UNIVERSIDAD PABLO DE OLAVIDE SEVILLA
ESCUELA POLITÉCNICA SUPERIOR

Robótica y Visión Artificial
Ingeniería Informática en Sistemas de Información

EPD 1: ROS introduction

Objectives

- First contact with ROS (Robot Operating System) and mobile robot control.

Preliminaries

1 Materials

In this and the remaining EPDs we will use the following tools:

- **ROS (Robot Operating System)**: http://www.ros.org
    - In particular, we will use the ROS **Noetic** distribution: http://wiki.ros.org/noetic/Installation/Ubuntu
    - For robot simulation we will use the **Gazebo Simulator**, which is included in ROS.

In order to use the ROS Noetic distribution in different machines (for instance, we have Ubuntu 22.04 in the computers of the Lab), we will provide a container (Docker in our case) with ROS Noetic.

NOTE1: The Noetic distribution works under Ubuntu version 20, so that If you do not want to use the container, you must use the indicated OS version.
NOTE2: ROS can be installed and used also in Windows (http://wiki.ros.org/noetic/Installation/Windows) but we do not provide support for that. You can use Windows under you own responsibility.

2 Docker installation

Docker provides the ability to package and run an application in a loosely isolated environment called a **container**. It is an open platform for developing, shipping, and running applications, and enables you to separate your applications from your infrastructure so you can deliver software quickly (https://www.docker.com/).

To install docker (in Ubuntu) we will follow the guide in: https://docs.docker.com/engine/install/ubuntu/. Preferably, we will use the installation method indicated in section “install using the repository”. Then, install the docker engine. Finally, do not forget to do the Linux post-install steps (https://docs.docker.com/engine/install/linux-postinstall/).

(Alternatively, If you want to use it in Windows follow the instructions here: https://docs.docker.com/desktop/install/windows-install/)

3 Use a dockerfile with the packages we need to build the container

We provide a base dockerfile to build our container. You can find this file in the EPD1 folder in the Aula Virtual of the subject. First, create a directory in your */home* directory called “rva”, and download the dockerfile to it.

```
$ mkdir rva
$ cd rva
```

Using this dockerfile, we will create a container in which ROS Noetic and the ROS packages for the robot Turtlebot3 will be installed. The content of the dockerfile is the following:

```
FROM osrf/ros:noetic-desktop-full

SHELL ["/bin/bash", "-c"]


# required ROS packages
RUN sudo apt update && apt install -y ros-noetic-joy ros-noetic-teleop-twist-joy \
    ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
    ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
    ros-noetic-rosserial-python ros-noetic-rosserial-client \
    ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
    ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
    ros-noetic-compressed-image-transport ros-noetic-rqt* ros-noetic-rviz \
    ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-markers

# ROS packages for Turtlebot3 robot
RUN sudo apt install -y ros-noetic-dynamixel-sdk \
    ros-noetic-turtlebot3-*

# we also install Git, just in case
RUN sudo apt install -y git

# MESA drivers for hardware acceleration graphics (Gazebo and RViz)
RUN sudo apt -y install libgl1-mesa-glx libgl1-mesa-dri && \
    rm -rf /var/lib/apt/lists/*

# we source the ROS installation
RUN echo “source /opt/ros/noetic/setup.bash” » ~/.bashrc
RUN echo “export TURTLEBOT3_MODEL=burger” » ~/.bashrc

# the work directory inside our container
WORKDIR "/home/rva_container"
```

IMPORTANT: we will use different graphical interfaces like the Gazebo simulator. These programs use hardware acceleration provided by the graphics cards of our computers. In the provided dockerfile, we have installed MESA drivers which are a basic graphics drivers. They should work (hopefully) in all our computers.
Anyway, if you have a Nvidia graphics card and want to use it, or you are having problems, you can check the following options to use different graphics acceleration hardware:
http://wiki.ros.org/docker/Tutorials (section 6)
http://wiki.ros.org/docker/Tutorials/Hardware%20Acceleration

To build the container using the dockerfile execute the following command from the directory in which the dockerfile is:

```
$ docker build -t rva_container .
```

The name of our container will be rva_container.
The dot indicates that the dockerfile is in the current directory.

4 Run the container with a bind mount

Now, we will create in our computer another directory. We will use the directory as a shared space with the container. Create another directory i**nside your rva directory**:

```
$ mkdir rva_exchange
```

Next, to run our container we can execute the following command in the terminal, but first we must change the text indicated in red color. We must indicate the user of our Ubuntu sesion.

```
 $ xhost +local:docker
 $ docker run -it \
      --env="DISPLAY=$DISPLAY" \
      --env="QT_X11_NO_MITSHM=1" \
      -volume="/tmp/.X11-unix:/tmp/.X11-unix"
      -name rva_container \
      --net=host --privileged \
      --mount
type=bind,source=/home/<your_user>/rva/rva_exchange,target=/home/rva_contai
ner/rva_exchange rva_container \
      bash

docker rm rva_container
```

With the command --mount we create a link between a directory in our host and a directory in the container. We will use this space to store our workspace with our ROS packages. This way, we do not need to push changes in the container image, and we can pass our code easily.
***IMPORTANT: The idea is to use the container to compile and to execute our ROS code, but we can edit and modify our code from our computer.**

Instead of writing or copying the previous command in the terminal each time we want to run the container, we also provide a bash script (fille “run_container.bash” in the EPD1 folder in “Aula Virtual”).
We must download it in our **rva directory**, and change its execution permissions:

```
 $ chmod a+x run_container.bash
```

Edit the file to put the correct user, and then you can execute it:

```
 $ ./run_container.bash
```

If everything was right, we are now inside our container.

5 Let’s run a Turtlebot simulation

At this point, we have a functional container with all the software we need. We can now try to launch a simulation example of the Turtlebot robot.

In your **container terminal** you should be able to run:

```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

The gazebo simulator will be launched. You will see the Turtlebot robot in the middle of a empty world.

Let’s try to move the robot! To do that we need to open another terminal in our container. To do that, type **in a terminal outside our container**:

```
 $ docker exec -it rva_container bash
```

Once we are **in our container**, you must source the ros environment again:

```
 $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

This latter will run a ROS node that allow us to send different velocity commands (linear and angular velocity) to the robot by using the keyboard keys W, A, S, D, and X.

6 The robot simulator Gazebo

For the development of ROS nos we will use Gazebo simulator. It allows us to safely test our programs for the Turtlebot robot in a simulated environment. After testing succesfully in simulation, we can try our programs in the real robot.

Gazebo implements a physical simulator, and can emulate sensors and motions of the robot. This simulator is integrated into ROS by creating several nodes and topics on which you can read the robot sensors or publish commands to the simulated robot. In the case of the simulator, we can now write code that works the same with the simulator and the real robot.

Further information:
https://gazebosim.org/home
https://emanual.robotis.com/docs/en/platform/turtlebot3/simulation/

7 The visualization tool Rviz

ROS has a visual tool to see the sensor data published into a ROS system, among other information. This tool is called RViz, and can be launched by typing:

```
$ rosrun rviz rviz
```

We will explain along the course how you can show information into this visualization tool. We will use it to see how the Gazebo simulator is able to simulate the robot as well as the robot sensors (like Kinect camera or lidars).

This visualization tool, RViz, is just this. It is not a simulator. It only displays information. The same tool can be used with the real and the simulated robot.

Example of a ROS Node

The main objective of this section is to show an initial ROS node that commands a simulated Turtlebot robot.

1 Create a ROS workspace

The first step is to create a ROS workspace **in our container**. We will create it in our exchange directory that we have just created inside our container (*/home/rva_container/rva_exchange*). Remind that this directory is linked to the directory */home/<your_user>/rva/rva_exchange* in your computer. The key idea is that the source code of all the packages that you develop for ROS should be contained in the "src" folder of a workspace. Then, you can compile them, generating the "build" and "devel" folders. Further information: http://wiki.ros.org/catkin/Tutorials/create_a_workspace

From the container, let's create the workspace and compile it:

```
$ cd /home/rva_container/rva_exchange
$ mkdir -p rva_ws/src
$ cd rva_ws
$ catkin_make
```

Catkin is the compilation tool employed in ROS Noetic to build our nodes.

2 Create a new ROS package

A package is the main software organization unit in ROS. Each package can contain libraries, executables, scripts, etc. To learn how to create a package, we must follow the next tutorial:
http://wiki.ros.org/ROS/Tutorials/CreatingPackage

First, we will create a package called *epd1* in our workspace. This package will use other ros packages and libraries (called dependencies). In our case, we need rospy (to be able to program ROS nodes using Python), TF and geometry_msgs.

```
$ cd %YOUR_CATKIN_WORKSPACE_HOME%/src
$ catkin_create_pkg epd1 rospy tf geometry_msgs
```

Build the workspace, and source your workspace to detect the new nodes:

```
$ cd %YOUR_CATKIN_WORKSPACE_HOME%/
$ catkin_make
$ source devel/setup.bash
```

Have a look into the new package by commanding **roscd epd1**

The two mandatory files of a package are the following:

- package.xml
- CMakeLists.txt

Package file

The package file is used to list the package name, version numbers, authors, maintainers, and **the dependencies of our code**. We can have internal dependencies to other ROS packages, and/or external dependencies to third-party libraries. This file is used by the ROS building system to install and/or compile these dependencies.

Open the file package.xml

First, the information about the name, etc, is indicated in the following tags:

```
<package>
  <name>foo_core</name>
  <version>1.2.4</version>
  <description>
  This package provides foo capability.
  </description>
  <maintainer email="ivana@willowgarage.com">Ivana Bildbotz</maintainer>
  <license>BSD</license>
</package>
```

Then, the dependencies are listed. Packages can have four types of dependencies:

- **Build Tool Dependencies** specify build system tools which this package needs to build itself. Typically the only build tool needed is **catkin**. They are indicated as:

```
<buildtool_depend>catkin</buildtool_depend>
```

- **Build Dependencies** specify which other packages are needed to build this package. This is the case when any file from these packages is required at build time.

```
<build_depend>”package name”</build_depend>
```

- **Run Dependencies** specify which packages are needed to run code in this package, or build libraries against this package.

```
<run_depend>”package name”</run_depend>
```

- **Test Dependencies** specify only *additional* dependencies for unit tests. They should never duplicate any dependencies already mentioned as build or run dependencies.

```
<test_depend>”package name”</test_depend>
```

CMakeLists file

In CMakeLists.txt, we indicate the files to compile and the libraries to link. ROS uses **catkin** (http://wiki.ros.org/catkin/conceptual_overview) as building system, which itself employs **CMake** (http://www.cmake.org). It is out of the scope of this course to explain in detail CMake. We will provide you with the adequate information when needed.

Now, we will add a first node to our package. Inside our new ROS package epd1, we will create a folder called “scripts”.

```
$ cd /home/rva_container/rva_exchange/rva_ws/src/epd1
$ mkdir scripts && cd scripts
```

Now, we can download an example code from the Aula Virtual. It is the file “*control.py*” inside the EPD1 folder.
From outside of the container, we can move the downloaded Python file to the new scripts directory.
Finally, we can execute our node from our container:

```
$ rosrun epd1 control.py
```

The functioning and the code will be explained in the class.

Exercises

1 A node controlling Turtlebot to reach a Goal position

In this exercise, we will extend the example node for controlling the Turtlebot in order to reach a goal position given in the odometry frame (odom).
To do so, we will need what we have learnt in Topics 1 and 2.

1.1 Coordinate frames

The simulated robot uses many coordinate frames to operate. All of them are published through the ROS /tf topic, and can be used by means of the TF library.

Among them, we will be using two main frames:

- /odom, which is the world frame. The origin (0,0,0) is set at the first position of the robot when Gazebo is launched
- /base_footprint (or /base_link) is a local frame attached to the robot (that is, it moves with the robot), with the X axis aiming to the direction of motion of the robot.

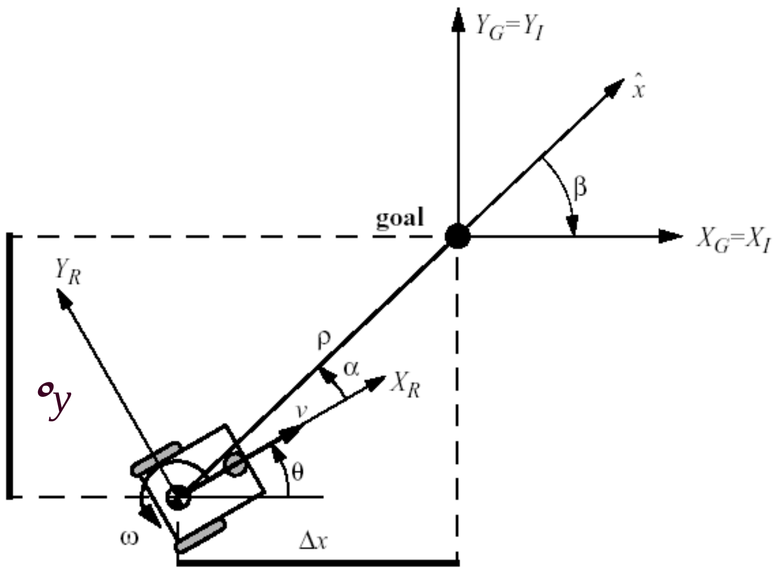

Figure 1: Coordinate frames. The world frame /odom (denoted as X_G and Y_G) and the robot local frame /base_link (denoted as X_R and Y_R)

Our objective will be to reach goals expressed in the world frame, in odom. One way to do it is to transform those points to the local frame, and then reasoning about the linear (v) and angular (ω) velocities that we have to command the robot.

For the conversion between connected coordinates system, we use the TF library. In particular, we may employ the TransformListener class. It allows to ask for the transformation between any two given coordinate frames.

For instance, the following code transforms a point expressed in the odom frame to the base_link frame:

```
import tf

[…]

goal = PointStamped();
base_goal = PointStamped();
goal.header.frame_id = "odom"

[…]

listener = tf.TransformListener()
listener.transformPoint(‘base_link’, goal, base_goal);
```

During the class the code will be explained. The main objective is to control the robot in order to reach a goal. The coordinates of the goal are passed as parameters to the node. Then, you have to complete the code for, given the goal coordinates and the robot coordinates, control the robot to reach the goal.

We will use a **proportional controller**. The idea is that the velocities (linear and angular) to be commanded to the robot will be proportional to the error between the robot position and the goal position. In particular, to the distance (ρ in Fig. 1) and the angle (α in Fig. 1) between the robot heading and the goal.

Download the script “controlGoal.py” from the EPD1 directory in the Aula Virtual, and put it in the epd1 package of your workspace. You have to understand the code and complete the function **command**.

Complete the function command so that the robot reaches the goal