ROS-I Basic Training "Mobility"
ROS Filesystem Tutorial

Instructor: MASCOR Institute

2017ff

FH AACHEN
UNIVERSITY OF APPLIED SCIENCES

Contents

H2020 funded GA no. 732287

1 Introduction

During this tutorial, you will learn how to navigate through your ROS file system. In addition, you will create your own ROS workspace, compile your first C++ ROS Nodes and start them up. Further more you will learn how to use the command line interface of ROS to inspect alle running topics and messages.

- Lines beginning with $ are terminal commands
- Lines beginning with # indicate the syntax of the commands

2 Terminal usage

- opening a new terminal : ctrl+alt+t
- opening a new tab inside an existing terminal : ctrl+shift+t
- killing an active process inside a terminal: ctrl+c

3 Catkin Workspace

Catkin is the official build system of ROS. A catkin workspace is a folder, in which you modify, build, and install packages. It is the place to create your own packages and nodes or modify existing ones to fit your application. In the further tutorials, you will work in your own workspace.

3.1 Create a workspace

```
$ cd
$ mkdir -p ~/robot_ws/src
$ cd ~/robot_ws/
$ catkin_make
```


H2020 funded GA no. 732287

After running *catkin_make*, you should notice two new folders in the root of your workspace: build and devel. The build folder is where cmake and make are invoked. The devel folder contains any generated files, targets and setup.bash files. The setup.bash files are used to add the workspace to the ROS environment of your system.

3.2 Sourcing a workspace

A workspace must be sourced for each terminal session. This is done by calling the setup.sh file of the devel-subfolder of the workspace you want to source. The location of the workspace you want to work with will then be added to the **ROS_PACKAGE_PATH**. By adjusting this environment variable, you can add ROS workspaces and even separate ROS packages to your ROS system. It can contain one or more paths, which tells the ROS system where to search for ROS packages. If there are multiple packages of the same name, the first one appearing in the **ROS_PACKAGE_PATH** is chosen.
The following command will add the robot_ws workspace to the **ROS_PACKAGE_PATH** *only for this* terminal session:

```
$ source ~/robot_ws/devel/setup.bash
```

It is recommended to auto-source the workspace each time a new terminal is opened. This can be done by adding the command to the .bashrc file. The .bashrc file is located in your home directory. You can directly add the command to the file by entering:

```
$  echo 'source ~/robot_ws/devel/setup.bash' >> ~/.bashrc
```

If you want to edit your .bashrc later, you can use any editor of your choice. This tutorial is assuming nano, which is installed on most linux distributions by default.

```
$ nano ~/.bashrc
```

nano usage:

- opening a file : nano <filename>

- save : ctrl+o, enter filename, then enter
- exit nano: ctrl+x

4 Catkin Packages

Two types of ROS packages exist: binary packages and build-from-source packages. ROS binary packages are in Ubuntu provided as debian packages, which can be managed via apt commands, where as build-from-source packages have to be deployed in the src folder of your ROS workspace.

4.1 Installing binary packages

Since all ROS packages do have a naming convention, you can easily install them with the following syntax:
sudo apt install ros-<distribution>-<package_name>
Example:

```
$ sudo apt install ros-kinetic-turtlesim
```

Hint: The package turtlesim may already be installed on your system, so executing the command might result in installing 0 new packages.

4.2 Installing build-from-source packages

Build-from-source packages can be splitted into packages provided by ROS and your own developments. Both have to be placed into the src folder of a ROS workspace. The ROS wiki page provides information about nearly any available package including a link to download the build-from-source package – mostly via GIT. The command for cloning a git repository has the following syntax:
git clone <path-to-repository>
respectively

git clone -b <branch> <path-to-repository>

H2020 funded GA no. 732287

Example:

```
$ cd ~/robot_ws/src
$ git clone -b {indigo-devel} https://github.com/ros/ros_tutorials.git
```

Download the ros_tutorials repository and try to compile it!

4.3 Creating build-from-source packages

Packages can be created by using the *catkin_create_pkg* command:

catkin_create_pkg <package_name> <dependency1> <dependency2> ...

Create your own build-from-source package:

```
$ cd ~/robot_ws/src/
$ catkin_create_pkg myfirstpackage roscpp std_msgs
```

This will create the package *myfirstpackage* with the ROS dependencies roscpp and std_msgs.

4.4 Building build-from-source packages

By executing the command *catkin_make*, all packages inside the workspace are built. Sometimes new built packages are not instantly listed after building them. To force the listing of the new built packages, use rospack profile, if necessary.
Build the packages in the workspace:

```
$ cd ~/robot_ws
$ catkin_make
$ rospack profile
```

5 File System Navigation

ROS also provides a lot of helpful tools to quickly navigate through the filesystem. Here is a selection of the most important helpers:
Locating a ROS package:
rospack find <package_name>

```
$ rospack find myfirstpackage
```

Listing package content:
rosls <package name>

```
$ rosls myfirstpackage
```

Navigating to a specific package directory:
roscd <package_name>

```
$ roscd turtlesim
```

A ROS package should be simple designed in order to be adjustable and reusable. Most ROS packages include only functionalities to fulfil specific tasks. Therefore, mostly all ROS packages are depending on other packages, which are offering functionalities for tasks on a lower level. These packages have their own dependencies, which ends up in a tree structure.
Listing the level one dependencies:
rospack depends1 <package_name>

```
$ rospack depends1 myfirstpackage
```

Listing all dependencies:

rospack depends <package_name>

```
$ rospack depends myfirstpackage
```

Automatic installing of package dependencies:
rosdep install <package_name>

```
$ rosdep install myfirstpackage
```

More useful file system tools can be found on the ROS Cheatsheet.

5.1 Creating your own Custom Message

Allthough ROS offers a build-in message type for nearly every sensor or other robot data, sometimes you might need to create a custom message that fits your special needs.
Go to the path of your package and create a *msg* folder. Inside this folder, create a file *MyMessage.msg*:

```
$ roscd myfirstpackage
$ mkdir msg
$ cd msg
$ touch MyMessage.msg
```

Now open this file and compose your own message using the build-in ROS message. You can choose any message types you want. The following message consists of a header (incl a timestamp), and two data fields *value* of type <std_msgs/Uint8> and *text* of type <std_msgs/String>:

```
Header header
uint8 value
string text
```

To let ROS generate a header file out of that message definition, which then can be included later in a ROS Node for example, you need to edit the *CMakeLists.txt* of your package. At first make sure that the following ROS dependencies are included:

```
find_package(catkin REQUIRED COMPONENTS
  message_generation
  std_msgs
  roscpp
)
```

then locate the *add_ message_ files* section and comment it in like this:

```
# Generate messages in the 'msg' folder
add_message_files(
  FILES
  MyMessage.msg
)
```

Locate the *generate_ messages* section and tell CMake to generate the messages :

```
generate_messages(
  DEPENDENCIES
  std_msgs  # Or other packages containing msgs
)
```

Finally, go to the root of your workspace and execute ţextitcatkin_make:

```
  $ cd ~/robot_ws
  $ catkin_make
```

You should now see the message beeing generated. If you like, you can inspect the result located in the devel-folder of you workspace:

```
  $ nano ~/robot_ws/devel/include/myfirstpackage/MyMessage.h
```

6 ROS Nodes

A ROS node is an executable in the ROS environment. A node is always part of a ROS package. A single ROS node can be started using the *rosrun* command. Groups of nodes, or even a whole system, can be launched using the *roslaunch* command.

6.1 Running single nodes

rosrun <package> <node>

Start the ROS master:

```
$ roscore
```

Hint: Before starting a node, you have to start the ROS master.
Start the talker node from package roscpp_tutorials:

```
$ rosrun roscpp_tutorials talker
```

Start the listener node from package roscpp_tutorials:

```
$ rosrun roscpp_tutorials listener
```

To get some introspection into running nodes, available topics or used message types, ROS provides the *rostopic* tool, which has the following syntax:

rostopic echo | list | info

A list of all available topics can be retrieved with

```
$ rostopic list
```

Both nodes that we just started are communicating using the topic */chatter*.

Retrieve some more information about this topic:

```
$ rostopic info /chatter
```

Take a look at the raw data that is send over this topic:

```
$ rostopic echo /chatter
```

Close all running nodes (ctrl+c) and also stop the roscore.

6.2 Launchfiles

roslaunch <package> <launchfile>

Go to your own package and create a folder *launch*:

```
$ roscd myfirstpackage
$ mkdir launch
```

Create a file example.launch inside the launch folder:

```
$ roscd myfirstpackage
$ mkdir launch
$ cd launch
$ nano example.launch
```

Open the file and insert the following:

```
<?xml version="1.0"?>
<launch>
<node pkg="roscpp_tutorials" type="talker" name="talker"
                output="screen"/>
<node pkg="roscpp_tutorials" type="listener" name="listener"
                output="screen"/>
</launch>
```

Now try launching all the nodes by just calling the launchfile:

```
$ roslaunch myfirstpackage example.launch
```

Hint: When using launchfiles, roscore will automatically be started first if not already running.

6.3 Creating your own C++ ROS Node

Copy the files *talker.cpp listener.cpp* to the src folder of your package:

```
$ roscd myfirstpackage
$ cd src
$ cp ../../ros_tutorials/roscpp_tutorials/talker/talker.cpp ./
$ cp ../../ros_tutorials/roscpp_tutorials/listener/listener.cpp ./
```

rename the source files:

```
$ mv talker.cpp mytalker.cpp
$ mv listener.cpp mylistener.cpp
```

Now lets make the talker node use our previously created custom message instead of just a string.
Edit *mytalker.cpp*:

```
$ nano mytalker.cpp
```

Replace the include of std_msgs/String with your own Message Header:

```
//#include <std_msgs/String.h>
#include <myfirstpackage/MyMessage.h>
```

Since a node name must be unique in ROS, we have to change the name of the node:

```
  ros::init(argc, argv, "mytalker");
```

Now change the message type of the publisher:

```
  ros::Publisher chatter_pub =
      n.advertise<myfirstpackage::MyMessage>("chatter", 1000);
```

Locate the instantiation of the published message and change it to the message type of your custom message:

```
myfirstpackage::MyMessage msg;
```

When filling the created message with data, we have to consider that the names of the data fields have changed, so we need to fill them accordingly to the message definiton:

```
msg.text = ss.str();
msg.value = 15;
msg.header.stamp = ros::Time::now();
```

```
ROS_INFO("%s", msg.text.c_str());
```

Finally we have to edit our *CMakeLists.txt* and tell CMake to compile our new node.
Edit *CMakeLists.txt*:

```
  $ roscd myfirstpackage
  $ nano CMakeLists.txt
```

Locate the *add_executable* section and add the following lines:

```
add_executable(mytalker src/mytalker.cpp)
```

Locate the *target_link_libraries* section and add the following lines:

```
  target_link_libraries(mytalker
```

```
        ${catkin_LIBRARIES}
)
```

Make sure that the dependencies are properly set for the mytalker node (this makes sure that the message generation for this particular package is executed before the mytalker node is compiled):

```
add_dependencies(mytalker ${$PROJECT_NAME}_EXPORTED_TARGETS} ${
    ↪ catkin_EXPORTED_TARGETS}})
```

Try to compile your node:

```
$ cd ~/robot_ws
$ catkin_make
```

Run your node:

```
$ rosrun myfirstpackage mytalker
```

Optional tasks:

- Also modify the mylistener.cpp to subscribe to the custom message
- modify CMakeLists.txt to also compile mylistener.cpp
- update your launchfile to auto-launch the new ros nodes
- use rostopic list / info / echo on the new message

H2020 funded GA no. 732287