CASE STUDY REVIEW

Privacy Level

1

CONTENTS

REVISION HISTORY

Date	Revision No.	Definition	Author(s)
10/04/2023	1.0	Initial document	Buse Nur EMİR

1. INTRODUCTION

Robot Operating System (ROS) is a collection of tools, libraries, and rules aimed at simplifying the task of creating complex and robust robot behavior across a wide variety of robotic platforms. It provides a flexible framework for robot software. ROS is built on various conceptual structures, including nodes, topics, services, and parameters.

1. Nodes: They are functionally independent units, and a ROS application can include multiple nodes.
2. Topics: They are communication channels and enable data sharing between nodes.
3. Services: They allow one node to call another node to perform a task.
4. Parameters: They are used to change various features of the ROS application

The case study is built in that the talker unit shared the data of battery level and the listener unit takes this data to display it. By using ROS, the units are defined as nodes. The nodes communicate with each other via “/battery_topic” topic as shown in Figure 1.

Figure 1. The structure of the project with rqt_graph

2. INSTALLATION

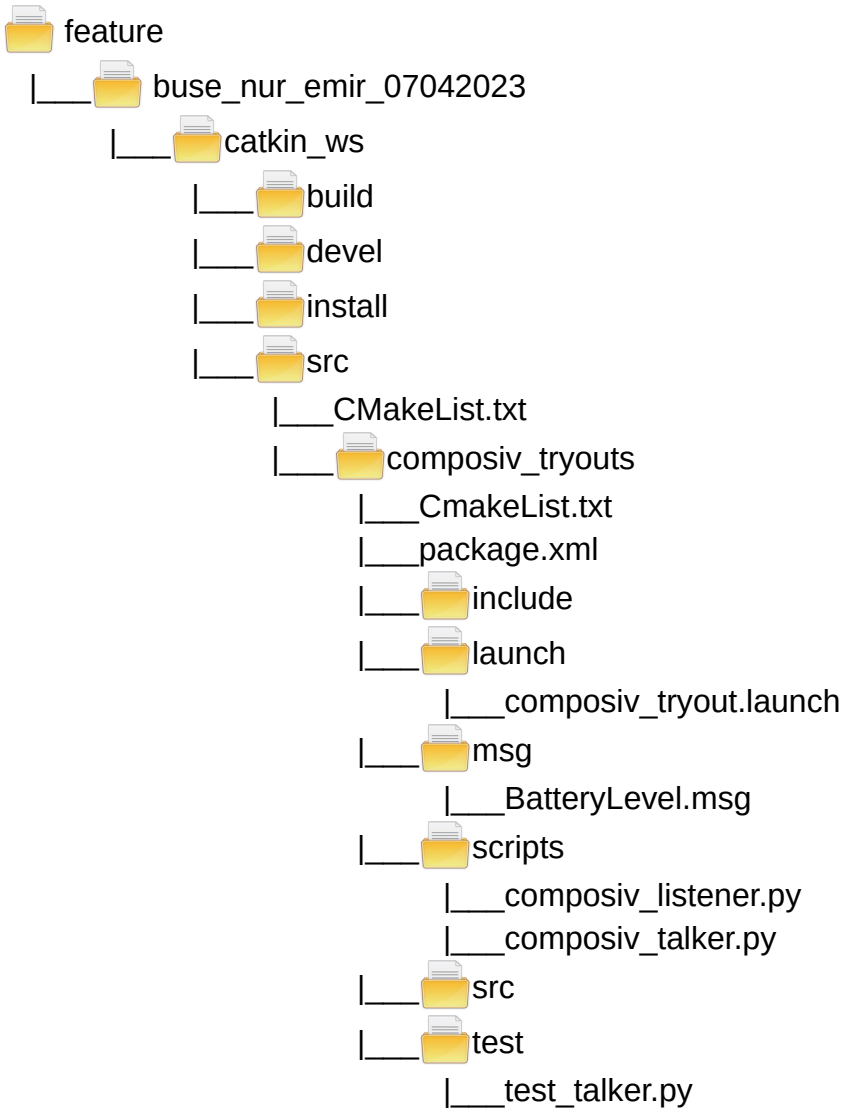
ROS is supported by many operating systems. However, the officially supported operating system is Ubuntu. ROS has different versions for different operating systems. This project is built on ROS Noetic Ninjemysm recommended for Ubuntu 20.04.

For installation, please visit this page and follow the steps.

3. STRUCTURE

The environment-based system in ROS provides modularize the code and dependencies, which makes it easier to develop, test, and deploy robotic applications. In ROS, the environment refers to a set of packages and their dependencies, which are installed and used together. Each package contains nodes, libraries, and configuration files that enable different functionalities. The ROS environment is structured as a hierarchical file system, where each package has its directory with subdirectories for different types of files. These files can include source code, configuration files, launch files, documentation, and more. Each package directory also contains a package.xml file, which defines the package name, version, dependencies, and other package-specific information.

The structure of the project is as below. The "catkin_ws" workspace under the feature/buse_nur_emir_07042023 directory is in the ROS environment. It includes the “composiv_tryouts” package. It was created that one test, one message, one launch, and two scripts file in this package.



4. IMPLEMENTATION

4.1 Creating Workspace and Package

First of all, the necessary conditions must be created to work in the ROS environment. The first thing to do is to define the configuration settings. This definition is made in the bashrc file in order not to do the same thing every time the terminal is opened. The bashrc file is opened by typing the following command into the terminal:

- gedit ~/.bashrc

The text below is added to the bottom line of the file:

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

Then the workspace catkin_ws is created under the feature/buse_nur_emir_07042023 directory as mentioned under the Structure heading. New folders must be created for this. To perform this operation from the terminal, mkdir, cd, and ls commands are used. “mkdir” command is used to create a new directory or folder. “cd” command is used to change the current directory. “ls” command is used to list the files and directories in the current working directory. The steps followed to create catkin_ws and the src folder in it are shown in Figure 2.

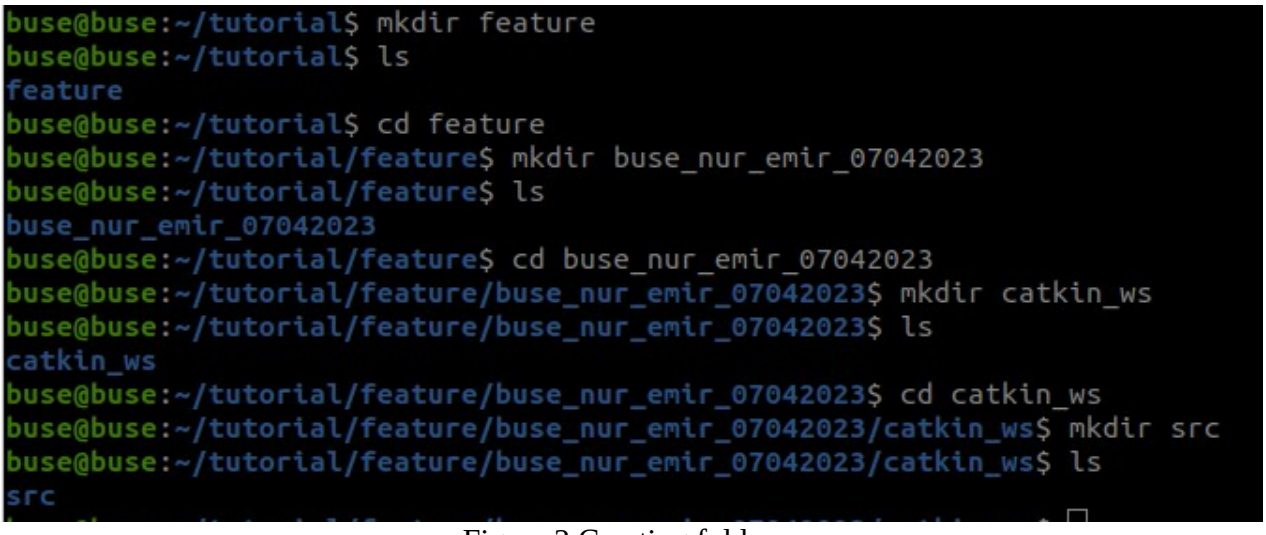


Figure 2.Creating folders

The catkin_make command is run under catkin_ws to compile the workspace. As a result of the process, "build" and "devel" folders are created under the folder. The "build" folder is where temporary files generated during the compilation process of the project are stored, while the "devel" folder contains development materials used during the development process. The "src" folder is where the project's source code is stored. The following text is added to the last part of the bashrc file to declare the source of the working environment:

- source /home/buse/feature/buse_nur_emir_07042023/catkin_ws/devel/setup.bash

Packages must be in the workspace under the src folder. After going to this directory, the following command is used to create the "composiv_tryouts" package. Since it is foreseen that cpp and python languages can be used for the developments to be made in the package, the package is created depending on rospy and roscpp.

- catkin_create_pkg composiv_tryouts rospy roscpp

“include” and “src” folders, “package.xml” and “CMakeLists.txt” files are created automatically in the package. Then, the package is compiled with the catkin_make command after going to the workspace folder.

4.2 Creating Message File

ROS msg (message) files define the structure of the data that can be exchanged between ROS nodes over a topic. They specify the name and data type of each field in a message. In this project, it is aimed to transfer battery level information over the "battery_topic" topic.

First, the msg folder is created in the composiv_tryouts folder. A message file named “BatteryLevel.msg” is created inside this folder. The contents of this file are as follows. It is stated that the value kept in the battery variable is of string type.

- string battery

Then edits are made in package.xml and CMakeLists.txt files of the package. The changes shown in Figure 3 have been made in the CMakeLists.txt file.

1. “message_generation” dependence is added because it works with messages.

2. The message file is determined

3. It is provided that generating message by uncomment the “generate_message()” function

4. The package is depended to message_runtime.

Open | CMakeLists.txt ~/feature/buse_nur_emir_07042023/catkin_ws/src/composiv_tryo... | Save

```
cmake_minimum_required(VERSION 3.0.2)
project(composiv_tryouts)

## Compile as C++11, supported in ROS Kinetic and newer
# add_compile_options(-std=c++11)

## Find catkin macros and libraries
## if COMPONENTS list like find_package(catkin REQUIRED COMPONENTS xyz)
## is used, also find other catkin packages
find_package(catkin REQUIRED COMPONENTS
  roscpp
  rospy
  message_generation
)
```

```
##   * add "message_runtime" and every package in MSG_DEP_SET to
##     catkin_package(CATKIN_DEPENDS ...)
##   * uncomment the add_*_files sections below as needed
##     and list every .msg/.srv/.action file to be processed
##   * uncomment the generate_messages entry below
##   * add every package in MSG_DEP_SET to generate_messages(DEPENDENCIES ...)

## Generate messages in the 'msg' folder
add_message_files(FILES BatteryLevel.msg)
#   FILES
#   Message1.msg
#   Message2.msg
# )

## Generate services in the 'srv' folder
```

```

## Generate actions in the 'action' folder
# add_action_files(
#   FILES
#   Action1.action
#   Action2.action
# )

## Generate added messages and services with any dependencies listed here
generate_messages()
#   DEPENDENCIES
#   std_msgs  # Or other packages containing msgs
# )

################################################
```

```
## The catkin_package macro generates cmake config files for your package
## Declare things to be passed to dependent projects
## INCLUDE_DIRS: uncomment this if your package contains header files
## LIBRARIES: libraries you create in this project that dependent projects also need
## CATKIN_DEPENDS: catkin_packages dependent projects also need
## DEPENDS: system dependencies of this project that dependent projects also need
catkin_package(CATKIN_DEPENDS message_runtime
#  INCLUDE_DIRS include
#  LIBRARIES composiv_tryouts
#  CATKIN_DEPENDS roscpp rospy
#  DEPENDS system_lib
)

###########
```

Figure 3.Changes in CMakeLists.txt

The changes have been made in the package.xml file is shown in Figure 4. Since message_generation is needed in compilation and message_runtime is needed in operation, they have been added to the file.

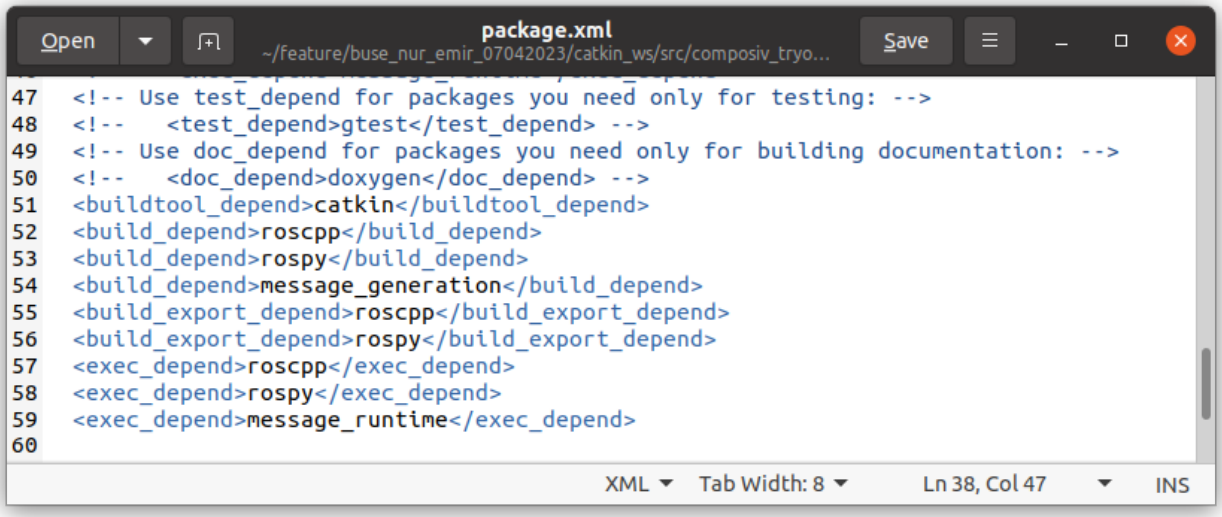


Figure 4.Changes in package.xml

4.3 Creating Publisher-Subscriber Nodes

Package source code is stored in the src folder. Source codes can be written in python and cpp languages. To separate these files, cpp files are in the src folder and python files are in the scripts folder. For this reason, the scripts folder is created in the same directory. Since the python language is preferred in the project, the codes are located in the scripts folder. The files must be made executable with the example command below, otherwise the files will not work.

- chmod +x composiv_talker.py

4.3.1 Publisher Node

A python script file named “composiv_talker” is created for the publisher node. This script shown in Figure 5 uses the rospy library to publish messages to a ROS topic called "battery_topic" with the message type of "BatteryLevel". The script starts by initializing the ROS node with the name "talker_node" and creating a publisher object for the "battery_topic" topic.

The script then enters a while loop that will run until the node is shutdown. Within the loop, the variable "level" is assigned the value of "%25", which represents a battery level of 25%. The "rospy.loginfo()" function is used to print the current battery level to the console.

Finally, the "publish()" method of the publisher object is called with the "level" variable as the argument to publish the message to the "battery_topic" topic. The "rate.sleep()" function is used to pause the loop for 1 second between each iteration to control the publishing frequency. The script ends by calling the "publishMsg()" function.

composiv_talker.py

```python
#!/usr/bin/env python3

import rospy
from composiv_tryouts.msg import BatteryLevel

def publishMsg():
    rospy.init_node("talker_node")
    pub = rospy.Publisher("battery_topic", BatteryLevel)
    rate = rospy.Rate(1)
    while not rospy.is_shutdown():
        level = "%25"
        rospy.loginfo(level)
        pub.publish(level)
        rate.sleep()

publishMsg()
```

Figure 5.Publisher node

4.3.2 Subscriber Node

A python script file named “composiv_listener” is created for the subscriber node. This script shown in Figure 6 uses the rospy library to subscribe to a ROS topic called "battery_topic" with the message type of "BatteryLevel". The script starts by initializing the ROS node with the name "listener_node" and creating a subscriber object for the "battery_topic" topic, with a callback function "subsCallback" specified as the second argument.

The "subsCallback" function takes a single argument, which is the message received from the "battery_topic" topic. In this case, the function prints a log message to the console, which includes the battery level of the vehicle obtained from the received message.

The "subscribeMsg()" function is then called, which enters a loop that will run until the node is shutdown. Within the loop, the subscriber object listens for messages on the "battery_topic" topic and calls the "subsCallback" function whenever a message is received.

The "rospy.spin()" function is used to keep the script running and handle any incoming messages in the background. Finally, the script ends by calling the "subscribeMsg()" function.

composiv_listener.py

```python
#!/usr/bin/env python3

import rospy
from composiv_tryouts.msg import BatteryLevel

def subsCallback(message):
    rospy.loginfo("Battery level of vehicle: %s" %message.battery)

def subscribeMsg():
    rospy.init_node("listener_node")
    rospy.Subscriber("battery_topic", BatteryLevel, subsCallback)
    rospy.spin()

subscribeMsg()
```

Figure 6.Subscriber node

4.3.3 Test

At this stage, 3 different terminal screens are needed, Figure 7. First, the roscore command shown on the upper terminal is run to start ROS. Then the nodes are run by rosrun command. The outputs show that both nodes are working fine.

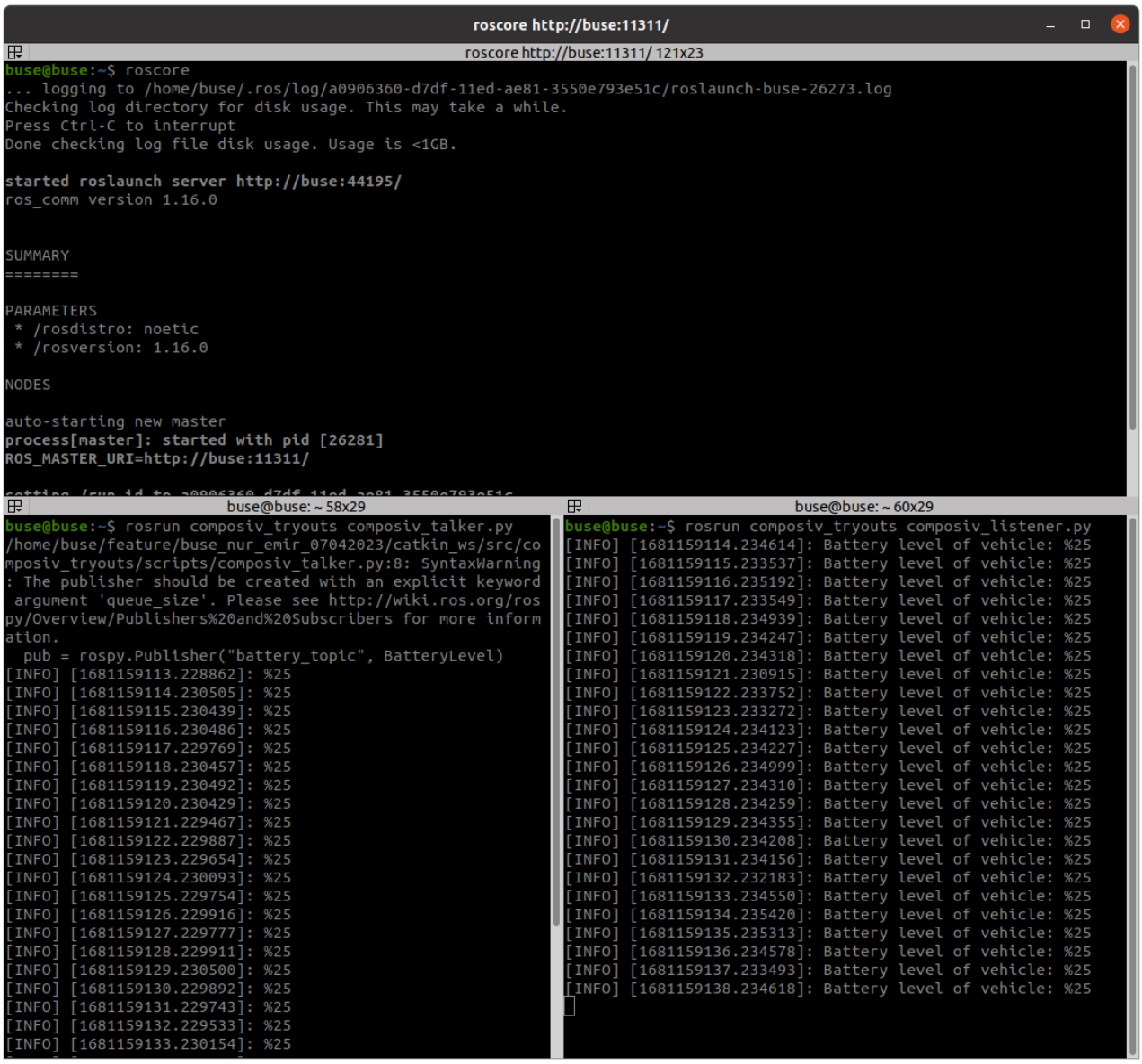


Figure 7.Test screen

4.4 Creating Unit-Test

The "rostest" and "unittest" modules are Python modules that help test software units on ROS. The "unittest" module provides a framework for defining test cases. These states describe the behavior, errors, and success of the functions to be tested. These test cases are used to determine that a particular part of your program is working correctly. The "rostest" module provides a framework for testing ROS packages. This module enables automatic coordination of tests between ROS packages using the "roslaunch" command. With these tests, dependencies between ROS packages, messages are sent and received correctly, and functions are called correctly can be tested. The test folder is created in the composiv_tryouts folder for the test scripts.

A python script file named “test_talker” shown in Figure 8 defines a unit test case for a ROS (Robot Operating System) node that publishes messages to a topic named "battery_topic". The test case is defined using the "unittest" module, which provides a framework for writing and executing tests.

The "TalkerTestCase" class extends the "unittest.TestCase" class and defines a test method called "test_talker_publishing". This method initializes a ROS node with the name "test_talker" and creates a subscriber object for the "battery_topic" topic. It also defines a callback function called "callback" that will be called when messages are received on the topic.

The "test_talker_publishing" method then enters a loop that will run until the node is shutdown, a maximum of 5 seconds have elapsed, or a message has been received on the "battery_topic" topic. If a message is received, the "self.talker_ok" flag is set to True, indicating that the test has passed.

The "if name == 'main':" block at the end of the script uses the "rostest" module to run the "composiv_talker" node with the "TalkerTestCase" test case. This allows the test case to be executed as part of a larger test suite for the ROS package "composiv_tryouts".

It is benefitted from the [ROS Q&A] 098 - How to see if my ROS Publisher works using ROS Unit Testing video to write this script.

test_talker.py

run > user > 1000 > doc > af49d4e5 > test_talker.py

```python
#!/usr/bin/env python3

import unittest
import rospy
from composiv_tryouts.msg import BatteryLevel
from time import sleep
import rostest

class TalkerTestCase(unittest.TestCase):
    talker_ok = False

    def callback(self,data):
        self.talker_ok = True

    def test_talker_publishing(self):
        rospy.init_node('test_talker')
        rospy.Subscriber("battery_topic", BatteryLevel, self.callback)
        counter = 0
        while not rospy.is_shutdown() and counter<5 and (not self.talker_ok):
            sleep(1)
            counter += 1

        self.assertTrue(self.talker_ok)

if __name__ == '__main__':
    rostest.rosrun('composiv_tryouts','composiv_talker', TalkerTestCase)
```

Figure 8.Test script

The test can be performed with the rosrun command using three separate terminals, as shown in Figure 9. It can also be done with the rostest command by creating a launch file.

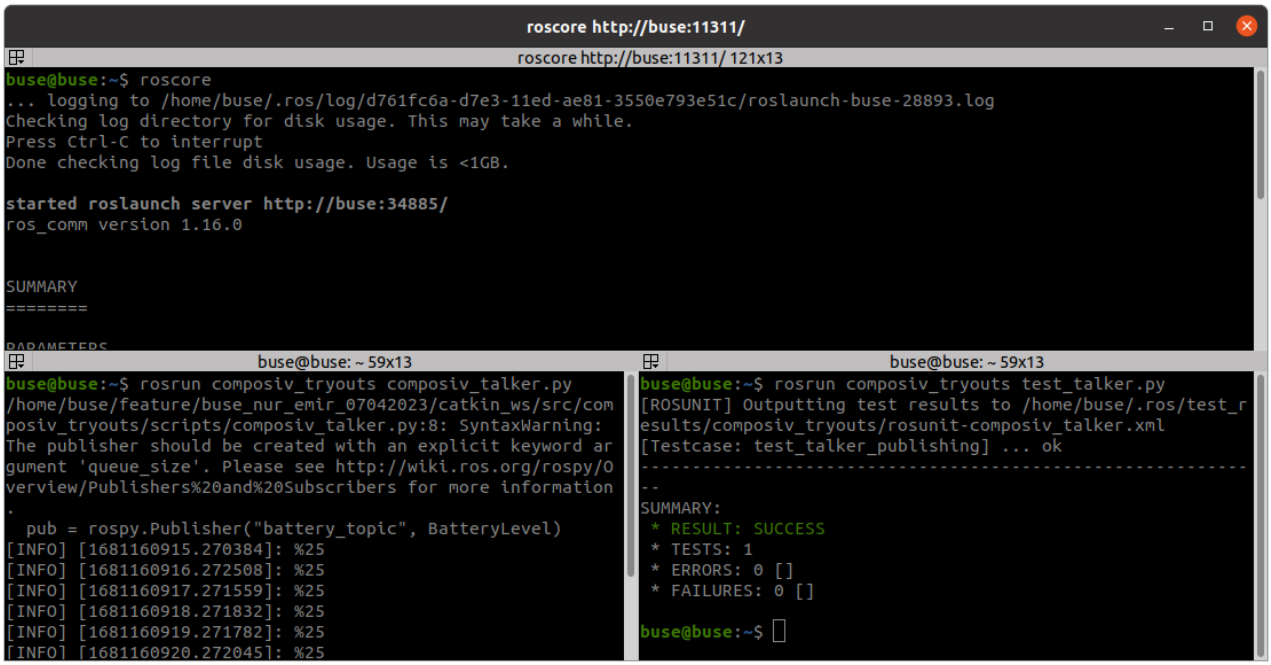


Figure 9.Running test script

4.5 Creating Launch File

A launch file specifies the dependencies between ROS packages and starts the ROS nodes you want to run. The launch folder is created in the composiv_tryouts folder for the launch files. In the launch file named “composiv_tryout” shown in Figure 10, three nodes are started from the "composiv_tryouts" package.

The first node launches a ROS node named "talker" from the "composiv_talker.py" file. This node publishes messages to a ROS topic named "battery_topic" at certain intervals.

The second node launches a ROS node named "listener" from the "composiv_listener.py" file. This node listens to messages published on the "battery_topic" ROS topic and prints the messages it receives.

The third node launches a ROS test case named "test_talker.py". This test case checks whether the "talker" node in the "composiv_talker.py" file sends messages to the "battery_topic" topic correctly. This way, it tests whether the "talker" node is working properly.

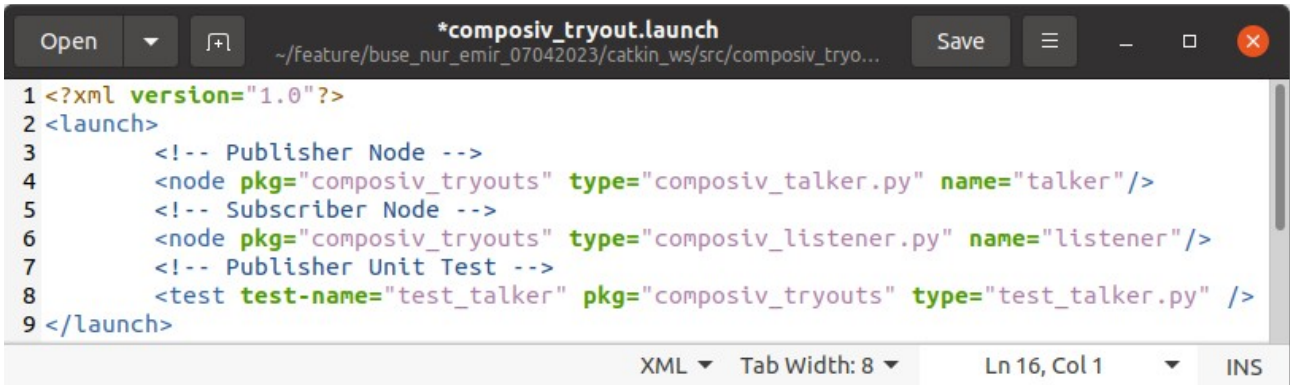


Figure 10.The launch file

This "launch" file can be run with the "roslaunch" command, which will automatically start all the nodes and test cases. In addition, testing can be performed by running the launch file with the rostest command, Figure 11.

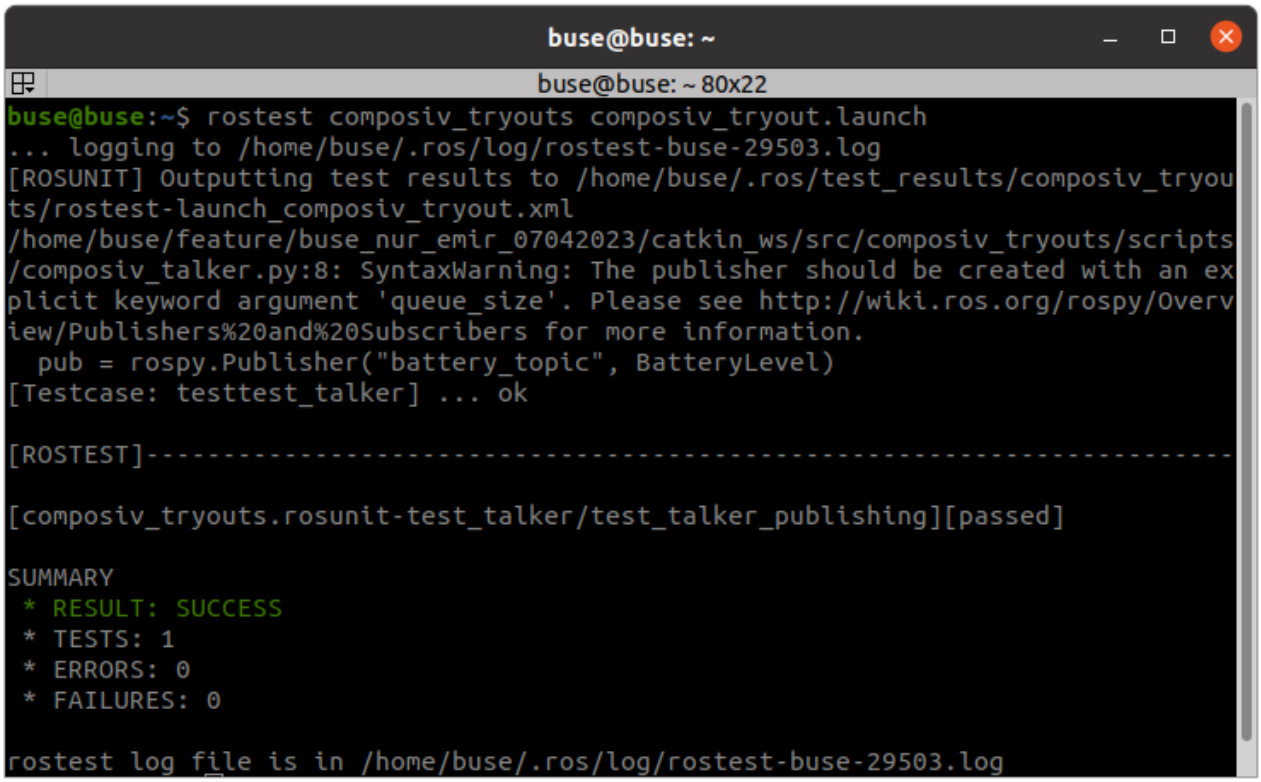


Figure 11.Usage of rostest