How to Define Custom Messages in ROS
Wyatt Newman
July, 2015

Our minimal nodes illustrated use of standard messages (std_msgs) for communicating via publish/subscribe. In fact, the std_msgs package defines 32 message types. Some of these messages (e.g. Header.msg) are composed of more primitive messages. In this way, one can build up quite complex message types. Some additional useful (more complex) messages are defined in additional packages, such as: geometry_msgs, sensor_msgs, nav_msgs, pcl_msgs, visualization_msgs, trajectory_msgs, and actionlib_msgs. Defined messages can be examined interactively using "rosmsg show", followed by the package/message_name of the message of interest. E.g., entering:

```
rosmsg show std_msgs/Header
```

outputs:

```
uint32 seq
time stamp
string frame_id
```

which shows that "Header" is comprised of 3 fields: seq, stamp and frame_id. These message names store data of primitive types uint32, time and string, respectively.

If a message type already exists in the standard distribution of ROS, you should use that existing message. However, it is sometimes necessary to define one's own message. Defining custom messages is described at: http://wiki.ros.org/ROS/Tutorials/DefiningCustomMessages, which can be consulted for more details. The present document introduces the basics of how to define custom messages, with reference to corresponding code in the class repository, within the package "example_ros_msg."

The package "example_ros_msg" was created using:

```
catkin_create_pkg example_ros_msg roscpp catkin_simple std_msgs
```

This creates a directory structure under "example_ros_msg." By depending on "catkin_simple" (a dependency that will appear in the "package.xml" file), we will be able to use the the abbreviated "Cmakelists.txt."

To define a new message type, we create a subdirectory in this package called "msg." Within this msg directory, we create a new text file by the name of "example_message.msg." The example message file contains only 3 relevant lines:

```
Header header
int32 demo_int
float64 demo_double
```

This message type will have three fields, which may be referred to by the names: header, demo_in and demo_double. Their types are Header, int32 and float64, respectively, which are all message types defined in the package "std_msgs."

To inform the compiler that we need to generate new message headers, the "package.xml" file must be edited. Insert (or uncomment) the following lines:

```
<build_depend>message_generation</build_depend>
```

and

```
<run_depend>message_runtime</run_depend>
```

The abbreviated CMakeLists.txt file is simply:

```
cmake_minimum_required(VERSION 2.8.3)
project(example_ros_msg)

find_package(catkin_simple REQUIRED)

catkin_simple()

# Executables
#cs_add_executable(example_ros_message_publisher src/example_ros_message_publisher.cpp)

cs_install()
cs_export()
```

Note the cs_add_executable is commented out. We will enable this once we have our anticipated source code for a test node, “example_ros_message_publisher.cpp.”

Having defined a message type, we can generated corresponding header files suitable for C++ file inclusion. Compiling the code with “catkin_make” produces a header file, which it installs in the directory:

ros_ws/devel/include/example_ros_msg/example_message.h

Source code for nodes that want to use this new message type should depend on the package “example_ros_msg” (in the corresponding package.xml file) and should include the new header with the line:

#include <example_ros_msg/example_message.h>

in the source code of the node using this message type.

Example node using custom message: The class code includes a source file under example_ros_msg/src/example_ros_message_publisher.cpp. This node uses the new message type as follows. It defines a publisher object as:

```
ros::Publisher my_publisher_object =
    n.advertise<example_ros_msg::example_message>("example_topic", 1);
```

which says that topic “example_topic” will carry messages of type “example_ros_msg::example_message.”

We also instantiate an object of type example_ros_msg::example_message with the line:

```
example_ros_msg::example_message  my_new_message;
```

Note that when referring to the header file, we use the notation example_ros_msg/example_message.h (path to the header file), but when instantiating an object based on this definition (or referring to the datatype for publication), we use the class notation: example_ros_msg::example_message.

Within the source code of example_ros_message_publisher.cpp the various fields of the new message object, my_new_message, are populated, and then this message is published.

Populating some fields of the new message type is simple, e.g.:

```
my_new_message.demo_int= 1;
```

But accessing elements of hierarchical fields requires drilling down deeper, as in:

```
my_new_message.header.stamp = ros::Time::now(); //set the time stamp in the header;
```

Here, “stamp” is a field within the “Header” message type for the field “header.” Additionally, this line of code illustrates another useful ROS function: ros::Time::now(). This looks up the current time and returns it in a form compatible with “header” (consisting of separate fields for seconds and nanoseconds). Note: the absolute time is essentially meaningless. However, differences in time can be used as valid time increments.

By uncommenting the line in CmakeLists.txt,

```
cs_add_executable(example_ros_message_publisher src/example_ros_message_publisher.cpp)
```

and re-running “catkin_make”, a new node is created, with the name “example_ros_message_publisher.”

Running this node (assuming roscore is running):

```
rosrun example_ros_msg example_ros_message_publisher
```

produces no output. However, (from a separate terminal), running:

```
rostopic list
```

reveals that there is a new topic, “example_topic”. We can examine the output of this topic with:

```
rostopic echo example_topic
```

which produces the following output:

```
header:
  seq: 1
  stamp:
    secs: 1435969105
    nsecs: 735021607
  frame_id: base_frame
demo_int: 4
demo_double: 3.16227766017
---
header:
  seq: 2
  stamp:
    secs: 1435969106
    nsecs: 735011756
  frame_id: base_frame
demo_int: 8
demo_double: 1.77827941004
---
header:
  seq: 3
  stamp:
    secs: 1435969107
    nsecs: 735049942
  frame_id: base_frame
```

```
demo_int: 16
demo_double: 1.33352143216
---
header:
  seq: 4
  stamp:
    secs: 1435969108
    nsecs: 735050946
  frame_id: base_frame
demo_int: 32
demo_double: 1.15478198469
---
```

…
We see that our new node successfully uses the new message type. Sequence numbers increase monotonically. The demo_int field is doubled each iteration (per the logic of the source code). The demo_double field displays sequential square-roots (starting from 100). The “secs” field of the header increments by 1 second each iteration (since the iteration rate timer was set to 1Hz). The string “base_frame” appears in the “frame_id” field.

Conclusion: This presentation has shown how to define and use a custom message type. Having defined a new message type, nodes within the same package or nodes in other packages can use the new message type—provided the external packages list “example_ros_msg” as a dependency (in the corresponding package.xml file).

Defining message types for ROS services and for ROS action servers follows a similar procedure.