RETO SEMANAL I
Manchester Robotics

Abraham Ortiz Castro
Dpto. de Ingenierías Tec de Monterrey
Tecnológico de Monterrey
Puebla, México
A01736196@tec.mx

Alan Iván Flores Juárez
Dpto. de Ingenierías Tec de Monterrey
Tecnológico de Monterrey
Puebla, México
a01736001@tec.mx

Jesús Alejandro Gómez Bautista
Dpto. de Ingenierías Tec de Monterrey
Tecnológico de Monterrey
Puebla, México
A01736171@tec.mx

Ulises Hernández Hernández
Dpto. de Ingenierías Tec de Monterrey
Tecnológico de Monterrey
Puebla, México
A01735823@tec.mx

I. Resumen

ROS 2 ha demostrado ser una plataforma avanzada para el desarrollo de software en robótica, que mejora y se adapta a los cambios en la industria desde su predecesor ROS 1. Basado en una arquitectura compuesta por nodos, facilita la interacción entre componentes robóticos con su estructura de grafo y la capa de abstracción RMW. El entendimiento de este entorno, así como su funcionamiento y estructura general nos permitirá el desarrollo de esta primera actividad introductoria.

II. Objetivo.

Desarrollar las capacidades de manejo del framework ROS 2 y comenzar a familiarizarse con la metodología de trabajo de éste, así como reconocer la arquitectura básica de su funcionamiento, por medio de la creación de nodos y tópicos para enviar una señal, modificarla y graficarla.

III. Introducción.

Aunque los elementos previamente enlistados sirven como parteaguas para ubicarnos en el propósito de este reto, es momento de profundizar en conceptos sumamente importantes que de otra forma dificultarian el entendimiento de los elementos proximos a desarrollar, particularmente para aquellos que no esten familiarizados con ello, y con ello, facilitar la demostración de los resultados de la práctica. Primordialmente nos centraremos en explicar y dar contexto sobre nuestro entorno de trabajo, ROS2.

A. ¿Qué es ROS 2?

El Sistema Operativo de Robots (RO S, por sus siglas en inglés) es un conjunto de bibliotecas/librerías de software además de herramientas para construir aplicaciones de robots. Su primera versión, ROS 1, ha sido un elemento clave de un gran número de sistemas robóticos durante más de una década, decada que trajo cambios significativos en el paradigma de la robótica y en el propio ROS, como resultado de ello surge ROS 2, una mejora de las limitaciones de ROS 1 y manteniendo lo funcional de dicha primera versión así como adpatandose a las necesidades crecientes de la industría misma. [1].

B. Estructura básica de funcionamiento

La estructura de funcionamiento de ROS2 se puede sintetizar como se expresa a continuación:

- **Nodos:** Son la unidad más pequeña en el funcionamiento de ROS, cada nodo es un programa con tareas específicas. [2].
- **DDS Middleware:** ROS 2 emplea Data Distibution Service como middleware para comunicarse entre los nodos, permitiendo su comunicación. [2].
- **Estructura de grafo:** La estructura nodo, tema, mensaje y descubrimiento es la estructura distribuida de ROS 2. [2].
- **Capa de abstracción RMW:** Usa una capa de abstracción propia en lugar de un middleware. [2].
- **Publicadores y Suscriptores:** Los nodos de ROS son capaces de publicar mensajes de un tema o suscribirse a estos y recibir mensajes. [2].
- **Herramientas y Bibliotecas:** ROS 2 incluye bibliotecas de código abierto empleadas para diferentes propósitos dentro del robot. [2].

C. Elementos que integran su red de comunicación

1) ***A nivel de sistema de archivos:*** ROS se compone de paquetes que son la unidad principal para organizar el software. Los metapaquetes representan un grupo de paquetes relacionados. Los manifiestos de paquete proporcionan metadatos sobre un paquete. Los repositorios son colecciones de paquetes que comparten un sistema VCS común. ROS utiliza tipos de mensajes y tipos de servicio para definir las estructuras de datos para los mensajes y servicios respectivamente. [3].

2) ***A nivel gráfico de cálculo:*** ROS se compone de nodos que son procesos que realizan cálculos. El Maestro ROS proporciona registro de nombres y búsqueda para la comunicación entre nodos. El servidor de parámetros permite

almacenar datos en una ubicación central. Los nodos se comunican entre sí pasando mensajes a través de un sistema de transporte con semántica de publicación/suscripción, conocido como temas. Para las interacciones de solicitud/respuesta, se utilizan servicios, que están definidos por un par de estructuras de mensajes. Finalmente, las bolsas son un formato para guardar y reproducir datos de mensajes ROS. [3].

*3) **A nivel de comunidad:**:* Las distribuciones ROS son colecciones versionadas de pilas que facilitan la instalación de software y mantienen la consistencia de las versiones, similar a las distribuciones de Linux. ROS opera en una red federada de repositorios de código, permitiendo a diferentes instituciones desarrollar y lanzar sus propios componentes de software robótico. [3].

IV. Solución del problema

Para dar solución al problema se comienza creando un paquete dentro de nuestra carpeta de proyecto, en este caso RoboInges-Manchester creada previamente con la función `mkdir <nombre_carpeta>`, de igual manera se crea la subcarpeta "Reto1", posteriormente se procede a crear el paquete con el siguiente comando en la terminal

```
ros2 pkg create --build-type ament_python
--node-name signal_generator courseworks
--dependencies rclpy std_msgs
```

Una vez echo esto, se utiliza el comando colcon build para construir el paquete, de la siguiente manera:

```
colcon build
source install/setup.bash
```

Hecho esto abrimos Visual Studio con el comando `code .`, una vez abierto el programa ingresamos a nuestra carpeta 'build' donde encontraremos nuestro programa .py con el nombre asignado previamente, en neustro caso `signal_generator` aquí es donde se escribe el código necesario para la creación de una señal senoidal y la creación de los dos topicos de señal y tiempo, esto con el fin de que el otro nodo pueda recibirla y así procesarla. El código se muestra a continuación:

```
#Se importan las librerías necesarias
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32
import math

class My_Publisher(Node):
    #Creación del constructor
    def __init__(self):
        super().__init__('wave_publisher')
        self.publisher_signal = self.create_publisher(Float32,
        'signal', 10)
        self.publisher_time = self.create_publisher(Float32,
        'time', 10)
        self.timer_period = 0.1  # Frecuencia de 10 Hz
        self.timer = self.create_timer(self.timer_period,
        self.timer_callback)
        self.get_logger().info('Wave Publisher node
        successfully initialized!!!')
        self.time = 0.0

    #Definción de timer_callback
    def timer_callback(self):
        # Calcular el valor de la onda sinusoidal
        wave_value = math.sin(self.time)

        # Publicar el valor de la onda sinusoidal en el topic 'signal'
        msg_signal = Float32()
        msg_signal.data = wave_value
        self.publisher_signal.publish(msg_signal)

        # Publicar el tiempo actual en el topic 'time'
        msg_time = Float32()
        msg_time.data = self.time
        self.publisher_time.publish(msg_time)

        # Incrementar el tiempo para el próximo ciclo
        self.time += self.timer_period

def main(args=None):
    rclpy.init(args=args)
    wave_publisher = My_Publisher()
    rclpy.spin(wave_publisher)
    wave_publisher.destroy_node()
    rclpy.shutdown()

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

Posteriormente se procede a crear el nodo resposanble de modificar nuestra señal, que en nuestro caso se llama `signal_process` dentro de nuestra carpeta build, este nodo será el responsable de modifcar nuestra señal recibida a través de los dos nodos de 'signal' y 'time'. El código de este nodo se muestra a continuación:

```
#Se importan las librerías necesarias
import rclpy
from rclpy.node import Node
from std_msgs.msg import Float32

class SignalProcessor(Node):
    #Constructor
    def __init__(self):
        super().__init__('process')
        #Se crea la suscripción para la señal
        self.subscription_signal = self.create_subscription(
            Float32, #Tipo flotante de 32 bits
            'signal',
            self.signal_callback,
            10)
        #Se crea la suscripción para el tiempo
        self.subscription_time = self.create_subscription(
            Float32, #Tipo flotante de 32 bits
            'time',
            self.time_callback,
            10)
        self.publisher_proc_signal = self.create_publisher(Float32,
            'proc_signal', 10)  # Crear publicador para proc_signal
        self.alpha = 10.0  # Valor inicial del offset
        self.phase_shift = 20.0  # Valor inicial del cambio de fase
        self.get_logger().info(
        'Signal Processor node successfully initialized!!!')

    #Declaracion de callback para señal y tiempo
    def signal_callback(self, msg):
        # Procesar la señal recibida
        processed_signal = msg.data + self.alpha  # Aplicar offset
        processed_signal /= 2  # Reducir la amplitud a la mitad
        processed_signal *= self.phase_shift  # Aplicar cambio de fase
        # Publicar la señal procesada en el nuevo tópico proc_signal
        msg_proc_signal = Float32()
        msg_proc_signal.data = processed_signal
        self.publisher_proc_signal.publish(msg_proc_signal)
        # Imprimir la señal procesada en la terminal
        self.get_logger().info(
            'Processed Signal: %f' % processed_signal)

    def time_callback(self, msg):
        # No es necesario hacer nada con el tiempo en este nodo
        pass

def main(args=None):
```

```
    rclpy.init(args=args)
    signal_processor = SignalProcessor()
    rclpy.spin(signal_processor)
    signal_processor.destroy_node()
    rclpy.shutdown()

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

Dado que este nodo fue creado directamente en Visual Studio dentro de la carpeta 'Build', es necesario agregar la siguiente línea de código dentro del archivo `setup.py` dentro de `entry_points={}` resultando de la siguiente manera:

```
entry_points={
        'console_scripts': [
            'signal_generator = courseworks.signal_generator:ma
            'signal_process = courseworks.signal_process:main'
        ],
    },
```

Es ahora que podemos realizar la construcción del paquete y correr cada uno de nuestros nodos y corroborar que funcionan de manera correcta, para esto ejecutaremos en nuestra consola los siguientes comandos:

```
colcon build
source install/setup.bash
ros2 run courseworks signal_process
```

De ejecutarse de la manera correcta el mensaje de 'Signal Process succesfully initialized!!' debe aparecer en la terminal, seguido por los datos recibidos por el tópico 'signal'. Debemos repetir el paso anterior en otra ventaja de terminal para el nodo de 'signal generator' apareciendo un mensaje similar de ejecutarse de manera correcta. Dentro de otra ventana en la terminal podemos ejecutar `ros2 topic list` para asegurarnos que todos nuestros topicos aparezcan, así como la función `ros2 topic echo /<nombre_topico>` para asegurarnos que se estan enviando los datos correctos. De misma forma podemos ejecutar `ros2 run rqt_plot rqt_plot` y `ros2 run rqt_graph rqt_graph` para visualizar los datos numéricos de forma gráfica [4]. y para graficar el grafo de computación de ROS respectivamente [5].

V. Resultados

Como resultado obtenemos dos nodos activos, donde se genera la señal y donde se recibe y modifica, tambien obtenemos tres topicos para la señal, el tiempo y la señal procesada. A continuación se muestran las imagenes del resultado, en la cual se puede apreciar los comandos realizados en la terminal, así como una grafica con dos señales, una es de la señal original que se genera en el nodo de "signal generator" y la otra señal procesada del nodo "process"

A. Process y Signal Generator node

En la figura 1 se puede observar el nodo de 'signal generator' corriendo en terminal, así como el nodo de 'signal process' en la figura 2.

B. Tópicos

En la figura 3 podemos observar la lista de tópicos de nuestro paquete, en nuestro caso nos interesan los tópicos de '/signal','/time' y `'/proc_signal'`

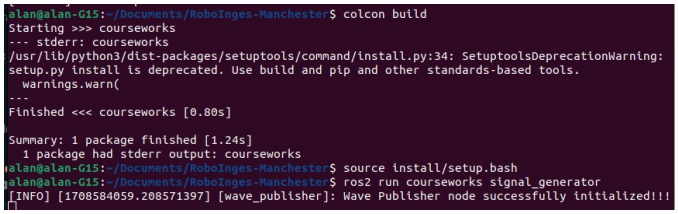

Fig. 1. Signal generator node

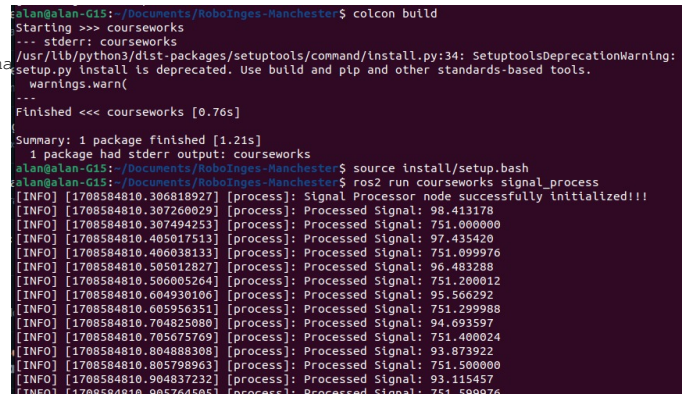

Fig. 2. Signal process node

Fig. 3. Topic List

C. Graficación señales

Para la graficación de señales se utiliza la herramienta rqt plot, esto con la intención de gráficar ambas señales, tanto la producida por el nodo 'signal generator', así como la modificada por el nodo 'signal process'. Para esto se utiliza la siguiente linea de comando:

```
ros2 run rqt_plot rqt_plot
```

Arrogando el resultado visto en la figura 4.

D. Visualización de nodos y tópicos

Para la graficación de los nodos y tópicos se utilizó la herramienta 'rqt graph', aquí es donde se muestra la conexión y los mensajes que se rtansmiten entre los diferentes nodos, es una gran herramienta para hacer depuración y visualzar las difetentes conexiones de nuestro paquete en ROS. Para esto se utiliza la siguiente línea de comando:

```
ros2 run rqt_graph rqt_graph
```

Y el resultado se puede observar en la figura 5.

E. Interpretación de resultados

Como podemos observar en la figura 4, se despliegan ambas señales senoidales, aquella generada por el nodo 'signal generator' con el tópico '/signal/data' mostrada en azul, así

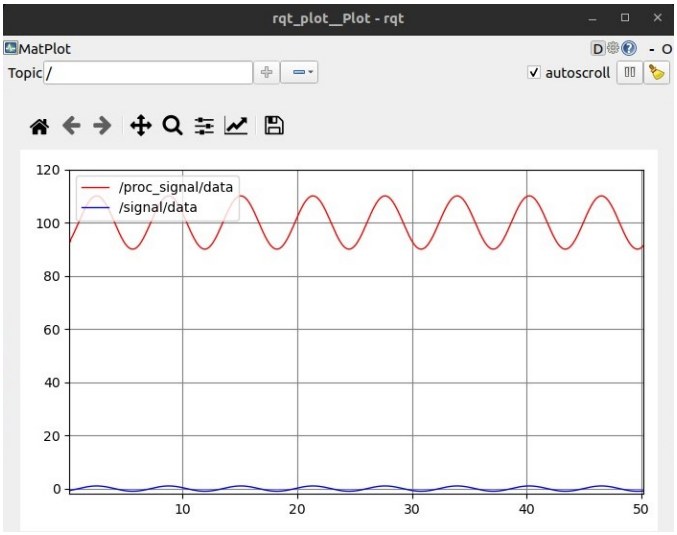

Fig. 4. Graficación de señales

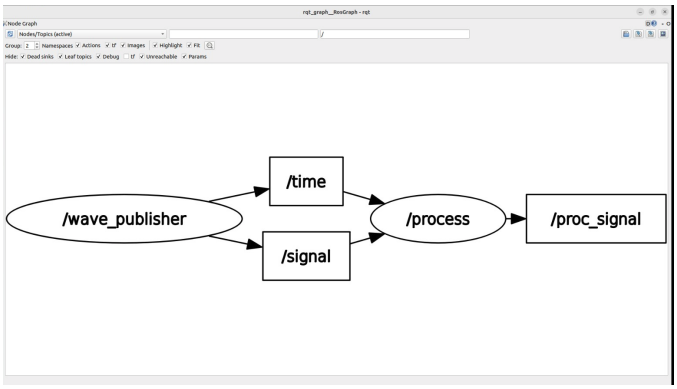

Fig. 5. Conexiones

como aquella modificada por el nodo 'signal process' con el tópico '/proc signal/data' mostrado en rojo, aquí corroboramos que los cambios realizados en este último nodo a la señal son correctos, esto ya que se pude obsevervar un cambio de amplitud de por 10, visto en la variable 'alpha', así como el desplazamiento de fase, denotado en la variable 'phase.shift' con un valor de 20, ambas definidas en el constructor del nodo 'signal process'. De igual forma podemos observar en la figura 5 ambos nodos, el generador así como el modificador de la señal senoidal, en conjunto con el tópico '/signal' que es utilizado para enviar nuestra señal senoidal entre ambos.

VI. Conclusiones

Como conclusión obtenemos el desarrollo de un proyecto basado en ROS 2, donde se implementaron nodos y topicos para transmitir una señal y su posterior modificación, en equipo se logró cumplir los objetivos establecidos, principalmente gracias al uso de herramientas de ROS 2 para compilación, ejecución y obtención de información. A lo largo del proceso se identificó un área de mejora en la creación de archivos launch para simplificar la ejecución de múltiples nodos y detalles de configuración específicos. Finalmente, se pudo obbtener la gráfica con ambas señales, la original y la modificada, lo que demostraba el exito del reto propuesto

References

[1] ROS 2 Documentation — ROS 2 Documentation: Foxy - documentation. (s.f.). link.
[2] Kutluca, H. (2022, Marzo 14). Robot Operating System 2 (ROS 2) Architecture - Software Architecture Foundations - Medium. Medium. link.
[3] es/ROS/Conceptos - ROS Wiki. (s.f.). link.
[4] rqt plot - ROS Wiki. (s. f.). link.
[5] rqt graph - ROS Wiki. (s. f.). link.
[6] ManchesterRoboticsLtd. (s. f.). GitHub - ManchesterRoboticsLtd/TE3001B-Robotics-Foundation-2024. GitHub. link.