International Journal of Parallel Programming (2020) 48:119–136
https://doi.org/10.1007/s10766-019-00647-w

Message Passing Optimization in Robot Operating System

Ziyue Jiang[1] · Yifan Gong[1] · Jidong Zhai[2] · Yu-Ping Wang[2] · Wei Liu[1] · Hao Wu[1] · Jiangming Jin[1]

Received: 5 August 2019 / Accepted: 4 November 2019 / Published online: 16 November 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
With the development of deep learning, autonomous robot systems grow rapidly and require better performance. Robot Operating System 2 (ROS2) has been widely adopted as the main communication framework in autonomous robot systems. However, the performance of ROS2 has become the bottleneck of these real-time systems. From our observations, we find that it can take a large amount of time to serialize complex message in communication, especially for some high-level programming languages, including Python, Java and so on. To address this challenge, we propose a novel technique, called adaptive two-layer serialization algorithm, which can achieve good performance in communication for different kinds of messages. Experimental results show that our algorithm can achieve significant performance improvement over traditional methods in ROS2, up to 93% improvement in our framework. We have successfully applied our proposed techniques in a real autonomous robot system.

Keywords Robot operating system · Message passing · Communication performance

✉ Ziyue Jiang
ziyue.jiang@tusimple.com

Yifan Gong
yifan.gong@tusimple.com

Jidong Zhai
zhaijidong@tsinghua.edu.cn

Yu-Ping Wang
wyp@tsinghua.edu.cn

Wei Liu
wei.liu@tusimple.com

Hao Wu
hao.wu@tusimple.com

Jiangming Jin
jiangming.jin@tusimple.com

1 Tusimple, Beijing, China

2 Tsinghua University, Beijing, China

1 Introduction

Autonomous robot system, which is very sensitive to communication latency, has long been a very hot topic. As a mainstream communication framework, Robot Operating System (ROS [1]), which provides publish/subscribe transport, multiple libraries, and tools to help software developers create nomous robot applications [2], is placed high hopes in such systems (e.g. Apollo [3]). Its fundamental function is to provide modularity and passing message between nomous robot applications.

However, there are two main problems in ROS: high latency in communication and low extensibility of the framework. For high latency, the performance of ROS is limited by its high-latency serialization method and socket communication with TCP/UDP, which brings many times of memory copy during the process of message passing. For low extensibility, ROS couples tightly and tackles applications in different programming language with an incompatible way, which makes it difficult to do general optimization for all the language. Furthermore, it is impossible to write customized plug-in to replace the traditional function, for example the serialization tools and communication protocols.

Robot Operating System 2 (ROS2), as an upgrade from ROS community, replaces the old framework to provide extensibility. It supports different programming language applications with a compatible way. In ROS2, there is a programming language interface layer for each of the programming languages and all these languages share the same lower layer, called ROS2 middleware layer, in the bottom. It means that if we do some general optimization in ROS2 middleware layer, all the applications implemented by different programming languages can share the benefits. Furthermore, ROS2 middleware layer, which adopts Data Distribution Service (DDS [4]) for message definition and serialization, is pluggable and customized from the users. It can be easily replaced by another user-defined plug-ins.

Although ROS2 removes the drawbacks of low extensibility from ROS, these new features bring new problems to the performance, which leads to even higher latency. The reason is that: in communication, the original messages from different programming languages need to be converted in ROS2 middeware layer, which entails large extra overhead.

In order to improve the performance of ROS2, a new efficient method from converting between different programming languages is required. Based on our observation for different types of messages from different programming languages, we find that the structure of the original message largely affects the converting efficiency and the more complex of the message will lead to larger converting overhead. However, reducing the complexity of the original message is very challenging. First, it will change the structure of the converted message after converting, which will affect the serialization efficiency. Second, the reduction will also lead to some extra overhead and how to achieve a good balance between the reduction and converting needs to be carefully designed.

To address the above challenges, we propose a novel technique, called adaptive two-layer serialization algorithm (denoted as ATSA), which can adaptively move parts of the serialization to the programming language interface layer instead of ROS2 middleware layer to reduce the complexity. It can achieve a good balance between serialization and converting for different types of message. We implement our opti-

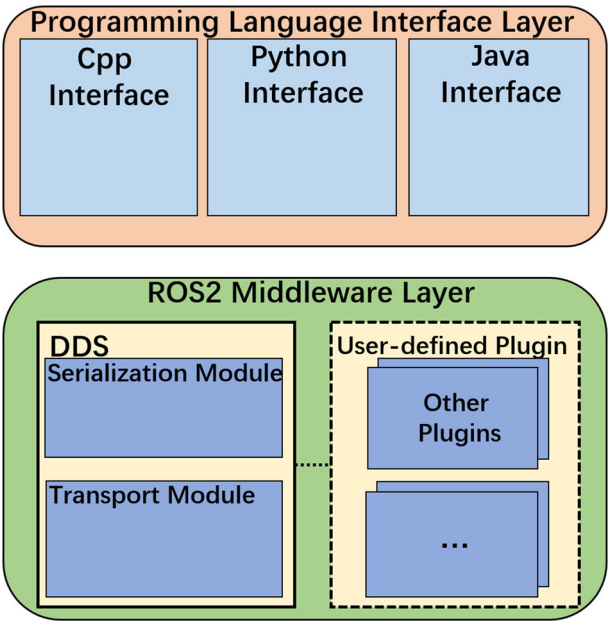

Fig. 1 ROS2 architecture

mization and realize a plug-in in ROS2 for both CPP and Python applications, and utilize the real workflow from autonomous robot system to evaluate the performance improvement. Experimental results show that our optimization can get up to 93% performance improvement over traditional methods in ROS2.

2 Preliminary and Background

2.1 Data Distribution Service

The DDS specification [5] is a group of definitions and standardized APIs for a publish/subscribe data-distribution system, defined and managed by the object management group (OMG). The details of implementation are out of the OMG's responsibility. Many vendors have provided different DDS implementations. The core of DDS is a data-centric publish-subscribe (DCPS) model designed for efficient data transport among processes in distributed platforms. In DDS, each process that publishes or subscribes is called a participant. A global data space in the DCPS model allows read and write operations from any participants with proper interfaces.

2.2 ROS2 Framework

Figure 1 illustrates the architecture of ROS2. We simplify the architecture into two layers. (1) The upper layer is called Programming Language Interface Layer, which is

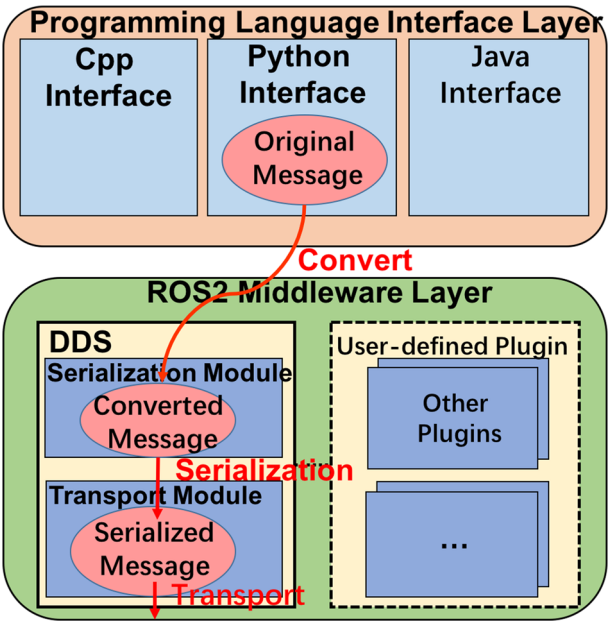

Fig. 2 Message passing process in ROS2

composed of several interfaces from different programming language, including CPP, Python, Java and so on. (2) The lower layer is ROS2 Middleware Layer. It provides some APIs (application programming interfaces) to the upper layer, and these APIs are required to be implemented by those who are preparing to realize the user-defined plugins. ROS2 utilizes DDS to realize the APIs provided by ROS2 Middleware Layer and users do not need to be aware of DDS due to the abstraction layer. This layer also allows ROS2 to have high level configurations and optimizes the utilization of DDS.

2.3 Message Passing Process in ROS2

ROS2 uses `nodes` to organize key components in a distributed system. Nodes in ROS2 applications are composed of independent computing processes, corresponding to a participant in DDS. Nodes contribute to rapid development, fault isolation, modularity, and code reusability. A publish/subscribe model is adopted for communication among nodes. In this model, nodes transmit messages through a topic to communicate. A message is defined by a .msg file with a simple data format (much like C structs).The topic name indicates the content of the message. A node publishes a message to a topic, and then the message is available for another node subscribing to the topic.

Figure 2 shows a detailed process of message publishing in ROS2. We utilize Python message as an example. The original message, used by the users, comes from programming language interface layer of Python. Then the message will be passed to ROS2 Middleware Layer, and converted into the converted message to satisfy DDS's requirements. After that, the message will be serialized by the serialization module and send to another node through socket by TCP/UDP protocol. Subscribing is the dual process,

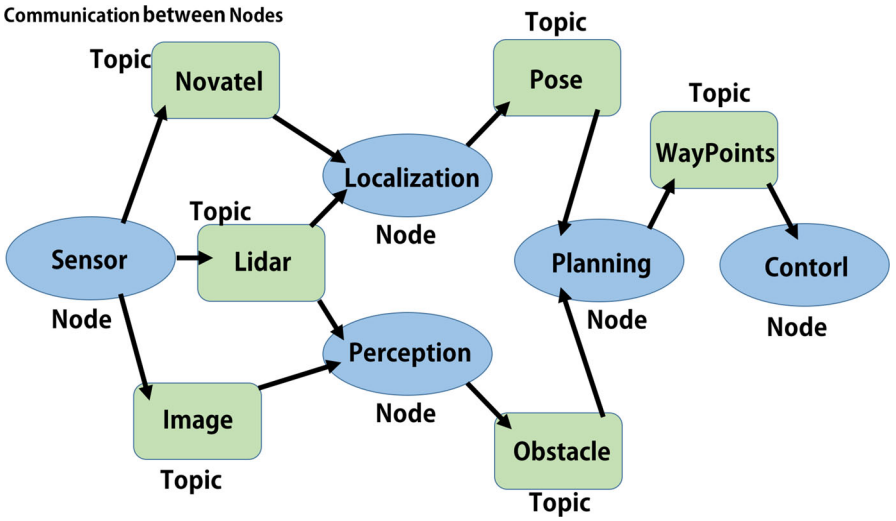

Fig. 3 Communication in autonomous robot systems

which contains three steps: (1) receiving message from socket, (2) deserializing and (3) de-converting to Python.

Figure 3 shows the communication in a typical autonomous robot system. Sensor nodes publish camera figures to the `Image` topic, GPS information to the `Novatel` topic and lidar information to the `Lidar` topic. The localization node subscribes to the `Image` and `Lidar` topic and then publishes position to the `Pose` topic. The perception node generates messages for the `Obstacle` topic through `Novatel` topic. The planning node subscribes to both `Pose` and `Obstacle` as its input, passing the `WayPoints` message to the control node. To be modular at a fine-grained scale makes the publish/subscribe model a good design for distributed systems.

3 Observation

3.1 Communication Cost

We analyze the ROS2's performance problems with a specific example shown in Fig. 3. It originates from a real-world scenario in our autonomous robot system built on the ROS2. We select two types of message, `Image` and `WayPoints`, to measure the total communication cost. There may be some differences in the message structure in other systems, but the problem generally exists. `Image` has only one dimension, who consists of an array of bytes. `WayPoints` contains an array of submessage called `WayPoint`. `WayPoint` includes `Timestamp` (two integer value) and `Pose` data. `Pose` includes much float64 information and `Point` data. Finally, `Point` includes 3 float64 data. `WayPoints` message is complex due to its nested structure but small in the terms of data size. One `WayPoint` only has 1 int32, 1 uint32 and 8 float64, totally 72 bytes.

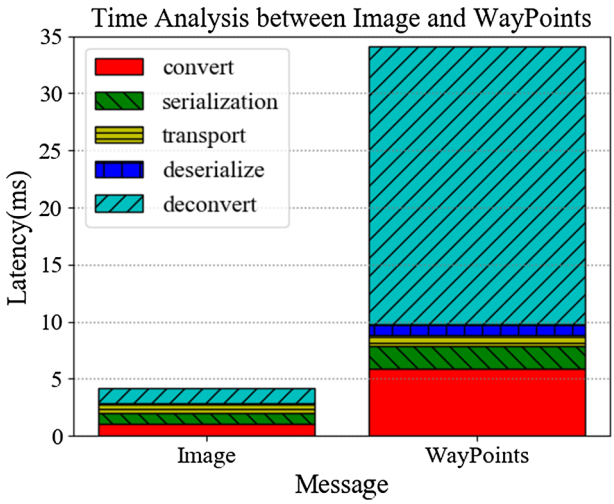

Fig. 4 Communication cost of Image and WayPoints

Figure 4 shows that the communication cost for `Image` and `WayPoints`. The total size of `Image` is 8 MB; while the size of `WayPoint`, who contains 2000 `WayPoint` message, is only 144 KB. We observe that (1) The communication cost `WayPoints` is 34.1 ms, which means the bandwidth is about 4.2 MB/s. `Image` uses just 4 ms but nearly 55 times the data size of the former as a contrast and the bandwidth is 1.97 GB/s. Compared with `Image`, `WayPoints` has much higher latency and lower bandwidth. (2) After analyzing the communication cost of `WayPoints` for each step, we observe that "convert" and "de-convert" accounted for 89.1% of total communication cost. The terrible 24.3 ms of "de-convert" shows that this part is a key bottleneck to improve ROS2's performance in our workflow.

3.2 Converting Cost Study

We further evaluate the converting efficiency for different types of messages due to the high cost in passing `WayPoints` message. To explore this problem, we construct a three-level message shown in Fig. 5 (denoted as A-Msg). One A-Msg includes an array of a two-level message (denoted as B-Msg) and one B-Msg's submessage is an array of C-Msg, which consists of an array of float64. We vary the size of array in each layer and calculate the converting cost. In our experiment, we fix the total number of elements to 10,000 float64.

Fig. 5 Structure of the three-layer message

```
A:
    -B[] b_array
        -C[] c_array
            -float64[] f
```

Table 1 Converting cost for different types of messages

Structure	Number of converting	Cost
1 × 1 × 10,000	2	1.5
1 × 10 × 1000	11	1.6
1 × 20 × 500	21	1.7
1 × 50 × 200	51	2.2
1 × 100 × 100	101	2.8
1 × 200 × 50	201	3.9
1 × 500 × 20	501	8.2
1 × 1000 × 10	1001	14.4
1 × 2000 × 5	2001	27.6
1 × 2500 × 4	2501	35.0
1 × 5000 × 2	5001	63.1
1 × 10,000 × 1	10,001	126.1
10 × 10 × 100	110	2.9
100 × 10 × 10	1100	15.7
500 × 5 × 2	3000	41.6
500 × 10 × 2	5500	73.4

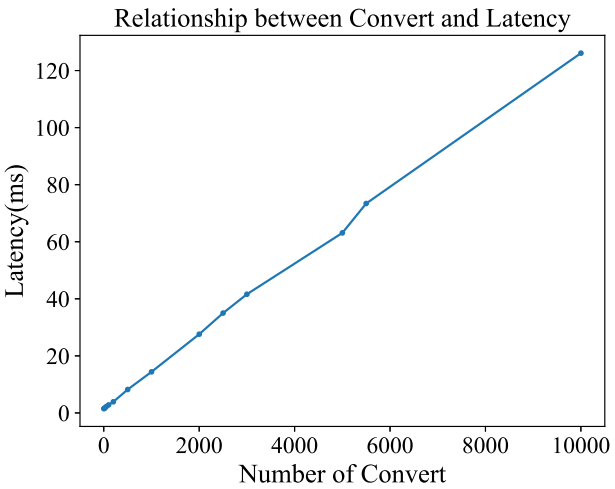

Fig. 6 Results of converting experiment

Table 1 shows the structure of the message, the number of converting and the converting cost. We have two observations: (1) For the fixed size of messages, the converting cost varies from 1.5 ms to 126.1 ms, which means it is very significant to do optimization for converting in the worst case and it can achieve up to 124.6 ms performance improvement. (2) We further calculate the ratio, where the number of converting divides the converting cost and the results are shown in Fig. 6. We observe that the results fit into a line, which means the converting cost linearly increases when the number of converting increases.

3.3 Motivation

In summary, the more complex the message is, the larger cost of converting (de-converting), which means if we can simplify the structure of messages, we can largely reduce the converting (de-converting) overhead. As the simplest structure of message is serialized message, which only includes a continuous buffer in memory, we consider to doing the serialization before converting, so that we can obtain a serialized message as the input of converting process.

Based on this idea, we put the serialization step into Programming Language Layer. However, we find that although the converting cost decreases, the serialization cost increases. The reason is that the high level programming language, for example Python and Java, is not as efficient as C, and the serialization in C is much faster.

Based on these results, we find that there is a trade-off between serialization performance loss and converting performance improvement. Therefore, for each type of message, we can select when to convert and serialize the message to obtain best performance. In other word, there is an opportunity in determining the order of converting and serialization to minimize the total overhead.

4 Adaptive Two-Layer Serialization Algorithm

4.1 Cost Formulation

In the traditional message passing process, the serialization process is in the lower layer (ROS2 Middleware Layer) and we call it **Low-layer Serialization Algorithm** (denoted as LSA). On the contrary, we propose to put the serialization step into Programming Language Layer in order to reduce the converting cost. This method is denoted as **High-layer Serialization Algorithm** (HSA). As publishing and subscribing process are decoupled, we only analyze the publishing process as an example. We can obtain the same conclusions for subscribing.

In **Low-layer Serialization Algorithm**, we consider to publish message x with n layers, where $k_i(x)$ $(i = 1, 2, \ldots, n)$ denotes the number of elements in the i-th layer. We denote converting cost, serialization cost and transport cost as $C(x)$, $S(x)$ and $T(x)$ and the total cost $Cost(x)$ can be expressed in equations below:

$$Cost(x) = C(x) + S(x) + T(x) \tag{1}$$

4.1.1 Converting Cost

In our observations, we find that in the fixed length of message (denoted as $l(x)$), converting cost linearly increases when the number of converting increases. Note that the length of message means the total number of bytes the message occupies. Furthermore, for the fixed message type, the cost is linearly related to the length of message. We utilize a fixed value t_c to represent the converting coefficient and converting cost can be expressed as:

$$C(x) = \sum_{i=1}^{n-1} k_i(x) \times t_c \times l(x) \tag{2}$$

4.1.2 Serialization Cost

Serialization process is very complex, the cost is related to the type of each element and the length of message $l(x)$. We can use a linear function to simulate the cost. In our experiments, we find the total difference from serialization cost and our approximate linear function is very small (usually less than 5%). We use the fixed serialization coefficient (t_s) to describe the linear relationship and serialization cost can be expressed as:

$$S(x) = t_s \times l(x) \tag{3}$$

4.1.3 Total Cost

Based on the analyzing for each step in publishing process, the total cost can be calculated as:

$$Cost(x) = \sum_{i=1}^{n-1} k_i(x) \times t_c \times l(x) + t_s \times l(x) + T(x) \tag{4}$$

In **High-layer serialization algorithm**, we do serialization (denoted as advanced serialization) in programming language interface layer, which will take more time. Compared with the serialization cost $S(x)$ in ROS2 Middleware Layer, the cost $\bar{S}(x)$ of advanced serialization is α times larger. But the number of converting can be reduced to one copy and the converting cost $\bar{C}(x)$ is equal to $t_c l(x)$. The transport cost $\bar{T}(x)$ is equal to $T(x)$. Based on these analysis, the cost can be expressed as:

$$\begin{aligned} \bar{Cost}(x) &= \bar{C}(x) + \bar{S}(x) + \bar{T}(x) \\ &= t_c \times l(x) + (1+\alpha) \times t_s \times l(x) + T(x) \end{aligned} \tag{5}$$

4.2 Threshold Definition

The difference $D(x)$ between the two algorithms can be expressed as:

$$\begin{aligned} D(x) &= Cost(x) - \bar{Cost} \\ &= \left(\sum_{i=1}^{n-1} k_i(x) \times t_c l(x) + t_s l(x) + T(x)) - (t_c l(x) + (1+\alpha) t_s l(x) + T(x) \right) \\ &= \left[\left(\sum_{i=1}^{n-1} k_i(x) - 1 \right) t_c - \alpha t_s \right] l(x) \end{aligned}$$

$$= \left(\sum_{i=1}^{n-1} k_i(x) - \frac{t_c + \alpha t_s}{t_c} \right) \times l(x) \times t_c \tag{6}$$

We define $K(x) = \sum_{i=1}^{n-1} k_i(x)$ and a constant value $Threshold = \frac{t_c + \alpha t_s}{t_c}$ and Eq. 6 can be simplified to:

$$D(x) = [K(x) - Threshold] \times l(x) \times t_c \tag{7}$$

From Eq. 7, we can conclude that if $K(x)$ is larger than $Threshold$, $D(x)$ will be larger than 0, which means the total cost of **LSA** is larger than the cost of **HSA**. On the contrary, if $K(x)$ is smaller than $Threshold$, **LSA** is better because of the lower cost.

4.3 Algorithm Design

Algorithm 1 Adaptive Two-layer Serialization Algorithm in publisher

```
1: Function: Publish
2: Parameter: Threshold
3: Input: Message x
4: Output: Serialized Message y
5: if K(x) > Threshold then
6:     x = Advanced-Serialization(x)
7: end if
8: x = Converting(x)
9: if K(x) > Threshold then
10:    y = Memcpy(x)
11: else
12:    y = Serialization(x)
13: end if
```

4.3.1 Basic Algorithm

Based on the analyzing of total cost, if the message x is given, we can select the suitable algorithm to do publishing. In fact, function $K(x)$ can represent the complexity of message x. If $K(x)$ is larger than $Threshold$, which means message x is complex enough, we can utilize advanced serialization in the upper layer and only do memory copy after converting. Otherwise, we use the traditional algorithm to publish message. We propose the Adaptive Two-layer Serialization Algorithm in publishing, as illustrated in Algorithm 1, to improve the performance in message passing. In Algorithm 1, we combine **HSA** and **LSA** together and adaptively determine which approach to select based on the **Selection Condition** in Line 1 and 5.

4.3.2 Subscribing Problem

For the subscribing process, we cannot obtain the message before de-converting step and we cannot determine when to do deserialization based on the same method as publishing. The solution is that, we add the information, that whether $K(x)$ is larger or smaller than $Threshold$, to the serialized message and send to the subscriber. In the subscribing process, we add an pre-deserialzation step to obtain this value (bool value) to decide which method to use. This design introduces a small overhead due to packing and unpacking the bit into a message. The overhead will be calculated as a part of de-converting cost when evaluating the latency of the whole message passing process. The Algorithm is shown in Algorithm 2.

Algorithm 2 Adaptive Two-layer Serialization Algorithm in subscriber

```
1: Function: Subscribe
2: Input: Serialized Message x
3: Output: Message y
4: Condition, x = Pre-deserialization(x)
5: if Condition then
6:     x = Memcpy(x)
7: else
8:     x = Deserialization(x)
9: end if
10: y = De-converting(x)
11: if Condition then
12:     y = Advanced-Deserialization(y)
13: end if
```

4.4 System Overview

Figure 7 shows our proposed system, which is based on ROS2. We create our plugin in ROS2 Middleware layer and add an advance-serialization module in Programming Language Interface Layer. In publishing the message, we first calculate the **Selection Condition** (Line 1 and 5 in Algorithm 1) and then determine the order of serialization and converting.

The red solid line shows the communication process for relatively complex messages. The publisher do serialization in advance, extracting this module from the DDS implementation to the programming language interface layer. Then the serialized messages are passed to the bottom layer without converting. Messages are copied into transport slots using in-memory representation. The subscriber receives the messages and directly pass them to the top layer. The de-serialization module in Programming Language Interface Layer deserializes the messages. The blue dotted line represents the communication process for flat-structure messages. This process is similar to the common ROS2, except that in-memory representation is replaced with our implementation for compatibility.

Before publishing messages, we pack the **Selection Condition** into the messages as their first bit and unpack it while subscribing.

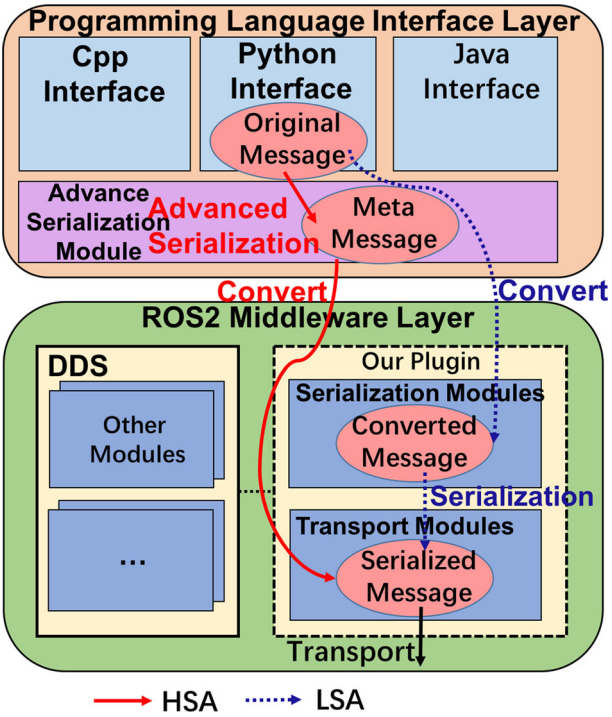

Fig. 7 Two-layer Serialization System Overview (Color figure online)

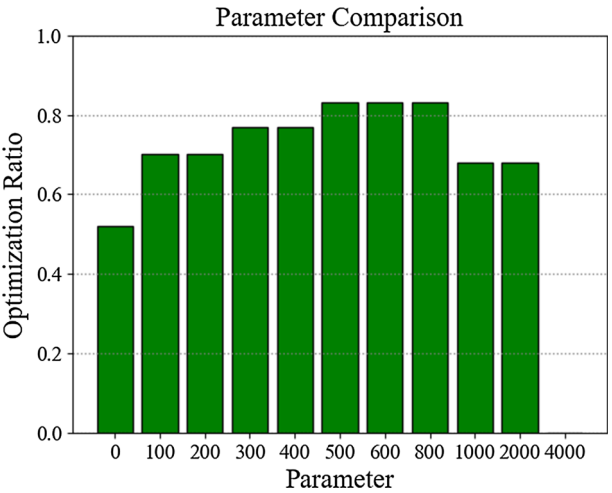

Fig. 8 The average optimization effect of different parameters

5 Evaluation

5.1 Experimental Setup

5.1.1 Platform

Our experiment is based on ROS2 Bouncy Version [6]. We run our experiments on a server with Interl(R) Core(TM) i7-7700K CPU (4.2 Ghz 8 cores 32 GB memory). The operating system is Ubuntu 16.04. All the publishers and subscribers are implemented by Python.

5.1.2 Comparisons

We select ROS2, which use Low-level Layer Serialization Algorithm, as **Baseline**. The Low-level Layer Serialization Algorithm is denoted as LSA. The High-level Layer Serialization Algorithm is denoted as **HSA** and our proposed Adaptive Two-layer Serialization Algorithm is denoted as **ATSA**.

5.1.3 Dataset

We compare different algorithms in our real autonomous robot system workflow. The structure of the workflow is shown in Fig. 3. Our dataset is based on the real data recorded from the workflow. It contains several different types of message, including `Image`, `Lidar`, `Novatel`, `Pose` and `WayPoints`. `Image` has only one dimension, who consists of an array of bytes. The total size of `Image` is 8 MB. `Lidar` is the most complex one in the five message types. It contains 1000 packets and each packet contains 1250 B. The total size is 1220 KB `Novatel` is a frame of GPS information, which consists of some simple sub-messages like timestamp and other geometry information. The total size is 142 B. `Pose` is a relatively flat message, which includes two-layer sub-messages like position(x,y,z). The total size is 400 B. `WayPoints` contains an array of submessage called WayPoint, which includes timestamp and Pose data. 2000 WayPoints' total size is 140 KB. Furthermore, we evaluate the end-to-end performance of the whole workflow.

5.2 Experimental Results

5.2.1 Parameter Study

An experiment is designed to determine the best $Threshold$ parameter. We first find the range [0, 8000], which cause the ATSA to downgrade into HSA and LSA. Then we vary the value and calculate the average performance improvement for all the message in our selected Dataset.

Figure 8 shows the optimization effect of different $Threshold$ parameter. When the $Threshold$ is set to 500–800, the average cost of all messages in our system becomes lowest, achieveing 83% optimization effect. The parameters smaller than 400 cause

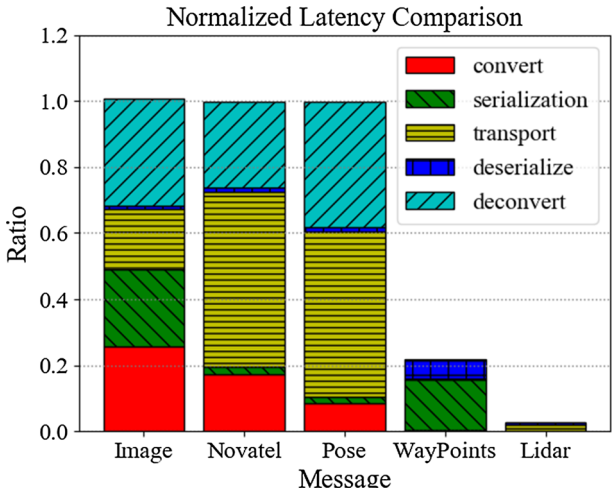

Fig. 9 Overall performance comparison between ROS2 and ATSA

many flat meesages to use HSA, which means unnecessary cost of serialization. And those parameters larger than 1000 may not cover all complex messages, thus suffering from converting costs.

5.2.2 Overall Performance Comparison

This experiment is designed to evaluate the overall performance. A publisher sends to a subscriber in a single machine. We send 2000 number of message for each selected type of message in our dataset and calculate the average performance improvement.

Figure 9 shows the overall performance comparison between ROS2 and ATSA. It includes the detailed latency for each step of message passing. As the sizes of message are different for each type, we normalize the communication latency to **Baseline**. We have two major observations: (1) For `Image`, `Novatel` and `Pose`, Baseline and ATSA have similar performance. Because these messages are not complex and ATSA will select LSA to pass the message. The extra overhead for packing and unpacking a subscribing bit, which we have introduced before, has a tiny impact. (2) For `WayPoints` and `Lidar`, ATSA outperforms the Baseline with 78–95% performance improvement. The reason is that the converting cost for the two messages are very large in LSA. `Lidar` achieves a better optimization effect, given that converting cost dominates its message passing process.

5.2.3 Performance Comparison for Different Algorithms

This experiment is designed to evaluate the efficiency of our proposed **adaptive** method. We select `Image` and `WayPoints` to represent two relatively extreme cases for different message types. For both types, we send 2000 number of message and calculate the average performance improvement.

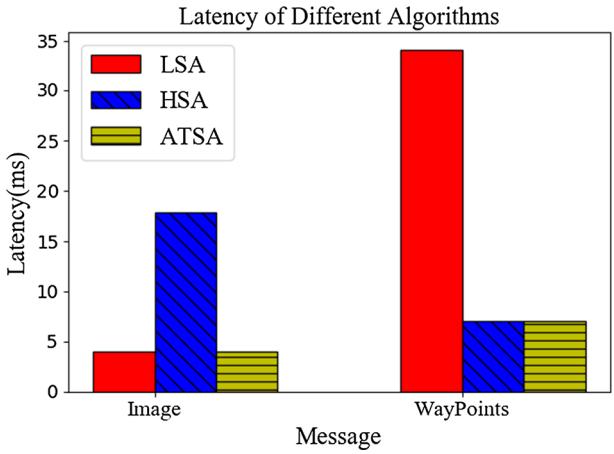

Fig. 10 Algorithm between LSA, HSA and ATSA

Figure 10 shows the performance comparison for different algorithms, including LSA, HSA and ATSA. The results show that ATSA has the best performance in both of the two cases. For `Image`, ATSA can obtain 77.6% performance improvement compared with HSA; while for `WayPoints`, ATSA can get 79.2% performance improvement compared with LSA. Using either LSA or HSA alone can't achieve the best results.

5.2.4 Workflow Performance Comparison

This experiment is designed to evaluate ATSA's effect in a real scene. We use ROS2 and ATSA respectively as the middleware in our autonomous robot system. There are more than ten nodes with publishers and subscribers in the test environment. The message is passed in accordance to the workflow. The total latency in our system is the sum of latency in the critical path, which includes three types of message, `Lidar`, `Pose` and `WayPoints`: The lidar sensor nodes publish `Lidar` to localization node. The localization node subscribes `Lidar` and publishes `Pose`. The planning node subscribes `Pose` as its input, passing the `WayPoints` message to the control node.

Figure 11 shows the performance comparison results.

Our proposed algorithm can achieve up to 93% performance improvement in communication. ATSA can largely reduce the latency of `Lidar` and `WayPoints` decrease the total communication latency from 160 ms to about 10 ms. The performance improvement is very significant in real-time autonomous robot system, which is very sensitive to latency.

5.3 Summary

Based on the experiment, our proposed algorithm indeed improves the performance of message passing process with complex structures. Besides, our design still take

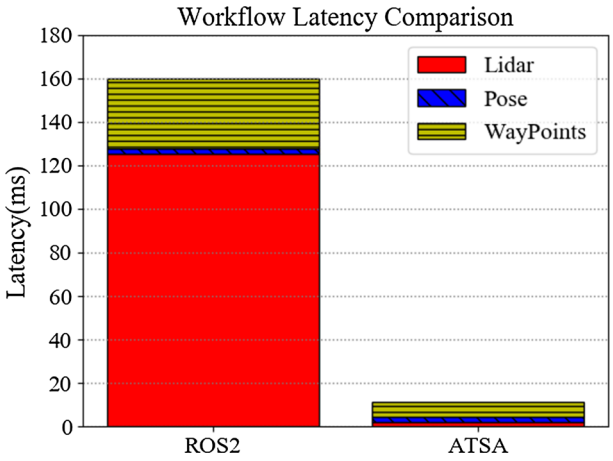

Fig. 11 Workflow latency comparison

advantage of the raw ROS2's message passing process, thus keeping low latency when transporting flat messages. Overall, ATSA can obtain 93% performance improvement in the whole workflow.

6 Related Work

6.1 Serialization in Robotics Middleware

Many projects designed for robotics middleware have been developed in recent years. A survey and comparison of the different projects is given in [7]. Message serialization are important functions for these robotics middleware.

Robot Operating System 1 (ROS1) aims to provide a general communication method with sockets. Users define a message simply with a file and the ROS1 automatically generates message types as well as the serialization functions for different programming languages. However, the serialization and de-serialization functions are strictly determined by the ROS1, which means users are not allowed to use the serialization method they want.

Middleware for Robotic Applications (MIRA) [8] aims to provide a middleware with low-latency communication mechanism. MIRA adopts a "reflect" serialization method in high-level languages, which allows complex objects with functionalities to be transported. However, no matter how simple a user-defined class is, MIRA requires users to implement a "reflect" method themselves to help do serialization.

The Lightweight Communications and Marshalling (LCM) library [9] aims to simplify the development of low-latency message passing systems. LCM provides marshalling library to do data pack. Extra information is written into data to verify the marshalled serialized messages. LCM mostly takes the convenience into account when developing instead of serialization performance.

Yet Another Robot Platform (YARP) [10] aims to contribute to humanoid robotics applications by promoting the development efficiency. YARP provides serialization

and deserialization for simple data types. However, users have to write their own C++ templates to provide serialization and deserialization for complex data types, which brings some inconvenience.

This survey shows different serialization mechanisms adopted in current robotics middleware. Most robotics middleware provide unscalable serialization methods. To provide scalability, ROS2 is designed to support customized DDS plug-ins and more programming languages, suffering the loss when passing complex messages during converting between different languages and message types. We summary communication mechanisms mentioned on the above middleware and try to balance the scalability and performance of ROS2. This optimization makes an effort to reduce the latency of passing complex messages while preserving the advantage of ROS2.

6.2 Latency in the ROS

Mush emphasis has been placed on the latency in the ROS like [11,12]. RT-ROS [13] constructs a real-time ROS architecture to allow real-time and non-real-time ROS nodes separately running with different processor cores respectively, achieving a higher performance than the ROS1.

Towards Zero-Copy (TZC) [14] aims to reduce the times of data copy and serialization in the ROS2. TZC uses an algorithm called partial serialization, cutting a message into two parts. By only serializing the necessary information in one part, TZC improves the general serialization process and the whole latency.

Hardware ROS-compliant FPGA Component [15] implements a FPGA-based message passing accelerator to improve the publish/subscribe performance in the ROS. The research uses small amount of hardware and utilizes the high performance of hardware algorithm to reduce the latency in message passing process.

The uROSnode [16] is designed to help the ROS communicate with embedded systems. The embedded system can do a part of work that ROS does in a fast and lightweight way.

7 Conclusion and Future Work

Since the latency of the ROS2's message passing process is limited by the converting process, we propose an Adaptive Two-layer Serialization Algorithm to adaptively determine the order of converting and serialization. The ATSA aims to minimize the total costs of converting and serialization for different message types. Experimental results show that our design can obtain 93% performance improvement in our real autonomous robot workflow.

Currently, we only implement the ATSA on ROS2 for Python and C++. By modifying the programming language interface layer and evaluating the serialization efficiency of target programming languages, we could implement the ATSA for other languages that ROS2 supports, such as Java and JavaScript. And we verify the optimizations and do the experiments on Linux now. We plan to evaluate ATSA-ROS2 on

more platforms like Windows and Raspberry Pi. It would be interesting and important to support more programming languages and platforms.

In addition, we set the threshold parameter through many experiments and manually select the best parameter. To make the process automatically, a basic idea is to simulate what we do by ATSA itself before a system starts, generating a proper parameter with binary search.

Furthermore, we find that transport cost becomes one of the most important factors since converting cost has been reduced by ATSA. Shared memory is generally a faster transport method than socket-based method. We plan to take efforts to optimize the transport cost in future.

Acknowledgements We would like to thank the anonymous reviewers for their valuable comments. This work is partially supported by the National Key R&D Program of China (Grant No. 2016YFB0200100), National Natural Science Foundation of China (Grant No. 61722208).

References

1. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., Ng, A.Y.: ROS: an open-source robot operating system. In: Proceedings of IEEE International Conference on Robotics and Automation Workshop on Open Source Software, vol. 3 (2009)
2. Maruyama, Y., Kato, S., Azumi, T.: Exploring the performance of ROS2. In: Proceedings of the 13th International Conference on Embedded Software. ACM (2016)
3. Baidu Appollo. http://apollo.auto/
4. Pardo-Castellote, G.: OMG data-distribution service: architectural overview. In: Proceedings of IEEE International Conference on Distributed Computing Systems Workshops (2003)
5. Data Distribution Services (DDS) v1.4, (2015). https://www.omg.org/spec/DDS/1.4/PDF
6. ROS2 Bouncy Bolson. https://index.ros.org/doc/ros2/
7. Elkady, A., Sobh, T.M.: Robotics middleware: a comprehensive literature survey and attribute-based bibliography. J. Robot. (2012)
8. Erik, E., et al.: Mira-middleware for robotic applications. In: 2012 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE (2012)
9. Huang, A.S., Olson, E., Moore, D.C.: LCM: lightweight communications and marshalling. In: 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems. IEEE (2010)
10. Metta, G., Fitzpatrick, P., Natale, L.: YARP: yet another robot platform. Int. J. Adv. Robot. Syst. **3**(1), 8 (2006)
11. Wei, H., Huang, Z., Yu, Q., Liu, M., Guan, Y., Tan, J.: RGMP-ROS: a real-time ROS architecture of hybrid RTOS and GPOS on multi-core processor. In: 2014 IEEE International Conference on Robotics and Automation (ICRA) May 31, pp. 2482–2487. IEEE (2014)
12. Saito, Y., Sato, F., Azumi, T., Kato, S., Nishio, N.: ROSCH: real-time scheduling framework for ROS. In: 2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA) Aug 28, pp. 52–58. IEEE (2018)
13. Wei, H., Shao, Z., Huang, Z., Chen, R., Guan, Y., Tan, J., Shao, Z.: RT-ROS: a real-time ROS architecture on multi-core processors. Fut. Gener. Comput. Syst. **1**(56), 171–8 (2016)
14. Wang, Y.-P., Tan, W., Hu, X.-Q., Manocha, D., Hu, S.-M.: TZC: efficient inter-process communication for robotics middleware with partial serialization (2018). arXiv:1810.00556
15. Sugata, Y., Ohkawa, T., Ootsu, K., Yokota, T.: Acceleration of publish/subscribe messaging in ROS-compliant FPGA component. In: Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies Jun 7, p. 13. ACM (2017)
16. Maruyama, Y., Kato, S., Azumi, T.: Exploring the performance of ROS2. In: Proceedings of the 13th International Conference on Embedded Software Oct 1, p. 5. ACM (2016)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.