Pregunta #1: ¿Cuál es la versión y el nombre de la distribución de ROS instalada en el contenedor que se le ha proprocionado? ¿Cuándo finaliza su soporte? ¿Por qué?

Nuestra versión de ROS es ROS 2. Estamos en la distribución Humble Hawksbill, que finaliza soporte en mayo de 2027.

Se libera una distribución de ROS cada 23 de mayo, y las distribuciones de los años pares tienen soporte extendido de 5 años. Puesto que Humble salió en 2022, su soporte finaliza en 2027.

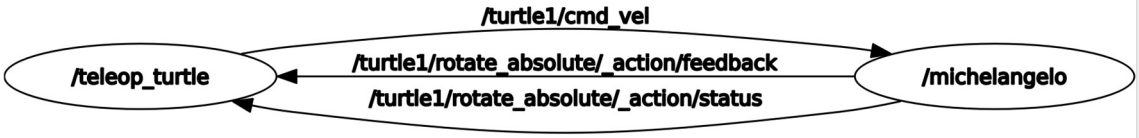
Pregunta #2: ¿Qué otro comando se puede ejecutar con ros2 node? Escriba ros2 node-h para obtener ayuda.

El otro comando es:

`ros2 node info /turtlesim`.

Este nos da la información sobre el nodo.

Pregunta #3: ¿Cómo se llama el tema (topic) donde /teleop_turtle publica los comandos de movimiento? Adjunte una captura de pantalla de rqt_graph donde se vea el nombre.



Se puede observar que el tema donde `/teleop_turtle` publica los comandos de movimiento es: `/turtle1/cmd_vel`.

Pregunta #4: ¿Qué recibe /michelangelo de /teleop_turtle?

Recibe mensajes indicando la velocidad linear y angular. El siguiente es un ejemplo:

```
linear:
  x: -2.0
  y: 0.0
```

```
    z: 0.0
  angular:
    x: 0.0
    y: 0.0
    z: 0.0
```

Pregunta #5: ¿Qué tipos de datos se intercambian entre el nodo de teleoperación y TurtleSim?

Se intercambian datos de tipo `geometry_msgs/msg/Twist`, que sirven para expresar la velocidad en un espacio libre, descompuesta en sus partes lineal y angular. Estos siguen la estructura:

```
Vector3 linear
        float64 x
        float64 y
        float64 z
Vector3 angular
        float64 x
        float64 y
        float64 z
```

Pregunta #6: ¿Qué campos hay que completar en package.xml? ¿Qué es exactamente una licencia en este contexto? ¿Puede citar algunas licencias de software?

Hay que completar los campos con la descripción del paquete, la información de contacto de quien mantiene el paquete (su correo electrónico y nombre, que en este caso no vamos a poner) y la licencia bajo la que se distribuye el paquete, que en este caso es Apache License 2.0. También añadimos las dependencias de lo que vamos a importar, que son rclpy y std_msgs.

Una licencia es un conjunto de términos y condiciones que describen cómo se puede utilizar, modificar y redistribuir un programa o código. En este contexto indica lo que los

usuarios pueden o no hacer con el código. Existen distintos tipos de licencia: tenemos licencias más permisivas como la de MIT que solo requiere reconocimiento y otras más restrictivas como la de GLP que te obliga a liberar cualquier modificación que hagas al código.

Pregunta #7: ¿Qué hay que añadir a setup.py?

En setup.py hay que añadir también el nombre de quien mantiene el paquete, su email, la descripción del paquete y su licencia. Además, hay que añadir un punto de entrada para un nodo "talker" asociado al script "publisher_member_function" y otro punto de entrada para un nodo "listener" asociado al script "subscriber_member_function".

Pregunta #8: ¿Cuáles son los pasos para construir y ejecutar un nodo?

En primer lugar, hacemos el script que va a ejecutar el nodo. Después, añadimos un punto de entrada en el setup.py en el que asociamos el script a un nodo. Para ejecutar el nodo, instalamos los archivos de configuración mediante "`source install/setup.bash`" y corremos el comando "`ros2 run nombre_paquete nombre_nodo`"

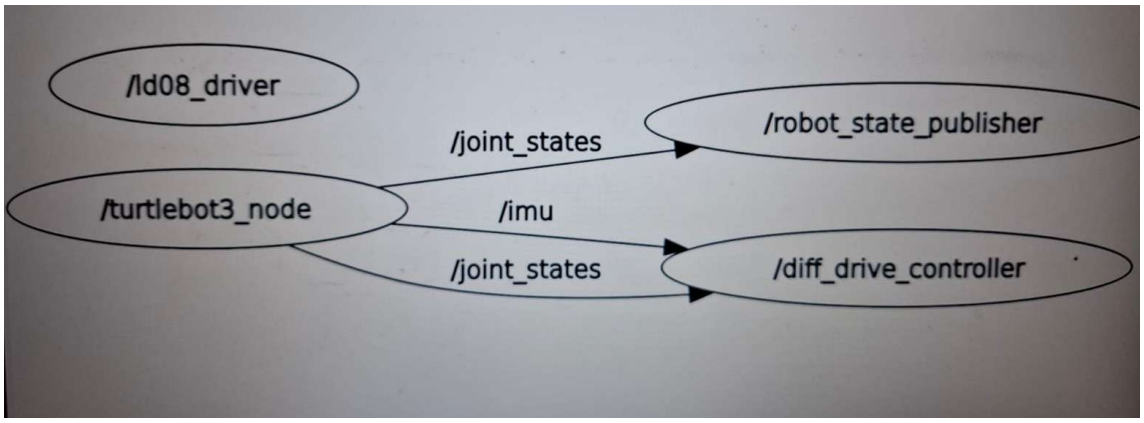
Pregunta #9: En sus propias palabras, explique brevemente cómo se comunica el nodo parlante (talker) con el oyente (listener).

Para que dos nodos puedan enviarse mensajes hay que hacerlo de forma indirecta. Se crea un tema (topic) y ambos nodos se suscriben a él, uno como publicador y otro como receptor de mensajes. A un mismo tema pueden suscribirse varios nodos parlantes y varios nodos oyentes, los nodos oyentes recibirán todos los mensajes que publiquen los distintos parlantes.

Pregunta #10: Utilice las herramientas de terminal de ROS 2 y rqt para descubrir los nodos que operan el TurtleBot3, así como los publicadores y suscriptores que tienen. Explore también la estructura y el contenido de los mensajes e intente explicar con sus palabras para qué sirve cada nodo a partir de esa información. Ignore todos los temas (topics) que empiecen con el prefijo rcl.

Observamos el rqtgraph para ver los nodos operativos. Vemos que hay tres nodos: el turtlebot3_node, que publica en el tema joint_states y en el tema imu. Un nodo

robot_state_publisher suscrito al tema joint_states; y un diff_drive_controller suscrito a los temas joint_states e imu.



Para conocer el tipo de mensaje del tema “joint_states” podemos usar el comando “ros2 topic info /joint_states”. Observamos que se trata de un mensaje de tipo JointState, y, además, observamos que hay un publicador y dos suscriptores, tal y como podíamos ver en el rqtgraph.

Utilizamos el comando “ros2 interface show sensor_msgs/msg/JointState” para ver la estructura de un mensaje de tipo JointState. Podemos ver que el mensaje contiene información sobre el estado de las articulaciones del robot, incluyendo posiciones, velocidades y esfuerzos.

Ahora comprobamos el tema imu “ros2 topic info /imu” y observamos que se trata de un mensaje de tipo Imu con un publicador y un suscriptor.

Y comprobamos la estructura de un mensaje de tipo Imu: “ros2 interface show sensor_msgs/msg/Imu”. Podemos ver que el mensaje contiene datos del sensor IMU (Unidad de Medición Inercial), que incluye aceleración, velocidad angular y orientación.

Sabiendo esta información, podemos deducir la función de cada nodo. El turtlebot3_node se encarga de publicar la información sobre el robot, incluyendo la información sobre las articulaciones y los datos del sensor IMU.

El robot_state_publisher recibe la información sobre las articulaciones y, en principio, no sabemos lo que hace. Sin embargo, por el nombre podemos deducir que utiliza esa información para calcular el estado del robot y publicarlo para que otros nodos puedan conocer la posición y orientación del robot en el espacio.

Finalmente, el diff_drive_controller recibe la información de las articulaciones y los datos del sensor IMU. Por su nombre, podemos deducir que el nodo se encarga del movimiento diferencial del robot.

Pregunta #11: ¿Qué criterio de signos utiliza el TurtleBot3 para la velocidad angular ω por defecto? ¿Coincide con el habitual?

En el TurtleBot3 el eje z está invertido respecto al sentido habitual. Por esto, cuando rotamos el robot respecto al eje z, la velocidad angular tiene el signo opuesto al que esperaríamos.

Código: Controlar al robot con teclado

1. **Crear mensaje personalizado para la lectura del teclado**

Hemos creado un mensaje Key.msg. El único contenido del mensaje es un string key, que contiene el carácter de la tecla que se ha pulsado.

2. **Nodo encargado de publicar cada pulsación detectada por sshkeyboard.**

El siguiente código define un publicador llamado KeyPublisher que llama al método press cada vez que sshkeyboard detecta una pulsación. El método crea un mensaje de tipo Key que hemos creado anteriormente, establece el contenido key del mensaje al carácter de la tecla que se ha pulsado y publica el mensaje en el topic “key”.

```
import rclpy
from rclpy.node import Node

from amr_msgs.msg import Key

from sshkeyboard import listen_keyboard

class KeyPublisher(Node):

    def __init__(self):
        super().__init__('minimal_publisher')
        self.publisher_ = self.create_publisher(Key, 'key', 10)
        listen_keyboard(
            on_press=self.press,
            delay_second_char=0.75,
            delay_other_chars=0.05,
        )

    def press(self, key):
```

```
            msg = Key()
            msg.key = f"{key}"
            self.publisher_.publish(msg)
            self.get_logger().info('Publishing: "%s"' % msg.key)

def main(args=None):
    rclpy.init(args=args)

    minimal_publisher = KeyPublisher()

    rclpy.spin(minimal_publisher)

    # Destroy the node explicitly
    # (optional - otherwise it will be done automatically
    # when the garbage collector destroys the node object)
    minimal_publisher.destroy_node()
    rclpy.shutdown()


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

3. **Nodo que se suscribe a las pulsaciones del teclado, modifica las consignas de velocidad lineal y angular, y las publica en el tema por el que el TurtleBot3 espera recibirlas.**
4. **Y se suscribe al tema que publica el LiDAR para supervisar que los comandos que se generan desde el teclado no van a provocar una colisión.**

En el siguiente código se define una clase KeySuscriber que se suscribe al tema “key” para recibir las pulsaciones del teclado y crea un publicador que publica en el tema “cmd_vel” en el que el robot recibe las consignas de velocidad lineal y angular. Según la tecla pulsada aumenta la velocidad lineal (w), la disminuye (s), aumenta la velocidad angular (a) o la disminuye (d). Además, si se pulsa la barra espaciadora se establecen ambas velocidades a 0 para parar el robot.

La clase LiDARSuscriber se suscribe a los mensajes enviados por el LiDAR en el tema “scan”, este mensaje contiene la información de distancia dada por 240 haces. Para comprobar que el robot no se choque en la parte delantera utilizaremos 40 haces (del -20 al 20), que representan los 60º delanteros del robot. Si alguno de estos haces mide una distancia inferior a la distancia de seguridad (que hemos establecido a 0.2) y la velocidad lineal es positiva, entonces se establecerá la velocidad lineal del robot a 0, publicándolo

en el tema cmd_vel. Para comprobar la parte trasera utilizaremos también 60º (del haz 100 al 140). Si alguno de ellos mide una distancia inferior a 0.2 y la velocidad lineal es negativa, se establecerá la velocidad lineal a 0. Para conocer la velocidad del robot en todo momento, este nodo estará también suscrito al tema “cmd_vel”.

```
import rclpy
from rclpy.node import Node
from rclpy.qos import QoSProfile, QoSReliabilityPolicy,
QoSDurabilityPolicy

from amr_msgs.msg import Key
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

class KeySubscriber(Node):

    def __init__(self):
        super().__init__('minimal_subscriber')
        self.subscription = self.create_subscription(
            Key,
            'key',
            self.listener_callback,
            10)
        self.subscription  # prevent unused variable warning
        self.publisher = self.create_publisher(Twist, '/cmd_vel',
10)
        self.twist = Twist()

    def listener_callback(self, msg):
        # Log the received key
        self.get_logger().info(f'I heard: "{msg.key}"')

        # Update velocities based on key input
        if msg.key == 'w':  # Move forward
            self.twist.linear.x += 0.1
        elif msg.key == 's':  # Move backward
            self.twist.linear.x -= 0.1
        elif msg.key == 'a':  # Turn left
            self.twist.angular.z += 0.1
        elif msg.key == 'd':  # Turn right
            self.twist.angular.z -= 0.1
        elif msg.key == 'space':  # Stop (spacebar)
            self.twist.linear.x = 0.0
            self.twist.angular.z = 0.0
        else:
```

```
            return

        # Publish the updated Twist message to the /cmd_vel topic
        self.publisher.publish(self.twist)

        # Log the published velocities for debugging purposes
        self.get_logger().info(f"Published velocities:
linear={self.twist.linear.x}, angular={self.twist.angular.z}")

class LiDARSubscriber(Node):

    def __init__(self):
        super().__init__('lidar_subscriber')

        # Publicador para enviar comandos de velocidad al
TurtleBot3
        self.cmd_vel_publisher = self.create_publisher(Twist,
'/cmd_vel', 10)

        # Suscriptor al LiDAR con un perfil de QoS adecuado para
datos de sensores
        qos_profile = QoSProfile(
            reliability=QoSReliabilityPolicy.BEST_EFFORT,
            durability=QoSDurabilityPolicy.VOLATILE,
            depth=10
        )

        self.lidar_subscription = self.create_subscription(
            LaserScan,
            '/scan',
            self.lidar_callback,
            qos_profile
        )
        self.cmd_vel_subscription = self.create_subscription(Twist,
'/cmd_vel', self.cmd_vel_callback, 10)

        # Inicialización de variables
        self.stop_distance = 0.2  # Minimum safe distance
        self.current_twist = Twist()
        self.logger_timer = 0

        self.get_logger().info("Teleoperation node initialized.")

    def cmd_vel_callback(self, msg):
        self.current_twist = msg

    def lidar_callback(self, msg):
        front = msg.ranges[0:20] + msg.ranges[-20:]
        back = msg.ranges[100:140]
```

```
        safe_to_move_forward = all(d > self.stop_distance for d in
front if d > 0)
        safe_to_move_backward = all(d > self.stop_distance for d in
back if d > 0)

        stop_twist = Twist()
        stop_twist.linear = self.current_twist.linear
        stop_twist.angular = self.current_twist.angular

        if not safe_to_move_forward and self.current_twist.linear.x
> 0:
            self.current_twist.linear.x = 0.0
        if not safe_to_move_backward and
self.current_twist.linear.x < 0:
            self.current_twist.linear.x = 0.0

        self.cmd_vel_publisher.publish(self.current_twist)

        self.logger_timer += 1
        if self.logger_timer % 10 == 0:
            self.get_logger().info(f"Safe forward:
{safe_to_move_forward}, Safe backward: {safe_to_move_backward}")
            self.get_logger().info(f"Published velocities:
linear={self.current_twist.linear.x},
angular={self.current_twist.angular.z}")

def main(args=None):
    rclpy.init(args=args)

    key_subscriber = KeySubscriber()
    lidar_subscriber = LiDARSubscriber()

    # Use a MultiThreadedExecutor to spin both nodes concurrently
    executor = rclpy.executors.MultiThreadedExecutor()
    executor.add_node(key_subscriber)
    executor.add_node(lidar_subscriber)

    try:
        executor.spin()
    finally:
        key_subscriber.destroy_node()
        lidar_subscriber.destroy_node()
        rclpy.shutdown()



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