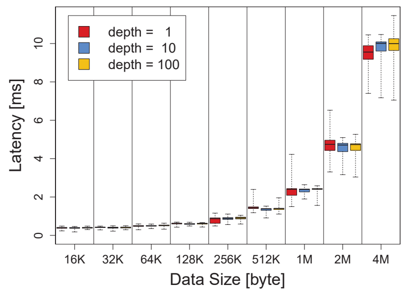

Figure 15: (2-b) Configured `*-depth policy` in ROS2 with OpenSplice.

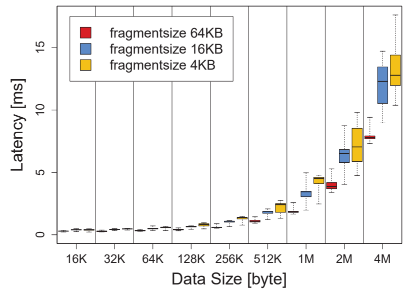

Figure 16: (2-b) Different fragment sizes in ROS2 with OpenSplice `best-effort policy`.

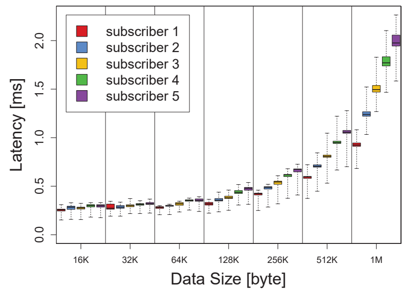

Figure 17: (1-b) ROS1 multiple destinations publisher.

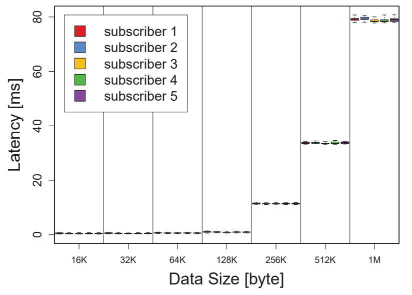

Figure 18: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

and `best-effort policy`. The impact of the *QoS Policy* is shown in Figures 11 and 12. In this evaluation, the network is ideal, i.e., *publisher-nodes* resend *messages* very infrequently. If the network is not ideal, latencies with the **`reliable policy`** increase. The differences in RELIABILITY and DURABILITY in the *QoS Policy* lead to overhead at the cost of reliable communication and resilience against late-joining *Subscribers*. 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.

Finally, fragment overhead is measured using OpenSplice in (2-b) by changing the fragment size to the maximum UDP datagram size of 64 KB. A maximum payload for Connext and OpenSplice originates from this UDP datagram size, because dividing large data into several datagrams has significant impact on many implementations of the *QoS Policy*. As shown in Figure 16, the end-to-end latencies are reduced, as fragment data size increases. With a large fragment size, DDS does not need to split large data into many datagrams, which means fewer system calls and less overhead. In terms of end-to-end latencies, we should preset the fragment size to 64 KB when using large data.

3.3.4 Multiple Destinations Publisher in `local` cases

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 users. 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 much 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 depend on the number of packets. This is same characteristic we learned from Figure 10. Figure 19 also indicates fair latencies and dependency of packets. Although we cannot say that latency 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 *subscriber-nodes* than ROS1 one.

3.4 Throughput of ROS1 and ROS2

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. *Publisher-nodes* repeatedly transport each *message* with 10Hz.

In small data from 256 B to 2 KB, we can observe constant gaps 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 gaps do 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. ROS1 and ROS2 are able to utilize all of available bandwidth and similarly behave in this situation. Throughput is limited by the network and not by DDS.

3.5 Thread of ROS1 and ROS2

In this section, we measure the number of threads on each *node*. Table 6 shows the result of measurements. Note that

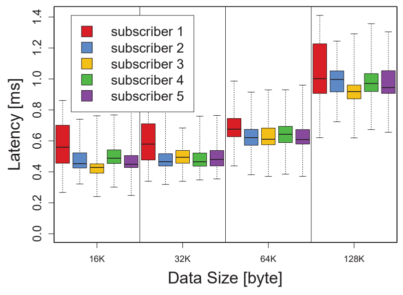

Figure 19: (2-b) ROS2 multiple destinations with OpenSplice `reliable policy`.

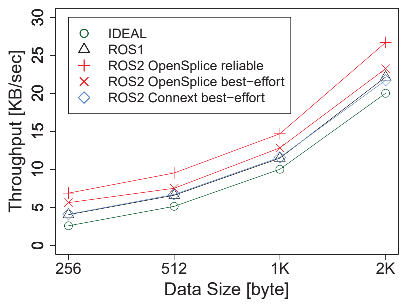

Figure 20: (1-a) and (2-b) `remote` cases throughput with small data.

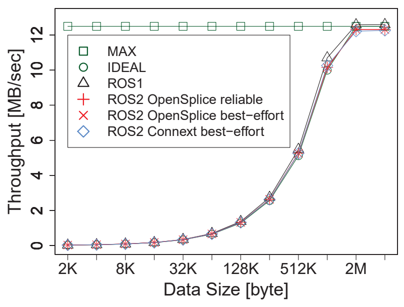

Figure 21: (1-a) and (2-b) `remote` cases throughput with large data.

Table 5: Depth Configurable QoS Policies

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

Table 6: The Number of Thread on ROS1 or ROS2

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

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

the number described in Table 6 depends on DDS configurations including *QoS Policy*. The number does not be fixed by vendors.

First of all, we can observe that a 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. A 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 a *master-node*.

3.6 Memory consumption of ROS1 and ROS2

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.

3.7 Lessons Learned

So far, we have clarified characteristic of DDS implementations through ROS2 from several standpoints: ROS2 capability, latencies, throughput, the number of threads and memory consumption. We can get insight and guidelines for DDS through ROS2 from experimental results. They will be meaningful for DDS and ROS users.

DDS supports *QoS Policy* but there is trade-off between end-to-end latencies and throughput. In the `local` case, overhead latencies of ROS2 are not trivial. From Section 3.3, the latencies are caused by two data conversions for DDS and DDS transaction. DDS end-to-end latencies are 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 increase 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 are trivial, we must consider throughput for bandwidth. As shown in Figure 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 of 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 need a *master-node*. DDS guarantees fair latencies as shown in Figures 18 and 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.

Since ROS2 is under development, we have clarified room for improvement of ROS2 performance and capability to

Table 8: Comparison of ROS2 to Related Work

	Small Embedded	Real-Time	Publish/ Subscribe	Frequent Update	Open Source	Library and Tools	RTOS	Mac/ Windows	QoS
RTM [8]					✓	△		✓	
Extended RTC [10]		✓			✓				
RT-Middleware for VxWorks [17]	✓	△			✓	△	✓	✓	
RTM-TECS [16]	✓	✓				△	✓	✓	
rosc [13]	✓	△	✓		✓		✓		
μROS [19]	△	△	✓		✓	✓	✓		
ROS Industrial [4]	△		✓	✓	✓	✓			
RT-ROS [36]		✓	✓				✓		
ROS1 [28]			✓	✓	✓	✓			
ROS2 [23]	✓	✓	✓	✓	✓	△	✓	✓	✓

maximize DDS potential. First, current *QoS Policies* supported 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. FreeRTPS [22] [27] is a good candidate for this issue but it is under development. 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 configurations options.

4. RELATED WORK

In addition to the ROS, the Robot Technology Middleware (RTM) [8] is well known and widely used for robotics development. In this section, we discuss research related to the ROS and RTM.

RTM: RTM applications consist of Robotic Technology Component (RTC), whose specifications are managed by the OMG [1]. RTM cannot handle hard real-time and embedded systems because it generally uses not real-time CORBA [31] but CORBA [35]. CORBA is an architecture for distributed object computing standardized by the OMG. CORBA manages packets in a FIFO manager and requires significant resources. CORBA lacks key quality of service features and performance optimizations for real-time constraints.

Extended RTC: [10] extends RTC for real-time requirements using GIOP packets rather than CORBA packets. The interface of the Extended RTC provides additional options such as priority management and multiple periodic tasks. However, it is difficult to implement Extended RTC in embedded systems because it is based on only an advanced real-time Linux kernel.

RT-Middleware for VxWorks: Using lightweight CORBA and libraries, [17] enables RTM to run on VxWorks, which is an RTOS, and embedded systems. Nonetheless, [17] did not consider real-time requirements. Furthermore, it uses global variables and cannot run on distributed systems.

RTM-TECS: RTM-TOPPERS Embedded Component Systems (TECS) [16] proposes a collaboration framework of two component technologies, i.e., RTM and TECS. TECS [9] has been added to RTM to satisfy real-time processing requirements. [16] adapted RPC and one-way data transport between TECS components and RTC. RTM-TECS enhances the capability for real-time embedded systems.

rosc: rosc [13] is a portable and dependency-free ROS client library in pure C that supports small embedded systems and any OS. rosc was motivated by a bare-metal, low-memory reference scenario, which ROS2 also targets. While rosc is available as an alpha release, it is in development and has not been updated since 2014.

μROS: μROS [19] is a lightweight ROS client that can run on modern 32-bit micro-controllers. Targeting embedded systems, it is implemented in ANSI C and runs on an RTOS, such as ChibiOS. μROS supports some of the features of ROS and can coexist with ROS1. However, as of 2013, development has ceased.

ROS Industrial: ROS-Industrial [4] is an open-source project that extends the advanced capabilities of ROS software to manufacturing. This library provides industrial developers with the capabilities of ROS for economical robotics research under the business-friendly BSD and Apache 2.0 licenses.

RT-ROS: RT-ROS [36] provides an integrated real-time/non-real-time task execution environment. It is constructed using Linux and the Nuttx Kernel. Using the ROS in an RTOS, applications can benefit from some features of the RTOS; however, this does not mean that the ROS provides options for real-time constrains. To use RT-ROS, it is necessary to modify legacy ROS libraries and nodes. In addition, RT-ROS is not open-source software; therefore, it is developed more slowly than open-source software.

Table 8 briefly summarizes the characteristics of several related methods and compares them to ROS2. ROS1 has more libraries and tools for robotics development than RTM. At present, ROS2 has only a few libraries and packages because it is currently in development. However, by using multiple DDS implementations, ROS2 can run on embedded systems. In addition, by utilizing the capabilities of DDS and RTOSs, ROS2 is designed to overcome real-time constraints and has been developed to be cross-platform. ROS2 inherits and improves the capabilities of ROS1.

5. CONCLUSION

This paper has conducted proof of concept for DDS approach to ROS and arranged DDS characteristic, guidelines, and room for improvement. From various experiments, we have clarified the capabilities of the currently available ROS2 and evaluated the performance characteristics of ROS1 and DDS through ROS2 in various situations from several aspects: latencies, throughput, the number of threads, and memory consumption. Furthermore, we have measured the influence of switching DDS implementations and the *QoS Policies* in ROS2. Understanding each DDS characteristic, we should use a different DDS implementation for different situations. DDS gives ROS2 fault tolerance and flexibility for various platforms. Utilization of DDS is not limited in ROS because ROS2 is one of systems using DDS. Above contributions are valuable for many people.

In future work, we will evaluate real-time applications such as an autonomous driving vehicle [18] 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.

6. REFERENCES

[1] Object Management Group (OMG). http://www.omg.org/.

[2] Open Source Robotics Foundation (OSRF). http://www.osrfoundation.org/.

[3] Point Cloud Library (PCL). http://pointclouds.org/.

[4] ROS Industrial. http://rosindustrial.org/.

[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.

[7] Willow Garage. https://www.willowgarage.com/.

[8] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, and W.-K. Yoon. RT-middleware: distributed component middleware for RT (robot technology). In *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3933–3938, 2005.

[9] T. Azumi, Y. Nagahara, H. Oyama, and N. Nishio. mruby on TECS: Component-Based Framework for Running Script Program. In *Proc. of IEEE International Symposium on Real-Time Distributed Computing*, pages 252–259, 2015.

[10] H. Chishiro, Y. Fujita, A. Takeda, Y. Kojima, K. Funaoka, S. Kato, and N. Yamasaki. Extended RT-component framework for RT-middleware. In *Proc. of IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing*, pages 161–168, 2009.

[11] S. Cousins. Exponential growth of ROS [ROS Topics]. *IEEE Robotics & Automation Magazine*, 1(18):19–20, 2011.

[12] S. Cousins, B. Gerkey, K. Conley, and W. Garage. Sharing software with ROS [ROS topics]. *IEEE Robotics & Automation Magazine*, 17(2):12–14, 2010.

[13] N. Ensslen. Introducing rosc. In *ROS Developers Conference*, 2013.

[14] eProsima. FastRTPS. http://www.eprosima.com/index.php/products-all/eprosima-fast-rtps.

[15] A. Garg. Real-time Linux kernel scheduler. *Linux Journal*, 2009(184):2, 2009.

[16] R. Hasegawa, H. Oyama, and T. Azumi. RTM-TECS: Collabolation Framework for Robot Technology Middleware and Embedded Component System. In *Proc. of IEEE International Symposium on Real-Time Computing*, 2016.

[17] A. Ikezoe, H. Nakamoto, and M. Nagase. OpenRT Platform/RT-Middleware for VxWorks. *ROBOMECH2010*, pages 2A1–F19, 2010.

[18] S. Kato, E. Takeuchi, Y. Ishiguro, Y. Ninomiya, K. Takeda, and T. Hamada. An Open Approach to Autonomous Vehicles. *IEEE Micro*, 35(6):60–68, 2015.

[19] M. Migliavacca and A. Zoppi. μROSnode: running ROS on microcontrollers. In *ROS Developers Conference*, 2013.

[20] Object Management Group (OMG). the Real-time Publish-Subscribe Protocol (RTPS) DDS Interoperability Wire Protocol v2.2. http://www.omg.org/spec/DDSI-RTPS/2.2/, 2014.

[21] Object Management Group (OMG). Data Distribution Services (DDS) v1.4. http://www.omg.org/spec/DDS/1.4/, 2015.

[22] Open Source Robotics Foundation (OSRF). FreeRTPS. https://github.com/ros2/freertps.

[23] Open Source Robotics Foundation (OSRF). ROS2. https://github.com/ros2.

[24] G. Pardo-Castellote. OMG Data-Distribution Service: Architectural Overview. In *Proc. of IEEE International Conference on Distributed Computing Systems Workshops*, pages 200–206, 2003.

[25] PRISMTECH. OpenSplice DDS Community Edition. http://www.prismtech.com/dds-community.

[26] PRISMTECH. Vortex OpenSplice. http://www.prismtech.com/vortex/vortex-opensplice.

[27] M. Quigley. ROS2 on small embedded systems. In *ROS Developers Conference*, 2015.

[28] 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 *Proc. of IEEE International Conference on Robotics and Automation Workshop on Open Source Software*, volume 3, page 5, 2009.

[29] Real-Time Innovations. RTI Connext DDS Professional. http://www.rti.com/products/dds/index.html.

[30] J. M. Schlesselman, G. Pardo-Castellote, and B. Farabaugh. OMG data-distribution service (DDS): Aarchitectural Update. In *Proc. of IEEE Military Communications Conference*, volume 2, pages 961–967, 2004.

[31] D. C. Schmidt, D. L. Levine, and S. Mungee. The design of the TAO real-time object request broker. *Computer Communications*, 21(4):294–324, 1998.

[32] S. Sierla, J. Peltola, and K. Koskinen. Evaluation of a real-time distribution service. In *Proc. of the 3rd International Symposium on Open Control System*, 2003.

[33] D. Thomas, E. Fernandez, and W. Woodall. State of ROS2: Demos and the technology behind. In *ROS Developers Conference*, 2015.

[34] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, M. Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al. Autonomous driving in urban environments: Boss and the Urban Challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[35] S. Vinoski. CORBA: integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2):46–55, 1997.

[36] H. Wei, Z. Shao, Z. Huang, R. Chen, Y. Guan, J. Tan, and Z. Shao. RT-ROS: A real-time ROS architecture on multi-core processors. *Future Generation Computer Systems*, 56:171–178, 2015.

[37] M. Xiong, J. Parsons, J. Edmondson, H. Nguyen, and D. C. Schmidt. Evaluating the performance of publish/subscribe platforms for information management in distributed real-time and embedded systems. *omgwiki. org/dds*, 2009.