Intelligent Systems and Robotics:

ROS Noetic

Rayane KADEM

STUDENT ID: c7264434

Table of Contents

Table of Contents 1

Introduction 2

ROS Noetic 2

Installation and setup 2

installation 2

Build a workspace : Catkin 4

Packages in ROS 4

ROS Nodes 5

ROS publisher and subscriber 5

ROS parameter server 6

ROS Lunch files 7

ROS Bag files 8

ROS service: Application for image call 8

ROS service (server-client) 10

ROS Action 11

ROS Gazebo 12

Important Notes 14

Conclusion 14

Introduction

This report represents the practical part of the robotic and intelligent systems ISR module, in which we will present the ROS framework, explain the installation and setup of its noetic version,along with details on the structure and applications performed during the semester.

ROS Noetic

Robot Operating System (ROS) is a flexible platform for writing robot software's codes. It gathers a set of tools, libraries and conventions, to create an environment that eases the task of creating complex robot behavior. The programming language dependencies of ROS can be Python, C++ or Lisp. In all cases, it is an oriented object programming language (OOP).
ROS noetic is the thirteenth distribution released of ROS. This version is only compatible with Ubuntu Linux Operating System. There are some experimental versions of the framework for Windows but they are not supported yet.

Installation and setup

installation

Before the installation of ROS Noetic we created a Virtual Machine and installed the 64 bits Ubuntu 20.04 Linux OS. The VM was of 2GB RAM and 20 GB disk.

- **First a configuration of** Ubuntu repositories is needed to allow "restricted," "universe," and "multiverse" repositories. This configuration is presented in figure 1.

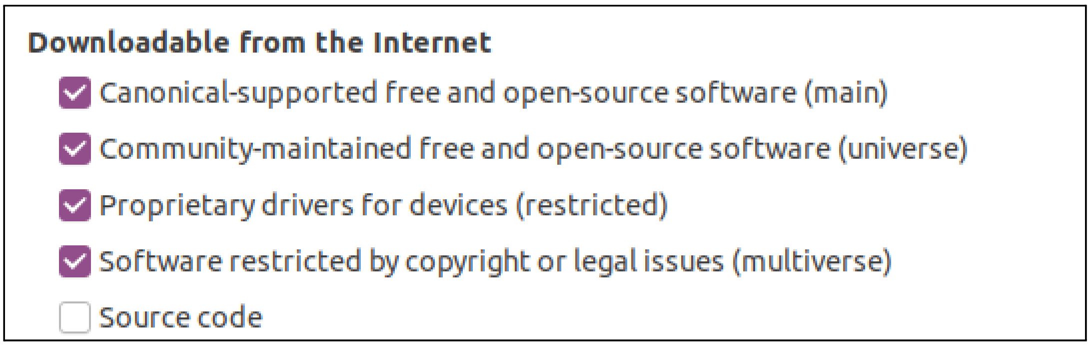


Figure 1- Configuration of Ubuntu repositories

- Setup the source list with the following command so that the VM accepts software form packages.ros.org.

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

- Setup the Keys.

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

- Make sure your packages are updated.

```
sudo apt update
```

- Install the full desktop version of ROS Noetic.

```
sudo apt install ros-noetic-desktop-full
```

- The next step is to source the scripts in every bash terminal where ROS is used in.

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

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

```
source ~/.bashrc
```

- Now ROS is well installed, Next we need to run ROS packages with the following command.

```
sudo apt install python3-rosdep python3-rosinstall
python3-rosinstall-generator python3-wstool build-essential
```

- Initialize ROS dependencies (rosdep)

```
sudo apt install python3-rosdep
```

```
sudo rosdep init
```

```
rosdep update
```

- Finally to start ROS on the terminal: we type `roscore` (figure 2).

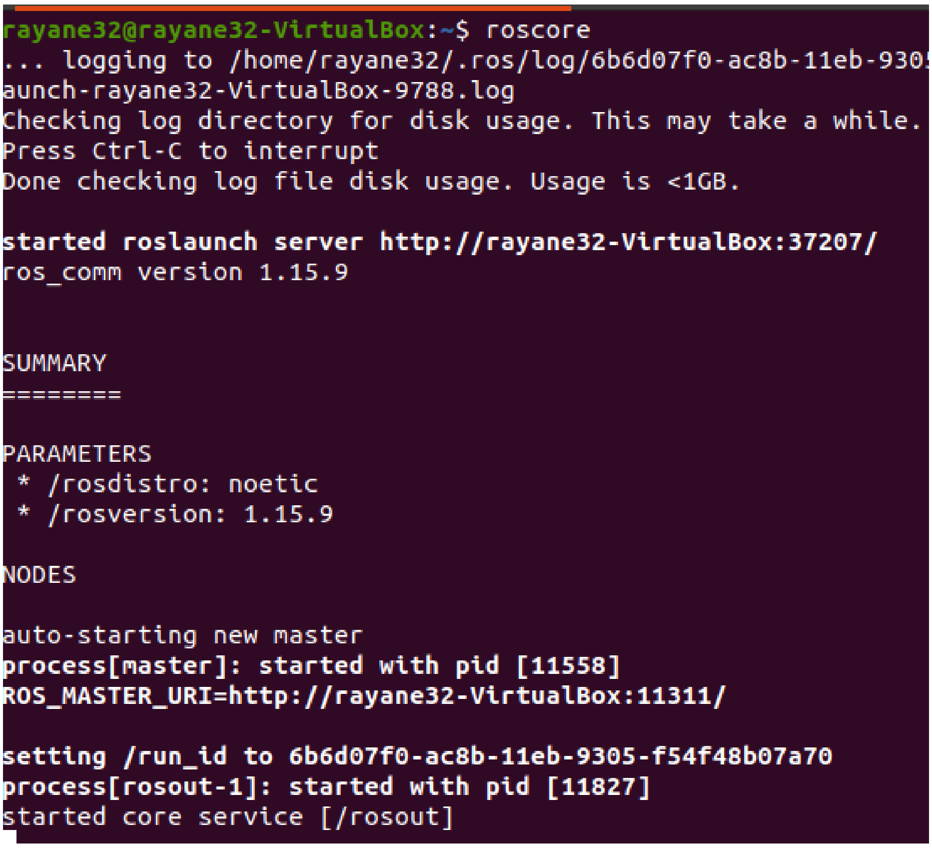


Figure 2- Start ROS with roscore

Build a workspace : Catkin

catkin is the main ROS **build** system. It combines **CMakemacros** and **Python** scripts to provide some functionality in addition to the normal CMakes workflow. Catkins workflow is very similar to ordinary CMakes but adds support for automatic find package infrastructure and building multiple, dependent projects at the same time.

To create the workspace, we create a folder named `catkin_ws` with the command `mkdir catkin_ws`,inside it we create the folder `src`. Then we pass the command `catkin_make.`

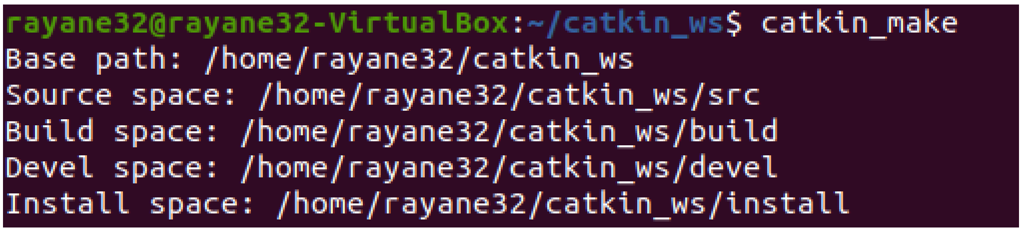


Figure 3- Demonstration of catkin

Everytime a catkin is made, all dependencies are updated on the `Cmakelists.txt` file placed in the `src` folder. A sourcing is needed before making a new catkin with the command `source Devel/setup.bash.`

It is also recommended to add the sourcing command on the `/.bashsrc` file.

Packages in ROS

ROS software is organized in packages in order to simplify the use and reuse of ROS functionalities. A package can include ROS nodes, datasets, configuration files, libraries, or anything that can build a useful model.

- To create a package with cpp, py and std_msg dependencies:

```
catkin_create_pkg package_name roscpp rospy std_msgs
```

- ROS also allows the smooth transition between packages and the interaction between them. Figure 4 is an example of the interaction of 3 ROS packages.

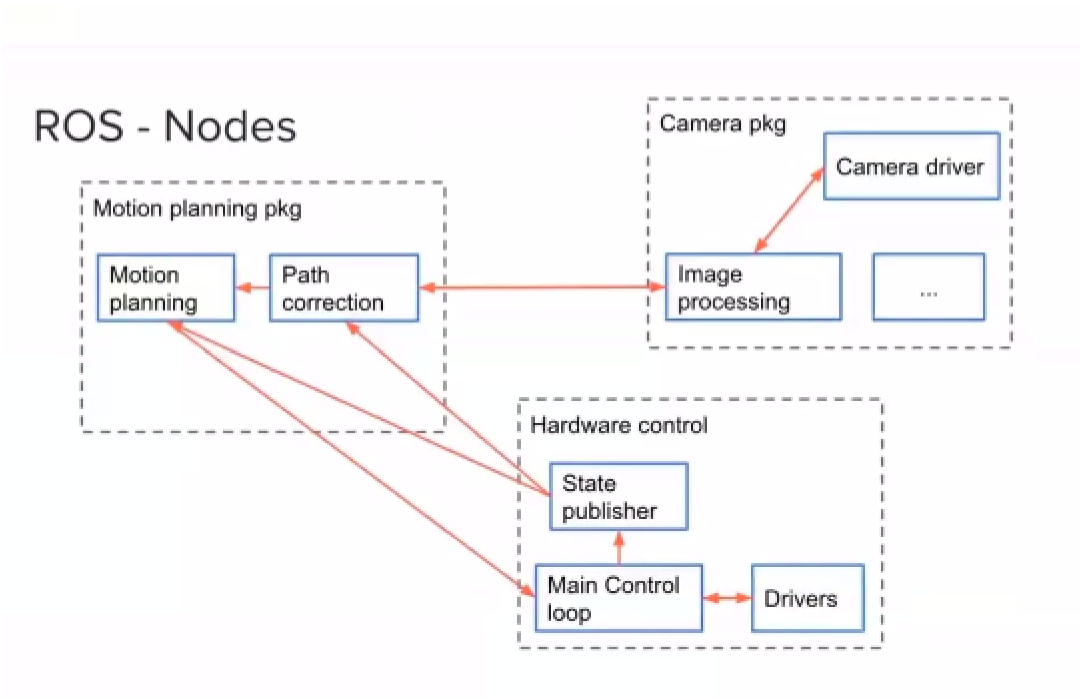


Figure 4 - Interaction between ROS packages

ROS Nodes

Next, within packages, we create nodes, which are an executable file that uses a ROS 'client' library in order to communicate with other nodes. A node can publish or subscribe to a Topic. A node can also provide or use a Service. All nodes of the presented project are located in the file ***catkin_ws/src/my_robot_tutorial/scripts/**** and are executable python codes.

```
import rospy

if __name__ == '__main__':
    rospy.init_node('my_first_python_node')
    rospy.loginfo('This node has feedback')

    rate = rospy.Rate(10)

    while not rospy.is_shutdown():
        rospy.loginfo('Hello there')
        rate.sleep()
```

```
^Crayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ python3 my_first_node.py
[INFO] [1620103787.624357]: This node has feedback
[INFO] [1620103787.625803]: Hello there
[INFO] [1620103787.726111]: Hello there
```

Figure 5- Example of running a Python node code

ROS publisher and subscriber

ROS nodes can communicate between each other via a **publisher/ subscriber** mechanism, in which data is sent from one of many possible publishers on a topic, and subscribers can access this information through the unique name of the topic. In the present project, a topic named ***rpm*** was created which contains the message **“Hello world”**+ the number of the publishement. Codes of `publisher.py/subcriber.py` are in the `scripts` folders.

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ python3 publisher.py
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ python3 subscriber.py
Msg received: data: "This is a published message95"
Msg received: data: "This is a published message96"
Msg received: data: "This is a published message97"
```

Figure 6- Publisher/Subscriber example

Another application was created using a publisher and publisher-subscriber. File names are `task1-pub.py, task1-sub-pub.py`. The first publisher publishes in topic ***rpm***: a coefficient to be multiplied by the radius of the wheels of a robot(for example if radius=10 and rpm is 0.5, we consider the radius for the speed as 0.5*10=5). Then the 2nd code subscribes to ***rpm***, recovers and displays the data, calculates the speed of the robot and publishes it on the topic ***speed***. Results are presented in figure 7.

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ python3 task1-pub.py
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ python3 task1-sub-pub.py
Msg received: data: 10.0
Msg received: data: 10.0
```

```
^Crayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ rostopic list
/rosout
/rosout_agg
/rpm
/speed
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ rostopic echo /speed
data: 31.399999618530273
---
data: 31.399999618530273
---
data: 31.399999618530273
```

Figure 7- Example of multiple topic publishing and subscribing

ROS parameter server

A parameter server in ROS is a shared dictionary that is accessible via network APIs, made to be globally viewable for easy access to retrieve at runtime or to be modified. However, it is not designed for high-performance, it is best used for static, non-binary data such as configuration parameters. In this project, a parameter server named “/wheel_radius” was set from the terminal as depicted in figure 8.
Using the previous codes(Pub/Sub codes), we retrieve both information which are ***rpm*** coefficient and ***wheel_radius***, and employ them in a new node named ***“speed calc_pub_node”*** which calculates the speed and publishes it on “***speed”*** topic. codes employed for this application are `rpm_pub.py, speed_calc.py`.

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ rosparam list
/rosdistro
/roslaunch/uris/host_rayane32_virtualbox__37207
/rosversion
/run_id
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ rosparam set /wheel_radius 2
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ python3 rpm_pub.py
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ python3 speed_calc.py
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial/scripts$ rostopic echo speed
data: 0.20943933333333334
---
data: 0.20943933333333334
```

Figure 8- Speed calculation example, using topics and parameter server

ROS Launch files

Roslaunch is a tool for ease launching multiple ROS nodes locally and remotely, as well as setting parameters on the Parameter Server. It includes options to automatically respawn processes that have already died. **Roslaunch** launches all nodes and parameter servers it contains along with ROS itself, thus no need to start **roscore**. Figure 9 depicts a launch file created for the previous application and its results.

```
<launch>
  <param name="wheel_radius" type="double" value="0.325" />
  <node name="rpm_pub_node" pkg="my_robot_tutorial" type="rpm_pub.py" />
  <node name="speed_calc_sub_node" pkg="my_robot_tutorial" type="speed_calc.py" />
</launch>
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ roslaunch my_robot_tutorial speed_sim.
launch
... logging to /home/rayane32/.ros/log/2f534e0a-aec5-11eb-a629-67356060c2d5/rosl
aunch-rayane32-VirtualBox-3186.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://rayane32-VirtualBox:40945/
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ rosparam list
/rosdistro
/roslaunch/uris/host_rayane32_virtualbox__40945
/rosversion
/run_id
/wheel_radius
rayane32@rayane32-VirtualBox:~/catkin_ws$ rostopic list
/rosout
/rosout_agg
/rpm
/speed
rayane32@rayane32-VirtualBox:~/catkin_ws$ rostopic echo speed
data: 0.03403389166666667
---
data: 0.03403389166666667
```

Figure 9 - Example of the launch file of the previous application and its execution

ROS Bag files

ROS bag file is a storing file for ROS data with an extension *.bag*. To launch a bag file we use the following command: `rosbag record -a -o test.bag` and in order to play its content we use `rosbag play test.bag.`
If we want to play it in a loop, we just add `-l` to the line as follows: `rosbag play -l test.bag.`

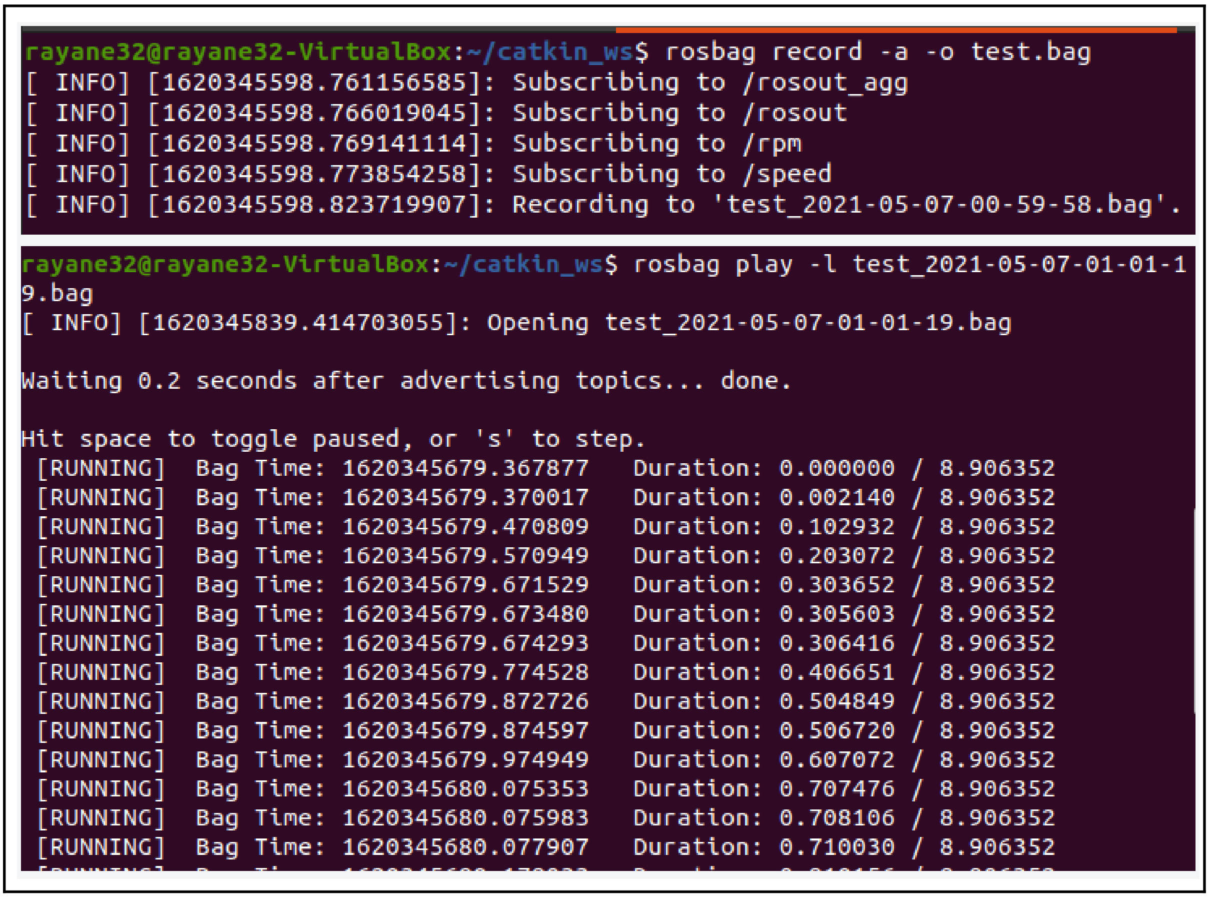


Figure 10- example of a ROS bag

Using predefined ROS packages: Application for image call

As previously explained, we can create packages in ROS and add multiple nodes to it. Moreover, we can use pre-existing packages from the so many list of packages of ROS framework.The list of used packages is saved in `catkin/src/my_robot_tutorial/package.xml`
We can also display the list of ROS packages in terminal using the command:
`rospack list-names`
Figure 11 depicts some of the packages included in our project(the list is very long).

```
rayane32@rayane32-VirtualBox:~/catkin_ws/src/my_robot_tutorial$ rospack list-nam
es
actionlib
actionlib_msgs
actionlib_tutorials
angles
bond
bondcpp
bondpy
camera_calibration
camera_calibration_parsers
camera_info_manager
catkin
class_loader
cmake_modules
compressed_depth_image_transport
compressed_image_transport
```

Figure 11- Example of some packages included in our project

In order to add new packages, for example `usb-cam`, a package that accesses to a usb camera, and the `find-object-2d` package that detects a 2d object from the image retrieved using usb-cam functions, we use the following command lines:

```
>>sudo apt install ros-noetic-usb-cam
>>sudo apt-get install ros-noetic-find-object-2d
```

To run the usb_cam node we run :

```
>>rosrun usb_cam usb_cam_node
```

Unfortunately, due to access problems we could not use the camera, even after installing **v4l2loopback** which is a driver for the camera device. Thanks to the driver, the camera could be found in the list of devices **dev/***. However, ROS could not use the device due to access problems. When running, it shows an error that the device or resource is busy. When investigating this, we found that no process is using the device **dev/video0**. Note that we could have a look at information about the camera. (Figure 12).

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ ls -ltrh /dev/video0
crw-rw----+ 1 root video 81, 0 May  7 02:08 /dev/video0
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ fuser /dev/video0
rayane32@rayane32-VirtualBox:~/catkin_ws$
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ rosrun usb_cam usb_cam_node
[ INFO] [1620350973.261498295]: using default calibration URL
[ INFO] [1620350973.262581502]: camera calibration URL: file:///home/rayane32/.r
os/camera_info/head_camera.yaml
[ INFO] [1620350973.262683765]: Unable to open camera calibration file [/home/ra
yane32/.ros/camera_info/head_camera.yaml]
[ WARN] [1620350973.262737031]: Camera calibration file /home/rayane32/.ros/came
ra_info/head_camera.yaml not found.
[ INFO] [1620350973.262787248]: Starting 'head_camera' (/dev/video0) at 640x480
via mmap (yuyv) at 30 FPS
[ERROR] [1620350973.262890614]: VIDIOC_S_FMT error 16, Device or resource busy
```

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ v4l2-ctl --info --device /dev/video0
Driver Info:
        Driver name      : v4l2 loopback
        Card type        : Dummy video device (0x0000)
        Bus info         : platform:v4l2loopback-000
        Driver version   : 5.8.18
        Capabilities     : 0x85208003
                Video Capture
                Video Output
                Video Memory-to-Memory
                Read/Write
                Streaming
                Extended Pix Format
                Device Capabilities
        Device Caps      : 0x05208003
                Video Capture
                Video Output
                Video Memory-to-Memory
                Read/Write
                Streaming
                Extended Pix Format
```

Figure12 -Information about running usb_cam and the device **dev/video0**

ROS service (server-client)

ROS services is a paradigm that connects a server to its client, thus it allows a request/replay operation between a pair of nodes. Services are defined with **.srv** files, which are compiled into source code by a ROS client library.

- Before creating services, some dependencies must be added to the *package.xml* file which are :

```
<build_depend>message_generation</build_depend>
<exec_depend>message_runtime</exec_depend>
```

- when adding a service, we have to generate this service in the *Cmakelist.txt*.
- A **catkin** is mandatory to establish the added dependencies.

In our project, we have created a service that takes from the client an angle and the server returns an image associated to the closest angle to the input angle from the dataset (for example if the client choses 40° and the closest angle of camera in the dataset is 45°, it returns the image associated to the angle 45°). For that, we've created a service named `TurnCamera.srv`. The server node is `turn_camera_service` and the client node is `turn_camera_client`, both are found in `my_robot_tutorial/src/scripts`.

```
float32 turn_degrees
---
sensor_msgs/Image image
```

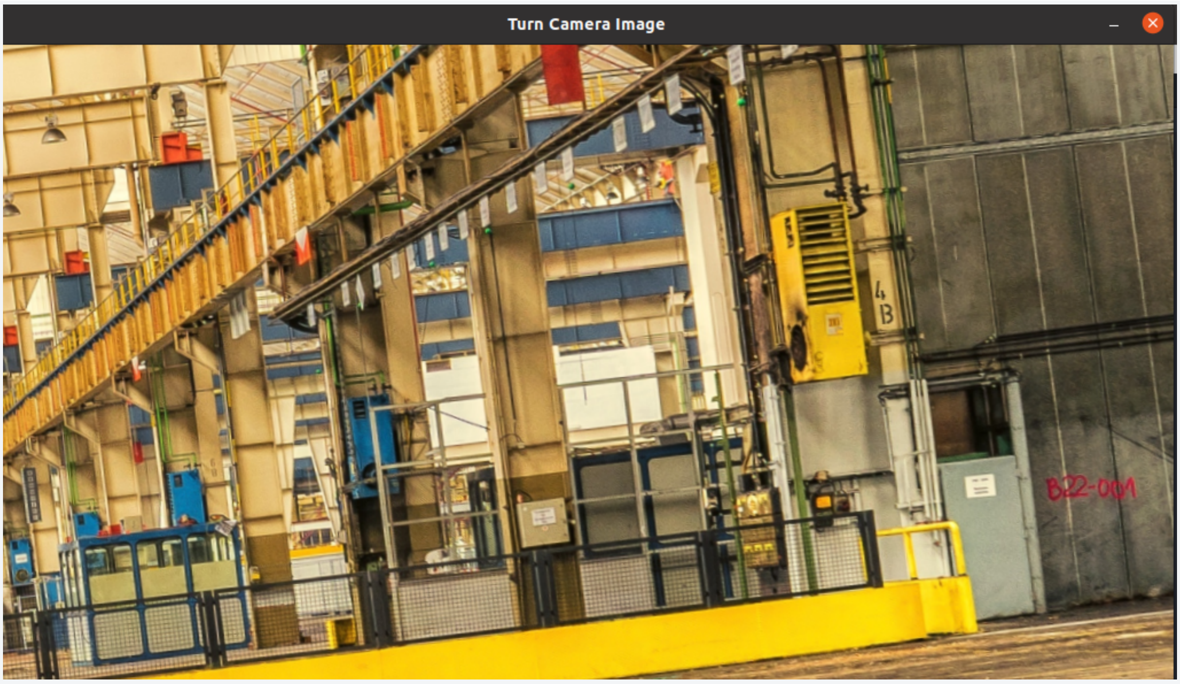
(a) TurnCamera.srv

```
^Crayane32@rayane32-VirtualBox:~/catkin_ws$ rosrun my_robot_tutorial turn_cameraervice.py
Turn Camera service is Running
```

(b) running the server

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ rosrun my_robot_tutorial turn_camera_client.py
 Enter an angle degrees to move the robot Camera:40
```

(c) running the client



(d) result: closest angle to 40 image

Figure 13- The running of ROS turn camera service

ROS Action

ROS action is a client-server service that provides feedback. Thus to create an action, we have to specify 3 types of information:

- **Goal** : sent to an ActionServer by an ActionClient.
- **Feedback** : Allows the ActionServer to inform the ActionClient about the incremental progress of a goal. Very useful when the result isn't very important and the ActionClient wants to track the achievement of the goal
- **Result** : sent upon completion of the goal

Some dependencies must be added to *package.xml*:

```
<build_depend>actionlib</build_depend> <build_depend>actionlib_msgs</build_depend>

<exec_depend>actionlib</exec_depend> <exec_depend>actionlib_msgs</exec_depend>
```

Once an action file is created (.action), it must be added to the CMakelist.text with the function : `add_action_files( FILES name_of_file.action)`.
In our project, we create an action called `navigate 2D.action` to help the ActionClient track the navigating of the robot to a goal point.
Codes of `ActionClient.py`, `ActionServer.py`, and `robot_point_pub.py` are found in */src/my_robot_tutorial/scripts/.* Figure 14 depicts the action file and the running of the action on the terminal.

```
#Goal
geometry_msgs/Point point
---
#Result
float32 elapsed_time
---
#feedback
float32 distance_to_point
```

(a) Action file

```
^Crayane32@rayane32-VirtualBox:~/catkin_ws$ rosrun my_robot_tutorial robot_point_pub.py
What is your current x-coordinates?: 1
What is your current y-coordinates?: 1
What is your current z-coordinates?: 1
Publishing
```

(b) robot point publisher

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ rosrun my_robot_tutorial  action_client.py
5What is your desired x-coordinate?: 5
What is your desired y-coordinate?: 8
What is your desired z-coordinate?: 7
Distance to Goal: 54.7813835144043
Distance to Goal: 54.7813835144043
Distance to Goal: 54.7813835144043
Distance to Goal: 54.7813835144043
```

(c) Action Client

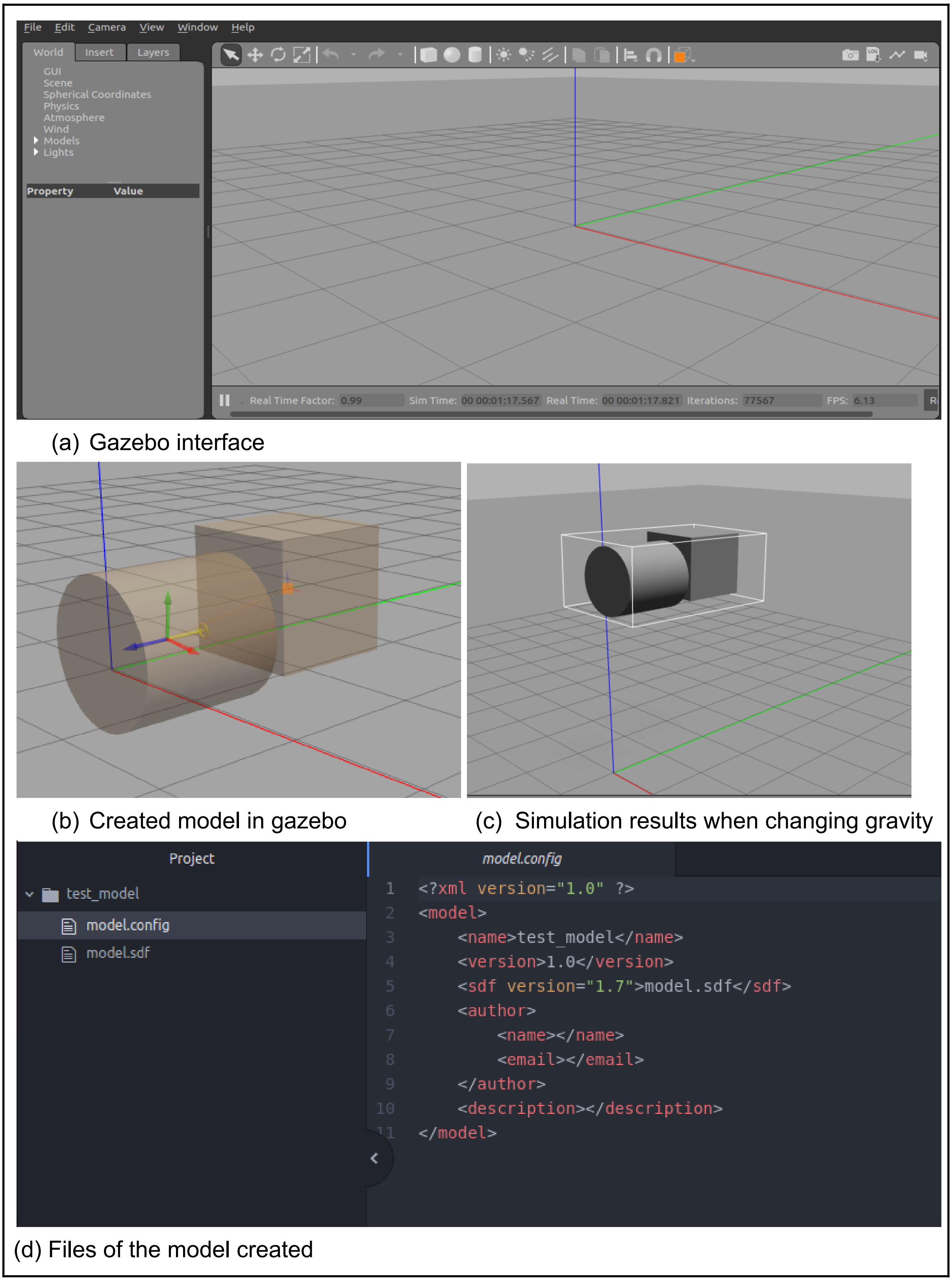
```
rayane32@rayane32-VirtualBox:~/catkin_ws$ rosrun my_robot_tutorial action_server.py
[WARN] [1620382603.823890]: You've passed in true for auto_start to the python action
onditions.
Robot Point Not Detected
Robot Point detected
```

(d) Action Server

Figure 14- ROS action example for tracking

ROS Gazebo

Gazebo is a cinematic, dynamic, 3D robot simulator. Its packages provide the necessary interfaces to simulate a robot in a 3D plane.To start gazebo on a roscore running machine, the command “gazebo” must be entered. The output interface is presented in figure 15-a.
With gazebo we create the model presented in figure 15-b. Figure 15-c shows the example of simulation in case we change the gravity to a positive value: the blocks will move to a higher point. Figure 15-d presents the resulting files from the creation of a test_model in gazebo.



(a) Gazebo interface

(b) Created model in gazebo

(c) Simulation results when changing gravity

(d) Files of the model created

Figure 15- Model creation using gazebo

Two dependencies are added via the following commands:

```
-  sudo apt install ros-noetic-joint-state-controller
-  sudo apt install ros-noetic-velocity-controllers
```

A **catkin** is mandatory to establish these new dependencies. (Figure 16)

```
rayane32@rayane32-VirtualBox:~/catkin_ws$ catkin_make
Base path: /home/rayane32/catkin_ws
Source space: /home/rayane32/catkin_ws/src
Build space: /home/rayane32/catkin_ws/build
Devel space: /home/rayane32/catkin_ws/devel
Install space: /home/rayane32/catkin_ws/install
####
#### Running command: "cmake /home/rayane32/catkin_ws/src -DCATKIN_DEVEL_PREFIX=
/home/rayane32/catkin_ws/devel -DCMAKE_INSTALL_PREFIX=/home/rayane32/catkin_ws/i
nstall -G Unix Makefiles" in "/home/rayane32/catkin_ws/build"
####
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- ~~  traversing 2 packages in topological order:
-- ~~  - my_robot_tutorial
-- ~~  - test_sim_pkg
-- ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
-- +++ processing catkin package: 'my_robot_tutorial'
-- ==> add_subdirectory(my_robot_tutorial)
-- Using these message generators: gencpp;geneus;genlisp;gennodejs;genpy
-- Generating .msg files for action my_robot_tutorial/navigate2D /home/rayane32/
catkin_ws/src/my_robot_tutorial/action/navigate2D.action
-- my_robot_tutorial: 7 messages, 2 services
-- +++ processing catkin package: 'test_sim_pkg'
-- ==> add_subdirectory(test_sim_pkg)
-- Configuring done
-- Generating done
```
Figure 16- Catkin of the new dependencies

Important Notes

- A catkin is mandatory after every modification of dependencies.
- all python code must be executable.
- Sourcing is necessary in every terminal : *source devel/setup.bash.*

Conclusion

In this report, we have presented the ROS implementations that we have achieved during the semester for the ISR module. We have detailed the installation setup, package adding and creation, node creation, publisher/subscriber and topics, parameter server, services, actions and gazebo usage. These functionalities allowed the realization of 4 main projects which are:

- Simple **publisher/subscriber** mechanism using **one topic.**
- **Publisher/publisher-subscriber** using **2 topics** for speed calculation and publishing.
- Object detection using **usb_cam package.**
- **Client/Server** application using **Services** to return whether a number is odd or even.
- **Client/Server** application using **Services** to return the images from the closest camera.
- **Client/Server** application using **Actions** to track a robot moving from a starting point to a goal point, while receiving feedback.
- Creation and simulation of a model using **gazebo**.

In the presented project, all python codes are commented and **a tree presentation** of the catkin_ws file is included under the name **catkin_tree_file.odt**.