Getting started with ROS at the DBL

Wouter Caarls

July 23, 2013

Contents

Preliminaries

Some notation: anything you should see on your screen is shown in `Courier`. All commands in Linux are *case sensitive*, which means that there is a difference between a small "a" and a capital "A". This also holds for file names. Anything you should type in is shown in Courier too, but it's `underlined` as well. Keys you should press are typed in boxes, for example `Enter`.

1 Overview

On the surface, ROS is just another communications middleware for robots. But it is much more than that. It is an ecosystem for the development of robotics software packages. Most importantly, it is a community of roboticists that develop such packages. The basic infrastructure is maintained by Willow

Garage, and documentation is kept at a central location (`http://www.ros.org`), but everyone can create their own repository of packages and submit it to be indexed by the ROS software browser.

ROS packages generally contain tools, libraries or *nodes*. Nodes perform certain tasks (such as path planning, or sending velocity control commands to a motor) and use the ROS middleware to communicate with other nodes. The complete robot control software then is a bunch of communicating nodes. ROS uses the publish/subscribe mechanism of communication: one node *publishes messages* to a *topic*, and another node *subscribes* to that topic, being notified on the reception of new messages. This way, *who* publishes the data becomes irrelevant, greatly reducing complexity and facilitating interchange.

Other conveniences include tools to navigate the filesystem (`roscd`), launch multiple nodes at the same time (`roslaunch`), view the communications graph (`rxgraph`), visualize information (`rviz`) and generate documentation (`rosdoc`).

2 Installation

We assume you're installing ROS on a recent Ubuntu system. If not, find the instructions for your particular system on `http://www.ros.org`. A basic introduction to Linux can be found at `http://ros.caarls.org/linux.pdf`.

There are two parts to the installation: the system installation needs to be done only once per computer, and requires administrator access. The user installation should be performed by user individually and doesn't need any special privileges.

2.1 System installation

Follow the installation instructions on `http://www.ros.org/wiki/fuerte/Installation/Ubuntu`. Use the *Desktop-Full* install. This will install ROS into the `/opt/ros` directory structure. Skip the section about *Environment setup*.

Then, install the *rosinstall* package (see `http://www.ros.org/wiki/rosinstall`), which we'll use to do the user installation:

```
wcaarls@vbox:~$ sudo apt-get install python-rosinstall python-rosdep
```

2.2 User installation

You will now create an *overlay* of the ROS distribution to create your own packages. This is just a directory in your homedir in which you can edit to your heart's content without breaking the system itself. Execute the following commands:

```
wcaarls@vbox:~$ rosinstall ~/ros /opt/ros/fuerte
wcaarls@vbox:~$ echo "source ~/ros/setup.bash" >> ~/.bashrc
wcaarls@vbox:~$ source ~/.bashrc
```

You may also want to edit `~/.bashrc` to add the following line at the end:

```
export ROS_PACKAGE_PATH=~/ros:$ROS_PACKAGE_PATH
```

This makes sure that any new packages you create in `~/ros` can by found by all the ROS utilities.

3 The repository

The DBL has its own repository for ROS packages, located at `https://svn.3me.tudelft.nl/dbl-ros-pkg`. To check out this repository, type the following two lines:

```
wcaarls@vbox:~$ cd ~/ros
wcaarls@vbox:~$ svn co https://svn.3me.tudelft.nl/dbl-ros-pkg/trunk dbl-ros-pkg-dev
```

If you later wish to update to the latest version of the DBL packages, simply type

```
wcaarls@vbox:~$ cd ~/ros/dbl-ros-pkg-dev && svn up
wcaarls@vbox:~$ rosmake --pre-clean dbl_repos
```

Some documentation of the packages in the DBL repository can be found at `http://ros.caarls.org`. When building the DBL ROS repository, it will pull some files from the general DBL repository. To make sure this works, you need to store your password for it in your keyring. To do that, simply type

```
wcaarls@vbox:~$ svn info https://svn.3me.tudelft.nl/BME-BMechE/DBL
```

and log in with your netid and password. If that doesn't work because of keyring issues, run

```
wcaarls@vbox:~$ roscd dbl_repos && make
```

4 Getting started

You should start by reading the introduction on the ROS site at `http://www.ros.org/wiki/ROS/Introduction` and further. Next, you should follow all the tutorials at `http://www.ros.org/wiki/ROS/Tutorials`. This will give you a basic understanding of the system and allow you to write small programs.

5 Useful packages

The power of ROS is its community. People all over the world are making packages for it. Make sure you use their work! Instead of immediately writing your own code, first look on the ROS wiki whether what you want is already available in a package. Often, it will not be entirely what you had in mind, but try to work around those differences. Too much fragmentation does not benefit the community.

Below is a list of some pre-installed packages that are useful generally. For more specific needs, browse the available software at `http://www.ros.org/browse/list.php`.

tf `http://www.ros.org/wiki/tf`
: tf is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

urdf http://www.ros.org/wiki/urdf/XML
: The Unified Robot Description Format (URDF) is an XML specification to describe a robot. This specification is designed to be as general as possible, but obviously the specification cannot describe all robots. The main limitation at this point is that only tree structures can be represented, ruling out all parallel robots. Also, the specification assumes the robot consists of rigid links connected by joints; flexible elements are not supported. See the tutorial at http://www.ros.org/wiki/urdf/Tutorials/Create%20your%20own%20urdf%20file.

rviz http://www.ros.org/wiki/rviz
: A 3d visualization environment for robots. Apart from displaying many different kinds of primitives, it can also show complete URDF robot models and their current state as published in a tf tree. See the tutorial at http://www.ros.org/wiki/urdf/Tutorials/Building%20a%20Visual%20Robot%20Model%20with%20URDF%20from%20Scratch.

actionlib http://www.ros.org/wiki/actionlib
: The actionlib package provides a standardized interface for interfacing with preemptible tasks. Examples of this include moving the base to a target location, performing a laser scan and returning the resulting point cloud, detecting the handle of a door, etc.

rosdoc http://www.ros.org/wiki/rosdoc
: rosdoc wraps documentation tools like doxygen, sphinx, and epydoc with ROS package semantics to auto-generate documentation for ROS packages. rosdoc also supports online documentation, like the ROS wiki.

5.1 Installing third-party packages

To install packages that are listed on the ROS wiki, we recommend you to use `rosinstall`:

```
wcaarls@vbox:~$ roslocate info pkg_name > /tmp/pkg_name.rosinstall
wcaarls@vbox:~$ rosinstall ~/ros /tmp/pkg_name.rosinstall
```

Other packages should be checked out in your `~/ros` directory, or an appropriate subdirectory structure thereof. For example, you could create a subdirectory structure using the repository names (such as dbl-ros-pkg). Make sure that you have all the version control software installed that such packages may need. For example, by running

```
wcaarls@vbox:~$ sudo apt-get install mercurial subversion git-core
```

To make sure all system dependencies of a package are installed, run your first `rosmake` command using

```
wcaarls@vbox:~$ rosmake --rosdep-install pkg_name
```

6 Cheat sheet

There's a cheat sheet for common ROS console commands at http://www.ros.org/wiki/Documentation?action=AttachFile&do=get&target=ROScheatsheet.pdf. Below there's a summary of the basic ROS C++ bindings.

Table 1: ROS naming conventions

Type	**Name**
Packages	lower_case_with_underscores[1]
Nodes	lower_case_with_underscores[2]
Topics	lower_case_with_underscores[1]
Message types	UpperCaseCamel
Message fields	lower_case_with_underscores
Classes	UpperCaseCamel
Methods	lowerCaseCamel
Variables	lower_case_with_underscores
Member variables	ending_in_underscore_

Table 2: ROS package structure

Subdirectory	**Contents**
package_name/bin	Compiled binaries
package_name/cfg	Configuration files
package_name/include/package_name	Header files
package_name/launch	Launch files
package_name/lib	Compiled libraries
package_name/msg	Message definitions
package_name/scripts	Executable scripts
package_name/share	Compiled libraries
package_name/src	C/C++ Source code
package_name/src/package_name	Python source code
package_name/srv	Service definitions
package_name/test	Unit tests and test source

6.1 Naming conventions

To ensure interoperability and readability, ROS has a few naming conventions. The full list and rationale is available at `http://www.ros.org/wiki/Naming`, but there's a summary in Table 1. Also read the C++ style guide at `http://www.ros.org/wiki/CppStyleGuide`.

The directory structure of your package is also important. We encourage you to put as few files as possible in the root of your package. Instead, separate subdirectories as in Table 2.

6.2 General structure

We encourage you to always use the same general structure when writing ROS programs. This makes it easier to share and improve eachother's code. Use the following template:

```
#include <ros/ros.h>
#include "my_class.h"
```

[1]Package and topic names should be descriptive – package names must be unique across the entire ROS ecosystem!

[2]Node names can be short because they're always inside a package.

```
int main(int argc, char **argv)
{
  ros::init(argc, argv, "my_node");

  MyClass my_class;
  my_class.spin();

  return 0;
}
```

my_class.h will then look something like this:

```
#include <ros/ros.h>

class MyClass
{
  protected:
    ros::NodeHandle nh_;

  public:
    MyClass() : nh_("~")
    {
      init();
    }

    ~MyClass()
    {
      nh_.shutdown();
    }

    // Advertise services, subscribe to topics, etc.
    void init();

    // Spin, either using ros::spin() or a
    // ros::spinOnce() loop using ros::Rate
    void spin();
};
```

In general, you should try to write most of your code as a ros-independent library, and then use that library in a node. To make your library accessible from other nodes, add the following lines to your manifest.xml:

```
  <export>
      <cpp cflags="-I${prefix}/include"
           lflags="-L${prefix}/lib␣-Wl,-rpath,${prefix}/lib␣-lpackage_name"/>
  </export>
```

6.3 Publisher

Definition

In the `protected` section of your class definition.

```
ros::Publisher my_pub_;
```

Registering

In the `init` function of your class

```
// String type, queue size 10
my_pub_ = nh_.advertise<std_msgs::String>("topic_name", 10);
```

Using

Probably somewhere in the `spin` function of your class

```
std_msgs::String msg;
msg.data = "test"
my_pub_.publish(msg);
```

6.4 Subscriber

Definition

```
ros::Subscriber my_sub_;

void myCallback(const std_msgs::String::ConstPtr& msg);
```

Registering

```
// Queue size 10
my_sub_ =
  nh_.subscribe("topic_name", 10, &MyClass::myCallback, this);
```

Using

```
void MyClass::myCallback(const std_msgs::String::ConstPtr& msg)
{
  puts(msg->data.c_str());
}
```

6.5 Service server

Definition

```
ros::ServiceServer my_service_;

void myCallback(roscpp_tutorials::TwoInts::Request& req,
                roscpp_tutorials::TwoInts::Response& res);
```

Registering

```
  ros::ServiceServer my_service_ =
    nh_.advertiseService("topic_name", &MyClass::myCallback, this);
```

Using

```
void MyClass::myCallback(roscpp_tutorials::TwoInts::Request& req,
                         roscpp_tutorials::TwoInts::Response& res);
{
  res.sum = req.a + req.b;
  return true;
}
```

6.6 Service client

Definition

```
    ros::ServiceClient my_client_;
```

Registering

```
  ros::ServiceClient my_client_ =
    nh_.serviceClient<roscpp_tutorials::TwoInts>("topic_name");
```

Using

```
  roscpp_tutorials::TwoInts srv;
  srv.request.a = 1;
  srv.request.b = 2;
  if (client.call(srv))
    printf("%d", srv.response.sum);
  else
    // Error handling
```

6.7 Parameters

Definition

In a `yaml` file:

```
string: 'foo'
numbers:
  integer: 1234
  float: 1234.5
  boolean: true
```

Registering

In the `launch` (global) or `node` (private) tags of a `roslaunch` file:

```
  <rosparam command="load" file="$(find␣package_name)/param_file.yaml"/>
  <param name="param_name" value="param_value"/>
```

Using

```
std::string param; // int, double, bool
if (!nh_.getParam("param_name", param))
{
  // Error handling
}
```

or

```
nh_.param<std::string>("param_name", param, "default_value");
```