Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Campus Puebla

TE3001B: Fundamentación de robótica (Gpo 101)

Profesor: Dr. Rigoberto Cerino Jiménez

Reto semanal 1. Manchester Robotics

Equipo 5

Tania Sofia Arrazola Bello	A01738124
Iñaki Ahedo Madrid	A01738231
Marcos Allen Martínez Cortés	A01737939

18 de febrero de 2026

Resumen

En este reporte se presenta el reto semanal 1 propuesto por Manchester Robotics que tuvo como propósito reforzar los conceptos fundamentales de ROS 2 mediante la implementación de un sistema simple basado en el modelo publicación-suscripción. La actividad consistió en desarrollar dos nodos: un generador de señal sinusoidal y un nodo procesador encargado de modificar dicha señal. En la solución se integraron herramientas propias de ROS 2 como `rqt_plot` para la visualización gráfica de los datos y un archivo launch para ejecutar simultáneamente todos los componentes del sistema.

Adicionalmente, el reporte presenta la metodología de solución del problema por etapas, junto con los resultados obtenidos. Por ende, el desarrollo permitió comprender la arquitectura básica de ROS, la interacción entre nodos, el manejo de tópicos y la importancia de herramientas dentro de ROS.

Objetivos

El objetivo principal es aplicar los conceptos básicos de ROS 2 mediante el desarrollo de un sistema compuesto por dos nodos que generen y procesen señales utilizando comunicación por tópicos. Entre los objetivos específicos se encuentran:

- Implementar un nodo generador de señal sinusoidal.
- Desarrollar un nodo procesador capaz de modificar una señal recibida.
- Visualizar las señales mediante la herramienta `rqt_plot`.
- Construir un archivo launch para ejecutar todos los elementos simultáneamente.
- Analizar la interacción entre nodos dentro de ROS

Introducción

Robot Operating System (ROS) es un conjunto de librerías y herramientas que permiten desarrollar aplicaciones en robótica; asimismo, ROS funciona como un framework de middleware para sistemas distribuidos (NVIDIA, s. f.).

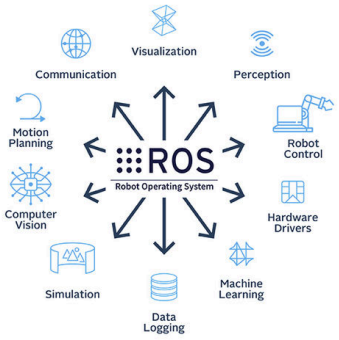

Figura 1: Robot Operating System (ROS).
https://img.directindustry.es/images_di/photo-mg/192516-16646354.jpg

ROS se basa en el mecanismo publicación-suscripción que permite la comunicación entre diferentes procesos a través de mensajes (Open Robotics, s.f.-a). Además, la arquitectura de ROS consiste en los paquetes de ROS, los nodos y los grafos (NVIDIA, s. f.); estos últimos son la base del funcionamiento de ROS, y se refieren a "la red de nodos en el sistema ROS y las conexiones entre ellos por las que se comunican" (Open Robotics, s.f.-a).

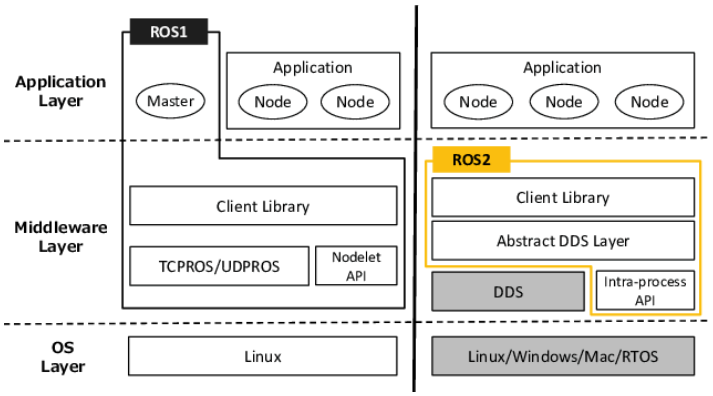

Figura 2: Arquitectura de ROS 1 y ROS 2.
https://www.researchgate.net/publication/309128426/figure/fig1/AS:416910068994049@1476410514667/ROS1-ROS2-architecture-for-DDS-approach-to-ROS-We-clarify-the-performance-of-the-data.png

Los nodos son participantes fundamentales en ROS y son la unidad de computación en un grafo ROS, ya que cada uno debería realizar una "cosa" lógica, y además, utilizan una client library para comunicarse con otros nodos del mismo o diferente proceso, o incluso en una máquina diferente (Open Robotics, s.f.-b).

Los nodos pueden publicar en tópicos para mandar datos a otros nodos, o pueden suscribirse a tópicos para recibir datos de otros nados; por otra parte también pueden actuar como clientes o servidores de servicios (Open Robotics, s.f.-b). De este modo, usualmente los nodos se entienden como "una compleja combinación de publicadores, suscriptores, servidores de servicios, clientes de servicios, servidores de acciones y clientes de acciones, todo al mismo tiempo" (Open Robotics, s.f.-b).

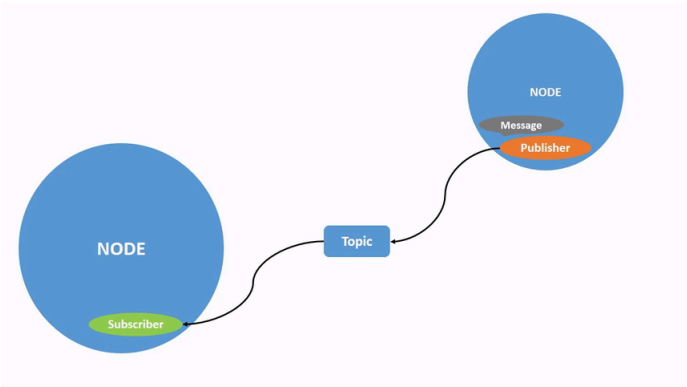

Figura 3: Nodos y tópicos en ROS.
https://docs.ros.org/en/rolling/_images/Topic-SinglePublisherandSingleSubscriber.gif

"Los tópicos son uno de los tres estilos principales de interfaces que ofrece ROS 2. Los tópicos deben utilizarse para flujos de datos continuos, como datos de sensores, estado del robot, etc.". (Open Robotics, s.f.-c)

Los tópicos en ROS son parte del modelo publicador-suscriptor en el que los publicadores producen datos y los suscriptores los consumen; y en ROS, los nodos publicadores y suscriptores pueden comunicarse entre sí cuando formen parte de un mismo tópico (Open Robotics, s.f.-c). Cuando se publican datos en un tópico, todos los suscriptores en el tópico reciben los datos; y por este motivo se le conoce al sistema como un "bus" (Open Robotics, s.f.-c).

Un tópico puede tener cero o más publicadores y cero o más suscriptores; y además, los tópicos cuentan con otras dos características importantes: en ROS 2 son anónimos, lo que significa que cuando un suscriptor reciba datos generalmente no sabrá qué publicador los envió originalmente, y la otra es que son fuertemente tipados ya que cada campo de un mensaje ROS está tipado, es decir, que tiene un tipo de dato asignado (Open Robotics, s.f.-c).

Solución del problema

La metodología realizada para lograr los objetivos del reto, consistió en las siguientes etapas:

1. Diseño del sistema

Se planteó una arquitectura compuesta por dos nodos:

- Nodo generador de señal
- Nodo procesador de señal

Ambos nodos se comunican mediante tópicos bajo el modelo publicación-suscripción.

La arquitectura contiene los siguiente: publicación de la señal, publicación del tiempo de simulación, subscripción a ambas variables, procesamiento de la señal (matemáticamente) y visualización en `rqt_plot`.

2. Organización del paquete y estructura de carpetas

El sistema se desarrolló dentro de un paquete ROS 2 en Python con la siguiente estructura:

```
Reto_1_Equipo_5/
├── Reto_1_Equipo_5/
│   ├── signal_generator.py
│   ├── signal_processor.py
├── launch/
│   └── challenge_launch.py
├── resource/
├── package.xml
├── setup.py
```

Esta estructura es la principal en ROS para crear un paquete, esto permite separar nodos en archivos independientes, así mismo ejecutar múltiples procesos mediante launch files y registrar ejecutables dentro del sistema ROS.

Por el otro lado el archivo `setup.py` se encargó de registrar los nodos como ejecutables mediante los `entry points:`

```
entry_points={
    'console_scripts': [

'signal_generator = Reto_1_Equipo_5.signal_generator:main',
'signal_processor = Reto_1_Equipo_5.signal_processor:main',

    ],
}
```

3. Implementación del nodo generador

Este nodo simula el comportamiento de un sensor o fuente de datos. El nodo generador se diseñó como una clase llamada: `SignalGenerator(Node)`

Su función principal es producir una señal sinusoidal periódica y publicarla continuamente.

3.1 Inicialización del nodo

En el constructor (`__init__`) se configuraron:

- Nombre del nodo
- Dos publishers
- Un temporizador
- Variable de tiempo

```
self.signal_publisher_ = self.create_publisher(Float32,
'/signal', 10)
self.time_publisher_ = self.create_publisher(Float32, '/time',
10)
```

Los publishers permiten enviar el valor de la señal y el tiempo de la simulación

3.2 Generación de la señal

La señal se genera en la función callback del timer:

```
signal = np.sin(self.t)
```

Esta función crea una onda sinusoidal dependiente del tiempo.

3.3 Publicación de datos

Los valores se empaquetan en mensajes de ROS:

```
msg = Float32()
msg.data = float(signal)
```

Y luego se publican en los tópicos correspondientes:

```
self.signal_publisher_.publish(msg)
self.time_publisher_.publish(time_msg)
```

3.4 Control temporal

El tiempo se incrementa cada ciclo lo cual permite generar una señal continua.:

```
self.t += 0.1
```

4. Implementación del nodo procesador

Este nodo realizó las siguientes funciones:

- Suscribirse al tópico del nodo generador.
- Aplicar modificaciones a la señal (cambio de fase y amplitud).
- Publicar la señal procesada en un nuevo tópico.
- Mostrar los resultados.

Este nodo se diseñó como la clase: `SignalProcessor(Node)`

4.1 Subscripción a Tópicos

Se implementaron dos suscriptores, para los tópicos `/signal y /time.`

```
self.subscription = self.create_subscription(
    Float32,
    '/signal',
    self.signal_callback,
    10
)

self.time_subscription = self.create_subscription(
    Float32,
    '/time',
    self.time_callback,
    10
)
```

Estos Callbacks almacenan los valores recibidos.

4.2 Procesamiento de la señal

El procesamiento lo generamos en el Timer Callback

```
phase_shift = 1.0
new_signal = np.sin(self.current_time + phase_shift)
```

Donde se aplican las modificaciones pedidas en el challenge las cuales son desfase de fase offset positivo y cambio de amplitud.

```
new_signal = new_signal + 1.0
new_signal = 0.5 * new_signal
```

4.3 Publicación de la señal procesada

La señal resultante de esta modificaciones se genera en:

```
self.publisher_ = self.create_publisher(Float32,
                  '/proc_signal', 10)
```

5. Visualización de señales
Se utilizó la herramienta rqt_plot para:

- Graficar ambas señales simultáneamente.
- Comparar la señal original y la señal procesada.
- Observar los tópicos del sistema
- Verificar el correcto funcionamiento del sistema.

6. Creación del archivo Launch
Se desarrolló un archivo launch que permitió:

- Ejecutar ambos nodos automáticamente.
- Abrir rqt_plot con los tópicos configurados.
- Facilitar la ejecución del sistema completo con un solo comando.

```
Node(
    package='first_challenge',
    executable='signal_generator',
    name='signal_generator_node'
)
```

Este archivo ejecuta, el nodo generador, el nodo procesador y `rqt_plot` además se configuran los tópicos que se visualizan:

```
arguments=[
    '/signal',
    '/proc_signal'
]
```

7. Validación del sistema
Se verificó el correcto funcionamiento mediante:

- Visualización en rqt_graph
- Comparación de señales en rqt_plot.
- Mensajes mostrados en terminal

Resultados

Utilizando la herramienta de `rqt_graph`, se logró visualizar los nodos lanzados con el archivo launch y los respectivos tópicos a los que se publican y suscriben. Se puede observar como el nodo signal_generator crea los tópicos `/time` y `/signal` a los cuales el nodo `signal_processor` se suscribe y a su vez, este publica el tópico `/proc_signal`.

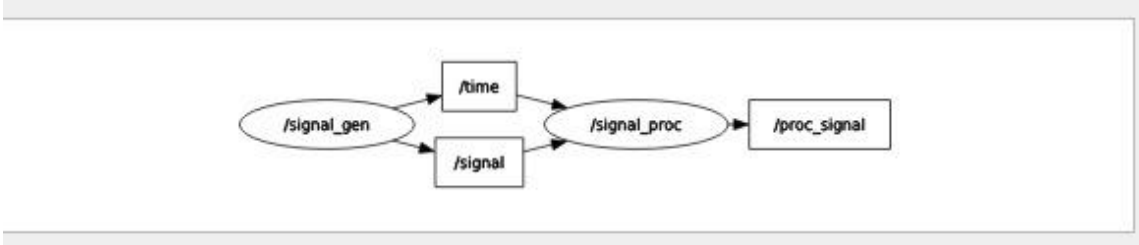

Figura 4: Gráfica en rqt_graph de los nodos y tópicos activos.

Después utilizando la herramienta de rqt_plot, en su interfaz gráfica se eligen los tópicos los cuales se quieren visualizar en una gráfica. En este caso, como indicamos en el archivo launch, se eligen los tópicos `/signal/data` y `/proc_signal/data`, se utiliza data debido a que se grafican con respecto al tiempo. Al elegir ambos, se comparan sus dos gráficas, estas se grafican en tiempo real con respecto al timer que se crea en los nodos. Gracias a esta interfaz gráfica, se pudo visualizar que se logró el objetivo de modificar la señal original utilizando el nodo que la procesa, se puede ver como la señal procesada si cambia con respecto a la original de acuerdo a las modificaciones que se hicieron en el nodo como su desfase y su cambio de amplitud.

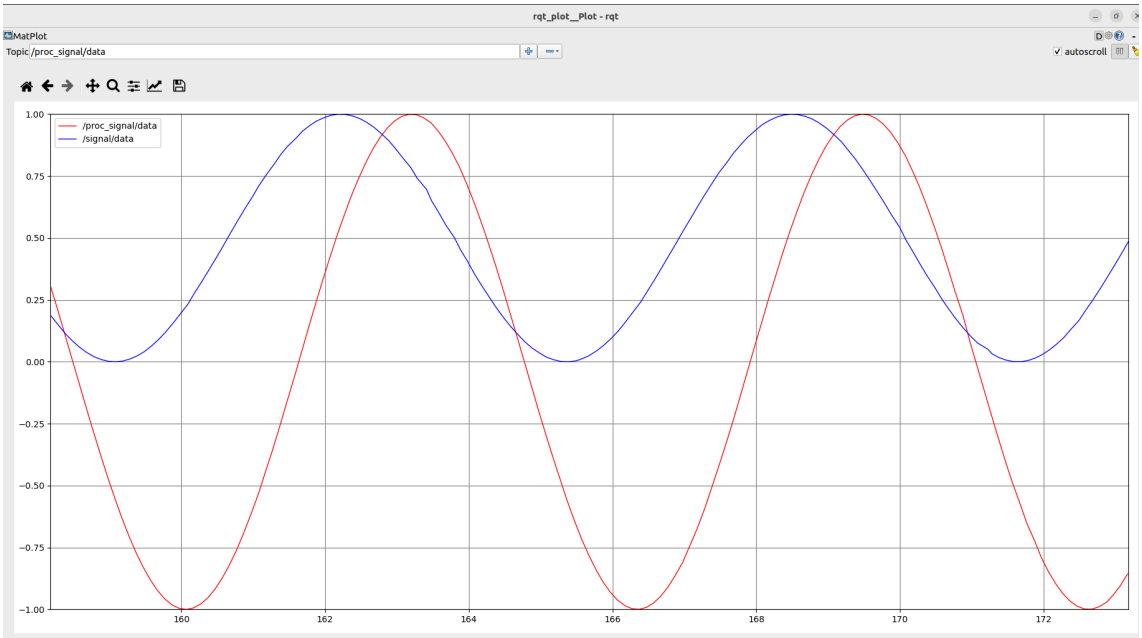

Figura 5: Gráfica en rqt_plot de la señal original y la señal procesada.

Por último, se utilizaron dos terminales para observar la información que imprimía cada nodo como se definió en su código utilizando `get_logger().info`

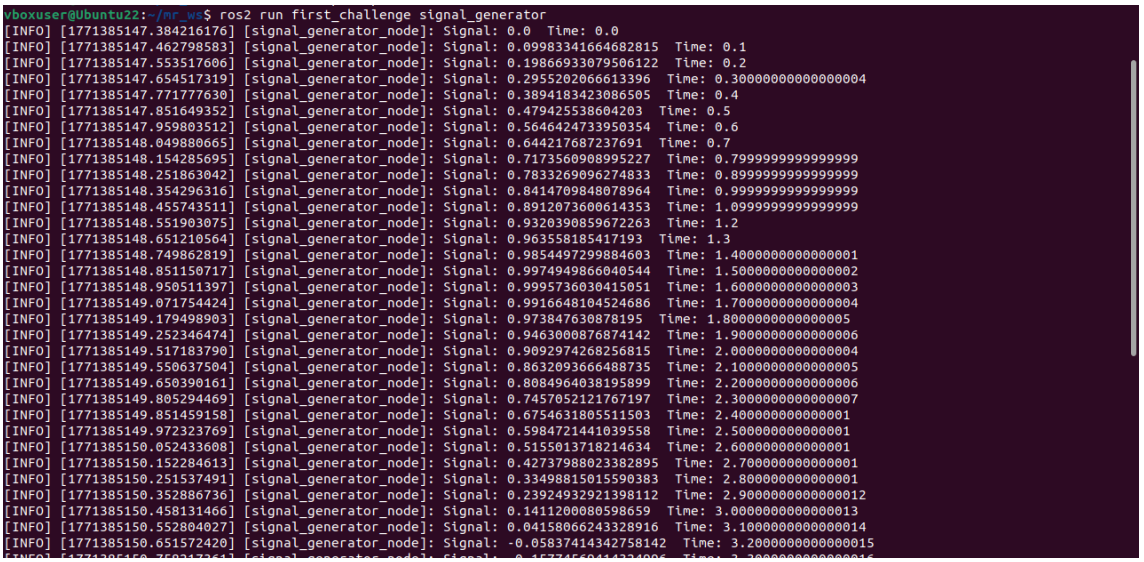

Figura 6: Terminal corriendo el nodo de `signal_generator`.

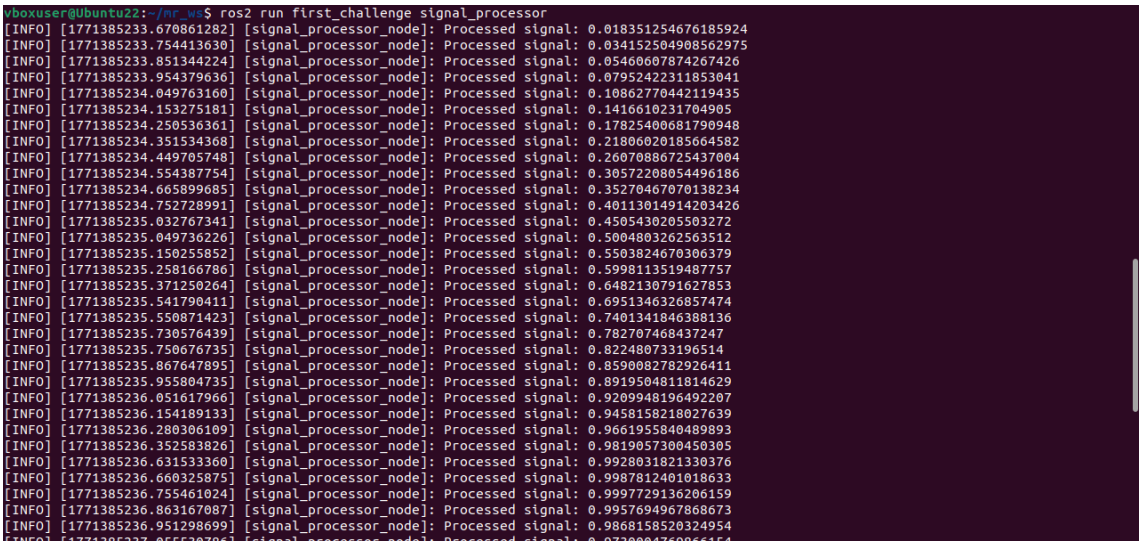

Figura 7: Terminal corriendo el nodo de `signal_processor`.

Conclusiones

Se lograron los objetivos de la creación de dos nodos utilizando el entorno de ROS los cuales intercambian información mediante tópicos. La información se pudo visualizar que iba de acuerdo a los parámetro establecidos en los nodos de manera exitosa. De igual forma, utilizando diferentes herramientas como impresiones en terminal o interfaces gráficas, se observó que la creación de nodos y el intercambio de información entre estos se satisfizo.

De acuerdo a la investigación de las herramientas que se pueden utilizar en ROS, una propuesta de mejora a la solución es utilizar parámetros para modificar los datos en tiempo real de los nodos, esto puede ser funcional en una aplicación real que utilice generación y procesamiento de señales.

Referencias

NVIDIA. (s. f.). *Robot Operating System (ROS)*. Getting Started With Isaac ROS. Recuperado 17 de febrero de 2026, de https://docs.nvidia.com/learning/physical-ai/getting-started-with-isaac-ros/latest/an-introduction-to-ai-based-robot-development-with-isaac-ros/03-what-is-ros.html

Open Robotics. (s. f.-a). *Basic Concepts*. ROS 2 Documentation: Humble Documentation. Recuperado 17 de febrero de 2026, de https://docs.ros.org/en/humble/Concepts/Basic.html

Open Robotics. (s. f.-b). *Nodes*. ROS 2 Documentation: Humble Documentation. Recuperado 17 de febrero de 2026, de https://docs.ros.org/en/humble/Concepts/Basic/About-Nodes.html

Open Robotics. (s. f.-c). *Topics*. ROS 2 Documentation: Humble Documentation. Recuperado 17 de febrero de 2026, de https://docs.ros.org/en/humble/Concepts/Basic/About-Topics.html