Taller 2: Introducción a ROS (Robot Operating System)

2.1 Objetivos

- Conocer el origen de ROS.
- Reconocer el concepto de ROS.
- Identificar la estructura de ROS.
- Desarrollar el primer programa en ROS de la tortuga.

2.2 Marco conceptual

2.2.1 Historia de ROS (Robot Operating System)

ROS nació en la Universidad de Stanford entre los años 2006 y 2007. Sus fundadores fueron Keenan Wyrobek y Eric Berger, estudiantes de Stanford. ROS nace a partir de que la robótica dedicaba demasiado tiempo a reimplementar controladores, comunicación y software base, debiendo tener un conocimiento avanzado del hardware y la electronica del mismo, por lo que quedaba poco tiempo para desarrollar algoritmos verdaderamente inteligentes. La solución inicial fue crear una infraestructura de software reutilizable para acelerar el desarrollo robótico. ROS se desarrolló durante seis años en la empresa de desarrollo y patrocinio Willow Garage, alcanzando gran popularidad y convirtiéndose en un estándar académico. Algunas de las distribuciones iniciales (2010–2012) fueron Box Turtle, C-Turtle, Diamondback, Electric, Fuerte y Groovy. En el año 2011 se lanzó el Turtlebot, que fue un robot económico que democratizó ROS. En 2013, ROS pasó a ser desarrollado por la Open Source Robotics Foundation (OSRF), lanzando nuevas distribuciones como : Hydro (2013), Indigo (2014), Jade (2015), Kinetic (2016), Lunar (2017), Melodic (2018) y Noetic (2020). Noetic marca el cierre de ROS 1 y la transición completa hacia ROS 2 que incorpora la comunicación DDS (Data Distribution Service; ver Fig. 2.1.

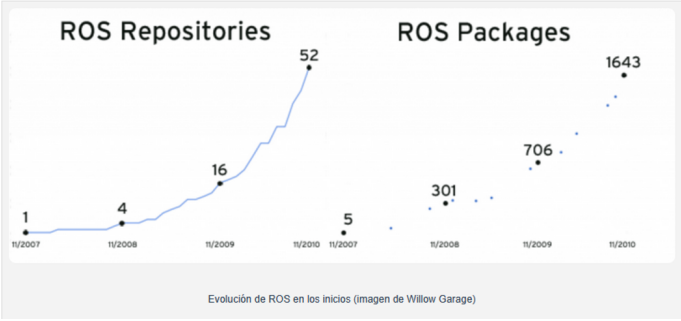

Figure 2.1: Evolución de ROS

ROS 2 buscaba superar las limitaciones de ROS 1 (falta de seguridad, tiempo real y

robustez para productos comerciales). Desde el año 2015, han existido varias distribuciones de ROS 2. Además, en el año 2017, la OSRF pasó a ser Open Robotics, con presencia global. Grandes empresas como Amazon, Google y Microsoft apoyan este ecosistema. ROS consolida su posición como estándar abierto para la robótica, con un fuerte respaldo académico e industrial, convirtiéndose casi en un estándar.

2.2.2 Que es ROS (Robot Operating System)

Tradicionalmente, los especialistas en robótica debían programar y, al mismo tiempo, resolver problemas de hardware, por lo que muchos desarrolladores de software evitaban la robótica debido a la fuerte dependencia entre el hardware y el software. Cada robot requería aprender la API y la electrónica propietaria del fabricante y programar desde cero, lo que limitaba la colaboración y la innovación; ver Fig. 2.2.

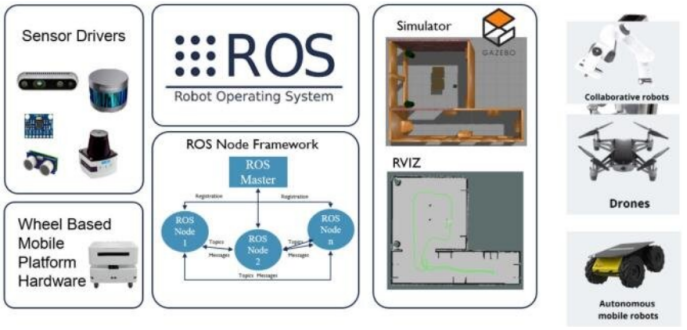

Figure 2.2: Evolución de ROS

ROS surge como un estándar que permite solventar esta limitación. ROS (Robot Operating System) es un marco de software, no un sistema operativo real, que se ejecuta sobre Linux, MacOS, Windows y Docker. Permite abstraer el hardware del software, separando completamente la programación de robots de la interacción directa con su electrónica. Este framework proporciona una API estandarizada basada en topics, services, messages y actions, que facilita el acceso a sensores y actuadores, el intercambio de datos entre módulos de software y la reutilización de código, sin importar el fabricante del robot. En la actualidad, la tendencia es "ROSificar" un robot, es decir, adaptarlo para que funcione con ROS y sea compatible con herramientas y paquetes existentes.

2.2.3 Ventajas de ROS

- Facilita la entrada de nuevos desarrolladores al campo de la robótica sin requerir conocimientos profundos de electrónica o mecánica.
- Permite simular robots y probar software sin hardware físico.
- Estandariza la programación robótica de manera similar a como Windows estandarizó las PC y Android estandarizó los teléfonos.
- Gran ecosistema de paquetes y herramientas, con miles de bibliotecas de código abierto disponibles y listas para usar.
- Favorece la colaboración académica e industrial, acelerando la innovación y reduciendo los costos de desarrollo.

2.2.4 Campos de aplicación de ROS

- Robots de servicio: adoptados ampliamente por universidades, startups y empresas, convirtiéndose en el estándar educativo y de investigación.
- ROS-Industrial: iniciativa internacional para aplicar ROS a la robótica industrial, ofreciendo software abierto para robots de fabricantes tradicionales.
- ROS-Agriculture: comunidad global que promueve el uso de ROS en robots agrícolas, como tractores autónomos y maquinaria pesada.

Entre algunos de los factores de éxito que han permitido el posicionamiento de ROS se encuentran los Tutoriales estructurados y las herramientas de depuración, que facilitan el aprendizaje; sistemas eficientes de gestión de paquetes, que permiten usar y compartir módulos fácilmente; el crecimiento acelerado del ecosistema y proyectos educativos, "como el "ROS Curriculum for the Campus" (2018), utilizado por cientos de universidades, consolidando a ROS como una habilidad esencial para ingenieros y desarrolladores robóticos.

2.2.5 Versionamiento y lenguaje

Hay que tener en cuenta que se necesita saber de antemano cómo programar en C++ o Python para empezar con ROS. Además, se debe conocer el shell de Linux; así como el despliegue de contenedores. En esta clase se tomara como referencia python como lenguaje base; luego se podrá proceder con C++, ya que es el lenguaje utilizado en la industria robótica. Además, se debe tener un conocimiento en el manejo de Docker, donde se llevarán a cabo los ejemplos prácticos.

También usaremos ROS en su version 2, ya que es el estándar actual para la robótica, con mayor presencia en la industria y con soporte a la fecha presente.

2.3 Introducción al funcionamiento de ROS

2.3.1 Comunicación entre nodos ROS

ROS utiliza una arquitectura distribuida donde cada nodo es un proceso independiente que puede ejecutarse en diferentes dispositivos dentro de una misma red. La comunicación entre nodos (por ejemplo, sensor_node, controller_node) se realiza mediante:

- **Topics** (publicadores/suscriptores). Para el intercambio continuo de datos.
- **Services** (cliente/servidor). Usado para peticiones y respuestas puntuales.
- **Actions**. Para tareas que toman tiempo (como moverse a una posición).

En la capa de transporte (ver Fig. 2.3), ROS utiliza TCPROS (sobre TCP/IP, garantiza la entrega). UDPROS (sobre UDP/IP, sin verificación de entrega. En la capa de red, Los nodos se conectan mediante direcciones IP y puertos dinámicos asignados por el Master ROS.

El ROS Master (nodo central) coordina las conexiones y mantiene el registro de quién publica o se suscribe a cada topic. No transmite los datos; solo facilita el handshake (inicio de sesión en la capa de sesión).

Finalmente, los datos se empaquetan utilizando formatos propios de ROS (std_msgs, geometry_msgs, etc. `https://wiki.ros.org/common_msgs?distro=noetic`). Estos

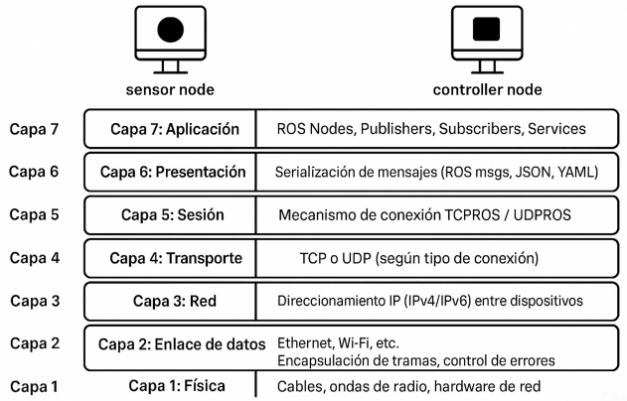

Figure 2.3: Comunicación entre nodos ROS

mensajes se serializan antes de enviarse por la red y se deserializan al recibirlos. En la capa de aplicación es donde se ejecuta la lógica del robot: percepción, control, navegación, etc.

La Fig. 2.3 resume los procesos que ejecuta ROS en las capas del modelo OSI. El Nodo 1 – Sensor (Publisher) publica la temperatura en el topic (temp_data) y el Nodo 2 – Controlador (Subscriber) se suscribe al topic (temp_data) y toma decisiones.

- Capas 7–6: El nodo sensor genera el mensaje ROS (por ejemplo, Float32 = 25 °C).
- Capa 5: ROS Master establece la conexión entre ambos nodos.
- Capas 4–3: El mensaje viaja por TCP/IP.
- Capas 2–1: El dato se transmite por la red física (Wi-Fi o Ethernet).

En las nuevas versiones de ROS2, en lugar de que los nodos envíen mensajes directamente, se usa DDS (Data Distribution Service) (ver Fig. 2.4). DDS es un middleware de comunicación estandarizado por la OMG (Object Management Group). DDS crea una infraestructura de publicación–suscripción donde su función es manejar la comunicación en tiempo real, distribuida y basada en datos entre múltiples nodos (programas o dispositivos).

Los publicadores envían datos a un tópico. Los suscriptores se suscriben a ese tópico y reciben los datos automáticamente. No necesitan conocerse entre sí, lo que hace que el sistema sea desacoplado, escalable y tolerante a fallos.

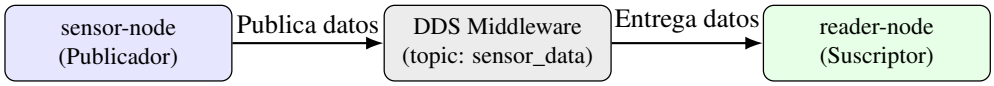

Figure 2.4: Flujo de comunicación en ROS 2 mediante DDS. El nodo `sensor-node` publica datos en el tópico `sensor_data`, gestionado por el middleware DDS, que los entrega al nodo `reader-node`.

2.3.2 Estructura de ROS

Para empezar a familiarizarnos con ROS, debemos tener en cuenta la estructura para iniciar un proyecto. Aparecen elementos como Workspace, topics, actions y message, los cuales se describen a continuación:

1. **WORKSPACE**: Es el directorio de trabajo donde se desarrollara todo el código ROS2. El directorio contiene todos los paquetes y programas que se requieren para el proyecto. Su función principal es organizar el código para compilarlo e integrarlo con ROS 2. La Fig. 2.5 muestra la estructura estandar del WOrkspace.

```
ros2_ws/
├── src/       ← Aquí van tus paquetes
├── install/   ← Archivos instalados al compilar
├── build/     ← Archivos intermedios de compilación
└── log/       ← Archivos de registro (logs)
```

Figure 2.5: Estructura de Workspace

2. **NODOS**: Es cada programa individual en ROS2 que tiene como objetivo una función concreta (cada nodo es un proceso independiente), como leer un sensor, procesar la información y controlar un actuador. Cada nodo publica o se suscribe a mensajes. La comunicación entre nodos se realiza a través de topics o servicios. Por ejemplo, un nodo ***sensor-node.py*** tiene la única función de publicar la temperatura cada segundo.
3. **COMUNICACIÓN**: los nodos requieren un canal de comunicación para publicar y suscribirse a mensajes. Por ejemplo, se puede tener el tópico `/temperatura`, que publica un dato de tipo `Float32`, o el tópico `/alarma`, que publica un dato de tipo `Bool` con valor true/false.
4. **MENSAJES** : Es la estructura de datos que viaja entre nodos. ROS 2 utiliza mensajes por defecto de tipo (std-msgs/String, std-msgs/Float32 , etc `https://wiki.ros.org/common_msgs?distro=noetic`), y también tiene mensajes de tipo personalizado.
5. **ACCIÓN**: Es un mecanismo para manejar tareas que toman tiempo en completarse y requieren retroalimentación continua mientras se ejecutan. Difieren de los SERVICES en los que la respuesta es inmediata o de los TOPICS que transmiten datos continuamente, sin saber cuando terminan. Los componentes de una acción definen tres tipos de mensajes:
   - **Goal**: indica lo que se quiere hacer.
   - **Feedback**: actualizaciones periódicas mientras se ejecuta la tarea.
   - **Result**: estado final cuando la tarea termina.

2.4 Desarrollo

2.5 Parte I: Red de sensores en ROS

El problema consiste en emular una red de sensores mediante la creación de una red de nodos. Para ello, se implementa un bucle en el que un nodo genera datos y otro los consume,

Imagen Docker	Contenido principal	Uso recomendado
`osrf/ros:jazzy-ros-core`	Librerías mínimas de ROS 2 (middleware DDS, `rclcpp`, `rclpy`, herramientas básicas).	Sistemas embebidos o contenedores ligeros donde se agregan solo los paquetes necesarios.
`osrf/ros:jazzy-ros-base`	Incluye `ros-core` + herramientas de compilación (`colcon`, `rosdep`, `launch`, etc.) + comunicación entre nodos.	Desarrollo y ejecución de nodos sin interfaz gráfica. Buen equilibrio entre peso y funcionalidad.
`osrf/ros:jazzy-desktop`	Incluye `ros-base` + herramientas gráficas (`rviz2`, `rqt`, `gazebo_ros_pkgs`, etc.).	Desarrollo completo con visualización y simulación en entornos de escritorio.

Table 2.1: Comparación de las imágenes Docker oficiales de ROS 2 Jazzy

de forma análoga a un sistema real con sensores (véase la Fig. 2.6). El nodo denominado `nodo-sensor` publica información de temperatura en el tópico `sensor-data`, mientras que el `nodo-reader` se suscribe a dicho tópico y muestra cada valor recibido.

- **Nodo-sensor**: Publica datos, simula un sensor de temperatura, posición, etc.
- **Nodo-reader**: se suscribe y se muestran los datos publicados. Ha de generar su propio workspace y nodos.

La Tabla 2.1 muestra un resumen de las principales imágenes de ROS 2 versión Jazzy que tenemos disponibles en Docker Hub. En lugar de instalar ROS 2 directamente en el sistema Ubuntu; que implica la instalación de una serie de bibliotecas y versiones precisas de paquetes, aprovechamos el uso de contenedores. Usaremos la imagen oficial de ROS 2 publicada por la OSRF (Open Source Robotics Foundation), que mantiene las imágenes de ROS/ROS 2 en DockerHub (ver Tabla 2.1). Este tiene todas las librerías probadas y validadas; por lo tanto, el proceso de puesta en marcha es mucho más rápido. Usaremos la versión Jazzy Jalisco, que es una distribución de ROS 2 lanzada en mayo de 2024, con soporte a largo plazo (LTS). Su soporte se extiende hasta mayo de 2029 y es compatible con la versión de Ubuntu 24.04 Noble.

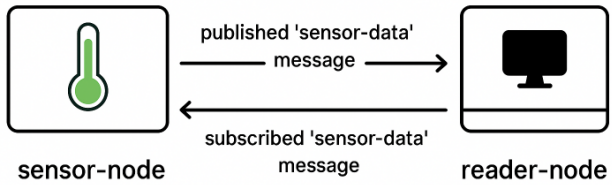

Figure 2.6: Red de sensores en ROS

2.5.1 Desarrollo: Parte II

Reporte en el informe

Reporte los pasos [1-8], que incluyen el proceso de preparación del contenedor, la creación del workspace y sus estructuras, así como la implementación de los nodos y su ejecución. Recuerde compartir el código en GitHub y utilizar pseudocódigo en el informe para explicar el funcionamiento del sistema. Además, compartir el enlace de las imagenes de Docker creadas en el Docker Hub.

1. **Preparando el contenedor**:
   - Primero, debemos ejecutar el contenedor de ROS 2, al que llamaremos ***ros_ws***, con acceso a una terminal interactiva ***bash***. Note que usaremos la imagen ***osrf/ros:jazzy-desktop***.
   - En segundo lugar, es necesario instalar el paquete ***python3-colcon-common-extensions***, el cual amplía las funcionalidades de la herramienta de compilación ***colcon*** de ROS 2. Estas extensiones permiten compilar distintos tipos de paquetes (como CMake o Python) dentro de un workspace. Su instalación es fundamental, ya que posibilita la definición de nodos y la ejecución del programa utilizando el lenguaje Python.
   - Finalmente, activamos la ubicación de las fuentes ROS 2 para compilar el proyecto.

```
# Ejecutamos contenedor:
docker run -it --name ros2_ws osrf/ros:jazzy-desktop bash

# Instalamos herramientas de compilacion:
apt update && apt upgrade -y && apt install -y python3-colcon-common-
    extensions nano

# Activamos la ubiacion de las fuentes de ROS2
#source /opt/ros/jazzy/setup.bash
```

2. **Creando el Workspace**: Dentro del contenedor, creamos la estructura básica para el proyecto ***ros2_ws*** donde se alojaran todos los archivos necesarios.

```
# Creamos el Workspace
```

```
mkdir -p ~/ros2_ws/src
cd ~/ros2_ws/src
```

3. **Creando estructuras y paquetes**: Dentro del contenedor, creamos el workspace de los archivos del proyecto con ***ros2 pkg create***. Las opciones del comando incluyen:

   - **ros2 pkg create**: Comando de ROS 2 para crear un nuevo paquete.
   - **–build-type ament_python**: Indica que el paquete se construirá usando ament con soporte para Python. Esto significa que será un paquete ejecutable de Python, en lugar de un paquete de CMake/C++.
   - **sensor_program**: Es el nombre del paquete que se está creando. Este será el nombre que usaremos para importar módulos o para ejecutar el nodo con ros2 run.

```
# Creamos el Workspace
ros2 pkg create --build-type ament_python sensor_program --license MIT
```

   El comando anterior crea la siguiente estructura del proyecto ***ros2_ws***:

```
# Se crea una estructura similar a la siquiente:
sensor_program/
      package.xml
      resource
         --sensor_program
      setup.py
      setup.cfg
      sensor_program/
         --__init__.py
      test
```

   La Tabla 2.2 resume la estructura generada conformada por archivos y carpetas esenciales para la compilación y ejecución del proyecto. El archivo ***package.xml*** define los metadatos y dependencias del paquete, mientras que ***setup.py*** configura el proceso de instalación y el registro de los nodos ejecutables. El directorio ***resource*** Sirve para que ament (el sistema de construcción de ROS 2) reconozca este paquete al instalarlo y resolver recursos (no borrar). La carpeta ***sensor_program***, que lleva el mismo nombre del paquete, contiene el código fuente en Python, incluyendo los nodos que implementan la funcionalidad del sistema y el archivo ***__init__.py*** permite que esta carpeta sea reconocida como un módulo de Python (paquete importable). El ***setup.cfg*** define dónde instalar los scripts y archivos.Por último la carpeta ***test*** contiene herramientas para verificar la calidad y el estilo del código.

4. **Creando el Nodo-Sensor**: Para desarrollar los nodos, utilizaremos el lenguaje Python. Vamos a simular un sensor de temperatura que emite valores aleatorios de temperatura cada segundo.
   - Creamos el archivo Python ***sensor_node.py*** en el que definimos la operación del sensor.

```
# Creamos el archivo python node-sensor
nano ~/ros2_ws/src/sensor_program/sensor_program/sensor_node.py
```

Table 2.2: Estructura generada por `ros2 pkg create`

Elemento	**Descripción**	**Qué se debe modificar**
`package.xml`	Archivo de metadatos del paquete. Contiene información como el nombre, versión, autor, licencia y dependencias necesarias (por ejemplo, `rclpy`, `std_msgs`). Es utilizado por ROS 2 para la gestión de dependencias y compatibilidad.	Actualizar los campos de autor y descripción. Agregar las dependencias necesarias usando etiquetas `<exec_depend>`.
`setup.py`	Archivo de configuración e instalación del paquete Python. Define cómo se construye el paquete, qué módulos se incluyen y qué scripts se ejecutan. Permite que `colcon` instale correctamente los nodos y dependencias.	Modificar el nombre del paquete, descripción y la lista de `console_scripts` para registrar los nodos creados.
`sensor_program/`	Carpeta principal del código fuente del paquete. Contiene los nodos, módulos y scripts en Python que implementan la funcionalidad del sistema.	Crear o modificar los archivos de los nodos dentro de esta carpeta (por ejemplo, `sensor_node.py`).
`__init__.py`	Archivo que indica a Python que la carpeta `sensor_program/` es un módulo o paquete importable. Generalmente está vacío, aunque puede incluir funciones o variables de inicialización.	Normalmente no requiere cambios, a menos que se necesite inicializar variables o importar funciones globales.

- Creamos la lógica del Nodo-sensor: El algoritmo 1 refleja de manera simplificada la lógica del código Python para el nodo sensor, mostrando la secuencia de inicialización de un nodo ROS2, la publicación periódica de datos y cierre del nodo.

  En el algoritmo 1, (i) primero, se inicializa el sistema ROS 2 (rclpy.init(args=args)) y se crea un nodo llamado ***sensor_node*** junto con un publicador en el tópico ***sensor_data*** para enviar mensajes de tipo ***String***. (ii) A continuación, mediante un bucle ***while true***, el nodo genera cada segundo un valor aleatorio de temperatura entre 20 y 30 grados (rand_num[20,30]), construye un mensaje con dicho valor, lo publica en el tópico y registra la información en la consola. (iii) Finalmente, cuando se detiene el nodo, se destruye y se apaga ROS 2 correctamente, asegurando una terminación ordenada del programa.

Algorithm 1 Node-Sensor en ROS 2

```
InicializarROS2();
nodo = CrearNodo(sensor_node);
publisher = CrearPublicador(sensor_data, String, cola=10);
while true do
    esperar 1 segundo;
    temperatura = rand_num[20,30] ;
    mensaje = "Temperatura: " + temperatura + "grados C";
    publisher.publish(mensaje);
    RegistrarEnConsola("Publicando: " + mensaje);
MantenerNodoActivo(nodo);
DestruirNodonodo;
ApagarROS2();
```

A continuación, se presenta un ejemplo del código del ***sensor_node.py*** :

```python
# Sensor-Node

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

class SensorNode(Node):
    def __init__(self):
        super().__init__('sensor_node')
        self.publisher_ = self.create_publisher(String, 'sensor_data',
            10)
        self.timer = self.create_timer(1.0, self.publish_data)

    def publish_data(self):
        msg = String()
        msg.data = f"Temperatura: {random.randint(20,30)} C"
        self.publisher_.publish(msg)
        self.get_logger().info(f' Publicando: {msg.data}')

def main(args=None):
```

```
    rclpy.init(args=args)
    node = SensorNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

5. **Creando el Nodo Lector (reader)**: Este nodo lee la información publicada del nodo-sensor y la muestra en pantalla. en futuros proyectos, se pueden realizar acciones con esta información, como alarmas o el encendido y apagado de dispositivos.
   - Creamos el archivo Python ***reader_node.py*** en el que definimos la operación del sensor reader.

     ```
     # Creamos el archivo python node-reader
     nano ~/ros2_ws/src/sensor_program/sensor_program/reader_node.py
     ```

   - Creamos la lógica del Nodo-reader: El algoritmo 2 refleja la lógica principal de la suscripción y manejo de mensajes en ROS 2.

     En el algoritmo 2, el reader_node está encargado de recibir y mostrar los datos publicados por otros nodos, como el nodo sensor. (i) Primero, se inicializa el sistema ROS 2 y se crea el nodo. (ii) A continuación, el nodo se **suscribe** al tópico **sensor_data** definiendo un callback (listener_callback) que **se ejecuta cada vez que llega un mensaje**; en este callback, el nodo registra en la consola el contenido del mensaje recibido almacenando hasta 10 msg en buffer. (iii) Finalmente, el nodo permanece activo mientras ROS 2 esté corriendo y, al finalizar, se destruye el nodo y se apaga el sistema de manera ordenada.

Algorithm 2 Nodo Lector en ROS 2

```
InicializarROS2()
nodo := CrearNodo(reader_node)
suscripcion := CrearSuscripcion(sensor_data, String, callback=listener_callback, cola=10)
Función listener_callback(mensaje):
  RegistrarEnConsola("Recibido: " + mensaje)
MantenerNodoActivo(nodo)
DestruirNodonodo
ApagarROS2()
```

A continuación, se presenta un ejemplo del código del ***reader_node.py***:

```
# Nodo de Lector

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

class ReaderNode(Node):
    def __init__(self):
```

```
        super().__init__('reader_node')
        self.subscription = self.create_subscription(
            String,
            'sensor_data',
            self.listener_callback,
            10)
        self.subscription

    def listener_callback(self, msg):
        self.get_logger().info(f' Recibido: {msg.data}')

def main(args=None):
    rclpy.init(args=args)
    node = ReaderNode()
    rclpy.spin(node)
    node.destroy_node()
    rclpy.shutdown()

if __name__ == '__main__':
    main()
```

6. **Agregar los nodos al setup.py**: Modifique el archivo ***setup.py*** para configurar los nodos creados (nodo sensor y nodo reader). Dentro de dicho archivo ubique la funcion ***entry_points***, para agregar los nuevos nodos, como se muestra en el siguiente código.

```
# Dentro del archivo setup.py
entry_points={
    'console_scripts': [
        'sensor_node = sensor_program.sensor_node:main',
        'reader_node = sensor_program.reader_node:main',
    ],
},
```

7. **Compilar el proyecto con** `Colcon`: Vamos a utilizar la herramienta Colcon para compilar nuestro proyecto. Colcon es una herramienta de ROS2 que proporciona una forma estandarizada de construir y gestionar paquetes de ROS 2.

```
# Nos ubicamos en la raiz del Workspace
cd ~/ros2_ws
colcon build

#  Cargamos el entorno de nuestro Workspace
source install/setup.bash
```

8. **Ejecutar los nodos del proyecto**: Primero, ejecutaremos el nodo que simula el sensor (***sensor_node.py***) y, después, el nodo reader (***reader_node.py***). Note que para ejecutar el segundo nodo, debemos abrir otra terminal de nuestro contenedor.

   Para ejecutar el nodo sensor:

```
# Ejecutamos el nodo sensor
ros2 run sensor_program sensor_node
```

   Para ejecutar el nodo reader hemos de abrir otra terminal lanzando el mismo contenedor (***ros2_ws***). Para ello, usaremos el comando `docker exec` y deberemos cargar nuevamente el entorno de ROS 2 en el sistema (***source /opt/ros/jazzy/setup.bash***), y cargar

nuestros propios desarrollos locales en ROS 2 (***source /ros2_ws/install/setup.bash***) para que se puedan ejecutar desde la terminal sin problemas de rutas o variables de nuestro proyecto.

```
# Ejecutamos el nodo reader en otra termina
docker exec -it ros2_ws bash          # ejecuta el comando bash en el
    contenedor activo
source /opt/ros/jazzy/setup.bash      # cargar entorno ROS2
source ~/ros2_ws/install/setup.bash   # cargar entorno de nuestro
    proyecto

# Ejecutamos el nodo reader
ros2 run sensor_program reader_node
```

9. **Modifique y compile nuevamente el proyecto ros2_ws**: Para este último punto, agregue un nuevo nodo reader 2 que se subscriba al tópico ***sensor_data***. Ejecute ambos nodos subscriptores y verifique su funcionamiento.

2.6 Actividad Reto

Reporte en el informe

Partiendo del ejemplo anterior, desarrolle un programa que incluya un nodo sensor encargado de publicar una temperatura simulada y un nodo lector que se suscriba a dicho sensor para recibir y procesar esa información.

Reporte la configuración de los siguientes desafíos del reto:

1. ***Dockerfile***: Desarrolle un archivo Dockerfile que automatice la mayor parte de los pasos [1-8]. Verifique la funcionalidad del script ***ros_entrypoint.sh*** que viene en el contenedor oficial. Ha de ejecutar este script antes de cualquier cambio.
2. ***Shared folder***: Cree una carpeta compartida entre su computadora y el contenedor. En esta carpeta, coloque los archivos sensor_node.py y reader_node.py. Al iniciar el contenedor, podrá copiar estos archivos a su ubicación definitiva; aunque también podrá editarlos directamente desde su IDE de programación favorito, lo que facilita su edición y desarrollo. Explique el proceso en detalle.
3. **Verificar** que todos los nodos estén funcionando: Todos los nodos deben estar ejecutándose y mostrando la información de temperatura.
4. **Cree un nodo adicional**, llamado ***plotter_node***, que se encargue de generar un gráfico de los datos recibidos por el nodo reader. Este nodo realiza el gráfico cada 5 segundos y guarda la imagen resultante en el volumen compartido, por ejemplo: ***plt.savefig('/ros2_ws/data/sensor_plot.png')***.