Worksheet 1 – ROS Ecosystem and Parameter Server

The concept of ROS ecosystem is introduced in this worksheet. A ROS workspace will be created and configured, followed by the process of compiling simple ROS nodes. Next, the parameter server is presented together with a few examples to conclude this worksheet.

***NOTE:** In order to better introduce the student with the instructions and commands given during practical classes, it is recommended that the student takes a look on the following guide about UNIX environments: http://www.ee.surrey.ac.uk/Teaching/Unix/. Certainly, this guide will ease the student's interaction with Linux operating systems.*

Create a ROS Workspace

After installing the ROS framework in your system, it's necessary to create and configure a ROS workspace that you'll use during the practical classes to create and develop new ROS nodes. To create a ROS workspace write the following commands in your system's console:

```
$ mkdir -p ~/trsa_ws/src
$ cd ~/trsa_ws/src
$ catkin_init_workspace
$ cd ..
$ catkin_make
$ source ~/trsa_ws/devel/setup.bash
```

***NOTE**: Each line contains a command that should be executed by the user when pressing the ENTER key.*

In order to permanently (every time you run a system terminal) integrate this workspace in your ROS framework, we need to add a specific instruction in the terminal's startup script (file .bashrc) to include (*sourcing*) a ROS *script* that was automatically generated when the ROS workspace was created. Open the file *.bashrc* (the "dot" means that this is a hidden file) through the text editor, *gedit*. To do that write the following command on the system's console:

```
$ gedit ~/.bashrc
```

Using the text editor, add the next instruction to the end of file and save the changes:

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

Restart the system's console to allow the changes to take effect (alternatively (to avoid restart), execute also the previous instruction on the system's console). This integration allows the user to compile and execute his own ROS nodes within the new workspace.

***NOTA:** Several console's command, available from the ROS framework, can be consulted on the following website: http://wiki.ros.org/ROS/Tutorials/NavigatingTheFilesystem. Also, the student should consult the tutorials written by*

the ROS authors in order to obtain additional and complementary information about this framework. These webpages are available at: http://wiki.ros.org/ROS/Tutorials.

Create, compile and execute ROS nodes

The next step consists of creating and compiling a ROS node in the workspace "trsa_ws". First, create a ROS package that will contain all the ROS nodes developed by the user. It's possible and recommended that several packages group different nodes that relate to each other according to a common functionality. For instance:

package navigation # Contains ROS nodes used for the robot's safe navigation

+> ***node*** global_planner # Global path planner to move the robot to a "goal" position

+> ***node*** local_planner # Local path planner that is used to avoid obstacles

+> ***node*** cost_map_builder # Cost map creator used by path planners

package laser_control # Exposes the functionalities of a *laser scanner* in the ROS network

+> ***node*** laser_driver # Logical controller for the physical laser scanner device (i.e., driver)

Let's start by creating a ROS package for this practical class, labeled as "lesson1". Write the following commands on the system's console:

```
$ cd ~/trsa_ws/src
$ catkin_create_pkg t1_package roscpp
```

As you can see, the command line *catkin_create_pkg* requires the package's name should be passed as a parameter, as well as, the package's dependencies (i.e., other packages, libraries, etc.):

catkin_create_pkg <package_name> [dependency-1] [dependency-2] …

Our package is very simple and it will depend only on another package: *roscpp*. The latter is a special package that implements the concepts and the functionalities of the ROS framework in the C++ programming language. In practice, all packages created here will depend on *roscpp*.

After executing *catkin_create_pkg* command, a new folder labeled as "t1_package" is created in the current directory (~/trsa_ws/src). This folder contains a *package.xml* and *CMakeLists.txt* files. Moreover, two sub-folders, "include" and "src", are created inside folder "t1_package". The user should put all source code files inside these two folders. The source code will be our implementation of the ROS nodes that we pretend to create on package "t1_package".

The *package.xml* file contains the metadata about the created package, while the *CMakeLists.txt* file contains all the information required to compile ROS nodes and/or runtime libraries. These two files

are interpreted by the ROS framework's compiler - *catkin*. The latter changes the semantic meaning of an apparent simple folder into a ROS package (also a folder from the point of view of our operating system). Despite being partially filled with information by the *catkin_create_pkg* command line, the files *package.xml* and *CMakeLists.txt* should be edited by the user in order to create ROS nodes from the source code. Additionally, these files provide more functionalities that will be explored during the next practical classe.

As a last note, it is possible to do not specify all the required dependencies of our package inside the command line *catkin_create_pkg*. Nevertheless, it'll be necessary to add them manually later on, in the *package.xml*, at *tag* <build_depend>:

```
<package>
    ...
    <buildtool_depend>catkin</buildtool_depend>
    <build_depend>roscpp</build_depend>
    ...
    <build_depend> dependency_x </build_depend>
    <run_depend> dependency_x </run_depend>
    ...
</package>
```

***NOTE:** For the interested reader, consult the following website in order to obtain more informations regarding the different tags present in a package.xml file: http://wiki.ros.org/ROS/Tutorials/CreatingPackage.*

Now that you have created your first ROS package, let's create and execute a node in the ROS network. Create the file "*node.cpp*" inside the folder "src" and write the following code in the file:

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

int main( int argc, char** argv )
{
    ros::init( argc, argv, "trsa_node" );
    ros::NodeHandle node;
    ros::Rate rate(10);

    while ( ros::ok() )
    {
        ROS_INFO( "Hello World!" );
        rate.sleep();
    }

    return 0;
}
```

Open the *CMakeLists.txt* file with a text editor and add the following lines (or uncomment and modify the original ones):

```
include_directories( include ${catkin_INCLUDE_DIRS} )

# Create a ROS node executable, named as "simple_node", using the source code in the file "node.cpp" and
# by linking to the ROS libraries  " ${catkin_LIBRARIES}".
add_executable( simple_node src/node.cpp )
target_link_libraries( simple_node ${catkin_LIBRARIES} )
```

After saving these modifications on the respective file, compile the package from your workspace by executing the following command:

```
$ catkin_make
```

NOTE: *The student is encouraged to read the following tutorial to assimilate more complementary information at http://wiki.ros.org/ROS/Tutorials/BuildingPackages and verify the advantage of using the catkin tool to compile ROS nodes instead of using directly the underlying standard CMake compiler tool. Moreover, a even more comprehensive guide regarding catkin compiling tool can be found at: http://docs.ros.org/hydro/api/catkin/html/index.html#.*

Before executing any ROS node, the user must first run the *roscore* node. this is will launch in the system the core mechanisms that implement the ROS framework. To do so, write the following command:

```
$ roscore
```

The current system's console will be occupied executing the *roscore* process and won't allow any execution of other processes (except the interrupt signal that can be sent to terminate a process by pressing the Ctrl+C shortcut on the user's keyboard). Now, open another terminal (a new instance of the system's console) and execute the ROS node that you just created:

```
$ rosrun lesson1 simple_node
```

Several messages of "Hello World!" will appear on the new system's console, as an evidence that the ROS node is in fact alive and running in the ROS network. As you can see, these ROS nodes are not so useful at their current development state, serving only to demonstrate the process of creating and executing ros nodes. During the next practical class, we will create ROS nodes that communicate with each other to build more complex behaviours in the ROS network. But for now, let's move on to other functionalities of the ROS framework: the *roslaunch* tool and the Parameter Server.

Parameter Server and Roslaunch

The Parameter Server is a runtime process available to every ROS nodes, and is composed of variables and folders. This server should not be used for data transactions of high bandwidth, and should be mainly used to pass constant values/variables as parameters. When parameters are passed to a node through a command line (i.e., ***rosrun example_node parameter1***) or through a *roslaunch* file, they get stored in the Parameter Server.

With *roscore* node already running on a terminal, write the following command on a new window:

```
$ rosparam list
```

As you can see, there are already a few parameters in the server, such as /rosdistro and /rosversion. You can read the value of a parameter (i.e., **/rosdistro**) by running:

```
$ rosparam get /rosdistro
```

Where you'll get the version of your ROS distribution, *Hydro* or *Indigo*. Now run this commands:

```
$ rosrun turtlesim turtlesim_node
```

(Open a new terminal window)

```
$ rosparam list
```

The **turtle_sim** node you have launched has added new parameters to the server. Try to read one of the background colors. The same way the node has created and written those parameters, you can also edit them. Try it by running:

```
$ rosparam set /background_b 50
```

You may not see any change on the background but that is due to the way **turtle_sim** is programmed. It only loads the background once, during its startup, or if you call its own exposed service to reset the background. Run :

```
$ rosservice call /reset
```

Now the background colour has changed to the RGB parameters you have introduced. You can try to change other parameters to see the results.

***NOTE:** Due to the way **turtle_sim** is programmed, it will overwrite any changes made to Parameter Server. When creating a node, on startup you should check for values on the server before overwriting it.*

Besides setting and reading parameters, you can also save and load parameters to be used. The saved file should be a *.yaml*, and you can save it by using *rosparam dump file_name.yaml*. Start by editing some parameters and then run:

```
$ rosparam dump test1.yaml
```

This will create a file on your home folder with the saved parameters. Open it to see its contents. If on a terminal you run *rosparam get /* without specifying any parameter, it will return all the values, and that is what the dump command is making, and storing it on a file.

To load the parameters, let's start by deleting the existing parameters. To do so, use *rosparam delete* followed by the name of the parameter you want to delete. You can also terminate and re-run the *roscore* process, to reset the Parameters Server to its default values. However, this is not recommended as the others ROS nodes that are running will be left aside from the new ROS network.

Now, to load the Parameter Server file we created, run:

```
$ rosparam load test1.yaml
```

All the parameters are now on the server and you can use it as before.

***NOTE:** As with the **get** command, you can just use a forward slash (/) and delete all the parameters, but this isn't recommended as it will delete other parameters like **/rosdistro** and **/rosversion**, that may be used by other ROS tools (i.e., compiling packages)*.

Roslaunch is a tool for easily launching multiple ROS nodes locally and remotely via SSH, as well as setting parameters on the Parameter Server. Roslaunch uses XML configuration files that specify the parameters to set and nodes to launch, as well as the machines that they should be run on.

http://wiki.ros.org/roslaunch

Using the aforementioned "simple_node" type let's create a .launch example file.

```
<launch>
  <node name="my_first_node" pkg="t1_package" type="simple_node" />
</launch>
```

With this example we are launching a "simple_node" from the "t1_package" and named as "my_first_node".

However, the simple_node is a very simple example that does not need parameters. Consequently, we will use roslaunch to launch a more complex node already installed on ROS framework.

Using the aforementioned *turtlesim* example we will create a *roslaunch* script file that launches the turtlesim node while setting the background colour to RGB (250,125.100).

```
<launch>
  <node name="turtle1" pkg="turtlesim" type="turtlesim_node" />
  <!-- set parameters  -->
 <param name="background_r" value="250" />
 <param name="background_g" value="125" />
  <param name="background_b" value="100" />
</launch>
```

Final Exercise

1. Create a new node, which executable is named as "simple_node_2", in the package "t1_package". This ROS node must write in the terminal the following text: "Yet another node";

2. Create a *roslaunch* script file to automatically launch three ROS nodes: "simple_node_2", "turtlesim_node" and "simple_node" (created in the practical class). Label them as you wish.

The resulting ROS package from this exercise must be zipped into a file named as "T1_<NUMBER1>_<NUMBER2>.zip", where NUMBER is your student's number (i.e., T1_44500_45124.zip). This file must be submitted at the Moodle platform until September 29th, 23:55.