



ROS-SF: A Transparent and Efficient ROS Middleware using Serialization-Free Message

Yu-Ping Wang* BNRist, Tsinghua University China wyp@tsinghua.edu.cn Yuejiang Dong BNRist, Tsinghua University China dongyj21@mails.tsinghua.edu.cn Gang Tan The Pennsylvania State University, University Park USA gtan@cse.psu.edu

ABSTRACT

In recent years, ROS becomes the dominant middleware for robotic systems. The performance of its message-passing paradigm is crucial to the robot's reaction time. However, previous works only focus on efficiency, but ignore the requirement for transparency. We present ROS-SF framework, which can transparently eliminate serialization and de-serialization under the ROS APIs. The key contributions are a new serialization format called SFM and a life-cycle management method for serialization-free messages. Evaluation results show that our ROS-SF framework can improve the message-passing performance of ROS by up to 76.3%. Application case study and applicability study show that our ROS-SF framework can be transparently applied to many existing ROS-based systems and packages. Even in the failure cases, our ROS-SF framework can provide modification guidance.

CCS CONCEPTS

 Software and its engineering → Message oriented middleware; Message passing.

KEYWORDS

serialization-free message; robot operating system; message passing middleware

ACM Reference Format:

Yu-Ping Wang, Yuejiang Dong, and Gang Tan. 2022. ROS-SF: A Transparent and Efficient ROS Middleware using Serialization-Free Message. In 23rd International Middleware Conference (Middleware '22), November 7–11, 2022, Quebec, QC, Canada. ACM, New York, NY, USA, 12 pages. https: //doi.org/10.1145/3528535.3531518

1 INTRODUCTION

Robot Operating System (ROS) [25] is an open-source robotics middleware suite that helps developers build robotic systems. Originally, developing a new robotic system is laborious, because robotic systems are usually involved in various fields, such as mechanical control, computer vision, natural language processing, etc. With the help of thousands of open-sourced ROS packages, developers

*Corresponding Author



This work is licensed under a Creative Commons Attribution International 4.0 License. *Middleware '22, November 7–11, 2022, Quebec, QC, Canada* © 2022 Copyright held by the owner/author(s). ACM ISBN 978-1-4503-9340-9/22/11. https://doi.org/10.1145/3528535.3531518 can select and reuse existing ROS packages to implement common functionality and focus on their own new features. Thus, ROS can greatly reduce the effort and cost of developing a new robotic system, and it has become the dominant middleware for robotic systems in recent years. One of the core features of ROS is its publish/subscribe message-passing paradigm, which decouples data dependencies within robotic programs. Robotic programs are connected by the message-passing paradigm to form a complete robotic system. The performance of the message-passing paradigm is crucial to the robot's reaction time.

Message-passing among software modules in the form of interprocess communication (IPC) is the foundation of distributed systems. Structured data, such as trees, usually occupies discontinuous memory segments that are connected by references or pointers. Such structured data is hard to be transmitted over the wire directly, and it has to be converted into a continuous buffer before transmission. The routine that converts structured data into a continuous buffer is called serialization (or marshalling). Correspondingly, the inverse routine that converts the continuous buffer back into structured data is called de-serialization (or unmarshalling). In order to transmit structured data, serialization techniques have become a standard procedure for decades. All kinds of transmission middleware, including ROS, can automatically generate serialization and de-serialization routines based on their interface definition language (IDL). However, serialization/de-serialization routines introduce time and space overhead, especially for large messages. Traditionally, the time cost is negligible compared to network transmission time. However, with the development of high-speed networks, the network bandwidth has increased from 100Mbps to 100Gbps. As a result, the network latency of transmitting large buffers has been reduced tenfold or even hundredfold, and the time cost caused by serialization is not negligible anymore.

If we can eliminate serialization/de-serialization routines by directly constructing and accessing a message as a serialized buffer, the message-passing performance can be greatly improved. The Flat-Data solution [4] in the Connext DDS [27] by RTI and the FlatBuffer by Google [9] have considered this idea and provide APIs for socalled *Serialization-Free Messages*. A serialization-free message is a message whose memory layout is the same as a serialized buffer. Thus, for a serialization-free message, serialization on the sender side and de-serialization on the receiver side can both be eliminated, and the overall transmission latency can be significantly reduced. However, their main drawback is that the message-constructing APIs are greatly different from the APIs for constructing ordinary messages. In order to benefit from those serialization-free messages, developers must rewrite their code to adapt. For ROS developers, this is a laborious and even impractical solution, since there have been over two thousands ROS packages, which are open-sourced and maintained by different developers.

In this paper, we present ROS-SF, an optimized ROS middleware that can transparently support serialization-free messages and improve the message-passing performance. To achieve our goal, we overcome two challenges. Firstly, for the transparency issue, we find that previous serialized message formats used by FlatData and FlatBuffer cannot meet the requirement. We design a new serialization format, named SFM (Serialization-Free Message). The key idea of our SFM format is to keep the memory layout of messages as similar as possible to the representation of regular messages. Secondly, for the efficiency issue, we carefully manage the life-cycle of serialization-free messages, especially when interacting with developers' code, so that it is guaranteed no memory copy introduced in the whole transmission process. Experimental results show that our ROS-SF framework can transparently improve the overall messagepassing performance for practical scenarios.

The main contributions of this paper include:

(1) We propose a new serialization format, namely SFM. By employing our SFM format, serialization-free messages can be constructed and accessed in a transparent way for ROS.

(2) We design and implement ROS-SF, which optimizes ROS with our SFM format and a life-cycle management method for serialization-free messages.

(3) We perform a performance evaluation. The results show that our ROS-SF framework can reduce the overall message-passing latency.

(4) We apply this technique to practical robotic scenarios and analyze some failure cases.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 discusses the background and motivation of our work. Section 4 introduces the details of our ROS-SF framework. Section 5 evaluates the message-passing performance of our ROS-SF framework. Finally, we conclude in Section 6.

2 RELATED WORK

2.1 Message Passing Middleware

Passing messages among software modules is a fundamental problem called inter-process communication (IPC) [5]. By the location of modules, IPC can be classified into 3 categories: intra-process, intra-machine, and inter-machine. Intra-process IPC handles the situation when modules are located in the same memory space. Our ROS-SF framework mainly focuses on improving the performance of intra-machine and inter-machine IPC.

When modules are located in different memory spaces of the same machine, the shared memory mechanism can provide solutions to achieve efficient intra-machine IPC [12, 16, 34]. These systems can achieve high performance by constructing messages in shared memory, and directly sharing them among modules. However, their main drawback is that these solutions are hard to keep compatible with the inter-machine IPC.

When modules are located across different machines, inter-machine IPC has to employ network as the transmission channel. In this aspect, most researchers have focused on improving the performance of transmitting a continuous buffer, rather than a structured message. Kurmann et al. [17] proposed a zero-copy solution for COBRA, which avoids message copying in kernel space. Huang et al. [15] proposed the LCM library, which employs the multi-path UDP protocol when possible to improve transmission performance. They also designed a lightweight IDL to define message classes, but still employed the traditional External Data Representation (XDR) [29] as the serialization format. Rao et al. [26] employed the multi-path TCP protocol to balance performance and reliability.

General-purpose transmission middleware strives to unify all of the 3 categories of IPC mentioned above, and relieve the burden of developers. Some transmission middlewares (such as DDS [31], CORBA [33], and ROS [25]) design their own IDL to generate message classes, serializers and de-serializers. Their performance can be improved by using customized approaches similar to our ROS-SF framework. Other transmission middlewares (such as MPI [11] and ZeroMQ [14]) focus on continuous buffer transmission. Their applicability can be improved by using our ROS-SF framework, since our ROS-SF framework can construct serialization-free messages, whose memory layout is a continuous buffer.

2.2 Serialization Methods

Serialization is a traditional technique that transforms abstract data types into byte buffers for communication or persistent storage [13]. Sun Microsystems first published the External Data Representation (XDR) in 1987 [20] which is currently standardized as RFC 4506 [6]. A serialized message of XDR can be divided into smaller continuous memory segments which are recursively defined with the format of each serialized field. This principle has a great influence on other serialization formats, such as Common Data Representation (CDR, used by CORBA) and its successor, Extended CDR encoding version 2 (XCDR2, used by DDS) [32]. Protocol Buffers (ProtoBuf) [10] and MessagePack [8] introduce prefix encoding into the serialization format, which can potentially reduce the size of messages with small values, and thus reduce the transmission latency. But it also introduces more time overhead to the serialization and de-serialization routine.

In the late 1990s, XML [3] was designed to make the serialized messages human-readable. The message in XML format can be easily embedded into a web page. The design principle of XML has a great influence on JSON [2] and YAML [1]. However, the size of a serialized message in plain-text formats is generally larger than that in binary formats.

Writing serialization routines manually is error-prone. Therefore, most serialization frameworks support to automatically generate serialization routines by IDL. Furthermore, serialization is supported by many programming languages, such as Java (Java Object Serialization) [23] and Python (Pickle) [24]. But their serialization formats share the same principles above.

2.3 Serialization-Free Frameworks

To the best of our knowledge, FlatData by RTI [4] and FlatBuffer by Google [9] are the only two serialization-free frameworks so far. FlatData uses the same serialization format with regular messages (i.e., XCDR2), to provide better compatibility. As a result, a FlatData message can only be constructed in a recursive order, although the order among fields is not restricted. Besides, constructing and ROS-SF: A Transparent and Efficient ROS Middleware using Serialization-Free Message

Middleware '22, November 7-11, 2022, Quebec, QC, Canada

accessing such a message require extra classes, i.e., *Builder* classes and *Offset* classes, which hurts the transparency. The serialization format designed for FlatBuffer is inspiring, but still has drawbacks. The FlatBuffer APIs for constructing and accessing a message are similar with FlatData, and have the same restrictions. Our SFM serialization format introduces a new principle that the memory layout of a serialized message is as much similar as possible to the regular messages. As a result, our SFM format enables source-code level transparency. Furthermore, both FlatData and FlatBuffer leave the problem of managing the life cycle of messages to developers, but our ROS-SF framework carefully addresses these problems and manages the life cycle of messages transparently under the ROS APIs.

Naos [30] is another interesting direction of serialization-free framework. Rather than organizing a message into a continuous buffer, Naos directly transmits the message segment by segment, and rebuilds connections between segments after received. Naos is transparent, but their transparency means that developers do not need to manually split messages into segments, and developers still need to rewrite their code to use Naos API.

3 BACKGROUND AND MOTIVATION

For better understanding, in this section, we take image transmission process as an example, to show the program pattern of ROS, FlatData, and FlatBuffer. For simplicity, the message used in this section is a simplified *Image*, whose representation is shown in Fig. 1. Each *Image* message contains fields named *encoding*, *height*, *width*, and *data*, where *encoding* is a string that indicates the encoding method, *height* and *width* are integers that indicate the height and the width of the image respectively, and *data* is a sequence of bytes that stores the color of pixels.

Image	
string	encoding
uint32	height
uint32	width
uint8[]	data

Figure 1: A simplified *Image* representation.

3.1 Program Paradigm of ROS

For ROS, the representation in Fig. 1 is automatically translated into a C++ structure, shown in Fig. 2. The size of C++ structures is fixed, so the variable-size data, such as *std::string* and *std::vector*, should be stored in allocated heap segments. Therefore, such structures need to be serialized before transmission, and de-serialized after received and before accessed.

A typical program pattern of ROS is shown in Fig. 3. A more detailed version can be found in the official tutorial of ROS¹. On the *Publisher* side, a *NodeHandler* object is created first, which represents the process that the program runs within. By calling the *advertise* API, a new *Topic* is declared, which is a communication channel in ROS, and a *Publisher* object is created which will be used to publish messages later. Then, an *Image* object is defined,

¹http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29

```
struct Image {
    std::string encoding;
    uint32_t height;
    uint32_t width;
    std::vector<uint8> data;
};
```

Figure 2: ROS representation.

which represents an image to be transmitted. In this example, the Image object is assigned with 10×10 pixels, and each pixel consists of 3 bytes, thus the total length of *data* is 300 bytes. It is then published via the Publisher object. Inside the publish API, the Image object is serialized into a buffer with the serialization routine automatically generated by ROS, and the buffer is then put into a transmission queue. On the Subscriber side, a callback function is defined, which takes a constant smart pointer (Image::ConstPtr is defined as std::shared_ptr<const Image>) as its argument. A Node-Handler object is also created first. By calling the subscribe API, the callback function is registered to ROS. When the buffer from the Publisher side arrives, it is de-serialized into an Image object, and the callback function is triggered with the Image object as its argument. The Image object is allocated on heap, and its life cycle is managed by the smart pointer. The Image object will be released when there is no smart pointer pointing to it.

Program Pattern of the Publisher side	Program Pattern of the Subscriber side
ros::NodeHandler nh;	<pre>void callback(Image::ConstPtr & img) {</pre>
<pre>ros::Publisher pub = nh.advertise();</pre>	<pre>cout << "\n Height: " << img->height;</pre>
	cout << "\n Width: " << img->width;
Image img;	// Accessing img
<pre>img.encoding = "rgb8";</pre>	}
<pre>img.height = 10;</pre>	
img.width = 10;	ros::NodeHandler nh;
img.data.resize(10 * 10 * 3);	ros::Subscriber sub =
<pre> // Assigning img.data</pre>	<pre>nh.subscribe(, callback);</pre>
<pre>pub.publish(img);</pre>	

Figure 3: A common program pattern for ROS.

3.2 Program Paradigm of FlatData

In fact, the API for ordinary RTI Connext DDS is very similar to the program pattern shown in Fig. 3. However, the FlatData API is greatly different, because messages are directly constructed as if they have been serialized. Fig. 4 shows an example of its program pattern which uses the FlatData API to create an *Image* message. Creating messages directly is not allowed in FlatData, and helper classes such as *ImageBuilder* must be used in the creating process. After creating the message, a *finish_sample* API must be called. The message can then be published with the *write* API.

The great difference in program pattern is mainly because Flat-Data uses XCDR2 as the message format for forward compatibility. The memory layout formed by the FlatData code in Fig. 4 is shown in Fig. 5. In the XCDR2 format, each field is corresponding to a memory segment. The memory segments share the same order with the message construction routine. An example is that the construction order in Fig. 4 is the same with the order of fields in the generated memory layout in Fig. 5.

Middleware '22, November 7-11, 2022, Quebec, QC, Canada

Publisher side for RTI FlatData	Subscriber side for RTI FlatData
<pre>ImageBuilder builder = rti::flat::build_data<image/>(image_writer);</pre>	<pre>const Image & sample = image_reader.take()[0].data();</pre>
<pre>builder.build_encoding().set_string("rgb8");</pre>	auto img = sample->root();
<pre>builder.add_height(10);</pre>	<pre>cout << "\n Height: " << img.height();</pre>
<pre>builder.add_width(10);</pre>	cout << "\n Width: " << img.width();
<pre>auto data_builder = builder.build_data();</pre>	
data_builder.add_n(10 * 10 * 3);	
<pre>auto data_offset = data_builder.finish();</pre>	
<pre>Image * img = builder.finish_sample();</pre>	
<pre>image_writer.write(*img);</pre>	

Figure 4: The corresponding program pattern for FlatData.

Start Addr.	Size	End Addr.	Value	Meaning	
			Start of	encoding	
0x0000	4	0x0004	0x40000002	Type and Index of encoding	
0x0004	4	0x0008	8	Length of encoding	
0x0008	8	0x0010	"rgb8"	Value and padding of encoding	
			Start of	height	
0x0010	4	0x0014	0x20000000	Type and Index of height	
0x0014	4	0x0018	10	Value of height	
			Start of	width	
0x0018	4	0x001c	0x20000001	Type and Index of width	
0x001c	4	0x0020	10	Value of width	
Start of data					
0x0020	4	0x0024	0x40000003	Type and Index of data	
0x0024	4	0x0028	300	Length of <i>data</i>	
0x0028	300	0x0154		Value of data	

Figure 5: The memory layout of an Image message generated by FlatData.

As we can see, the total size of these memory segments is not fixed, and even the offset of each field is not fixed, which is different from a C++ structure. This is why we have to use interfaces (such as *.height()*) to access fields in the Subscriber side of Fig. 4. Besides, these methods must traverse all fields until the desired field is found by its index. Overall, it is impossible to achieve transparency under such serialization format.

3.3 Program Paradigm of FlatBuffer

FlatBuffer has the same problems with FlatData. Its APIs are similar with FlatData, where *Builder* classes are needed to build a serialized message. But the serialization format for FlatBuffer is different. The program pattern for FlatBuffer is similar with that of FlatData shown in Fig. 4. The memory layout of the resulting message is shown in Fig. 6.

In FlatBuffer, the message is generated with a stack. The firstassigned field is stored at the end of the serialized message. The highlight of this serialization format is the use of tables to look up fields and offsets to reference other data.

However, this serialization format still has drawbacks in terms of transparency. As we can see from Fig. 6, although the size of the *vtable* and the offset of each field in the *vtable* are fixed, the values of fields are stored in the *root* table and can only be found indirectly from the *vtable*. Therefore, we still have to use interfaces to access fields, rather than accessing it as accessing a field in a C++ structure.

Yu-Ping Wang, Yuejiang Dong, and Gang Tan

Start Addr.	Size	End Addr.	Value	Meaning
0x0000	4	0x0004	16	Offset to root table
			Star	t of vtable
0x0004	2	0x0006	12	Size of vtable
0x0006	2	0x0008	16	Size of inline data
0x0008	2	0x000a	16	Offset of encoding from the root table
0x000a	2	0x000c	12	Offset of height from the root table
0x000c	2	0x000e	8	Offset of width from the root table
0x000e	2	0x0010	4	Offset of data from the root table
			Start	of <i>root</i> table
0x0010	4	0x0014	12	Offset to vtable (negative)
0x0014	4	0x0018	16	Offset to data
0x0018	4	0x001c	10	Value of width
0x001c	4	0x0020	10	Value of height
0x0020	4	0x0024	16	Offset to encoding
			Sta	rt of data
0x0030	4	0x0034	300	Length of data
0x0034	300	0x0160		Value of data
			Start	of encoding
0x0024	4	0x0028	4	Length of encoding
0x0028	8	0x0030	"rgb8"	Value and padding of encoding

Figure 6: The memory layout of an Image message generated by FlatBuffer.

3.4 Challenges for Transparency and Efficiency

Neither FlatData nor FlatBuffer can achieve transparency, because of their serialization format. Our key idea is to design a serialization format that is as similar as possible with the memory layout of a C++ structure, so that developers are able to access fields just like accessing fields of a C++ structure, without using any other functions or interfaces.

Besides, there is an issue on managing the life cycle of serialized messages. Take the code shown in Fig. 3 as an example. On the Publisher side, after calling the publish API, the message object can be released by the developer's code, and the serialized message is left waiting for transmission. However, in our ROS-SF framework, the message object and the serialized message are the same object. Otherwise, copying between them would bring almost the same time overhead as serialization and de-serialization, which hurts efficiency. When the message object is released by the developer's code, we have to find a way to prevent the memory of it from being freed, because it still might be used as the serialized message. On the Subscriber side, once ROS de-serializes the buffer into a message object, the buffer is not used any more and is released by ROS before triggering the callback function. However, in our ROS-SF framework, the buffer and the message object are the same object. Similarly, copying between them hurts efficiency. Thus, a proper time to release it must be found. Freeing the memory when it is still used as a buffer or a message object would lead to program crash. The official documentation of FlatData emphasizes that after calling the write API, the serialized message is owned by the writer and developers should not release it. FlatBuffer just leaves the life cycle problem to developers.

Overall, it is a challenging problem to ensure transparency and efficiency simultaneously. We design a new serialization-free framework, ROS-SF, that is more transparent than existing ones, while remaining efficient. Transparency and efficiency are ensured by designing SFM serialization format and run-time management of message life cycle.

4 APPROACH

In this section, we firstly introduce our SFM serialization format, and explain how it solves the transparency problem. Then we introduce our management of message life cycle, and explain how it solves the efficiency problem. And finally, we introduce some implementation details and further discussions.

4.1 SFM Serialization Format

For better understanding, we also use the example in Section 3. By our SFM serialization format, the resulting memory layout is shown in Fig. 7. The key to our SFM format is to recursively define the *skeleton* of a field or a message.

Start Addr.	Size	End Addr.	Value	Meaning	
0x0000	4	0x0004	8	Length of encoding	
0x0004	4	0x0008	20	Offset to the value of encoding	
0x0008	4	0x000c	10	Value of height	
0x000c	4	0x0010	10	Value of width	
0x0010	4	0x0014	300	Length of <i>data</i>	
0x0014	4	0x0018	12	Offset to the value of data	
	Start of the value of encoding				
0x0018	8	0x0020	"rgb8"	Value and padding of encoding	
	Start of the value of data				
0x0020 300 0x014c Value of data			Value of data		

Figure 7: The memory layout of an Image message under our SFM format.

All basic types that ROS supports are fixed-size types (e.g. uint32_t), except for string (i.e. *encoding* in the example) and vector (i.e. *data* in the example). The memory layout of a field with fixed-size basic types is the same as a ROS serialized message. The skeleton of such field is equivalent to its memory layout.

The memory layout of a string field starts with two 32-bit integers. The combination of these two integers is defined as the *skeleton of the string field*. The first integer is the length of the string, which indicates how many bytes the string content occupies, including the terminal zero and padding bytes. In this example, the first integer of the *encoding* field is stored at the address 0x0000. Its actual content is "rgb8", which indicates that each pixel contains 3 color channels and each channel is an 8-bit number. Therefore, its length is 8 (4 bytes for content, 1 byte for the terminal zero, and 3 bytes for padding). The second integer is the offset to the content of the string. In this example, it is stored at the address 0x0004, and its value 20 indicates that the actual content of the string is at 0x0004 + 20 = 0x0018.

It is similar for vectors. The memory layout of a vector field starts with two 32-bit integers. The combination of these two integers is defined as the *skeleton of the vector field*. The first integer is the number of elements in the vector. In this example, the first integer of the *data* field is stored at the address 0x0010. Its value 300 indicates that there are 300 bytes in the *data* field. The second integer is the offset to the elements of the vector. In this example, it is stored at the address 0x0014. Its value 12 indicates that the elements of the vector start from 0x0014 + 12 = 0x0020.

Then, we can define the *skeleton of a message* as the combination of the skeletons of all fields in the order of the message definition.

The memory layout of such a message starts with the skeleton of it. And we define the *whole* message as the memory layout that contains all data of the message. In this example, the skeleton of the message is from the address 0x0000 to the address 0x0018, and the whole message is from the address 0x0000 to the address 0x014c.

Besides, our SFM format supports nested messages. When a field of a message A is another message B, the skeleton of this field is the skeleton of the message B. When elements of a vector are message B, the skeleton of each element is the skeleton of the message B. The offset in a vector (the second integer in the skeleton of the vector) points to a memory segment which contains the skeletons of all elements in the ascending order of index.

Thus, our SFM format has the following features.

- The size of the skeleton of a message is fixed. Because the only variable-length types that ROS supports are string and vector, and both their skeletons occupy a fixed-length memory of 8 bytes. Therefore, the size of a message composed of basic types is fixed. When a field of a message *A* is a message *B*, only the skeleton of *B* is contained in the skeleton of *A*. Recursively, the size of the skeleton of a nested message is fixed.
- The offset of a field in the skeleton of a message is fixed. Because the order of fields is fixed, the size of the skeleton of each field is fixed.
- The size of each vector element is fixed. Even if vector element is a message, only the skeleton of the message is contained in the vector, whose size is fixed. Since the elements of a vector are continuously stored in the memory region pointed to by the vector offset, they can be accessed as elements of a C++ array.

The above features show that the skeleton of a message can be expressed as a C++ structure, which is defined as the SFM message class. Specifically, the skeleton of a string is expressed as *sfm::string*, and the skeleton of a vector is expressed as *sfm::vector*. In our design, we keep the interfaces of sfm::string and sfm::vector the same with those of std::string and std::vector, respectively, eliminating the code rewriting cost for developers to use our format.

4.2 Management of Message Life Cycle

The life cycle of a serialization-free message is more complex than ordinary message objects. A message manager class is designed (namely *sfm::mm*), and a global message manager object (namely *sfm::gmm*) is defined to manage the life cycle of serialization-free messages. Each serialization-free message in our ROS-SF framework has three states, namely *Allocated*, *Published*, and *Destructed*.

On the *Publisher* side, the life cycle of an *Image* message is shown in Fig. 8. When a message is defined, a memory segment should be allocated for it. Our ROS-SF framework assumes that at run time, all serialization-free messages are allocated in the heap. This assumption is ensured by our compile-time checker and converter, which will be introduced in Section 4.3.2. When a message is initialized, a memory segment is allocated in the heap, whose size is large enough for the largest message of this message type. This size is defined by developers in the IDL. This is also the solution used by FlatData and FlatBuffer to avoid memory reallocation. But they implement this initial memory allocation by creating a *Builder* object, which is not transparent. In our ROS-SF framework, the initial memory allocation is implemented by overloading the global *new* operator and explicitly specializing the *std::make_shared* template function. The allocated memory segment is then registered into the message manager, and the message enters the *Allocated* state.



Figure 8: Message state changes at the *Publisher* side. A record in the global memory manager holds a smart pointer to the message memory. When the message is published, a copy of the smart pointer is provided to the ROS transmission queue. The message memory is freed only when the reference count becomes zero.

Based on our SFM format, the size of the whole message could increase during its life cycle. This only occurs when extra memory is required for vector elements or string contents. At this time, we only know the start address of the field that requests memory. We need to find the current end of the whole message and expand the whole message. When a message is initially allocated in the heap, the message manager is informed. The start address of the message and its initial size (i.e. the size of the skeleton of the message) are recorded within the message manager. Whenever a field requests for extra memory, the message manager is informed to find the corresponding record of the message based on the address of the requesting field. The start address and the size in the record are used to find the current end of the whole message. The record is then updated to the increased size.

When a message at the *Allocated* state is published, it enters the *Published* state. A message at the *Published* state acts as two roles, i.e. as a message object, and as a serialized buffer waiting for transmission. In ROS, the message object and the serialized buffer are independent of each other. The message object is managed by the developer's code. The serialized buffer is allocated in the heap and managed with a smart pointer of type *std::shared_array*. In the following explanation, this smart pointer is referred to as the *buffer pointer*. In our ROS-SF framework, since the message object and the serialized buffer occupy the same memory, the message object cannot be freed when the developer's code releases it, because the serialized buffer is still in use and waiting for transmission. Only when the developer's code has released the message object and the buffer has been transmitted, can the underlying message memory be freed. To solve the releasing issue, our solution is to construct a buffer pointer in the message manager, when the message is initially allocated. When the *publish* interface is called, a copy of the buffer pointer is provided to ROS. The serialized buffer is then waiting for transmission and the reference count of it is increased. When the message object is released by the developer's code, the message manager is informed. This is achieved by overloading the global delete operator. The message manager releases the record of the message along with its buffer pointer, and thus the reference count of the serialized buffer is decreased. If ROS has not finished transferring the serialized buffer, ROS will still hold a copy of the buffer pointer, and the reference count of the serialized buffer will not be zero. Only when the reference count becomes zero, will the message memory be actually freed, and the message will enter the Destructed state. If a message is released by the developer's code before published, the reference count instantly becomes zero, and the message memory is freed.



Figure 9: Message state changes at the Subscriber side.

On the Subscriber side, the life cycle is shown in Fig. 9. When a buffer is received, ROS also manages it with a smart pointer of type std::shared_array. Before ROS triggers the callback function, the de-serialization routine is called. This de-serialization routine is generated by ROS. In this routine, a message object is generated from the received buffer, and managed with a smart pointer of type shared_ptr. In the following explanation, it is referred to as the object pointer. The object pointer is passed as the argument to the callback function. In our ROS-SF framework, a dummy de-serialization routine is generated, where the message manager is informed to obtain the ownership of the buffer pointer. It is also called before the callback function is triggered. Then, the message directly enters the Published state. At this time, there are two smart pointers. One of them is the buffer pointer that manages references to the message memory, which is stored in a record in the global message manager. The other is the object pointer that manages references to the message object, which is provided to the callback function. After returning from the callback function, the object pointer is released. But the message object can still exist, because inside the callback function, the developer's code can add references of the message object by creating copies of the object pointer. When the reference count of the message object becomes zero, the message manager is informed, because of the overloaded global delete operator. The message manager then releases the record, thus the reference count to

the message memory becomes zero, and finally the message memory is freed.

Overall, the management of message life cycle is achieved by designing a message manager and overloading message-associated code. There is no need to modify the developer's code, thus the code-level transparency is maintained. Furthermore, messages are never copied or improperly freed, thus the whole-process efficiency is guaranteed.

4.3 ROS-SF Framework

In summary, our ROS-SF framework consists of three modules, namely SFM Generator, ROS-SF Converter, and ROS-SF Library. The routine of applying our ROS-SF framework is shown in Fig. 10b, which is straightforward compared with the normal compiling routine of ROS (shown in Fig. 10a).

- The SFM Generator generates header files which define message classes that follow our SFM format based on ROS's existing IDL.
- The ROS-SF Converter checks the source code, and automatically modifies it if necessary.
- The ROS-SF Library provides the implementation of auxiliary classes, including sfm::vector, sfm::string, and sfm::mm. It is linked into the final executable.



Figure 10: (a) Normal compiling routine of ROS. (b) Compiling routine of our ROS-SF framework.

4.3.1 SFM Generator. The SFM Generator is implemented based on the ROS message generator *genmsg*. Most of the message class generation routine is the same, except for the following features.

- Overloaded global *new* and *delete* operator. They are used to help the management of the message life cycle.
- Copy constructor and an overloaded = operator. Since the message class expresses the skeleton of messages, the default copy constructor and = operator can only copy the skeleton of a message. They are generated to find the current size of the whole message from the message manager and copy the message.
- Overloaded ROS serialization routine. It is generated to avoid normal serialization, and add a reference to the buffer array.
- Overloaded ROS de-serialization routine. It is generated to avoid normal de-serialization, and inform the message manager to add a reference to the buffer array.

4.3.2 ROS-SF Converter. As we have stated in Section 4.2, serialization-free messages must be allocated in the heap. This is not always true, even for the program pattern shown in Fig. 3. Our ROS-SF Converter is designed to convert a program so that all serialization-free messages are allocated on heap in the resulting program.

Source code level analysis would misidentify classes with the same name as our message class. Therefore, we implement the ROS-SF Converter based on LLVM [18, 19]. At the LLVM IR (Intermediate Representation) level, all C++ features that could affect the class name (e.g. namespace, macro, typedef, template, etc) are all translated into their demangled identity name. And we can find the location within the source code from the debugging information associated with IR instructions. As a result, ROS-SF Converter can automatically find messages that are defined as local variables, and modify the code to perform heap allocation instead. An example is shown in Fig. 11. The original local variable is changed into a reference to the message allocated in the heap. There is no need to change the following instructions, because the C++ grammar for the variable and the reference are the same. Besides, when the life cycle of the local variable ends in the original code, the life cycle of the smart pointer also ends in the modified code. Therefore, their semantics are consistent. Note that the global new operator is overloaded by the header file generated from the SFM Generator, so that "new Image" would allocate enough memory for the message.

Before Modification	After Modification
	<pre>std::shared_ptr<image/>ptmp_img(new Image);</pre>
Image img;	<pre>Image & img = *ptmp_img;</pre>
<pre>img.encoding = "8UC3";</pre>	<pre>img.encoding = "8UC3";</pre>
img.height = 10;	img.height = 10;
img.width = 10;	<pre>img.width = 10;</pre>
img.data.resize(10 * 10 * 3);	img.data.resize(10 * 10 * 3);
<pre> // Assigning img.data</pre>	<pre> // Assigning img.data</pre>
<pre>pub.publish(img);</pre>	<pre>pub.publish(img);</pre>

Figure 11: The code before modification is the same as Fig. 3. The ROS-SF Converter automatically finds the message defined as a local variable and modifies it to perform heap allocation.

4.3.3 *ROS-SF Library.* In the ROS-SF Library, we implement three main auxiliary classes, namely sfm::string, sfm::vector, and sfm::mm.

For the sfm::string class, it contains the length of the string and an offset whose initial values are both 0. When it is assigned for the first time, the whole message is expanded with the help of the message manager. When it is assigned again, the memory that stores the string content needs to be reallocated. This could be done by further expanding the whole message, but this solution would waste memory and possibly break the memory limit. Our solution is that when the length of the reassigned string is not 0, an alert is raised to developers. We assume that there is no reassignment to strings, because a reassignment also hurts the performance of the program. This assumption is referred to as the *One-Shot String Assignment Assumption*; we will discuss our applicability study of checking this assumption in Section 5.4. For better transparency, we also implement other interfaces to keep consistency with std::string, such as *constructors, operator* =, *length(), c_str(), operator []*, etc. Besides, type conversions to *char* * and *std::string* are also implemented.

For the sfm::vector template, it contains the size of the vector and an offset whose initial values are both 0. When it is resized for the first time, the whole message is expanded with the help of the message manager. When it is resized again, the situation is similar to string reassignments. Our solution is that when the size of the resized vector is not 0, an alert is raised to developers. This assumption is referred to as the *One-Shot Vector Resizing Assumption*, and we will discuss our applicability study of checking this assumption in Section 5.4. For better transparency, we also implement other interfaces to keep consistency with *std::vector*, such as *constructors, operator* [], *begin*(), *end*(), and etc. We also implement the corresponding iterator template.

Besides string assignment and vector resizing, there are other modifier interfaces that may trigger memory reallocation, such as *push_back()* and *pop_back()*. Our solution is to advise developers not using them, and developers will get compilation errors if they are called. We assume that none of these modifier interfaces are called. This assumption is referred to as the *No Modifier Assumption*, and we will discuss our applicability study of checking this assumption in Section 5.4.

These three assumptions aforementioned are also implicitly assumed in FlatData and FlatBuffer. There are no modifier interfaces in their APIs, and they also raise alerts when a string in a message is reassigned or a vector is resized.

The sfm::mm class is implemented to manage the life cycle of messages. Looking for the record of a message with its start address can be easily implemented by maintaining a std::map. But when expanding the whole message, it is needed to find the record of a message with an address in the middle of the message. Currently, we implement it as a binary search from a std::vector of ordered records. It could be further optimized, but according to our evaluation result in Section 5, it appears to be efficient enough. Note that we cannot implement the message manager as an STL compatible allocator. When allocating memory, a normal allocator only accepts the desired size. But we need to know "who" is getting more memory, so that the message manager can find the end of the message and expand it.

4.4 Discussion

4.4.1 Endianness. When communicating among machines with different architectures, the endianness issue has to be considered. In our ROS-SF framework, the endianness of a serialization-free message is the same as the publisher side. Therefore, it is up to the subscriber side to decide whether the endianness of the serialized message needs to be converted. This is a common issue that all serialization-free frameworks are facing. If the endianness conversion is inevitable, it could be time-consuming, and could even counteract the efficiency brought by serialization-free frameworks.

4.4.2 Other Data Structures. Some serialization formats support advanced features and/or data structures. Currently, our SFM format does not support them because they are not supported by ROS. But we would like to discuss two examples for potential extensions.

In ProtoBuf, each field can be labelled as *optional*. If a message is constructed without assigning this field, the message will not contain it. For our SFM format, an optional field with fixed-size basic types

could be assigned as a user-defined default value; an optional field with other types could be treated as a vector with its upper bound set as 1.

ProtoBuf also supports the type of "*map*", which represents a keyvalue map. The in-memory representation of a map field is complex. Our SFM format can treat it as a vector of key-value pairs, which is also the solution used by ROS.

5 EVALUATION

In this section, we firstly conduct two simple experiments to show the performance improvement by our ROS-SF framework. Then, we conduct an application case study, to show that our ROS-SF framework can transparently handle practical applications. We further show and discuss some failure cases.

5.1 Intra-Machine Performance

The first group of experiments is performed on one machine to show the performance improvement to intra-machine transmission. In this experiment, as shown in Fig. 12, one ROS node (named *pub*) acts as a publisher and the other (named *sub*) acts as a subscriber. A topic with the type of *sensor_msgs::Image* connects these two nodes. The code of these two nodes are similar to those in Fig. 3. At the *pub* node, an *Image* message is created, and the creation time is stored into the message. Then, after the content of the message is properly set, it is published. At the *sub* node, whenever a new message arrives, a callback function is triggered, and the time difference between current time and the time stored in the message is recorded. The above routine is repeated 2,000 times at a frequency of 10 Hz.





For the original ROS, the recorded time difference includes the time of message construction, serialization, loopback transmission via a TCP/IP socket, and de-serialization. For our ROS-SF, the same code is used, but the recorded time difference includes only the time of message construction and transmission, because serialization and de-serialization are eliminated. Since the serialization and deserialization time depends on the size of the message, we repeat our experiment for three different image sizes: $\sim 200 \text{KB} (256 \times 256 \times 24 \text{bits}), \sim 1 \text{MB} (800 \times 600 \times 24 \text{bits}), \text{ and } \sim 6 \text{MB} (1920 \times 1080 \times 24 \text{bits}).$ The results are organized into Fig. 13.

In Fig. 13, the boxes represent the average latency in milliseconds, and the black lines represent the standard deviation. As we can see, our ROS-SF framework can greatly reduce the transmission latency. When the image size increases, the serialization and de-serialization time increase, and thus the performance improvement by our ROS-SF framework also increases. Especially, when the image size is 6MB, our ROS-SF framework can reduce the average transmission latency by about 76.3%.

We also run the same test with 6MB image size using different middleware, and the results are organized into Fig. 14. The differences between serialization-free framework and its corresponding



Figure 13: Comparison of intra-machine transmission latency with different message size.

serialization framework indicate the time reduced by eliminating serialization. The difference between FlatBuf and ProtoBuf is the smallest among the three groups, indicating the serialization routine of ProtoBuf is well optimized. The transmission latency of RTI-FlatData is the smallest, indicating the transmission routine of RTI / RTI-FlatData is well optimized. Using ROS-SF, the transmission latency could be reduced to the same scale as FlatData and FlatBuffer, without introducing any code rewriting burden for developers, which is inevitable when using FlatData or FlatBuf.



Figure 14: Comparison of intra-machine transmission latency with different middleware.

5.2 Inter-Machine Performance

The second group of experiments is performed on two machines connected with Intel82599 10 Gigabit Ethernet controller, to show the performance improvement to inter-machine transmission. We cannot simply repeat the intra-machine performance experiment by running the *pub* and the *sub* node on different machines, because synchronizing time between two machines is very difficult, especially at the millisecond level. Therefore, we employ the commonly used ping-pong methodology. In this experiment, as shown in Fig. 15, there are 3 nodes and 2 topics. The first node (*pub*) publishes an *Image* message via the first topic. The second node (*trans*) subscribes

to the first topic. Once the second node receives a message, it creates another *Image* message, whose timestamp is set to be the same as the received message, and the message is published via the second topic. The third node (*sub*) subscribes to the second topic and records the time difference between current time and the time stored in the message. The *pub* and the *sub* node run on machine A, and the *trans* node runs on machine B. Both the final received time and the time stored in the message are the time at machine A, so it is reasonable to subtract them directly. The above routine is repeated 2000 times at a frequency of 10 Hz.



Figure 15: The relationship of the nodes and topics in our intermachine performance test.

For the original ROS, the recorded time difference includes the time of two message constructions, two serializations, two loopback transmissions via TCP/IP sockets, and two de-serializations. For our ROS-SF, the recorded time difference includes only two message constructions and two transmissions. We also repeat our experiment for 3 different image sizes. The results are organized into Fig. 16.



Figure 16: Comparison of inter-machine transmission latency with different message size.

In Fig. 16, the boxes represent the average ping-pong latency in milliseconds, and the black lines represent the standard deviation. Approximately, we can divide the ping-pong latency by 2 to obtain the one-way transmission latency. As we can see, our ROS-SF framework can greatly reduce the transmission latency. When the image size increases, the serialization and de-serialization time increases, and thus the performance improvement by our ROS-SF framework also increases. Especially, when the image size is 6MB, our ROS-SF framework can reduce the average transmission latency by about 69.9%.

Middleware '22, November 7-11, 2022, Quebec, QC, Canada

Yu-Ping Wang, Yuejiang Dong, and Gang Tan

5.3 Application Case Study

To show that our ROS-SF framework can transparently handle practical application cases, we employ ORB-SLAM [21] to conduct an application case study.

SLAM (Simultaneous Localization and Mapping) is one of the most important components for robotic systems [28]. The main purpose of SLAM algorithms is to "identify" where the robot is, based on input sensor data and a known map. In addition, when the robot moves to an area outside the known map, SLAM algorithms can simultaneously expand the map. If input sensors contain one or more cameras, such an algorithm is called a visual SLAM algorithm. ORB-SLAM is one of the state-of-the-art visual SLAM algorithms.

As we can see from its purpose, ORB-SLAM requires a sequence of images as its input, and generates the location and orientation of the camera. In its open-sourced version for ROS, these are realized by subscribing a ROS topic with the type of *sensor_msgs::Image*, and publishing a ROS topic with the type of *geometry_msgs::PoseStamped*. ORB-SLAM also generates several other outputs. A point cloud, which contains the corresponding 3D points of the feature points on the 2D input image, is generated for further processing by other software modules, such as the obstacle avoidance module. This is also implemented by publishing a ROS topic with the type of *sensor_msgs::PointCloud2*. An image, which combines the input image and the feature points, is generated for debugging purpose. The input images, output point clouds, and debugging images are usually large in size, therefore suitable for applying our ROS-SF framework.

Overall, Fig. 17 shows the simplified relationship of the nodes and topics in our application case study. We employ a ROS node called *pub_tum* to provide input images to ORB-SLAM. These images are from the TUM RGBD dataset [7], which is one of the most commonly used datasets to evaluate SLAM algorithms. To evaluate the overall latency, we design 3 ROS nodes that subscribe to the output poses, the point clouds, and the debug images, respectively. These 3 ROS nodes record the time differences from the time when the input image is created to the time when the output messages are received.



Figure 17: The relationship of the nodes and topics in our application case study.

All of the 5 ROS nodes can be supported by our ROS-SF framework without any manual modification on the source code. The latency results are shown in Fig. 18. These latency results include the time of input image construction, input image transmission, calculation of the ORB-SLAM algorithm, the output message construction, and output messages transmission. Among them, the calculation time of the ORB-SLAM algorithm is about 30-40 ms which is the major part of all latencies, and the cause of the overlap of confidence intervals. Under such harsh condition, our ROS-SF framework can still reduce the overall latency by about 5%.



Figure 18: Comparison of the overall latency of ORB-SLAM using ROS and ROS-SF.

5.4 Applicability Study

As we have stated, our ROS-SF framework can transparently improve the message-passing performance under 3 assumptions. When these assumptions are violated, our ROS-SF framework would fail, either by compilation error (the No Modifier Assumption) or runtime prompt (the One-Shot String Assignment and the One-Shot Vector Resizing Assumptions). To show the applicability of our ROS-SF framework, we conduct a study on the software modules maintained by the official ROS team (125 packages, 486 source files). We manually check the usage of some message classes, and check whether they satisfy our assumptions. The results are organized in Table 1. The "Total" column shows the number of files that use the corresponding message class; the "Applicable" column shows the number of files that satisfy all our assumptions; the "String Reassignment" column shows the number of files that violate our Assumption 1; the "Vector Multi-Resize" column shows the number of files that violate our Assumption 2; the "Other Methods" column shows the number of files that violate our Assumption 3.

As shown in Table 1, most of the application scenarios of the Image message class satisfy our assumptions. We would also like to discuss some of the failure cases.

The first failure case (shown in Fig. 19) comes from a software module in which an image is transformed with specific affinetransformation parameters and the result image is published to other software modules. The transformation is implemented with a commonly used image-processing library named OpenCV [22]. After the transformation, line 218 states that it will convert the OpenCV image (*out_image*) into a ROS *Image* message (*out_img*) with supplementary information of *msg->header* and *msg->encoding*. Inside this conversion routine, an *Image* message is constructed and all of its fields are filled. Our ROS-SF framework can handle this kind of conversion well. However, line 219 modifies the *header.frame_id* field of the output *Image* message. The *header.frame_id* field indicates the coordinate system used by this message, which is important in

Message Class	Total	Applicable	String Reassignment	Vector Multi-Resize	Other Methods
sensor_msgs::Image	49	40	8	6	0
sensor_msgs::CompressedImage	7	2	5	5	0
sensor_msgs::PointCloud	14	0	13	12	2
sensor_msgs::PointCloud2	15	1	7	7	8
sensor_msgs::LaserScan	18	5	13	12	1

Table 1: The results of applicability study.

robotic systems because there could be dozens of coordinate systems involved with a robot, such as the world coordinates, the motor base coordinates, the camera coordinates, etc. This modification violates our One-Shot String Assignment Assumption. In this case, our ROS-SF framework would prompt the developer to avoid this violation. The modification is simple: prepare a temporary *header* before line 218, do the assignment of line 219 as the first assignment of line 218. This failure case represents a major kind of our failure cases. We can rewrite the code to a more efficient version which satisfies our assumption (shown in the lower part of Fig. 19).

https:// image_ro	<pre>https://github.com/ros-perception/image_pipeline/blob/noetic/ image_rotate/src/nodelet/image_rotate_nodelet.cpp</pre>				
line 218	<pre>8 sensor_msgs::Image::Ptr out_img = cv_bridge::CvImage(msg->header, msg->encoding, out_image).toImageMsg();</pre>				
line 219	<pre>9 out_img->header.frame_id = transform.child_frame_id;</pre>				
line 220	<pre>img_pubpublish(out_img);</pre>				
Rewritten					
	<pre>Header header_tmp = {msg->header.seq, msg->header.stamp,</pre>				
	<pre>transform.child_frame_id};</pre>				
	<pre>sensor_msgs::Image::Ptr out_img = cv_bridge::CvImage(header_tmp, msg->encoding, out_image).toImageMsg();</pre>				
	<pre>img_pubpublish(out_img);</pre>				

Figure 19: Failure case that violates our One-Shot String Assignment Assumption. A string field is assigned twice.

The second failure case (shown in Fig. 20) comes from a software module in which input data is packed into specific message classes. In this failure case, two images with the OpenCV format (left_rect and right_rect in line 83) are converted into a DisparityImage message (disparity in line 85). Part of this disparity image message is an Image message. Line 104 creates a reference (dimage) of the Image message within the DisparityImage message. And line 109 resizes the data field of *dimage*. If the data field of the input argument disparity has been resized before calling this method, this resizing operation may violate our One-Shot Vector Resizing Assumption. In this case, our ROS-SF framework would also prompt the developer to avoid this violation. In fact, the disparity argument of the process-Disparity method, serves as an output reference. We can confirm that, in this software module, all callers of the processDisparity method pass an argument without a resized data field. However, this source file is also compiled into a separate library, which will be used by other software modules. Therefore, we cannot confirm that other software modules also follow the assumption. This failure case also represents a major kind of our failure cases. When a constructed message is passed as an output argument, we cannot confirm that all

callers follow our assumption, especially if the code is compiled to a separate library. For the sake of rigor, we count them all as failure cases.

https:// stereo_i	<pre>https://github.com/ros-perception/image_pipeline/blob/noetic/ stereo_image_proc/src/libstereo_image_proc/processor.cpp</pre>			
line 83	<pre>void StereoProcessor::processDisparity(const cv::Mat& left_rect, const cv::Mat& right_rect,</pre>			
line 84	<pre>const image_geometry::StereoCameraModel& model,</pre>			
line 85	<pre>stereo_msgs::DisparityImage& disparity) const</pre>			
line 104	<pre>sensor_msgs::Image& dimage = disparity.image;</pre>			
line 109	<pre>dimage.data.resize(dimage.step * dimage.height);</pre>			

Figure 20: Failure case that violates our One-Shot Vector Resizing Assumption. A vector field is possibly resized before.

The third failure case (shown in Figure 21) comes from the same software module with the second failure case. It packs input data into a PointCloud message. In this failure case, the "push_back" method is used in line 164, which violates our No Modifier Assumption. In this case, our ROS-SF framework would generate a compilation error, since our auxiliary class sfm::vector has not implemented the *push_back* method. The main reason of using the *push_back* method is that not all points in the "dense_points_" will be added to the message because of line 158. Point clouds usually contain thousands of points in practice, and the push_back method would trigger reallocation many times, especially when the "points" field is initially resized to 0 at line 147. Even if we replaced line 147 with code that reserves enough memory for the "points" field, it would lead to a waste of memory space; in the extreme case, if none of the points passes the check at line 158, the actual usage of memory would be 0. This failure case also represents another major kind of our failure cases. Rewriting it to satisfy our No Modifier Assumption is possible (shown in the lower part of Figure 21. In the rewritten code, we first count the number of the valid points, resize the "points" field of the message to a proper size, and then "push" them into the message. We believe that the rewritten code is more efficient even for the original ROS, because less reallocation would be triggered.

6 CONCLUSION

In this paper, we present ROS-SF, a middleware which can transparently improve the message-passing performance of ROS. It is based on a new serialization format, namely SFM. The main design principle of SFM is to make the memory layout of serializationfree messages keep the same as regular messages. Based on our

Middleware '22, November 7-11, 2022, Quebec, QC, Canada

https://github.com/ros-perception/image_pipeline/blob/noetic/					
lie 147 pointe pointe posize(0).					
1100 147	points.points.resize(0);				
line 156	<pre>for (int32_t u = 0; u < dense_pointsrows; ++u) {</pre>				
line 157	<pre>for (int32_t v = 0; v < dense_pointscols; ++v) {</pre>				
line 158	<pre>if (isValidPoint(dense_points_(u,v))) {</pre>				
line 164	<pre>points.push_back(pt);</pre>				
Rewritter					
	<pre>int cnt = 0, total_valid = 0;</pre>				
	<pre>for (int32_t u = 0; u < dense_pointsrows; ++u)</pre>				
	<pre>for (int32_t v = 0; v < dense_pointscols; ++v)</pre>				
	<pre>if (isValidPoint(dense_points_(u,v)))</pre>				
	<pre>total_valid++;</pre>				
	<pre>points.points.resize(total_valid);</pre>				
	<pre>for (int32_t u = 0; u < dense_pointsrows; ++u) {</pre>				
	<pre>for (int32_t v = 0; v < dense_pointscols; ++v) {</pre>				
	<pre>if (isValidPoint(dense_points_(u,v))) {</pre>				
	<pre>points.points[cnt++] = pt;</pre>				

Figure 21: Failure case that violates our No Modifier Assumption. Other modifier method (*push_back*) is used.

SFM format, our ROS-SF framework can transparently eliminate serialization and de-serialization under the ROS APIs.

Evaluation results show that our ROS-SF framework can improve the message-passing performance of ROS by up to 76.3%, and can be transparently applied to large-scale robotic projects such as ORB-SLAM. Applicability study results show that our ROS-SF framework can be easily applied to many existing ROS packages. Even in the failure cases, our ROS-SF framework can provide prompts, which can help developers to modify their code to satisfy our assumptions.

ACKNOWLEDGEMENTS

The authors deeply thank for the reviewers' comments which help to improve the quality of the paper. This work was supported by the National Natural Science Foundation of China under Grant No. 61872210.

REFERENCES

- Oren Ben-Kiki, Clark Evans, and Ingy döt Net. 2021. YAML Ain't Markup Language Version 1.2. (October 2021). Retrieved May 9, 2022 from https: //yaml.org/spec/1.2/spec.html
- [2] Tim Bray. 2017. The JavaScript Object Notation (JSON) Data Interchange Format. *RFC* 8259 (2017), 1–16. https://doi.org/10.17487/RFC8259
- [3] Tim Bray, Jean Paoli, and C. M. Sperberg-McQueen. 1997. Extensible Markup Language (XML). World Wide Web J. 2, 4 (1997), 27–66. http://www.w3.org/ TR/WD-xml-970807
- [4] Alex Campos. 2019. FlatData API. (March 2019). Retrieved May 9, 2022 from http://community.rti.com/examples/flatdata-api
- [5] Peter Druschel and Larry L. Peterson. 1993. Fbufs: A High-Bandwidth Cross-Domain Transfer Facility. In Proceedings of the Fourteenth ACM Symposium on Operating System Principles, SOSP 1993, The Grove Park Inn and Country Club, Asheville, North Carolina, USA, December 5-8, 1993. 189–202. https: //doi.org/10.1145/168619.168634
- [6] Mike Eisler. 2006. XDR: External Data Representation Standard. RFC 4506 (2006), 1–27. https://doi.org/10.17487/RFC4506
- [7] Jakob Engel, Vladyslav C. Usenko, and Daniel Cremers. 2016. A Photometrically Calibrated Benchmark For Monocular Visual Odometry. *CoRR* abs/1607.02555 (2016). http://arxiv.org/abs/1607.02555
- [8] Sadayuki Furuhashi. 2021. It's like JSON. but fast and small. (December 2021). Retrieved May 9, 2022 from https://msgpack.org/
- [9] Google. 2020. FlatBuffers. (January 2020). Retrieved May 9, 2022 from https://google.github.io/flatbuffers/
- [10] Google. 2022. Protocol Buffers. (May 2022). Retrieved May 9, 2022 from https://developers.google.com/protocol-buffers/

- [11] W. Gropp, E. Lusk, A. Skjellum, and R. Thakur. 1999. Using MPI: Portable Parallel Programming with the Message-passing Interface. MIT Press. https://books.google.ae/books?id=DFT1ngEACAAJ
- [12] Izzat El Hajj, Thomas B. Jablin, Dejan S. Milojicic, and Wen-Mei W. Hwu. 2017. SAVI objects: sharing and virtuality incorporated. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 45:1–45:24. https://doi.org/10.1145/3133869
- [13] Maurice Herlihy and Barbara Liskov. 1982. A Value Transmission Method for Abstract Data Types. ACM Trans. Program. Lang. Syst. 4, 4 (1982), 527–551. https://doi.org/10.1145/69622.357182
- [14] Pieter Hintjens. 2013. ZeroMQ: Messaging for Many Applications. O'Reilly Media.
- [15] Albert S. Huang, Edwin Olson, and David C. Moore. 2010. LCM: Lightweight Communications and Marshalling. In 2010 IEEE/RSJ International Conference on Intelligent Robots and Systems, October 18-22, 2010, Taipei, Taiwan. IEEE, 4057–4062. https://doi.org/10.1109/IROS.2010.5649358
- [16] Costin Iordache, Stephen M. Fendyke, Mike J. Jones, and Robert A. Buckley. 2021. Smart Pointers and Shared Memory Synchronisation for Efficient Inter-process Communication in ROS on an Autonomous Vehicle. *CoRR* abs/2108.07085 (2021). https://arxiv.org/abs/2108.07085
- [17] Christian Kurmann and Thomas Stricker. 2003. Zero-Copy for CORBA Efficient Communication for Distributed Object Middleware. In 12th International Symposium on High-Performance Distributed Computing (HPDC-12 2003), 22-24 June 2003, Seattle, WA, USA. 4–13. https://doi.org/10.1109/HPDC.2003.1210011
- [18] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In 2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2004), 20-24 March 2004, San Jose, CA, USA. 75–88. https://doi.org/10.1109/CGO.2004.1281665
- [19] LLVM. 2022. Clang: A C Language Family Frontend for LLVM. (March 2022). Retrieved May 9, 2022 from http://clang.llvm.org/
- [20] Sun Microsystems. 1987. XDR: External Data Representation standard. RFC 1014 (1987), 1–20. https://doi.org/10.17487/RFC1014
- [21] Raul Mur-Artal and Juan D. Tardós. 2017. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Trans. Robotics* 33, 5 (2017), 1255–1262. https://doi.org/10.1109/TRO.2017.2705103
- [22] OpenCV team. 2022. OpenCV Official Site. (May 2022). Retrieved May 9, 2022 from http://opencv.org/
- [23] Oracle. 2010. Java Object Serialization Specification. (January 2010). Retrieved May 9, 2022 from https://docs.oracle.com/javase/7/docs/platform/serialization/ spec/serialTOC.html
- [24] Python Software Foundation. 2022. pickle Python object serialization. (April 2022). Retrieved May 9, 2022 from https://docs.python.org/3/library/pickle.html
- [25] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. 2009. ROS: an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*.
- [26] Aniruddh Rao, Visali Mushunuri, Samar Shailendra, Bighnaraj Panigrahi, and Anantha Simha. 2017. Reliable robotic communication using multi-path TCP. In 9th International Conference on Communication Systems and Networks, COM-SNETS 2017, Bengaluru, India, January 4-8, 2017. IEEE, 429–430. https: //doi.org/10.1109/COMSNETS.2017.7945427
- [27] Real-Time Innovations. 2007. DDS: An Open Standard for Real-Time Applications. (January 2007). Retrieved May 9, 2022 from http://www.rti.com/products/ddsstandard
- [28] Muhamad Risqi U. Saputra, Andrew Markham, and Niki Trigoni. 2018. Visual SLAM and Structure from Motion in Dynamic Environments: A Survey. ACM Comput. Surv. 51, 2 (2018), 37:1–37:36. https://doi.org/10.1145/3177853
- [29] Raj Srinivasan. 1995. XDR: External Data Representation Standard. RFC 1832 (1995), 1–24. https://doi.org/10.17487/RFC1832
- [30] Konstantin Taranov, Rodrigo Bruno, Gustavo Alonso, and Torsten Hoefler. 2021. Naos: Serialization-free RDMA networking in Java. In 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 1–14. https://www.usenix.org/ conference/atc21/presentation/taranov
- [31] The Object Management Group. 2015. Data Distribution Service. (March 2015). Retrieved May 9, 2022 from http://www.omg.org/spec/DDS/1.4/
- [32] The Object Management Group. 2020. Extensible and Dynamic Topic Types for DDS. (February 2020). Retrieved May 9, 2022 from https://www.omg.org/spec/ DDS-XTypes/
- [33] The Object Management Group. 2021. Common Object Request Broker Architecture, 3.4 edition. (February, 2021). Retrieved May 9, 2022 from http://www.omg.org/spec/CORBA/
- [34] Yu-Ping Wang, Xu-Qiang Hu, Zi-Xin Zou, Wende Tan, and Gang Tan. 2019. IVT: an efficient method for sharing subtype polymorphic objects. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 130:1–130:22. https://doi.org/10.1145/3360556