Guía Detallada para el Desarrollo de un Proyecto Avanzado en ROS 2

August 22, 2025

Contents

1 Introducción

Este documento sirve como una guía detallada para el desarrollo de un proyecto avanzado utilizando ROS 2 (Robot Operating System 2). Esta orientado a estudiantes e ingenieros que buscan estructurar un sistema robótico complejo y modular, siguiendo buenas practicas de desarrollo.

2 Herramientas necesarias

2.1 Hardware sugerido

- Computadora con Ubuntu 20.04 o 22.04 LTS (recomendado).
- Procesador x86_64 (preferible con soporte de virtualización y al menos 4 GB de RAM).
- Opcional: Placa de desarrollo (Raspberry Pi 4, Jetson Nano, etc.).
- Sensores: LIDAR, cámaras, IMU, motores, etc.
- Microcontroladores: ESP32, ESP8266.

2.2 Software necesario

- Ubuntu 20.04 o 22.04 LTS
- ROS 2 (Foxy, Galactic, Humble o Iron, según tu versión de Ubuntu)
- Python 3 (viene por defecto en Ubuntu)
- Compilador C++ (g++)
- Visual Studio Code o cualquier editor con soporte para ROS 2
- Git
- Docker (opcional, para ambientes aislados)
- Mosquitto (broker MQTT local)

2.3 Recomendaciones generales

- Utilizar ROS 2 Humble Hawksbill o ROS 2 Jazzy Jalisco según la versión de Ubuntu.
  - Utilizar ROS 2 Humble si trabajas con Ubuntu 22.04.
  - Utilizar ROS 2 Jazzy si trabajas con Ubuntu 24.04.
- Siempre trabajar en un workspace de ROS 2 bien definido (por ejemplo, `ros2_ws`).
- Dividir el sistema en paquetes modulares según la funcionalidad.
- Usar entornos virtuales o Docker para evitar conflictos de dependencias.
- Dividir el sistema en paquetes modulares segun la funcionalidad.

2.4 Instalación de ROS 2 (Humble y Jazzy)

2.4.1 ROS 2 Humble Hawksbill (Ubuntu 22.04)

```
sudo apt update && sudo apt install curl gnupg lsb-release
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.
   asc | sudo apt-key add -
sudo sh -c 'echo␣"deb␣[arch=$(dpkg␣--print-architecture)]␣http://packages.
   ros.org/ros2/ubuntu␣$(lsb_release␣-cs)␣main"␣>␣/etc/apt/sources.list.d/
   ros2.list'
sudo apt update
sudo apt install ros-humble-desktop
source /opt/ros/humble/setup.bash
```

2.4.2 ROS 2 Jazzy Jalisco (Ubuntu 24.04)

```
sudo apt update && sudo apt install curl gnupg lsb-release
sudo curl -sSL https://raw.githubusercontent.com/ros/rosdistro/master/ros.
   asc | sudo apt-key add -
sudo sh -c 'echo␣"deb␣[arch=$(dpkg␣--print-architecture)]␣http://packages.
   ros.org/ros2/ubuntu␣$(lsb_release␣-cs)␣main"␣>␣/etc/apt/sources.list.d/
   ros2.list'
sudo apt update
sudo apt install ros-jazzy-desktop
source /opt/ros/jazzy/setup.bash
```

3 Creacion del Workspace y Paquetes

3.1 Crear el Workspace

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws
colcon build
source install/setup.bash
```

3.2 Crear un paquete en Python

```
cd ~/ros2_ws/src
ros2 pkg create --build-type ament_python my_py_pkg --dependencies rclpy
   std_msgs
```

3.3 Estructura del paquete

El paquete generado contiene:

- `package.xml`: metadatos del paquete
- `setup.py`: configuracion para instalar el paquete
- `my_py_pkg/`: carpeta fuente con `init.py` y codigo Python

3.4 Ejemplo de nodo Publisher

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String

class MinimalPublisher(Node):
def init(self):
super().init(minimal_publisher)
```

```
self.publisher_ = self.create_publisher(String, topic, 10)
self.timer = self.create_timer(0.5, self.timer_callback)
self.i = 0

def timer_callback(self):
    msg = String()
    msg.data = fHola ROS 2: {self.i}
    self.publisher_.publish(msg)
    self.i += 1

rclpy.init()
node = MinimalPublisher()
rclpy.spin(node)
node.destroy_node()
rclpy.shutdown()
```

3.5 Construir el workspace

```
cd ~/ros2_ws
colcon build
source install/setup.bash
```

3.6 Ejecutar el nodo

```
ros2 run my_py_pkg minimal_publisher
```

4 Siguientes pasos

En la siguiente seccion abordaremos el uso de archivos launch, URDF para descripcion de robots, creacion de mensajes personalizados (.msg), servicios (.srv), integracion con simuladores y pruebas automatizadas.

5 Estructura del Proyecto y Archivos Importantes

Un proyecto completo en ROS 2 puede involucrar múltiples paquetes, configuraciones de lanzamiento, descripciones de robots y más. A continuación se presenta una estructura recomendada para organizar un sistema modular.

5.1 Árbol de Archivos Típico

```
ros2_ws/
 install/
 build/
 log/
```

```
src/
   my_py_pkg/
      my_py_pkg/
         __init__.py
         minimal_publisher.py
         minimal_subscriber.py
      launch/
         my_launch_file.py
      resource/
         my_py_pkg
      package.xml
      setup.cfg
      setup.py
   robot_description/
       urdf/
          my_robot.urdf
       meshes/
          base_link.stl
       launch/
          display.launch.py
       rviz/
          my_config.rviz
       package.xml
       setup.py
       setup.cfg
```

5.2 Descripción de Componentes Clave

- **my_py_pkg/launch/**: contiene los archivos `.py` de lanzamiento.
- **my_py_pkg/my_py_pkg/**: código fuente del paquete.
- **robot_description/urdf/**: archivos URDF del robot.
- **robot_description/meshes/**: modelos 3D del robot (STL, DAE, etc.).
- **robot_description/rviz/**: configuración visual para RViz.

6 Lanzamiento de Nodos con Launch Files

6.1 Ejemplo de archivo `launch`

```python
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package=my_py_pkg,
            executable=minimal_publisher,
```

```
            name=publicador,
            output=screen
        ),
        Node(
            package=my_py_pkg,
            executable=minimal_subscriber,
            name=suscriptor,
            output=screen
        )
    ])
```

6.2 Ejecutar el archivo launch

```
ros2 launch my_py_pkg my_launch_file.py
```

7 URDF: Descripción del Robot

7.1 Estructura básica de un archivo URDF

```
<robot name="mi_robot">
  <link name="base_link">
    <visual>
      <geometry>
        <box size="0.2 0.2 0.1"/>
      </geometry>
    </visual>
  </link>
</robot>
```

7.2 Visualización en RViz

```
ros2 run robot_state_publisher robot_state_publisher my_robot.urdf
ros2 run rviz2 rviz2
```

7.3 Uso con archivos launch

Puedes incluir el URDF dentro de un archivo launch para automatizar su visualización junto con RViz y el estado del robot.

8 Mensajes Personalizados (.msg)

8.1 Creación de mensajes personalizados

En el directorio `msg/` de tu paquete:

```
mkdir msg
echo "string␣data" > msg/MensajeCustom.msg
```

Modifica `package.xml` y `setup.py` para incluir la generación de mensajes, y reconstruye el workspace.

9 Servicios Personalizados (.srv)

9.1 Creación de servicios personalizados

En el directorio `srv/`:

```
mkdir srv
echo -e "int64␣a\nint64␣b\n---\nint64␣suma" > srv/Suma.srv
```

También debes modificar los archivos de configuración e instalar dependencias.

10 Integración con Simuladores

Para simulaciones se puede usar Gazebo. Instálalo con:

```
sudo apt install ros-humble-gazebo-ros-pkgs
```

Puedes cargar tu robot en Gazebo directamente desde un archivo URDF o XACRO.

11 Pruebas Automatizadas

ROS 2 soporta pruebas con `launch_testing` y `pytest`. Un ejemplo básico:

```
import launch_testing
import pytest

@pytest.mark.rostest
def test_node_output(test_node):
    assert b"Hola␣ROS␣2" in test_node.output
```

—

£Deseas que continúe con una sección de depuración, despliegue en hardware real, o integración con MQTT/IoT?

12 Uso de Archivos Launch en ROS 2

Los archivos `launch` en ROS 2 permiten iniciar múltiples nodos, configurar parámetros y definir relaciones entre componentes del sistema robótico.

12.1 Estructura de un archivo launch (Python)

A diferencia de ROS 1, ROS 2 usa archivos de lanzamiento en Python. A continuación se muestra un ejemplo básico:

```
from launch import LaunchDescription
from launch_ros.actions import Node

def generate_launch_description():
    return LaunchDescription([
        Node(
            package=my_py_pkg,
            executable=minimal_publisher,
            name=publisher_node,
            output=screen
        ),
    ])
```

12.2 Ubicación y ejecución

El archivo debe colocarse en la carpeta `launch/` dentro del paquete. Para ejecutarlo:

```
ros2 launch my_py_pkg nombre_del_archivo.launch.py
```

13 Conectividad e Integracion con MQTT y ESP32

13.1 Instalacion de Mosquitto Broker

```
sudo apt update
sudo apt install mosquitto mosquitto-clients
sudo systemctl enable mosquitto
```

13.2 Publicar y suscribirse a un tema MQTT (prueba local)

```
En una terminal:

mosquitto_sub -h localhost -t test/topic

En otra terminal:

mosquitto_pub -h localhost -t test/topic -m "Hola␣desde␣ROS␣2"
```

13.3 Codigo ESP32 para publicar por MQTT

Este ejemplo en MicroPython permite que el ESP32 lea temperatura y humedad desde un sensor DHT22 y publique los datos al broker MQTT local.

```
import network
import time
from umqtt.simple import MQTTClient
import dht
import machine

Configurar red WiFi

ssid = NOMBRE_RED
password=CONTRASENA

sta = network.WLAN(network.STA_IF)
sta.active(True)
sta.connect(ssid, password)
while not sta.isconnected():
time.sleep(1)

Sensor DHT22 en GPIO 15

d = dht.DHT22(machine.Pin(15))

Conexion MQTT

client = MQTTClient(esp32_1, 192.168.1.100)  # IP del broker (ROS 2 o PC
    local)
client.connect()

while True:
d.measure()
t = d.temperature()
h = d.humidity()
payload = f"{{temp:␣{t},␣hum:␣{h}}}"
client.publish(sensor/ambiente, payload)
time.sleep(5)
```

13.4 Recepcion de datos MQTT desde ROS 2 con Python

Puedes usar paho-mqtt para recibir los datos en un nodo ROS 2.

```
import rclpy
from rclpy.node import Node
from std_msgs.msg import String
import paho.mqtt.client as mqtt

class MQTTBridge(Node):
def init(self):
super().init(mqtt_bridge)
self.publisher_ = self.create_publisher(String, mqtt_data, 10)
self.mqtt_client = mqtt.Client()
self.mqtt_client.on_connect = self.on_connect
self.mqtt_client.on_message = self.on_message
self.mqtt_client.connect(192.168.1.100, 1883, 60)
```

```
self.mqtt_client.loop_start()

def on_connect(self, client, userdata, flags, rc):
    self.get_logger().info(Conectado al broker MQTT)
    client.subscribe(sensor/ambiente)

def on_message(self, client, userdata, msg):
    self.get_logger().info(f"Mensaje recibido: {msg.payload.decode()}")
    ros_msg = String()
    ros_msg.data = msg.payload.decode()
    self.publisher_.publish(ros_msg)

rclpy.init()
node = MQTTBridge()
rclpy.spin(node)
node.destroy_node()
rclpy.shutdown()
```

```
my_py_pkg/
  __init__.py
  module1.py
  module2.py
```