Laboratory class #1: Introduction to ROS

1.- ROS architecture and philosophy

ROS, *Robot Operating System,* is a middleware linking the actual operating system (e.g., Linux Ubuntu) and the robot software. It was originally developed in 2007 at Stanford Artificial Intelligence Laboratory and since 2013 is managed by the Open Source Robotics Foundation, Inc. (https://www.openrobotics.org/). Today it is used by many universities and companies to develop programs for robots, being the standard for robot programming. Basically, it is composed by four different elements:

Plumbing which allows the execution of many programs in parallel and contains a publish-subscribe messaging infrastructure. This infrastructure allows communication between programs. Furthermore, this element provides drivers such that the programs can easily communicate with the available hardware.

Basic set of **Tools** provided by ROS that can be used for simulation, visualization, graphical user interface and data logging.

Capabilities that provide a broad collection of libraries that implement useful robot functionality for control, planning, perception, mapping, manipulation, etc.

Ecosystem meaning that the software is organized in packages such that it is easy to install and use it. Furthermore, ROS provides software distributions and it is supported and improved by a large community, with a strong focus on integration and documentation and tutorials.

ROS main characteristics

Peer to peer. The philosophy of ROS is to have many individual programs running in parallel that communicate over a defined API (ROS messages, services, etc.).

Distributed. The individual programs can be run on multiple computers communicating over the network. It is not important on which computer the programs (called ROS **nodes**) run since the API allows wireless communications.

Multi-lingual. ROS modules can be written in several programming languages (C++, Python, MATLAB, Java, etc.). In this course we will use **C++**.

Light-weight. The new algorithms can be easily integrated in ROS and communicate with other libraries.

Free and open-source. Most ROS software is open-source and free to use.

2. Installing Ubuntu 20.04.5 LTS virtual machine

For the laboratory classes we will use an Ubuntu virtual machine that we prepared for this purpose. It requires the Oracle VM VirtualBox application tool. Depending on your Operating System you should download the corresponding version of the program from: https://www.virtualbox.org/wiki/Downloads (version 6.1). Please make sure that you install the VirtualBox Extension Pack as well as the main program. It is also possible to get an error message when you try to run the virtual machine for the first time (i.e., VT-x is disabled in the BIOS for all CPU modes). In this case, you should enable the visualization in bios[1].

Once you start Oracle VM VirtualBox, you need to download the virtual machine using the following link: https://unizares-my.sharepoint.com/:u:/g/personal/riazuelo_unizar_es/Ed-2ltNLKYFMn5Yxen42_MEBtH83RsQUvN44tVdo7h4neA?e=X0J3hf

The next step is to use the 'Import' option of Oracle VM VirtualBox and choose the file you have already downloaded. This operation may take several minutes. When complete, on the left side you should see the new Ubuntu virtual machine added that can be started. When you are prompted in Ubuntu, log with the following credentials:

Username: 'arob'

Password: 'unizar'

3.- Ubuntu VM overview

Once the Ubuntu Virtual Machine is open, in the taskbar on the left you have shortcuts to the Firefox Web Browser, a shortcut to the files on your home and a shortcut to the Terminator (a console that we will use). Notice that this console application allows you to split the console vertically and horizontally in order to have several consoles open in the same window. This will be something useful when we need to run different programs at the same time.

If you experience problems when running the virtual machine (the window becomes black) you can try to **increase the video memory**.

4.- Catkin workspace and build system

One of the important elements of ROS is the workspace environment. The default workspace installed in the virtual machine can be loaded by executing the following command line in a console:

```
$ source /opt/ros/noetic/setup.bash
```

Notice that noetic distribution of ROS (http://wiki.ros.org/noetic) is installed. You need to specify the workspace every time you open a new terminal. If you want to do this by default, you can add the previous line in the .bashrc file by using the following command:

```
$ echo  >> ~/.bashrc
$ echo source /opt/ros/noetic/setup.bash >> ~/.bashrc
$ source ~/.bashrc
```

If you want to check that the workspace is well configured, use the following command:

```
$ echo $ROS_PACKAGE_PATH
```

If the result is '*/opt/ros/noetic/share*' the default workspace was successfully associated.

This is the default workspace where the ROS is installed but in order to create our own programs we need to create our own workspace that can be used to modify, build, and install new packages. In order to create the working workspace, use the following commands:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
```

[1] This is something different for every computer, so you will have to check how to access the BIOS of your computer and search in the menu to enable virtualization

```
$ catkin build
```

Looking in the ‘`catkin_ws`’ directory (use ‘`ls`’ linux command to see the content of a folder) two new folders appear, namely '`build`' and '`devel`' (together with the ‘`src`’ created by ‘`mkdir`’ command). These folders are used for:

`src`: containing the source code of your packages. It is also possible to clone folders, for example a git repository and then edit it.

`build`: is a folder containing cache information and other files used by the compiler (CMake).

`devel`: is the folder where the executable files will be placed before being installed.

The `build` and `devel` folders shouldn’t be changed. If you need to clean them use the following command:

```
$ catkin clean
```

Inside the '`devel`' folder (use the linux command ‘`cd devel`’ to change the directory), several setup.*sh files exist. Sourcing any of these files will overlay this workspace on top of your environment (for more information see the general catkin documentation here: http://wiki.ros.org/catkin). Before continuing to source your new setup.*sh file by using the following command:

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

This `setup.bash` file is setting some environmental variables of the ROS environment. If you want to load it automatically in your terminal, you can add the line to your .bashrc file

After programming your own package in the ‘`src`’ folder, in order to build it, you should use the following two commands:

```
$ cd ~/catkin_ws
$ catkin build package_name
```

IMPORTANT. After you build a new package you should update your environment by using:

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

5.- ROS master, nodes and topics

One important element is the ROS master, which manages the communication between nodes. This is done after the registration of the node with the master, ensuring that all the nodes communicate with each other. A ROS node is an executable program which is individually compiled and can also be run individually. The ROS workspaces are organized in packages which contain a set of nodes.

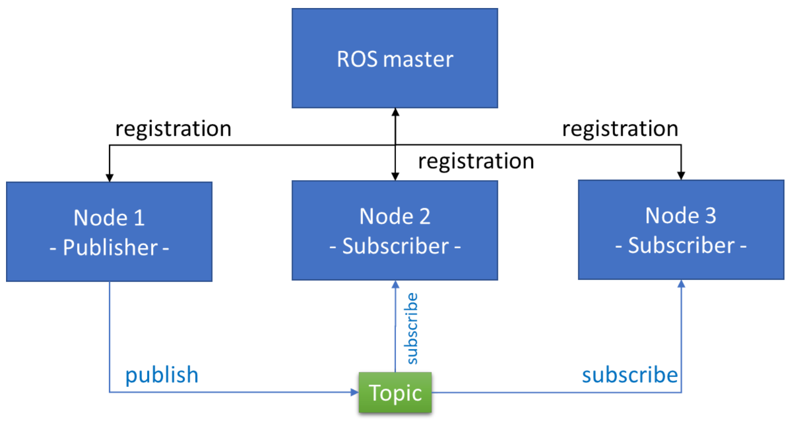

Figure 1: Example of three nodes registered with ROS master, one being publisher and the other two subscribers.

The ROS master can be initialized in a console using the command:

```
$ roscore
```

While to run a node we should use

```
$ rosrun package_name node_name
```

To see the active nodes use

```
$ rosnode list
```

And to retrieve specific information about a node,

```
$ rosnode info node_name
```

Nodes can communicate between them using **ROS topics**. Each node can **publish** or **subscribe** to one or more topics, being possible to have multiple subscribers to the same topic. Essentially, a topic is a stream of messages of the same type. In the example of Figure 1, once the three nodes have been registered with the ROS master, Node 1 advertises the topic, meaning that it will publish messages there. Nodes 2 and 3 subscribe to the topic and the messages will flow from node 1 to nodes 2 and 3. The ROS master is the one that makes sure that these connections happen.

To see the active topics use

```
$ rostopic list
```

To subscribe and print the contents of a topic use

```
$ rostopic echo /topic
```

To retrieve specific information about a topic use

```
$ rostopic info /topic
```

Example 1.

Let's start a simple example from the ROS tutorial to see all this information.

1. Start a console (Terminator) from the dock menu and let's start a roscore by using the command '`roscore`'. If you get an error message you should define the workspace context by using the following two instructions: '`source /opt/ros/noetic/setup.bash`' and '`source`

`/home/arob/catkin_ws/devel/setup.bash`'. If everything is going fine you should see the following message: '`started core service [/rosout]`'

2. Split horizontally the console by choosing the corresponding option after right-clicking on the console window.
3. In this new working space, you should define the working context again if the `.bashrc` file has not been updated as explained before.
4. Run a talker demo node with '`rosrun roscpp_tutorials talker`'. The message 'hello world nn' should appear on your screen.
5. Let's split vertically one of the previous consoles by just right-clicking on the console and choosing the corresponding option.
6. In the new console, execute '`rosnode list`' in order to see the list of nodes that are currently executed. You should be able to see the '`/talker`' node that has been started in step 4.
7. More information on the node can be obtained using '`rosnode info /talker`'. Notice that this node is publishing on the '`/chatter`' topic.
8. More information on the '`/chatter`' topic can be obtained by using '`rostopic info /chatter`'. Notice that the topic has one publisher ('`/talker`' node) and no subscriber.
9. Checking the type of the topic can be done by using the following command: '`rostopic type /chatter`' and showing the message content of the topic using '`rostopic echo /chatter`'. Stop it by using CTRL+C.
10. Finally, the frequency of the topic can be obtained by using '`rostopic hz /chatter`'. Stop showing the information by using CTRL+C.
11. Split vertically the other console and let's start a listener node by using '`rosrun roscpp_tutorials listener`'.
12. In the previous console, if you check again the list of nodes by using '`rosnode list`' you should be able to see the new node '`/listener`' that is executing.
13. Checking again the information on the '`/chatter`' topic by using '`rostopic info /chatter`' you can see that the '`/listener`' node is a subscriber of the '`/chatter`' topic.
14. Close the '`/talker`' node in the second console by using CTRL+C.
15. Publish manually a message using: `rostopic pub /chatter std_msgs/String "data: 'UNIZAR Autonomous Robots'"`. This message should be listened to by the '`/listener`' node and displayed on the last created console.

Example 2.

Let's see a little bit more complicated example, learning also about git repositories and launch files. First, we will go to a GitHub website and we will clone a git repository. Git is a typical tool when working with software. Close all terminals that you used in Example 1 and open a new one before executing the following commands:

```
$ git clone https://github.com/luisriazuelo/arob_lab1.git
~/catkin_ws/src/turtle_comunic
```

In general, if you go to the GitHub website (e.g., https://github.com/luisriazuelo/arob_lab1.git), and push the green button "Code" you can either download a .zip file containing the repository or you can copy the link address used after that for '`git clone`' command. The next step after downloading the package is to compile it, following the following commands:

```
$ cd ~/catkin_ws
$ catkin build turtle_comunic[2]
$ source ~/catkin_ws/devel/setup.bash
$ roslaunch turtle_comunic turtlesim2.launch
```

Notice that now we execute nodes using a **launch file** that can be found in the turtle_comunic folder named '`turtlesim2.launch`' written in XML language. If you want to launch the simple simulation in Example 1, use the following command:

```
$ roslaunch roscpp_tutorials talker_listener.launch
```

The `talker_listener.launch` file has the following structure

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

The nodes in the XML file have the following interpretation,

launch: Root element of the launch file

node: Each <node> tag specifies a node to be launched

name: Name of the node (free to choose)

pkg: Package containing the node

type: Type of the node, there must be a corresponding executable with the same name

output: Specifies where to output log messages (screen: console, log: log file)

Is it possible to create reusable launch files with the `<arg> tag` option that is very similar to passing a parameter to a function (see more details here: http://wiki.ros.org/roslaunch/XML/arg).

6.- Robot simulator

In order to test the different algorithms, there exist two robot simulators: Stage (https://github.com/rtv/Stage) which is a 2D simulator and a more advanced 3D simulator called Gazebo (http://gazebosim.org/). Gazebo is more mature and is under active development but requires a powerful computer. An example of a gazebo scene is here: https://youtu.be/oP39r2PQ4Vk

In this course we will use Stage simulator that can be started with an existing world file by using:

```
$ roscore
$ rosrun stage_ros stageros $(rospack find
stage_ros)/world/willow-erratic.world
```

[2] **Important**: the name of the package is not necessarily the same as the name of the folder. See package.xml for info.

Click on the stage window and press R to see the perspective view. Notice that the previous command requires 'roscore' to be running. Remember that roslaunch automatically starts 'roscore', hence in the case of starting Gazebo with the launch file it is not necessary to start roscore.

7. ROS Packages

ROS software is organized into packages, which can contain source code, launch files, configuration files, message definitions, data, and documentation. A package that requires other packages (e.g., message definitions), declares these as dependencies. In order to create a package, use:

```
$ catkin_create_pkg package_name {dependencies}
```

An XML file called package.xml defines the properties of the package (see http://wiki.ros.org/catkin/package.xml for more details). The following set of tags are necessary within <package> </package> XML tag.

```
    <name> - The name of the package

    <version> - The version number of the package (required to be 3 dot-separated integers)

    <description> - A description of the package contents

    <maintainer> - The name of the person(s) that is/are maintaining the package

    <license> - The software license(s) (e.g. GPL, BSD, ASL) under which the code is released.

    <buildtool_depend> - specify build system tools used to build the package. In general, the only build tool needed is catkin.

    <depend> - specify which packages are needed to build and run the package.
```

An example of an package.xml file is the following:

```
<?xml version="1.0"?>

<package format="2">

<name>ros_package_template</name>

<version>0.1.1</version>

<description>A template for ROS packages</description>

<maintainer email="riazuelo@unizar.es">Luis Riazuelo</maintainer>

<license>BSD</license>

<url type="website">https://github.com/luisriazuelo/arob_lab1.git</url>

<author email="riazuelo@unizar.es">Luis Riazuelo</author>

<buildtool_depend>catkin</buildtool_depend>

<depend>roscpp</depend>
```

```
        <depend>sensor_msgs</depend>

        </package>
```

The file CMakeLists.txt is the input to the CMake build system for building software packages. Your CMakeLists.txt file MUST follow this format otherwise your packages will not build correctly. The order in the configuration DOES count.

- Required CMake Version (cmake_minimum_required)
- Package Name (project())
- Find other CMake/Catkin packages needed for build (find_package())
- Enable Python module support (catkin_python_setup())
- Message/Service/Action Generators (add_message_files(), add_service_files(), add_action_files())
- Invoke message/service/action generation (generate_messages())
- Specify package build info export (catkin_package())
- Libraries/Executables to build (add_library()/add_executable()/target_link_libraries())
- Tests to build (catkin_add_gtest())
- Install rules (install())

8. Visual Studio Code

For building and compiling ROS programs it is possible to use many IDEs, during this course we propose you to use Visual Studio Code. In order to generate your codes, first you need to add your catkin workspace src folder to the Visual Studio Code. Start the Visual Studio Code (in the provided Virtual machine you can find a shortcut in the taskbar on the left) or open a terminal and execute “code”. Use the option “File -> Open folder ” and choose the src folder from your catkin workspace. If everything is fine, you should be able to see in the explorer window all your ROS packages.

From Visual Studio you can clone git repositories, create ROS packages, compile and execute ROS nodes. This can be done by using the command palette open by using the “View -> Command Palette” option or by using the shortcut

Ctrl + Shift + P

In the command palette, the following commands may be utils for this lab:

- Git: Clone - to clone a git repository from a given URL
- ROS: Create Catkin Package - to create a new catkin package
- ROS: Create Terminal - to open a terminal and execute ROS commands

Also it is important to have the headers files of the ROS distribution. In order to do this, open the C/C++ configuration by using Ctrl + Shift + P and choose “C/C++: Edit configurations (UI)”. In the Include path option introduce (if they are nor already there):

```
${workspaceFolder}/**

/opt/ros/noetic/include/

/usr/include/
```

While in the “C++ standard” option put “c++14” required to compile ROS nodes.

Exercise.

Clone the teleop_twist_keyboard_cpp package in your workspace in the catking_ws/src folder (the corresponding link to the ROS package is available on the ROS wiki: http://wiki.ros.org/teleop_twist_keyboard_cpp).

Second, create a new package from Visual Studio without any dependency called first_package, add a src/launch folder and create a launch file to start the stage simulator with the simple.world (the world file can be copied from the AROB_lab1 repository, folder world) and the teleop_twist_keyboard package. This last package should be compiled from source using the corresponding GitHub repository.

Exercise.

Create a new package called second_package depending on the following packages: std_msgs rospy roscpp. Add inside an src folder and create a new file with the name helloWorld.cpp.

Copy and paste the following c++ code (you can understand the code by reading the comments).

```
#include <ros/ros.h>
int main(int argc, char** argv)//includes ROS main header file
{
    ros::init(argc, argv, "helloWorld");//should be called before calling other ROS
    functions
    ros::NodeHandle nodeHandle;          //The node handle is the access point for communications
                                         // with the ROS system (topics, services, parameters)
    ros::Rate loopRate(10);          //ros::Rate is a helper class to run loops at a desired frequency
    unsigned int count = 0;
    while (ros::ok()) {   //ros::ok() checks if a node should continue running Returns false
                          //if Ctrl + C is received or ros::shutdown() has been called
        ROS_INFO_STREAM("Hello World " << count);//ROS_INFO() logs messages to the
                                                 // filesystem
        ros::spinOnce();//ros::spinOnce() processes incoming messages  via callbacks
        loopRate.sleep();
        count++;
    }
return 0;
}
```

Some additional configurations are necessary in order to compile the new node and execute it. Add/uncomment these lines in the CMakeLists.txt file from [Source directory] folder.

```
catkin_package(
 INCLUDE_DIRS include
 LIBRARIES second_package
#  CATKIN_DEPENDS roscpp rospy std_msgs
#  DEPENDS system_lib
)
include_directories(include ${catkin_INCLUDE_DIRS})
add_executable(helloWorld src/helloWorld.cpp)
target_link_libraries(helloWorld ${catkin_LIBRARIES})
```

Compile the project in the terminal available in Visual Studio Code and execute the ROS node created.

In a c++ file, it is possible to create different node handle as follows:

- Default (public) node handle: nh_ = ros::NodeHandle(); // resolved as /namespace/topic
- Private node handle: nh_private_ = ros::NodeHandle("~"); // resolved as /namespace/node/topic
- Namespaced node handle: nh_eth_ = ros::NodeHandle("eth"); // resolved as /namespace/eth/topic
- Global node handle: nh_global_ = ros::NodeHandle("/"); // resolved as /topic

Considering as example the following two files:

- https://raw.github.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/talker/talker.cpp
- https://raw.github.com/ros/ros_tutorials/kinetic-devel/roscpp_tutorials/listener/listener.cpp

Described on the ROS wiki (https://wiki.ros.org/roscpp_tutorials/Tutorials/WritingPublisherSubscriber), add two nodes to the previously created package second_pack, one node to publish the message “Autonomous Robot Course” and another node that should reply to the message before with “Hello, I am a student!”.

9. References

- ROS tutorial: http://wiki.ros.org/
- ROS /noetic navigation: http://wiki.ros.org/navigation