Estudiantes:
Axel Javier Rosas Rodríguez | A01738607
Elian Cantalapiedra Sabugal | A01738462
Edwin Emmanuel Salazar Meza | A01738380

Profesor:
Rigoberto Cerino Jiménez

Nombre de la Institución:
Instituto Tecnológico y de Estudios Superiores de
Tecnológico de Monterrey

Materia:
Fundamentación de la Robótica

Fecha:
18 de Febrero de 2026

Título del trabajo:
Reto semanal 1. Manchester Robotics

Resumen

En este reto se desarrollaron dos nodos en ROS2 con el objetivo de generar y procesar señales en tiempo real. El primer nodo debe funcionar como generador de una señal sinusoidal, publicando datos periódicamente en un tópico. El segundo nodo actúa como procesador, suscribiéndose a la señal original para modificarla y publicar una señal procesada en un nuevo tópico.

Ambas señales fueron visualizadas utilizando la herramienta, permitiendo comparar la señal original y la señal procesada en una misma ventana.Se creará un archivo launch que permite ejecutar automáticamente los dos nodos y la herramienta de visualización simultáneamente, facilitando la depuración y el análisis del sistema.

Objetivos

Objetivo General:

- Desarrollar e integrar dos nodos en ROS2 que generan y procesan una señal sinusoidal, visualizando sus resultados en tiempo real.

Objetivos específicos:

- Implementar un nodo generador de señal sinusoidal utilizando Python y NumPy que publique en dos tópicos el tiempo y la señal generada.
- Implementar un nodo procesador que se suscriba a la señal recibida y la publique una vez modificada.
- Visualizar las señales generadas y procesadas utilizando rqt_plot.
- Verificar el correcto funcionamiento del sistema mediante rqt_graph

Introducción

En los sistemas robóticos modernos, la comunicación entre procesos se realiza comúnmente mediante arquitecturas basadas en nodos, como las implementadas en ROS2. Ya que ROS 2 (Robot Operating System 2) se presenta no como un sistema operativo tradicional, sino como un middleware de código abierto diseñado para facilitar la interoperabilidad en aplicaciones robóticas, su arquitectura se basa en un sistema distribuido que utiliza el estándar DDS (Data Distribution Service), lo que garantiza una comunicación robusta, escalable y con capacidades de tiempo real.

Estructura Básica y Red de Comunicación

Para comprender el funcionamiento de ROS 2, es necesario identificar los elementos principales que integran su red de comunicación:

- **Nodos (Nodes):** Son la unidad mínima de ejecución en ROS. Cada nodo es un proceso independiente encargado de una tarea específica (por ejemplo, leer un

sensor o controlar un motor). Siendo que en este proyecto, se implementaron dos nodos, un generador y un procesador.

- **Tópicos (Topics):** Funcionan como canales de comunicación bajo un modelo de Publicación/Suscripción. Un nodo publica datos en un tópico determinado, mientras que otros nodos interesados se suscriben a él para recibir dicha información sin necesidad de que los nodos se conozcan entre sí directamente.

- **Mensajes (Messages):** Es el formato de datos estructurado que define qué tipo de información viaja a través de los tópicos (enteros, flotantes, strings, etc.).

- **Grafo de ROS (ROS Graph):** Es la red formada por la interacción de todos estos elementos simultáneamente.

En este reporte, se detalla la creación de un grafo de ROS 2 donde un nodo generador publica una señal sinusoidal, la cual es interceptada, transformada y re-publicada por un nodo procesador. Este flujo de datos permite analizar la eficiencia de la comunicación inter-procesal y la capacidad de visualización de datos en tiempo real mediante herramientas de diagnóstico como rqt_plot y rqt_graph.

Metodología

1. **Configurar el entorno**
   Configurar el workspace y crear un paquete en Python dentro de ROS 2, asegurando la inclusión de las dependencias necesarias (rclpy, std_msgs, numpy).

2. **Diseño de la arquitectura del sistema**
   Definir la interacción entre los nodos:
   - signal_gen como publicador de time y signal.
   - signal_proc como suscriptor y publicador de proc_signal.

3. **Implementación del nodo generador**
   Programar el cálculo de la señal sinusoidal mediante un Timer periódico. Publicar en cada ciclo el tiempo actual y el valor de la señal en sus respectivos tópicos.

4. **Implementación del nodo procesador**
   Configurar las suscripciones a “time” y “signal”. Procesar la señal aplicando un desfase, un offset y una reducción de amplitud. Publicar el resultado en proc_signal.

5. **Integración y ejecución conjunta**
   Crear un archivo launch para ejecutar simultáneamente ambos nodos y la herramienta de visualización rqt_plot.

6. **Validación del sistema**
   Verificar la correcta conexión entre nodos con rqt_graph y comparar gráficamente la señal original y la procesada en tiempo real.

Solución del problema

Nodo signal_gen

```python
#Imports
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32
import numpy as np

# Class Definition
class SignalGenerator(Node):
    def __init__(self):
        super().__init__('signal_gen')
        self.publisher_time = self.create_publisher(Float32, 'time', 10)
        self.publisher_signal = self.create_publisher(Float32, 'signal', 10)

        # Parámetros
        self.f = 10.0 # Hz
        self.A = 1.0 # Amplitud
        self.omega = 2 * np.pi * self.f
        self.t = 0.0 #

        #Timer
        timer_period = 0.00001 # Segundos
        self.timer = self.create_timer(timer_period, self.timer_cb)
```

Se define el nodo **SignalGenerator**, después con **self.create_publisher** se define dos canales de salida (tópicos). Uno llamado “time” y otro “signal”. Ambos usan el tipo de mensaje Float32.

Se establecen los parámetros de la señal, donde se define una frecuencia de 10 Hz (f) y una amplitud de 1.0 (A). Calcula la frecuencia angular como $\varpi = 2\pi f$

El Timer nos dice que se ejecuta cada 0.00001 segundos. Cada vez que pasa ese tiempo, se dispara automáticamente la función **timer_cb**.

Función Callback

En la función callback se instancia que los mensajes son objetos de tipo Float32().

Luego se calcula el valor de la señal usando la fórmula: $y(t) = A * sin(\varpi * t)$

Luego se envía el tiempo actual y el valor calculado a sus respectivos tópicos mediante **self.publisher_time.publish()** y **self.publisher_signal.publish()**

```python
# Timer Callback
def timer_cb(self):

    # Publicar tiempo
    msgT = Float32()
    msgT.data = self.t
    self.publisher_time.publish(msgT)
    self.get_logger().info(f"Tiempo: {msgT.data:.2f}")

    # Publicar señal
    msgS = Float32()
    msgS.data = self.A * np.sin(self.omega * self.t)
    self.publisher_signal.publish(msgS)
    self.get_logger().info(f"Señal: {msgS.data:.2f}")
    self.t += 0.00001
```

Se usa **self.get_logger().info()** para imprimir en la terminal lo que está sucediendo, lo cual es útil para depurar.

Función main

```python
def main(args = None):
    rclpy.init(args = args)
    signal_gen = SignalGenerator()

    try:
        rclpy.spin(signal_gen)
    except KeyboardInterrupt:
        pass
    finally:
        if rclpy.ok():
            rclpy.shutdown()
        signal_gen.destroy_node()

# Execute Node
if __name__ == '__main__':
    main()
```

En la función main, **rclpy.init** es quien inicia las comunicaciones de ROS2 en el script.

rclpy.spin(signal_gen) es una de las partes más importantes ya que mantiene al nodo vivo y escuchando o ejecutando sus timers. Sin esto, el programa terminaría inmediatamente tras crear el nodo.

Y el bloque finally, asegura que el nodo se destruya correctamente y los recursos del sistema se liberen.

Nodo signal_proc

```python
#Imports
import signal
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32
import numpy as np

# Class Definition
class SignalProcessing(Node):
    def __init__(self):
        super().__init__('signal_proc')
        self.subscription_time = self.create_subscription(Float32, 'time', self.time_cb, 10)
        self.subscription_signal = self.create_subscription(Float32, 'signal', self.signal_cb, 10)
        self.publisher_proc_signal = self.create_publisher(Float32, 'proc_signal', 10)

        # Parámetros de procesamiento
        self.f = 10.0 # Hz
        self.omega = 2 * np.pi * self.f
        self.theta = np.pi # Desplazamiento de fase
        self.scale = 0.5 # Escala de amplitud
        self.offset = 1.0 # Offset >= 0.5
        self.t = 0.0 # Tiempo Actual
```

El segundo nodo se configura para escuchar los tópicos que creó el nodo anterior.

self.create_subscription crea dos suscriptores, uno para el tópico “time” y otro para “signal”.

Este nodo funciona por eventos, cada vez que llega un mensaje nuevo a un tópico, se ejecutan automáticamente las funciones **time_cb** o **signal_cb.**

Funciones Callback

time_cb se encarga de actualizar la variable interna **self.t** con el tiempo exacto que viene del generador para mantener la sincronización.

Mientras que en **signal_cb** es donde se realiza el procesamiento de la señal para cada nuevo valor de la onda. Primero, se calcula la derivada teórica de la señal original para poder aplicar una identidad trigonométrica de desfase.

Para aplicar el desfase se utiliza la fórmula: $y = cos(\theta) + \frac{dx}{\omega} * sen(\theta)$.

```python
def signal_cb(self, msgS):
    self.get_logger().info(f'Señal recibida: {msgS.data:.2f}')

    # Procesar señal
    x = msgS.data
    dx = self.omega * np.cos(self.omega * self.t)
    y = x * np.cos(self.theta) + (dx / self.omega) * np.sin(self.theta)
    y = self.scale * y + self.offset # Escala y offset

    # Publicar señal procesada
    msgP = Float32()
    msgP.data = y
    self.publisher_proc_signal.publish(msgP)
    self.get_logger().info(f'Señal procesada: {msgP.data:.2f}')

def time_cb(self, msgT):
    self.get_logger().info(f'Tiempo recibido: {msgT.data:.2f}')
    self.t = msgT.data
```

Los parámetros escogimos para el procesamiento fueron: un desfase de $\pi = 180°$ (invierte la señal), un escalamiento de 0.5 de amplitud y un offset de 1 (lo que permite a la onda oscilar entre valores positivos).

Por último, una vez procesada la señal, el nodo crea un nuevo mensaje de tipo Float32 y lo publica en un tópico distinto llamado **proc_signal**.

Archivo de Lanzamiento (Launch)

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

def generate_launch_description():
    return LaunchDescription([

        Node(
            package='signal_processing',
            executable='signal_gen',
            name='signal_gen',
            output='screen'
        ),

        Node(
            package='signal_processing',
            executable='signal_proc',
            name='signal_proc',
            output='screen'
        ),

        Node(
            package='rqt_plot',
            executable='rqt_plot',
            name='rqt_plot',
            output='screen',
            arguments=['/signal/data', '/proc_signal/data', ]
        )
    ])
```

Se importa **LaunchDescription**, que define el conjunto de acciones a ejecutar, y Node, que permite lanzar nodos individuales.

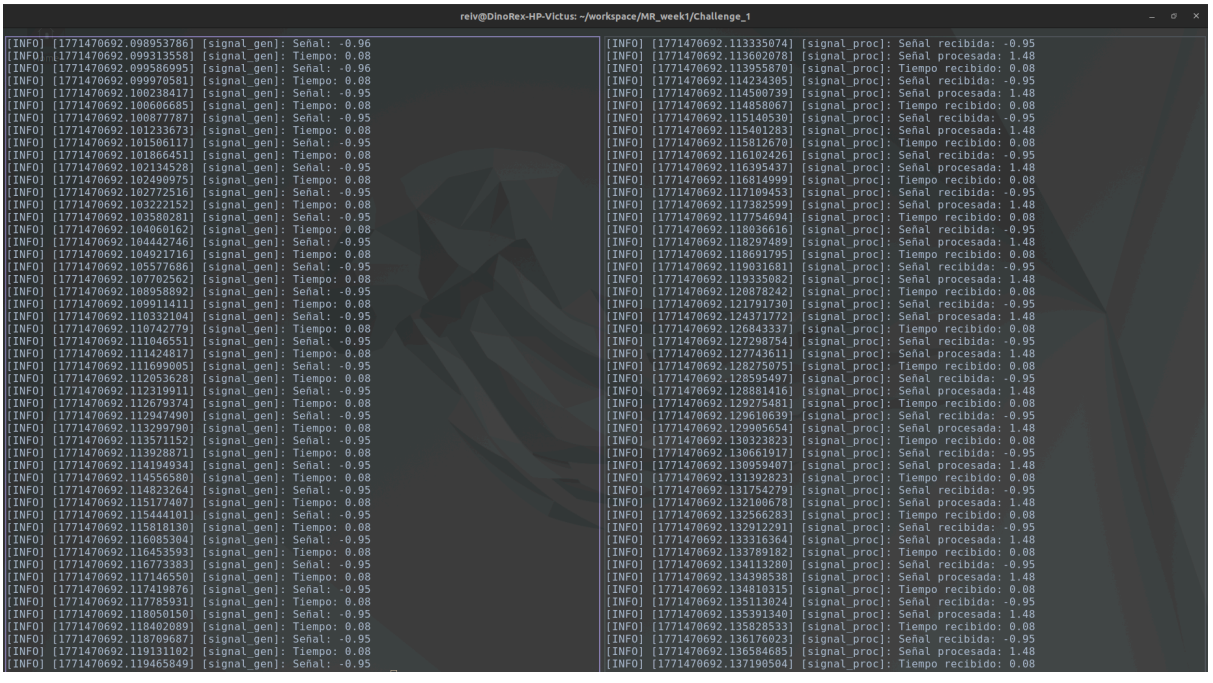
La función **generate_launch_description()** devuelve una lista con tres nodos que se iniciarán simultáneamente.

Los nodos que se ejecutan son **signal_gen**, **signal_proc** (pertenecientes al paquete signal_processing) y **rqt_plot** el cual se ejecuta automáticamente para graficar los tópicos /signal/data y /proc_signal/data, permitiendo visualizar en tiempo real la señal original y la procesada.

Resultados

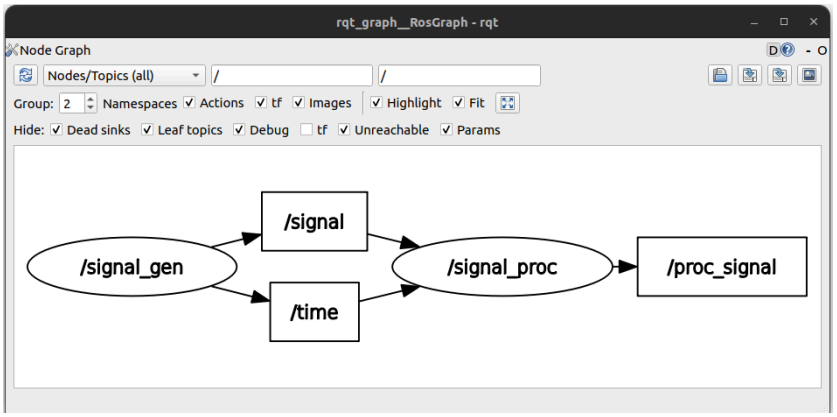
A continuación se mostrarán los resultados obtenidos tras la elaboración y ejecución de los nodos, incluyendo la publicación, procesamiento y visualización de las señales, así como verificar el cumplimiento de la arquitectura solicitada mediante rqt_graph.

- **Resultados del nodo generador de señal y procesamiento de la señal**



Cómo puede visualizarse, el nodo **signal_gen** publica el tiempo y el valor de la señal en ese instante. Por otra parte, el nodo **signal_proc** muestra los valores recibidos así como la señal modificada.

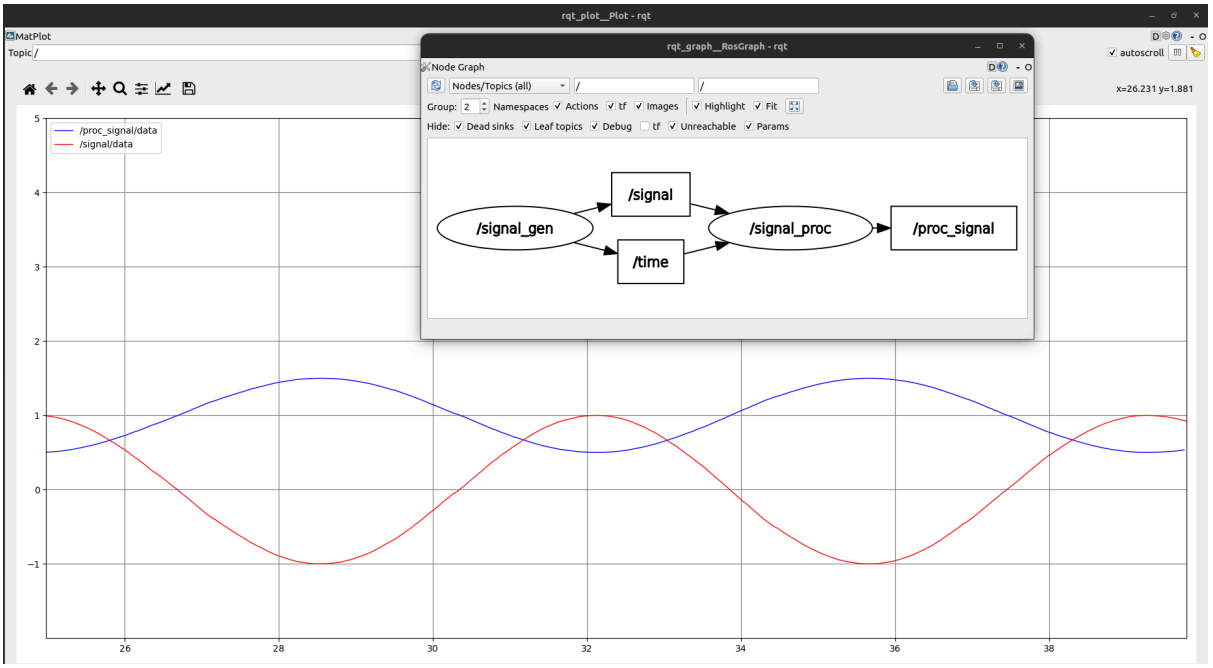
- **Integración de los nodos**



Al ejecutar rqt_graph en otra terminal, se visualiza que se cumple el comportamiento esperado, donde el primer nodo **signal_gen** genera dos tópicos “signal” y “time” que son recibidos por el nodo “signal_proc” y procesados para publicar una nueva señal.

- **Visualización de las señales**

Al ejecutar el archivo launch, se puede verificar de forma gráfica la diferencia entre ambas señales, la señal de color rojo es la original que oscila entre 1 y -1. Mientras que la señal azul se le aplicaron todos los procedimientos solicitados, teniendo un desfase, amplitud y desplazamiento diferentes a la señal original.



Conclusiones

Para este reto se lograron los objetivos planteados, que consistía en implementar dos nodos en ROS2 donde el primer nodo, el generador de señal sinusoidal, publica correctamente los datos de tiempo y señal, mientras que el nodo procesador recibe esta señal, aplicando desplazamiento, reducción de amplitud y desfase, para luego publicarla en un nuevo tópico.

La visualización en tiempo real mediante rqt_plot y la verificación de la arquitectura con rqt_graph confirmaron el correcto funcionamiento del sistema. Demostrando la comprensión de los procesos de publicación, suscripción y procesamiento de datos en ROS2.

Sin embargo, algunos objetivos podrían mejorarse. Por ejemplo, la modificación de la señal se realiza con parámetros fijos; introducir controles dinámicos permitiría ajustar la amplitud, desfase y desplazamiento en tiempo real.

Además, aunque el archivo launch facilita la ejecución simultánea, la sincronización exacta de la publicación de datos podría optimizarse para reducir posibles desfases en sistemas más complejos.