Evolving Messages Over Time - REP-2011

William Woodall <william@openrobotics.org>
Brandon Ong <brandon@openrobotics.org>
You Liang Tan <tan_you_liang@hotmail.com>
Kyle Marcey <kyle.marcey@apex.ai>

October 2022
Overview

- Conceptual Overview of REP-2011
- New Features Needed in ROS 2
- Dive into Run-Time Interface Reflection
- Future Work and Known Issues
Motivation for REP-2011

Reasons for this REP:

● It is natural for types to evolve over time
  ○ In your projects
  ○ And in ROS 2 itself
● We need tools to detect when this happens
● We need tools to help transition between versions
The Problem

In ROS 2 today:

- If you try to change a type unevenly across your system:
  - Some middlewares may allow communication, depending on the change
  - Warnings/errors when types are incompatible vary
  - Limited features exposed in ROS 2 to help you evolve types in a backward/forward way
The Problem - Example

What if we want to change this?

```
# Temperature.msg
uint64 timestamp
int32 temperature
```
The Problem - Example

What if we want to change this?

Add a new, but redundant, optional field?

Just change the type directly?

Note: you can’t actually do this right now.
The Problem - Example

What if we want to change this?

You can do either, if the middleware supports it.

We’ll focus on this option, for the REP.

Note: you can’t actually do this right now.
The Proposed Solution

- REP-2011:
  - [https://github.com/ros-infrastructure/rep/pull/358](https://github.com/ros-infrastructure/rep/pull/358)
- REP-2011 aims to help users:
  - Know when messages have changed
  - Convert between versions on demand
  - Write code to convert between versions
- It will do so by depending on the ability to:
  - Interact with types using only their description
- This REP does not try to:
  - Expose “advanced” serialization features like optional fields, extensible types, or inheritance
  - Prevent these “advanced” features from working
What will this look like in practice?

% ros2 topic echo /scan sensor_msgs/msg/LaserScan

[WARN] [1666081526.522630000] [ros2_bag]: Publisher 'gid...' on topic '/scan' is using a version of 'sensor_msgs/msg/LaserScan' ('abc123') that does not match the version used locally ('def456').

% ros2 interface transfer_functions info sensor_msgs/msg/LaserScan abc123 def456
Conversion available with transfer functions:
- [abc123 -> cba321]:
  - pkg: sensor_msgs_migration
  - description: new field added to describe ...
- [cba321 -> def456]:
  - pkg: sensor_msgs_migration
  - description: changed the type of field ...

Subject to change
What will this look like in practice?
What will this look like in practice?

% ros2 interface convert_topic_types \\ 
  --from /scan \\ 
  --to /updated/scan \\
# --component-container <container name>

% ros2 topic echo /updated/scan sensor_msgs/msg/LaserScan
...

---

openrobotics
What will this look like in practice?

- Ways to set up conversions:
  - as a stand-alone node
  - as a node component
  - syntactic sugar in a launch file
- Benefits of this approach:
  - keeps QoS and queuing in middleware
  - easy to observe from tools (e.g. rqt_graph)
- Downsides of this approach:
  - requires extra topics and hops through pub/sub
  - requires transfer functions to exist
Needed Underlying Technical Changes

- **TypeDescription.msg**
  - Description of Other Message/Service/etc.

- **Type Version Hashing and Enforcement**
  - Generation and Access via ROS Graph APIs

- **Type Description Distribution**
  - Accessing definition of types remotely

- **Run-Time Interface Reflection**
  - Interacting with types using only the TypeDescription, i.e. reading, writing, sending, and receiving
Run-Time Interface Reflection
The Context

Normally for ROS:
Message files $\rightarrow$ Compile-time Generated Code and Headers
- E.g. String.msg $\rightarrow$ std_msgs::msg::String
  (std_msgs/msg/string.hpp)

But what if…
- You don’t have the message headers
- But you obtain the message description at runtime
  - E.g. From a bagfile, published over a topic, etc.
Run-time Interface Reflection

“At run-time, given a byte buffer and its description… Can we access its members?”

- raw buffer: 6d 65 74 68 79 6c 44 72 61 67 64 6e ...
- type description: {'fields': [(0, 'string')]}
- serialization lib name: "raw_byte_ser"

- List fields
- Get and set fields
- Construct new message instances
- …
Pub-Sub (At run-time)

Using **interface reflection** , and a **message description** , dynamically create at run-time...

![Diagram showing message description leading to publishers, subscribers, and messages, with notes about parsing and dynamic data]

- Must be parsed (Dynamic Type)
- Must allow get/set (Dynamic Data)
Pub-Sub (At run-time)

Pub
1. Parse *description* to create a *dynamic type*
2. Use *dynamic type* to create *dynamic data*
3. Publish *dynamic data*

Sub
1. Receive *dynamic data*
2. Parse *description* to create a *dynamic type*
3. Use *dynamic type* to access *dynamic data*
How will run-time interface reflection be implemented?
Reflection for Different Technologies

For most technologies, we can just create a wrapper

<table>
<thead>
<tr>
<th>Technology</th>
<th>Dynamic Type</th>
<th>Dynamic Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastRTPS (C++)</td>
<td>DynamicType</td>
<td>DynamicData</td>
</tr>
<tr>
<td>RTI Connext (C)</td>
<td>DDS_TypeCode</td>
<td>DDS_DynamicData</td>
</tr>
<tr>
<td>Protobuf (C++)</td>
<td>FileDescriptorProto</td>
<td>DynamicMessage</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

For middlewares that don’t have structured messages, we can just piggyback off any serialization library (e.g. FastCDR)

The type description helps retain type information!
Run-time Interface Reflection Library

This functionality should be a standalone C library that can be used separately from RMW.

The run-time interface reflection library should abstract away serialization! (By piggybacking off a middleware or wrapping a serialization lib!)

This library should be able to support non-DDS, non-XTypes libs! (e.g. Protobuf/Zenoh)
Demo

We made a prototype to check for feasibility and refine the interfaces

🌟 It **WORKS** with FastDDS pub-sub!!🌟

(And there’s a protobuf dynamic example too!!)

methylDragon/rostrype-introspection-prototype
Future Work
Future Work

- More prototypes (e.g. Connext)
- Create the interfaces to abstract away getters and setters
- Type description distribution
- Plug it all into rcl/rmw!
Known Issues

● Some bugs need to be fixed in middlewares related to run-time interface reflection
● Some middlewares lack the necessary interfaces right now
● Some conceptual discrepancies in the type description message, e.g. bounded sequences of bounded strings
Questions

This presentation: https://bit.ly/3dFtIlg
The REP PR: https://tinyurl.com/rep-2011-pr
Pub-Sub

Publisher

.type description

Dynamic Type

creates instance of

Middleware

Dynamic Data

Subscription

.type description

Dynamic Type

reads

Dynamic Data
Demo: It works! (FastDDS Pub-Sub)

`$ ros2 run evolving_serialization_examples fastrtps_evolving_pub`

`$ ros2 run evolving_serialization_examples fastrtps_evolving_sub`

Pub-sub Demo Code

[QR Code]
tinyurl.com/fastdds-ets-pubsub
Demo

We made a prototype to check for feasibility and refine the interfaces

🌟 It **WORKS** with FastDDS pub-sub!! 🌟

(And there’s a protobuf dynamic example too!!)

methylDragon/rostop-type-introspection-prototype
Demo: Type Description

= [PRINTING TYPE DESCRIPTION] =
  referenced_type_descriptions_count: 2

== [MAIN DESCRIPTION] ==

[INDIVIDUAL DESCRIPTION]
  type_name: ExampleMsg
type_version_hash: (null)
field_count: 3

[FIELD]
  field_name: string_field
  field_type: 15
  field_array_size: 0
  nested_type_name: (null)

[FIELD]
  field_name: bool_static_array_field
  field_type: 34
  field_array_size: 5
  nested_type_name: (null)

[FIELD]
  field_name: nested_field
  field_type: 1
  field_array_size: 0
  nested_type_name: inner

[INDIVIDUAL DESCRIPTION]
  type_name: inner
  type_version_hash: (null)
  field_count: 1

[FIELD]
  field_name: doubly_nested_field
  field_type: 1
  field_array_size: 0
  nested_type_name: inner_inner
FastDDS Prototype: Type Description

```c
// TYPE DESCRIPTION =====================================
typedef struct type_description_t
{
    individual_type_description_t * type_description;
    GHashTable * referenced_type_descriptions;
} type_description_t;
```

The `type_description_t` struct allows us to iterate through the fields and obtain necessary information to construct the type.

```c
// INDIVIDUAL TYPE DESCRIPTION =========================
typedef struct individual_type_description_t
{
    char * type_name;
    char * type_version_hash;

    type_description_field_t ** fields;
    size_t field_count;
} individual_type_description_t;
```

```c
// TYPE DESCRIPTION FIELD ===============================
typedef struct type_description_field_t
{
    char * field_name;
    uint8_t field_type;

    uint64_t field_array_size;
    char * nested_type_name;
} type_description_field_t;
```
FastDDS Prototype: ETS

The EvolvingTypeSupport (ETS) is a C interface to be filled by any downstream implementations!
FastDDS Prototype: ETS

With the type_description_t struct, we can iterate through the fields and call the necessary methods to create the type!

```c
auto example_msg_type = eprosima::fastrtps::types::DynamicType_ptr{
    std::move(
        *reinterpret_cast<eprosima::fastrtps::types::DynamicType_ptr*>(
            ets_construct_type_from_description(ets, full_description_struct)
        )
    );
}
```

```c
void fastrtps_add_char_member(
    EvolvingFastRtpsTypeSupportImpl *ets_impl, void *builder, uint32_t id, const char *name)
{
    static_cast<DynamicTypeBuilder*>(builder)->add_member(
        id, name, ets_impl->factory_->create_char8_type() );
}

void fastrtps_add_float32_member(
    EvolvingFastRtpsTypeSupportImpl *ets_impl, void *builder, uint32_t id, const char *name)
{
    static_cast<DynamicTypeBuilder*>(builder)->add_member(
        id, name, ets_impl->factory_->create_float32_type() );
}

void fastrtps_add_float64_member(
    EvolvingFastRtpsTypeSupportImpl *ets_impl, void *builder, uint32_t id, const char *name)
{
    static_cast<DynamicTypeBuilder*>(builder)->add_member(
        id, name, ets_impl->factory_->create_float64_type() );
}
```
Demo: It works! (FastDDS Pub)

You can use the same interface on the subscription side!

This is grabbed at runtime!

```c
static EvolvingTypeSupport * ets = ets_init(
    create_fastrtps_evolving_typesupport_impl(),
    create_fastrtps_evolving_typesupport_interface());

type_description_t * full_description_struct = create_type_description_from_yaml_file(msg_path);

auto example_msg_type = eprosima::fastdds::types::DynamicType_ptr(
    std::move(
        *reinterpret_cast<eprosima::fastdds::types::DynamicType_ptr *>(
            ets_construct_type_from_description(ets, full_description_struct)
    )
);

// Create and Populate Data
this->msg_data_ = DynamicDataFactory::get_instance()->create_data(example_msg_type);

this->msg_data_->_set_string_value("A message!", 0);
auto bool_array = this->msg_data_->_loan_value(1);
for (uint32_t i = 0; i < 5; ++i) {
  bool_array->_set_bool_value(false, bool_array->_get_array_index({i}));
}
this->msg_data_->_return_loaned_value(bool_array);
```
Demo: It also works for protobuf (no protoc)!

In this case the `DynamicType` comes from a runtime-generated .proto file!

We use the `DynamicMessage` interfaces from protobuf to construct the message from the generated .proto file contents.

The demo repo includes the proto file generator library! (`protogen`)