Robotics Lab: ROS - Programming examples

Week 4

Mario Selvaggio

This documents contains ROS programming examples and good practices. It assumes you know how to how to create, init and build a catkin_ws.
In the following it is assumed that the path to your catkin_ws is `~/catkin_ws/` and you have sourced the `/opt/ros/noetic/setup.bash` file.

Downolad and compile a simple publisher/subscriber

Go to github, download compile.

Create your own package

First of all, create a new ros package called `~/my_package/`

```
$ catkin_create_pkg my_package std_msgs roscpp
```

Build your package and source your workspace

```
$ catkin build
$ source devel/setup.bash
```

After creation, you can customize your package by modifying the `package.xml` file. The final should look like this

```
<?xml version="1.0"?>
<package format="2">
<name>beginner_tutorials</name>
<version>0.1.0</version>
<description>The my_package package</description>

<maintainer email="you@yourdomain.tld">Your Name</maintainer>
<license>BSD</license>
<url type="website">http://wiki.ros.org/beginner_tutorials</url>
<author email="you@yourdomain.tld">Jane Doe</author>

<buildtool_depend>catkin</buildtool_depend>

<build_depend>roscpp</build_depend>
<build_depend>rospy</build_depend>
<build_depend>std_msgs</build_depend>

<exec_depend>roscpp</exec_depend>
<exec_depend>rospy</exec_depend>
<exec_depend>std_msgs</exec_depend>

</package>
```

Creating your own message

msg files are simple text files that describe the fields of a ROS message. They are used to generate source code for messages in different languages. msg files are stored in the msg directory of a package.
To create your own message you have to create a msg folder, with a .msg file inside

```
$ roscd my_package
$ mkdir msg
$ touch my_msg.msg
```

and fill it with some content, e.g.

```
  string name
  uint32 data
```

Open `package.xml`, and make sure these two lines are in it and

```
  <build_depend>message_generation</build_depend>
  <exec_depend>message_runtime</exec_depend>
```

Open `CMakeLists.txt` and add the `message_generation` dependency to the `find_package` call which already exists in your `CMakeLists.txt`

```
# Do not just add this to your CMakeLists.txt, modify the existing text to add message_generation before the
     closing parenthesis
find_package(catkin REQUIRED COMPONENTS
roscpp
rospy
std_msgs
message_generation
)
```

Also make sure you export the message runtime dependency.

```
catkin_package(
...
CATKIN_DEPENDS message_runtime ...
...)
```

Uncomment the `add_message_files` line

```
add_message_files(
FILES
my_msg.msg
)
```

Now we must ensure the `generate_messages()` function is called.

Publish your own message

Write a simple publisher. Start by creating a C++ file

```
  $ roscd my_package
  $ mkdir -p src
  $ touch my_package_node.cpp
```

and fill it with

```
#include "ros/ros.h"
#include "std_msgs/String.h"
#include "my_package/my_msg.h"

#include <sstream>

int main(int argc, char **argv)
{
  ros::init(argc, argv, "my_package_node");
  ros::NodeHandle n;

  ros::Publisher my_package_pub = n.advertise<std_msgs::String>("my_package_topic", 1000);

  ros::Rate loop_rate(10);
```

```
  int count = 0;

  my_package::my_msgs msg;
  msg.name = "my_msg_name";L
  while (ros::ok())
  {

    msg.data = count;

    ROS_INFO("%s", msg.data.c_str());

    my_package_pub.publish(msg);

    ros::spinOnce();
    loop_rate.sleep();
    ++count;
  }

  return 0;
}
```

Tell ROS to build your node by modyfing the CMakeLists.txt as follows

```
  cmake_minimum_required(VERSION 2.8.3)
  project(my_package)

  ## Find catkin and any catkin packages
  find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)

  ## Declare ROS messages and services
  add_message_files(FILES my_msg.msg)

  ## Generate added messages and services
  generate_messages(DEPENDENCIES std_msgs)

  ## Declare a catkin package
  catkin_package()

  ## Build talker and listener
  include_directories(include ${catkin_INCLUDE_DIRS})

  add_executable(my_package_node src/my_package_node.cpp)
  target_link_libraries(my_package_node ${catkin_LIBRARIES})
  add_dependencies(my_package_node my_package_generate_messages_cpp)
```

Run it by first starting running

```
  $ roscore
```

In another terminal, run the command

```
  $ rosrun my_package my_package_node
```

Exercise: create a subscriber node within the same package.

Use of parameters

You can use `rosparam set` to specify a parameter e.g.

```
  $ rosparam set my_int_param 10
```

Add the following lines to your node

```
int param;
if (n.getParam("my_int_param", param))
  ROS_INFO("my_int_param is %d",param);
else
  ROS_INFO("my_int_param not defined");
```

Run your node. You can alternatively specify the param as argument of the node as

```
$ rosrun my_package my_package_node _my_int_param:=10
```

but creating a private node handle `ros::NodeHandle n("~")`.

Create a launch file

A launch file is used to launch multiple nodes. It can have params as arguments or load a file of params. Create a `launch` folder with a `my_launch_file.launch`. From the `root` folder of your package

```
$ mkdir launch; cd launch
$ touch my_launch_file.launch
```

Open `my_launch_file.launch` in your favorite editor and paste the following content inside

```
<launch>
  <arg name="my_int_param" default="1"/>
  <param name="my_int_param" value="$(arg my_int_param)" />
  <node name="talker" pkg="my_package" type="my_package_node" output="screen"/>
</launch>
```

To launch

```
$ roslauch my_package my_launch_file.launch
```

If you want to additionally pass `my_int_param` from outside

```
$ roslauch my_package my_launch_file.launch my_int_param:=10
```

Load parameters from file

Create a `config/params.yaml` file and add content to it

```
$ mkdir config; cd config
$ touch params.yaml
$ echo "my_int_param: 10" >> params.yaml
```

Comment out the two following lines in the lauch file

```
<!--arg name="my_int_param" default="1"/-->
<!--param name="my_int_param" value="$(arg my_int_param)" /-->
```

Replace with

```
<rosparam command="load" file="$(find my_package)/config/params.yaml"/>
```

Object oriented programming

It is always a good idea to use object oriented programming.
To do so you have to work with classes and separate .h from .cpp files.
Let's create a .h file for our ros node

```
$ cd include/my_package
$ touch myPackage.h
```

open `myPackage.h` in your favorite editor and paste the following code inside

```
// ROS
#include <ros/ros.h>
#include "std_msgs/String.h"


/*!
* Main class for the node to handle the ROS interfacing.
*/
class MyPackage
{
  public:
  /*!
  * Constructor.
  * @param nodeHandle the ROS node handle.
  */
  MyPackage(ros::NodeHandle& nodeHandle);

  /*!
  * Destructor.
  */
  virtual ~MyPackage();

  private:
  /*!
  * Reads and verifies the ROS parameters.
  * @return true if successful.
  */
  bool readParameters();

  /*!
  * ROS topic callback method.
  * @param message the received message.
  */
  void topicCallback(const std_msgs ::String& message);

  //! ROS node handle.
  ros::NodeHandle& nodeHandle_;

  //! ROS topic subscriber.
  ros::Subscriber subscriber_;

  //! ROS topic name to subscribe to.
  std::string subscriberTopic_;

};
```

Now create a `myPackage.cpp` file into the `src` folder with the following content

```
#include "my_package/myPackage.h"

// STD
```

```
#include <string>

MyPackage::MyPackage(ros::NodeHandle& nodeHandle)
: nodeHandle_(nodeHandle)
{
  if (!readParameters()) {
    ROS_ERROR("Could not read parameters.");
    ros::requestShutdown();
  }
  subscriber_ = nodeHandle_.subscribe(subscriberTopic_, 1,
  &MyPackage::topicCallback, this);

  ROS_INFO("Successfully launched node.");
}

MyPackage::~MyPackage()
{
}

bool MyPackage::readParameters()
{
  if (!nodeHandle_.getParam("subscriber_topic", subscriberTopic_)) return false;
  return true;
}

void MyPackage::topicCallback(const std_msgs::String& message)
{
  ROS_INFO("I heard: '%s'", message.data.c_str());
}
```

Uncomment the following lines in your `CMakeLists.txt` file

```
include_directories(
  include
  ${catkin_INCLUDE_DIRS}
)
```

Change the `add_executable` line to

```
add_executable(${PROJECT_NAME}_node src/my_package_node.cpp src/myPackage.cpp)
```

We have created a subscriber within a class that awaits a topic whose name is to be specified into the params.yaml file

```
my_int_param: 1000
subscriber_topic: "my_package_subscriber_topic"
```

Now compile and execute with

```
$ catkin_make
$ roslaunch my_package my_lauch_file.launch
```

From another terminal

```
rostopic pub -r 10 /my_package_subscriber_topic std_msgs/String "data: 'Hello'"
```