

Práctica 3: Explorando nuevas herramientas

Mónica Sánchez Bellido
Jairo Madrigal Cutillas

Tecnología y Arquitectura Robótica

Grado en Ingeniería Informática

Contents

1	Enlaces	2
2	Parte 1 - SLAM con Turtlebot 3	3
3	Parte 2 - Migrando a ROS 2	25

1 Enlaces

Repositorio de GitHub con el código https://github.com/moniqueszcz/entrega_TAR_P2

Ejemplo de navegación mediante 2D Nav Goal https://youtu.be/3gXhPY_a1Xc

Ejemplo ejecución ejercicios ROS 2 https://drive.google.com/file/d/1mRajpaQkpGxd9Zo0Jjep19k_x94Zsw4I/view?usp=sharing

2 Parte 1 - SLAM con Turtlebot 3

Pregunta 1.

Analiza el archivo `.yaml` y explica que significa cada uno de los campos que se muestran.

Se ha generado el paquete `slam_pkg`, y en él se ha añadido las carpetas `launch` y `worlds`. Tras recorrer el mundo `maze_2.world` usando el paquete de teleoperación del Turtlebot 3, se ha conseguido el siguiente mapa mediante SLAM.

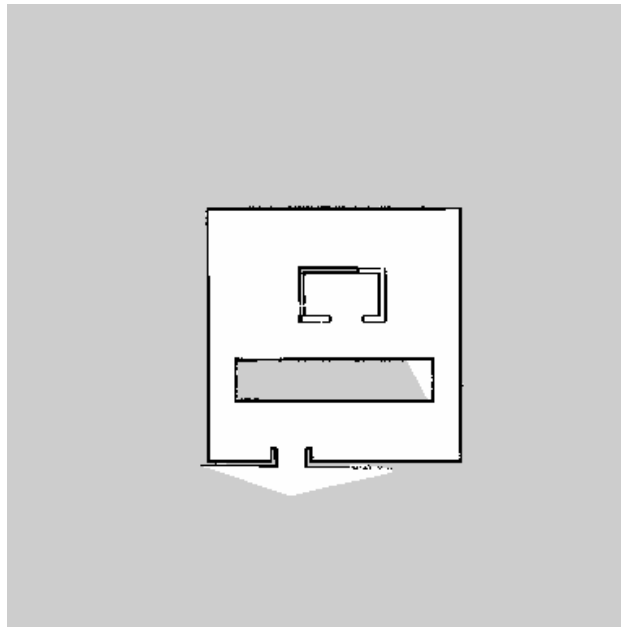


Figure 1: Mapa generado mediante SLAM

Tras generar el mapa y guardarlo, se generan dos archivos en la carpeta `slam_pkg/Maps`, una imagen 2D en escala de grises de tipo `.pgm` y un archivo de tipo `.yaml`. Se puede ver que el archivo `.yaml` contiene la siguiente información:

```
image: prueba_6.pgm
resolution: 0.050000
origin: [-10.000000, -10.000000, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

A continuación se va a explicar el significado de cada uno de los campos del `.yaml`.

- **image:** hace referencia al nombre del archivo `.pgm` que representa el mapa.
- **resolution:** indica el tamaño de cada celda del mapa en metros por píxel, en este caso, cada píxel representa 5cm.

- **origin:** indica las coordenadas de origen del mapa (correspondiente a la esquina inferior izquierda del `.pgm`) de la forma $[x, y, \theta]$, donde x e y se corresponden con la posición en metros del origen del mapa, y θ se corresponde con la orientación en radianes. En este caso, el mapa tiene origen en $(-10, -10)$, y no tiene rotación.
- **negate:** presenta utilidad cuando el mapa se ha generado con los colores al revés, así pues, si el valor del campo es 1, invierte los colores del `.pgm` (blanco \leftrightarrow negro).
- **occupied_thresh:** representa el umbral para considerar que una celda está ocupada, los píxeles más oscuros que este valor son considerados obstáculos.
- **free_thresh:** representa el umbral para considerar que una celda está libre, los píxeles más claros que este valor son considerados transitables.

Pregunta 2.

Investiga qué significa esa especie de "recuadro de colores" que aparece rodeando al robot cuando se pone a calcular la trayectoria y se va moviendo ¿qué significan los colores cálidos/fríos?

Se ha ejecutado RViz junto con el sistema de navegación, junto con el sistema de navegación utilizando el mapa previamente generado. En la imagen 2 se muestra la situación inicial tras el lanzamiento, donde el robot aparece en el centro de la imagen, se visualizan las capas del *costmap* local y global. Aún no se ha indicado la posición estimada del robot mediante el botón **2D Pose Estimate**, por lo que la localización no está inicializada y no aparece la nube de partículas.

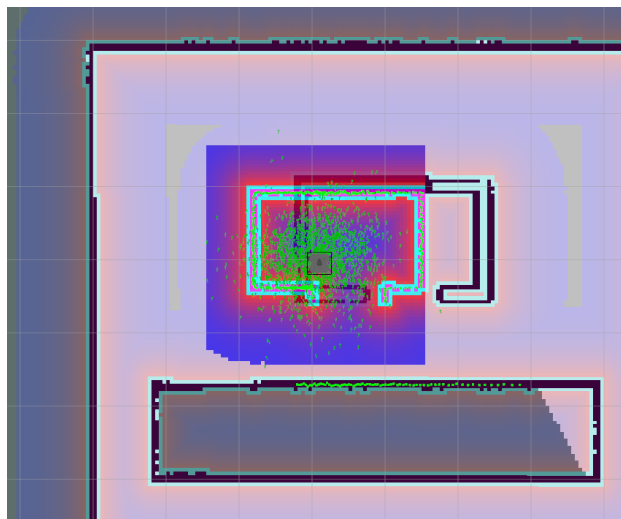


Figure 2: Estado inicial en RViz antes de estimar la posición del robot

Tras indicar la posición estimada del robot mediante el botón **2D Pose Estimate** el estado es el que se muestra en la imagen 3, puede observarse la aparición de una nube de partículas,

que simbolizan distintas posibilidades sobre la posible posición y orientación del robot en el mapa. Además, las paredes del laberinto coinciden.

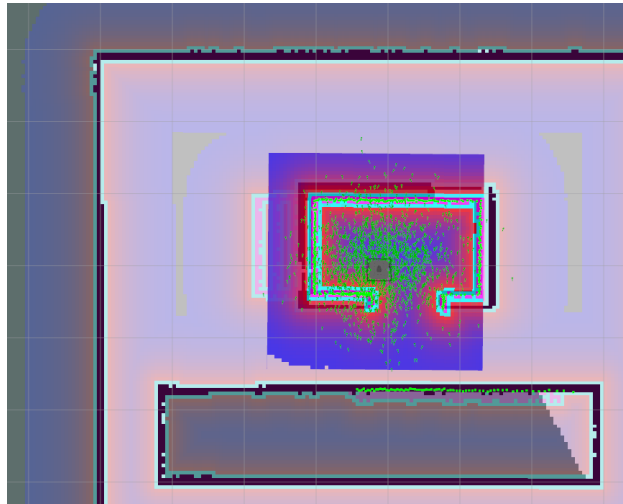


Figure 3: Estado inicial en RViz después de estimar la posición del robot

En el vídeo https://youtu.be/3gXhPY_a1Xc se puede ver la trayectoria seguida por el robot tras recibir una orden de navegación mediante el botón 2D Nav Goal en RViz. El sistema planifica automáticamente una ruta desde la posición actual hasta el destino indicado, teniendo en cuenta el mapa y los obstáculos. En el vídeo se puede ver cómo el recuadro de colores (llamado costmap) que rodea al robot va cambiando en función de los obstáculos que hay a su alrededor, en la figura 4 se muestra un ejemplo.

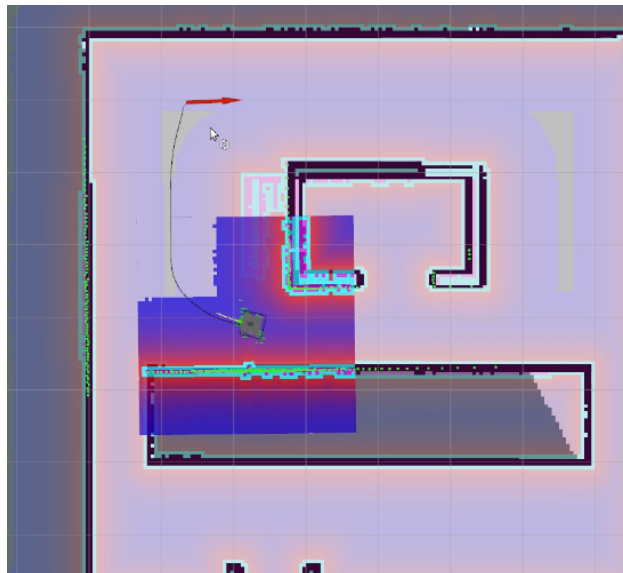


Figure 4: Trayectoria planificada tras definir un objetivo

De la imagen 4 se puede deducir que los colores cálidos (rojo/rosa) representan zonas cercanas a obstáculos, donde la probabilidad de colisión es alta según el costmap. Por el contrario, los colores fríos (azul/morado) indican zonas navegables, con baja probabilidad

de colisión. Se observa que las paredes están marcadas en rojo intenso, y el color se degrada hacia azul/morado a medida que aumenta la distancia al obstáculo.

Pregunta 3.

Investiga qué algoritmo usa ROS por defecto para calcular la trayectoria hasta el destino. Explica su funcionamiento lo más intuitivamente que puedas en aprox. 100-150 palabras (no el código línea por línea sino la idea de cómo funciona).

ROS utiliza dos algoritmos para determinar el movimiento del robot. Un algoritmo de navegación global y un algoritmo de navegación local. El algoritmo de navegación local determina la trayectoria a seguir para llegar desde la posición del robot hasta la posición destino, mientras que el algoritmo de navegación local calcula el movimiento inmediato teniendo en cuenta el entorno más cercano.

Teniendo abierto el entorno de RViz, podemos ejecutar en terminal el comando `rosparam get /move_base/base_local_planner` para conocer el algoritmo que usar ROS por defecto para la navegación local. En la imagen 5 se puede ver que el algoritmo usado es DWA (Dynamic Window Approach).

```
monica@ubuntu-monica:~/Desktop/TAR/practica_3_2425/catkin_ws$ rosparam get /move_base/base_local_planner  
dwa_local_planner/DWAPlannerROS
```

Figure 5: Algoritmo de navegación local por defecto

Por otro lado, podemos ejecutar el comando `rosparam get /move_base/base_global_planner` para conocer el algoritmo de navegación global usado por defecto, en este caso, como se puede ver en la imagen 6, ROS usa el algoritmo NavFn.

```
monica@ubuntu-monica:~/Desktop/TAR/practica_3_2425/catkin_ws$ rosparam get /move_base/base_global_planner  
navfn/NavfnROS
```

Figure 6: Algoritmo de navegación global por defecto

La trayectoria como tal es determinada por el algoritmo de navegación global, es decir, NavFn en este caso, por eso a continuación se va a explicar dicho algoritmo.

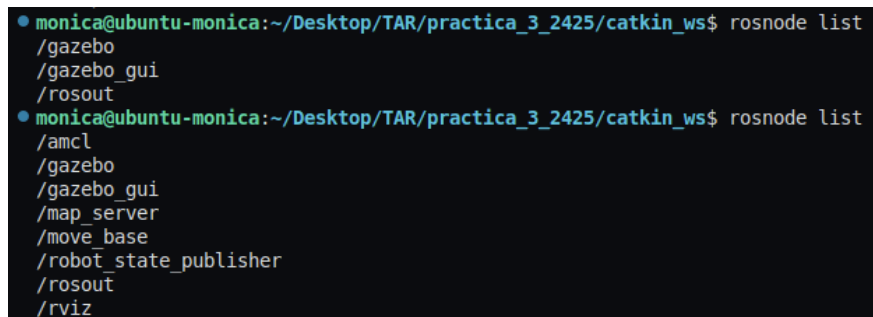
El algoritmo NavFn divide el mapa en una cuadrícula en la que cada celda puede estar ocupada, libre o ser desconocida, aquellas que contienen un obstáculo se marcan como ocupadas. A cada celda libre se le asigna un coste de travesía, un valor entre 0 y 252, donde los valores más altos indican mayor cercanía a un obstáculo. El valor 254 representa un obstáculo infranqueable, y 255 se utiliza para zonas desconocidas. Dado el mapa y las posiciones de origen y destino, el algoritmo calcula el camino más corto aplicando el algoritmo de Dijkstra. Para ello, desde la celda destino, el algoritmo crea un campo de costes acumulados hacia el resto del mapa, asignando a cada celda el valor mínimo necesario para alcanzarla desde el destino, evitando las celdas ocupadas. Esto se realiza mediante una búsqueda basada en el algoritmo de Dijkstra, que se asegura que los caminos más cortos

se evalúan antes. Una vez generado el campo, se decide la trayectoria desde la posición actual del robot eligiendo las celdas vecinas con menor valor, hasta alcanzar la posición destino.

Pregunta 4.

Averigua cuáles son esos nodos que necesitamos cargar en memoria para que funcione la navegación, pon los nombres y describe brevemente el papel de cada uno en 1-2 frases.

Una forma sencilla de ver los nodos necesarios para que funcione la navegación es ejecutar el comando `rostopic list` antes y después de lanzar el archivo de configuración de navegación con `roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=....`. Comparando los resultados se puede observar qué nodos se han añadido y, por tanto, forman parte del sistema de navegación.



```
monica@ubuntu-monica:~/Desktop/TAR/practica_3_2425/catkin_ws$ rostopic list
/gazebo
/gazebo_gui
/rosout
monica@ubuntu-monica:~/Desktop/TAR/practica_3_2425/catkin_ws$ rostopic list
/amcl
/gazebo
/gazebo_gui
/map_server
/move_base
/robot_state_publisher
/rosout
/rviz
```

Figure 7: Comparación de los nodos activos antes (arriba) y después (abajo) de lanzar la navegación

En la imagen 7 se puede ver que se han añadido los siguientes nodos:

- `/amcl`: Es un sistema de estimación de la localización en 2D para un robot en movimiento. Para ello utiliza la localización de Monte Carlo, la cual usa un filtro de partículas para estimar la posición del robot en un mapa conocido. Este nodo es el responsable de la nube de puntos verde que estima la posición del robot en RViz.
- `/map_server`: Se encarga de cargar el mapa previamente generado (en formato `.yaml` y `.pgm`) y publicarlo en ROS para que otros nodos puedan usarlo. Sin este nodo, el sistema no tendría ningún mapa sobre el que posicionarse o planificar trayectorias.
- `/move_base`: Es el nodo principal encargado de la navegación. Recibe un destino y se encarga de planificar y ejecutar la trayectoria hasta ese punto, evitando obstáculos en el camino. Para ello, combina el planificador global y el planificador local explicados anteriormente.
- `/robot_state_publisher`: Publica las transformaciones entre los distintos marcos de referencia del robot (entre la base, el láser, las ruedas, ...) a partir de la información de las articulaciones. Permite que RViz u otras herramientas puedan representar correctamente la posición y orientación de cada parte del robot.
- `/rviz`: Es el nodo asociado a RViz. Aunque no influye directamente en la navegación, permite observar el mapa, la posición estimada del robot, los sensores, y enviar objetivos de navegación de forma interactiva.

Ejercicio 1.

Como puedes observar, en la carpeta `worlds` proporcionada en el directorio `Parte_1`, existe un fichero llamado `muchos_obstaculos.world`. Prueba a mapear este entorno.

Se ha mapeado el entorno obteniendo el siguiente resultado, guardado con el nombre `mundo_obstaculos`.

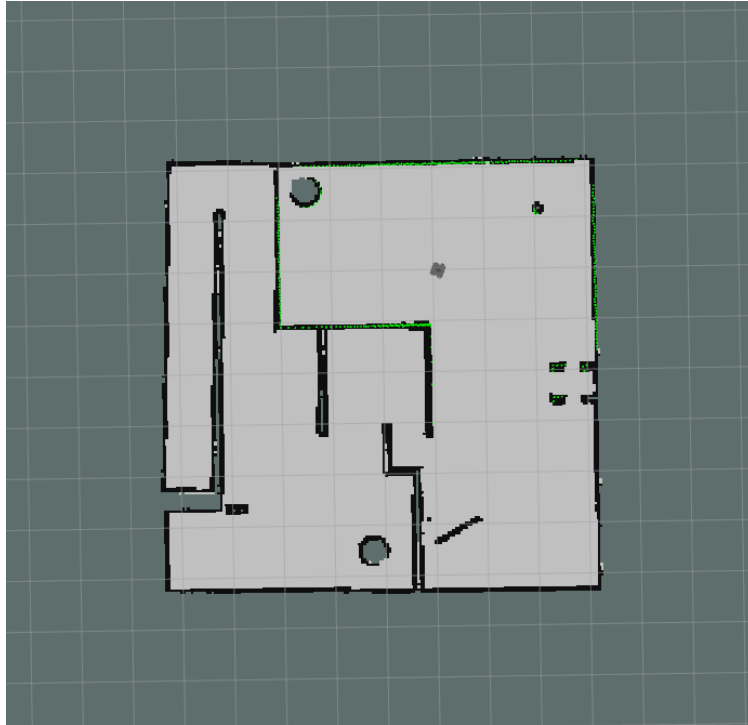


Figure 8: Mapeado del mundo con obstáculos

Pregunta 2.

¿Observas diferencias en el mapeado respecto al primer entorno probado anteriormente?

Hay dos diferencias que llaman la atención a la hora de generar los mapas mediante SLAM para ambos mundos. La primera, y la más relevante, es que el primer mundo contiene dos zonas aisladas con paredes no conexas, lo que dificulta la localización de los puntos detectados por el LiDAR correspondientes a estas paredes. Por ello, para que el mapa generado sea correcto, ha sido necesario recorrer zonas específicas del mundo varias veces, sobre todo las pareces que forman el "cajón" en el que aparece el robot inicialmente. Este problema no está presente en el mundo de obstáculos, dado que todas las paredes están conectadas entre sí, y con una simple pasada se puede generar un mapa preciso del entorno.

La segunda diferencia es que, en el mundo con obstáculos, algunos objetos tienen partes a distintas alturas. Esto supone un problema cuando la parte inferior de un objeto está por debajo del plano de escaneo del LiDAR, ya que no será detectada ni incorporada al

mapa. Este efecto se observa claramente en los conos, donde el sensor solo detecta su parte superior, que es estrecha, mientras que la base, que es mucho más ancha, queda fuera del alcance del escáner y no se mapea. Como resultado, el mapa representa como transitable una zona alrededor del cono que en realidad está ocupada por su base, lo que puede llevar a errores en la navegación del robot. Del igual manera, cualquier parte de un objeto que se encuentre por encima del plano del LiDAR tampoco será detectada. Aunque esto suele ser menos problemático, ya que el robot no podría atravesar esos objetos igualmente por su altura.

Ejercicio 2.

Genera un entorno propio con obstáculos y con diferentes configuraciones.

Se ha generado el siguiente entorno.

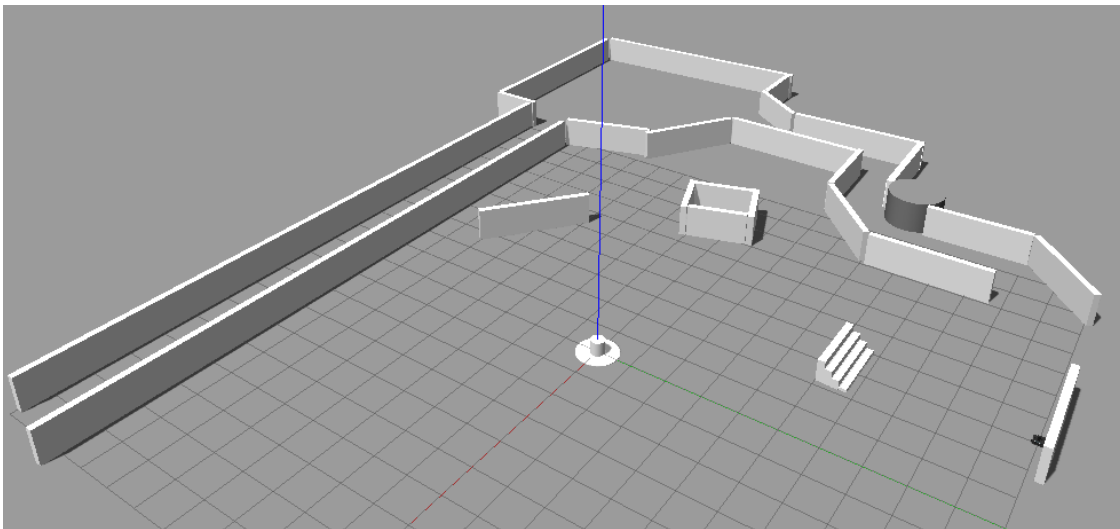


Figure 9: Entorno de prueba generado

Se ha diseñado de manera que permite probar el rendimiento de SLAM en diferentes entornos. Se han establecido las siguientes configuraciones.

1. Entornos cerrados
 - (a) Pasillo largo cerrado con paredes ininterrumpidas
 - (b) Área grande vacía con paredes completamente conectadas
 - (c) Pasillos con giros en forma de L y de U
2. Entornos abiertos
 - (a) Área abierta parcialmente rodeada de paredes completamente conectadas
 - (b) Escaleras con escalones a diferente altura
 - (c) Paredes aisladas
 - (d) Área cerrada inconexa
 - (e) Obstáculo con base más ancha que el resto del objeto

Pregunta 3.

¿Crees que hay cierto tipo de entornos en los que funciona mejor? (espacios abiertos, espacios pequeños, pasillos,...)

Definitivamente los entornos cerrados donde se puede relacionar diferentes puntos "característicos" del entorno es donde mejores resultados se obtiene. Para el mapeado del mapa creado se ha encontrado varios problemas, obteniendo el resultado de la imagen 10.

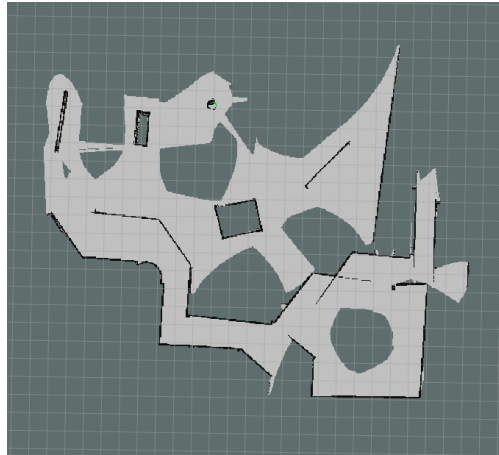


Figure 10: Mapa del mundo generado con SLAM

El primer problema es que el pasillo es muy largo, y no tiene ningún punto que sea diferente, ninguna esquina, u obstáculo. Por ello, se detecta el pasillo como si fuera mucho más corto (rodeado en naranja en la imagen 11), ya que puntos muy distantes se perciben como iguales. De hecho, se puede ver como cuando se escanea el pasillo desde fuera (rodeado en rojo en la imagen 11), sí que se interpreta correctamente su longitud porque hay puntos de referencia que lo permiten.

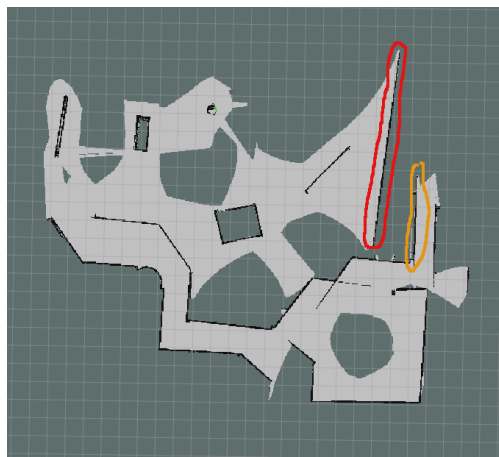


Figure 11: Mapeado del pasillo largo

El siguiente problema ha surgido a la hora de mapear el segundo espacio que explora el

robot, la habitación donde desemboca el pasillo. La estancia tiene un "radio" mucho mayor que rango de escaneo del LiDAR, por lo que cuando el robot circula por el centro de la estancia el LiDAR no detecta ningún punto. Además, no tiene puntos de referencia en los que apoyarse, salvo en los más cercanos, por lo que al dar una vuelta completa a la estancia, el mapa queda completamente descuadrado. En la imagen 12 se puede ver este problema.

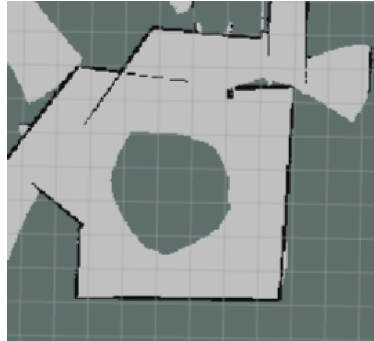


Figure 12: Problema en el segundo espacio del mapa

Como se puede ver en la figura 10, el pasillo con formas de U y L, al suponer de suficientes puntos de referencia, como serían las esquinas, se consigue mapear perfectamente. También se puede ver que aquellas estructuras aisladas presentan dificultades para ser mapeadas, aunque en el resultado mostrado ha sido necesario dar varias vueltas al rededor de ellas para conseguir un mapeado mínimamente aceptable.

Se puede comprobar también la problemática a la hora de mapear aquellos elementos del mundo con una altura inferior a la del robot. Por ejemplo, la base del cilindro no aparece en el mapa generado con SLAM, al igual que tampoco aparece el último escalón de las escaleras, aunque no se pueda apreciar mucho en la imagen 10.

Dados los resultados obtenidos, se puede concluir que SLAM funciona significativamente mejor en entornos cerrados y estructurados, como pasillos con paredes continuas y esquinas bien definidas. Estos elementos proporcionan referencias claras y constantes al LiDAR, permitiendo mantener una buena estimación de la posición del robot. En el mapa generado se observa que los pasillos en forma de L y U han sido correctamente mapeados gracias a sus formas angulares, que ofrecen puntos de referencia. En cambio, los entornos amplios sin suficientes referencias, como la sala abierta al final del pasillo, presentan mayores desafíos. Al carecer de elementos distinguibles dentro del rango de escaneo del LiDAR, la localización pierde precisión y el mapa resultante presenta deformaciones. También se observan errores en zonas con paredes aisladas o estructuras que no forman una topología cerrada, así como en objetos demasiado bajos para ser detectados correctamente por el sensor.

En resumen, SLAM funciona mejor con estructuras conectadas, formas irregulares y límites bien definidos, mientras que los espacios abiertos, simétricos o con obstáculos mal captados por el LiDAR tienden a dar lugar a peores resultados.

Ejercicio 3.

Elige uno de los entornos, puede ser uno de los proporcionados o el que hayas generado en el ejercicio anterior y construye el mapa variando los parámetros del algoritmo. Varía al menos `particles`, `linearUpdate` y `angularUpdate`. En el wiki de ROS tienes la lista completa de parámetros (sección 4.1.4). Para modificar un parámetro podemos hacerlo de varias formas, una de ellas es modificando el Parameter Server.

Además de los parámetros indicados en el enunciado, `particles`, `linearUpdate` y `angularUpdate`, que controlan la cantidad de partículas en el filtro de localización y la frecuencia con la que se procesan los escaneos en función del movimiento del robot, se han seleccionado otros parámetros relevantes para evaluar su impacto en la calidad del mapa generado por SLAM. Todos los parámetros probados son listados a continuación.

- **particles:** define el número de partículas utilizadas en el filtro de Monte Carlo para estimar la posición del robot. A mayor cantidad, mayor precisión en la localización, aunque también se incrementa el coste computacional.
- **linearUpdate:** determina la distancia mínima que debe recorrer el robot para que se procese un nuevo escaneo del LiDAR. Valores más bajos generan mapas más densos y precisos, pero pueden ralentizar el procesamiento.
- **angularUpdate:** indica el ángulo mínimo de rotación que debe realizar el robot para que se procese un nuevo escaneo. Disminuir este parámetro mejora la detección en zonas con muchas esquinas o giros, aunque incrementa el uso de recursos.
- **minimumScore:** permite filtrar escaneos de baja calidad y mejora la estabilidad en entornos abiertos. Un valor más alto puede evitar falsas asociaciones en zonas poco estructuradas, pero si se configura demasiado alto, puede ignorar escaneos válidos.
- **resampleThreshold:** controla cuándo se reamuestran las partículas y, por tanto, la precisión de la localización. Valores bajos provocan resampling más frecuente, lo que ayuda a mantener la convergencia del filtro en entornos dinámicos o ambiguos, pero puede generar mayor ruido y sobreajuste. Valores altos reducen la frecuencia de re-muestreo, disminuyendo la carga computacional, aunque pueden llevar a una pérdida de precisión.
- **delta:** establece la resolución del mapa y permite generar mapas más detallados. Disminuir este valor produce mapas con mayor resolución, capaces de representar mejor los detalles, pero requiere más memoria y tiempo de procesamiento. Aumentarlo reduce la precisión espacial pero mejora el rendimiento.
- **maxUrange:** limita el rango del sensor láser utilizado para construir el mapa. Un valor más bajo ignora lecturas lejanas, lo que puede evitar errores en sensores ruidosos o entornos con estructuras lejanas irrelevantes. Sin embargo, si se reduce demasiado, puede eliminar información útil y dificultar el mapeado de espacios amplios. Aumentarlo mejora la percepción de espacios abiertos, aunque puede incluir datos imprecisos o ruidosos.

En la tabla 1 se ha recogido los valores que se va a probar para cada uno de los parámetros.

Parámetro	Valores probados
particles	30 (defecto), 100
linearUpdate	1.0 (defecto), 0.5
angularUpdate	0.5 (defecto), 0.2
minimumScore	0 (defecto), 50
resampleThreshold	0.5 (defecto), 0.2
delta	0.05 (defecto), 0.01
maxUrange	3.5 (defecto), 2

Table 1: Valores probados para cada parámetro

Cabe mencionar que se va a usar el mundo `muchos_obstaculos.world`. También es necesario explicar que para modificar los valores de los parámetros anteriores se ha creado un script en bash que ejecuta los siguientes comandos. Para cada experimento, se han de cambiar los valores a los deseados.

```
rosparam set /slam_gmapping/particles 30
rosparam set /slam_gmapping/linearUpdate 1.0
rosparam set /slam_gmapping/angularUpdate 0.5
rosparam set /slam_gmapping/minimumScore 0
rosparam set /slam_gmapping/resampleThreshold 0.5
rosparam set /slam_gmapping/delta 0.05
rosparam set /slam_gmapping/maxUrange 3.5
```

Para poder ejecutar los experimentos con exactamente los mismos datos de entrada, y no tener que mover manualmente el robot por el entorno para generar el mapeado, se han ejecutado los movimientos necesarios para la generación del mapa y se han grabado los mensajes publicados en los topics `/scan` (lecturas del lidar), `/odom` (posición y orientación del robot) y `/tf` (mantiene la relación entre distintos frames de referencia, necesaria para alinear correctamente todos los datos en el espacio). Para ello, se ha utilizado la herramienta `rosbag`, que permite registrar mensajes de ROS y almacenarlos en un fichero `.bag`. Para poder guardar los datos, primero se ha abierto Gazebo con el mundo `muchos_obstaculos.world` en un terminal, en otro terminal se ha lanzado el nodo de teleoperación, y en otro terminal se ha ejecutado la siguiente secuencia de comandos desde `catkin_ws`.

```
cd /src/slam_pkg/
mkdir bagfiles
cd bagfiles/
rosbag record -O slam_experimento.bag /scan /odom /tf
```

Una vez se ha finalizado el recorrido por el mapa, basta con hacer `ctrl + C` en el terminal donde se ha lanzado `rosbag` para finalizar el registro y guardarlo en el archivo

slam_experimento_lento.bag.

Con el comando `rosbag info src/slam_pkg/bagfiles/slam_experimento_lento.bag` se puede ver la información general del archivo generado. En este caso, como se puede ver en la imagen 13, se trata de un registro de aproximadamente 47 minutos, en el que se han almacenado más de 320.000 mensajes. Los topics grabados fueron `/odom`, con 84.777 mensajes. `/scan`, con 14.129 mensajes. Y `/tf`, con 226.071 mensajes.

```
monica@ubuntu-monica:~/Desktop/TAR/practica_3_2425/catkin_ws$ rosbag info src/slam_pkg/bagfiles/slam_experimento_lento.bag
to.bag
path:      src/slam_pkg/bagfiles/slam_experimento_lento.bag
version:   2.0
duration:  47:05s (2825s)
start:     Jan 01 1970 01:05:12.13 (312.13)
end:       Jan 01 1970 01:52:17.99 (3137.99)
size:      145.0 MB
messages:  324977
compression: none [189/189 chunks]
types:     nav_msgs/Odometry [cd5e73d190d741a2f92e81eda573aca7]
           sensor_msgs/LaserScan [90cfef2dc6895d81024acba2ac42f369]
           tf2_msgs/TFMessage [94810edda583a504dfda3829e70d7eec]
topics:    /odom 84777 msgs : nav_msgs/Odometry
           /scan 14129 msgs : sensor_msgs/LaserScan
           /tf 226071 msgs : tf2_msgs/TFMessage (3 connections)
```

Figure 13: Información general del archivo .bag

Una vez registrado el fichero `.bag` con los datos de entrada del experimento, este se ha empleado para repetir el proceso de generación del mapa variando los valores de los parámetros mencionados anteriormente. Para ello, lo primero que se ha hecho es reiniciar roscore, y antes de inicializar ningún nodo, se ha ejecutado el comando `rosparam set /use_sim_time true`, para que no se use el reloj real, sino el de la simulación. A continuación, se ha lanzado el nodo de SLAM con `roslaunch turtlebot3_slam turtlebot3_slam.launch`. Después, en un terminal separado, se ha reproducido el archivo con el comando `rosbag play slam_experimento_lento.bag -clock`, que publica los mensajes almacenados en los topics `/scan`, `/odom` y `/tf`, tal y como ocurrieron durante el recorrido original.

En resumen de toda la información explicada anteriormente, se ha creado dos scripts bash, uno para lanzar SLAM (terminal1.sh) y otro para lanzar `roslaunch` `play` (terminal2.sh). Cada archivo hace lo siguiente.

- Establece los valores de los parámetros y de `/use_sim_time`. A continuación lanza SLAM.

```
rosparam set /use_sim_time true
rosparam set /slam_gmapping/particles 30
rosparam set /slam_gmapping/linearUpdate 1.0
rosparam set /slam_gmapping/angularUpdate 0.5
rosparam set /slam_gmapping/minimumScore 0
rosparam set /slam_gmapping/resampleThreshold 0.5
rosparam set /slam_gmapping/delta 0.05
rosparam set /slam_gmapping/maxUrange 3.5

cd catkin_ws/
source devel/setup.bash
export TURTLEBOT3_MODEL=waffle
```

```
roslaunch turtlebot3_slam turtlebot3_slam.launch
```

- Lanza la simulación con `roslaunch play`.

```
cd catkin_ws/  
source devel/setup.bash  
roslaunch play --clock --pause  
src/slam_pkg/bagfiles/slam_experimento_lento.bag
```

Pregunta 4.

¿Cómo afectan estos parámetros en la generación del mapa?

Para futura referencia, la imagen 14 muestra el mapa obtenido con todos los parámetros establecidos a sus valores por defecto.

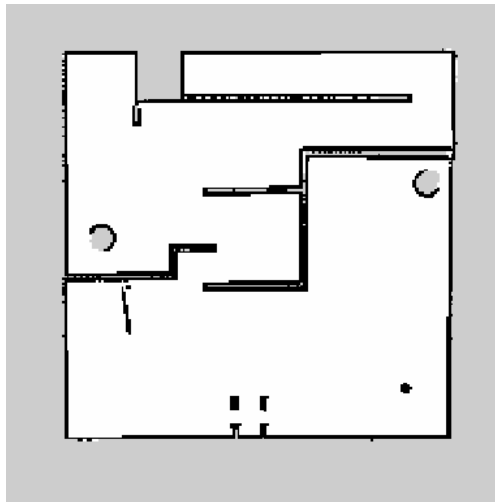


Figure 14: Mapa obtenido con valores por defecto de los parámetros

Para cada uno de los parámetros, se ha visto que afectan de la siguiente manera:

- **particles**: En la figura 15 se puede ver el mapa obtenido con los parámetros por defecto (izquierda) y con **particles** = 100 (derecha).

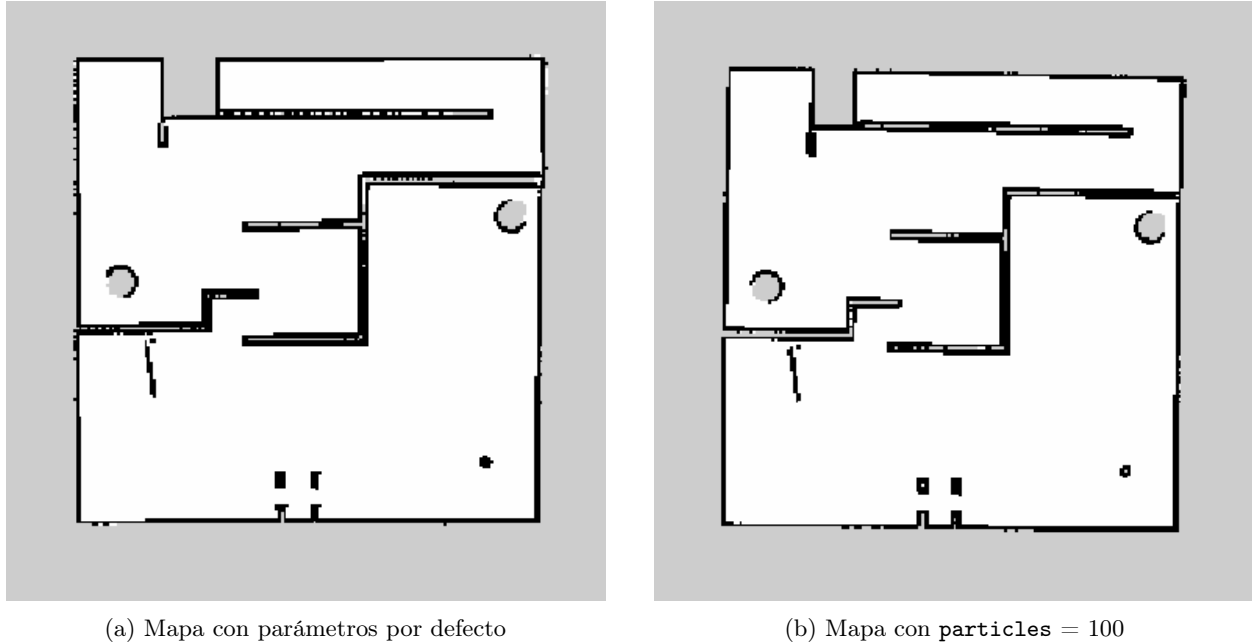


Figure 15: Comparación del mapa resultado cambiando **particles**

Al comparar los dos mapas obtenidos, se pueden observar algunas diferencias notables. En el primer mapa se aprecian algunas inconsistencias en las paredes, especialmente en los bordes y zonas con geometrías más complejas, lo que indica una localización menos precisa. En cambio, en el segundo mapa, las líneas que representan las paredes son más continuas y definidas. También se puede apreciar la mejora en la silueta de las ruedas del robot aparcado en el mapa, y en la silueta de la base del tronco del pino. Esto sugiere que un mayor número de partículas mejora la estimación de la posición del robot, lo que se traduce en un mapeado más limpio y coherente.

- `linearUpdate`: En la figura 16 se puede ver el mapa obtenido con los parámetros por defecto (izquierda) y con `linearUpdate = 0.5` (derecha).

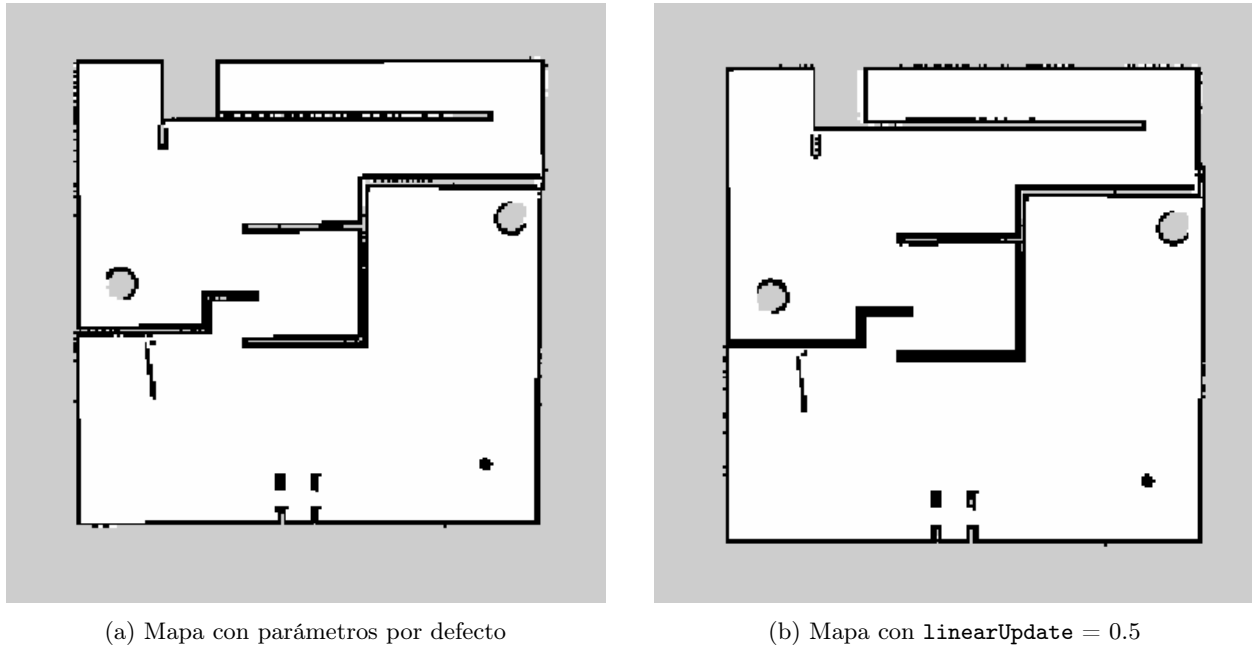


Figure 16: Comparación del mapa resultado cambiando `linearUpdate`

Se puede apreciar que este último ha dado lugar a un mapa más preciso y limpio, al menos en cuanto a las paredes interiores. En el mapa por defecto se observan líneas dobles, muros duplicados y ligeras incoherencias en las esquinas, indicios de errores de localización durante la construcción del mapa. Estos errores son típicos cuando el robot se mueve distancias apreciables entre actualizaciones del mapa. Al reducir `linearUpdate` a 0.5, se obliga al algoritmo a procesar un nuevo escaneo cada vez que el robot se desplaza tan solo medio metro, lo que permite una actualización más frecuente del mapa y una mejor corrección de la posición estimada.

- `angularUpdate`: En la figura 17 se puede ver el mapa obtenido con los parámetros por defecto (izquierda) y con `angularUpdate = 0.2` (derecha).

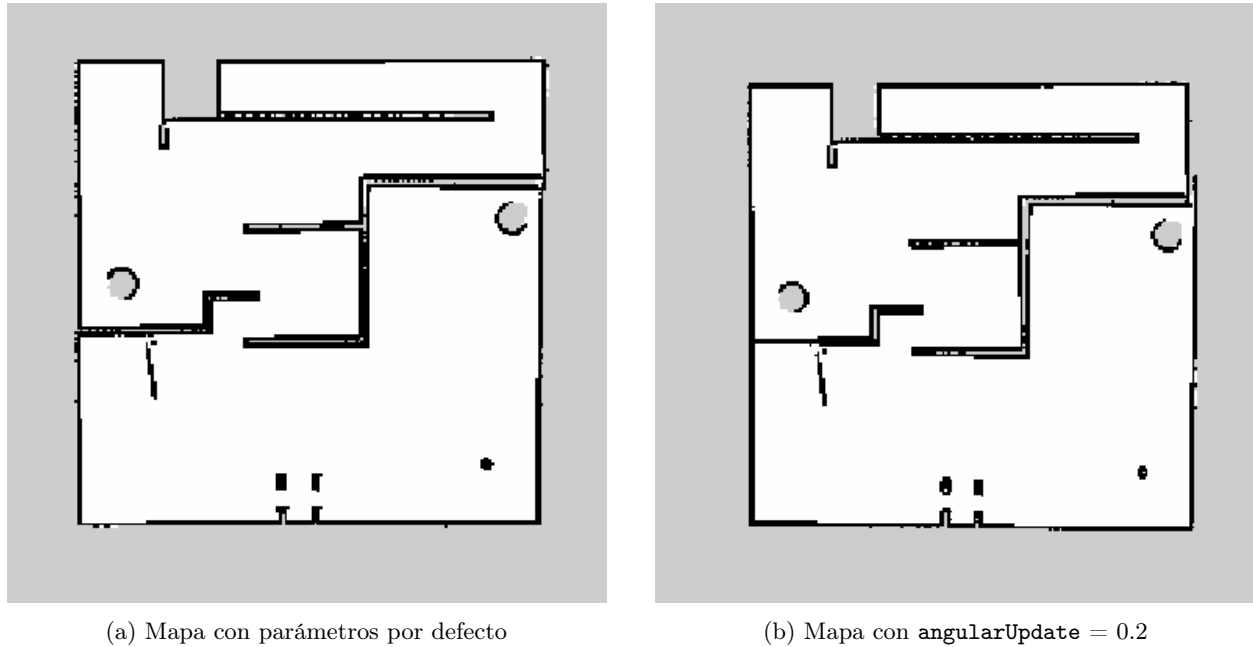


Figure 17: Comparación del mapa resultado cambiando `angularUpdate`

En la comparación entre ambos mapas, se aprecia que reducir este parámetro no ha supuesto una mejora significativa en la calidad del mapa. De hecho, el resultado muestra algunos trazos más imprecisos, especialmente en zonas donde el robot realiza giros, lo cual podría estar relacionado con una sobrecarga de actualizaciones que no aportan nueva información útil. Aunque en teoría disminuir `angularUpdate`, es decir, actualizar el mapa con cada rotación menor del robot, debería permitir capturar más detalle, en la práctica puede amplificar errores si la localización no es completamente estable. En contraste, el mapa con los parámetros por defecto muestra un resultado más uniforme y limpio, lo que indica que, al menos en este entorno, el valor original de 0.5 ofrece un mejor equilibrio entre frecuencia de actualización y estabilidad del sistema. Quizá con un valor mayor de `particles` se obtendría un mejor resultado.

- `minimumScore`: En la figura 18 se puede ver el mapa obtenido con los parámetros por defecto (izquierda) y con `minimumScore = 50` (derecha).

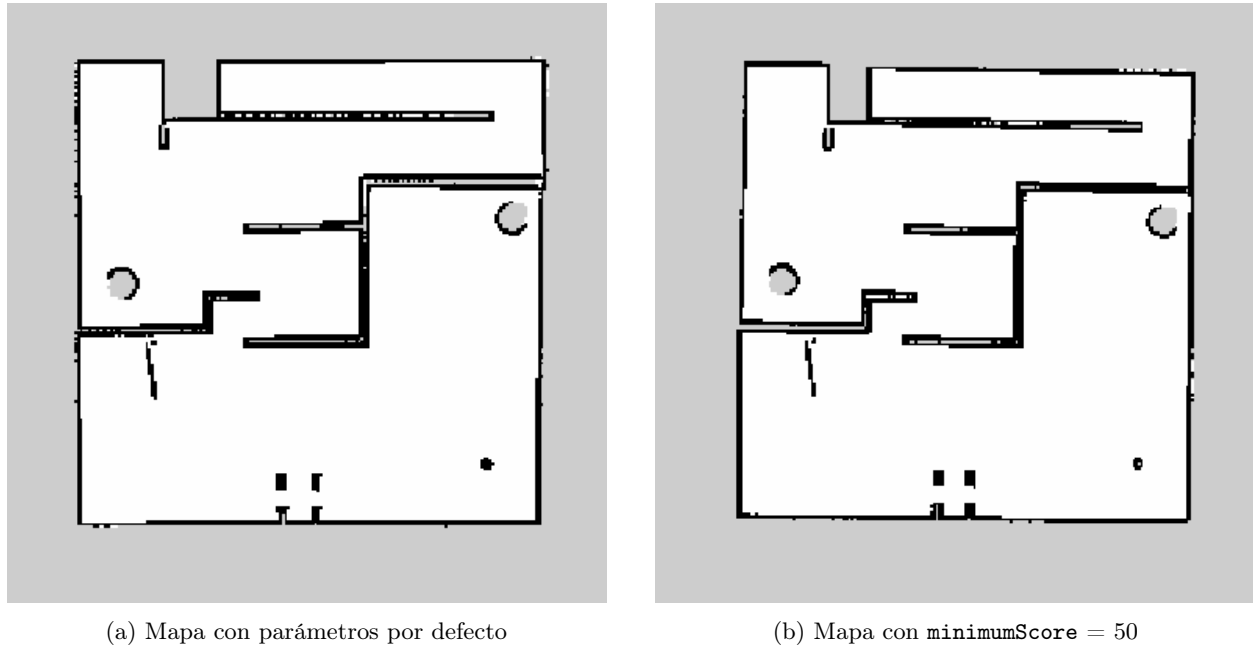


Figure 18: Comparación del mapa resultado cambiando `minimumScore`

Al comparar los resultados, se observa una leve mejora en la limpieza y definición de algunas zonas del mapa, especialmente en áreas abiertas. El parámetro `minimumScore` establece un umbral mínimo de calidad para aceptar una coincidencia entre escaneos, lo que ayuda a filtrar lecturas poco fiables. Al elevar este valor a 50, el algoritmo ignora asociaciones con baja probabilidad, lo que evita deformaciones en el mapa causadas por lecturas mal alineadas. Como consecuencia, se reducen los errores en la localización y mejora la coherencia del mapa, aunque puede conllevar omitir información útil si el umbral es demasiado alto. En este caso, el resultado muestra que establecer un `minimumScore` más exigente ayuda a obtener un mapa más limpio, aunque no presenta una mejora drástica frente a los valores por defecto. Se observa un efecto moderado, sobre todo en la estabilidad general de la estructura del mapa.

- `resampleThreshold`: En la figura 19 se puede ver el mapa obtenido con los parámetros por defecto (izquierda) y con `resampleThreshold = 0.2` (derecha).

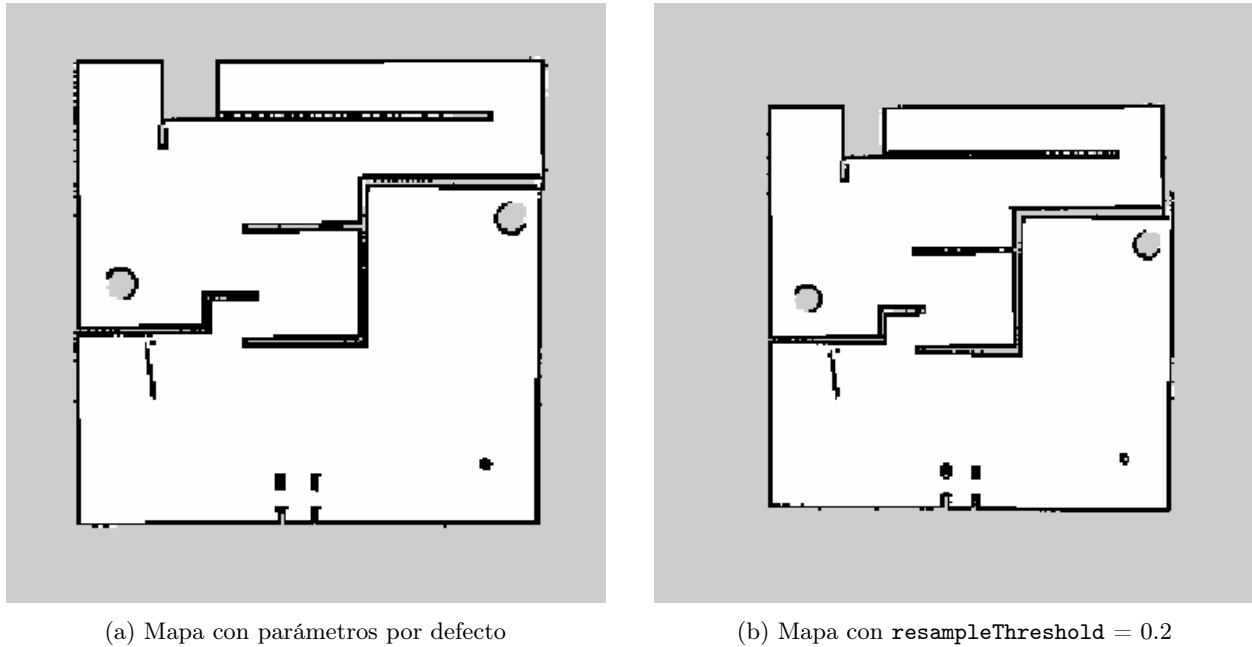


Figure 19: Comparación del mapa resultado cambiando `resampleThreshold`

Al comparar los mapas generados con el parámetro `resampleThreshold` por defecto y con un valor reducido a 0.2, no se aprecia una mejora significativa en la calidad general del mapa. Aunque en teoría un remuestreo más frecuente puede ayudar a mantener una mejor localización del robot, en este caso se observan ciertas inconsistencias, como en la esquina derecha del pasillo en forma de U, donde los contornos aparecen menos definidos, y las esquinas no coinciden. Esto sugiere que un valor demasiado bajo puede provocar una sobrecorrección del filtro de partículas, afectando negativamente a la estabilidad de la estimación de la pose y, en consecuencia, a la precisión del mapa generado.

- **delta**: En la figura 20 se puede ver el mapa obtenido con los parámetros por defecto (izquierda) y con **delta** = 0.01 (derecha).

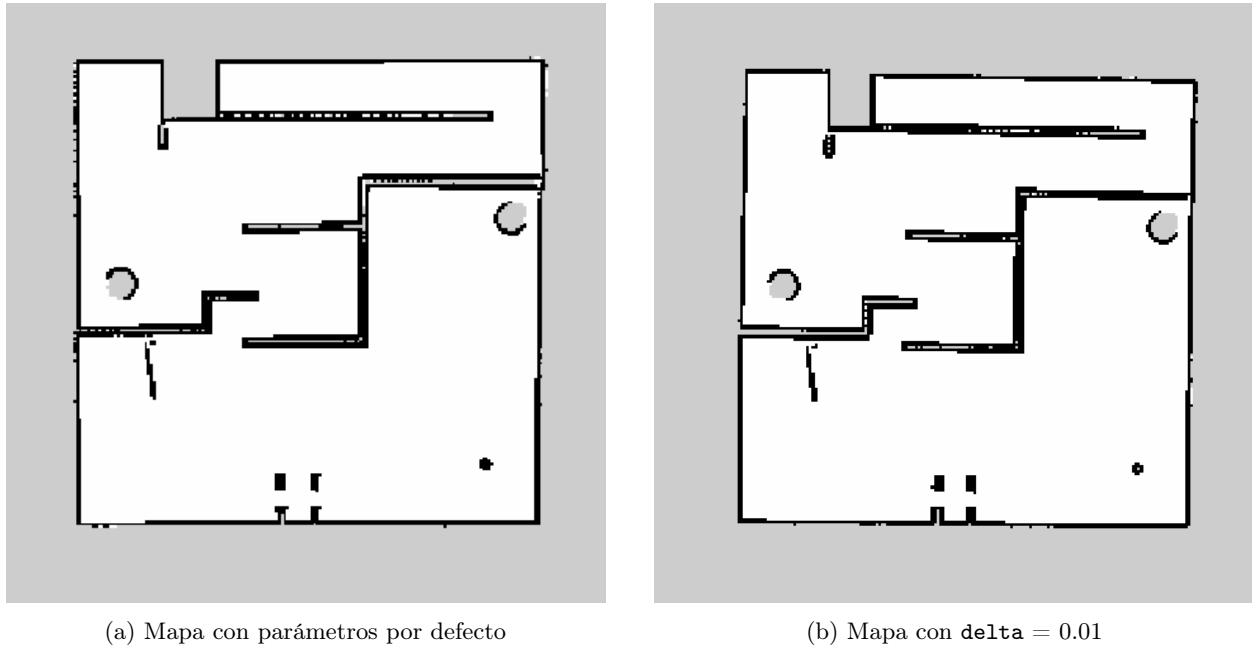


Figure 20: Comparación del mapa resultado cambiando **delta**

Al comparar los mapas obtenidos con el valor por defecto de **delta** = 0.05 y con un valor reducido a 0.01, se aprecia una (ligera) mayor resolución en el mapa resultante. Las líneas que representan las paredes aparecen más definidas y con bordes más precisos, especialmente en zonas curvas o con detalles más pequeños (como el cono del árbol, las ruedas del robot, o la rueda que hay en la estancia de arriba).

- **maxUrange**: En la figura 21 se puede ver el mapa obtenido con los parámetros por defecto (izquierda) y con **maxUrange** = 2 (derecha).

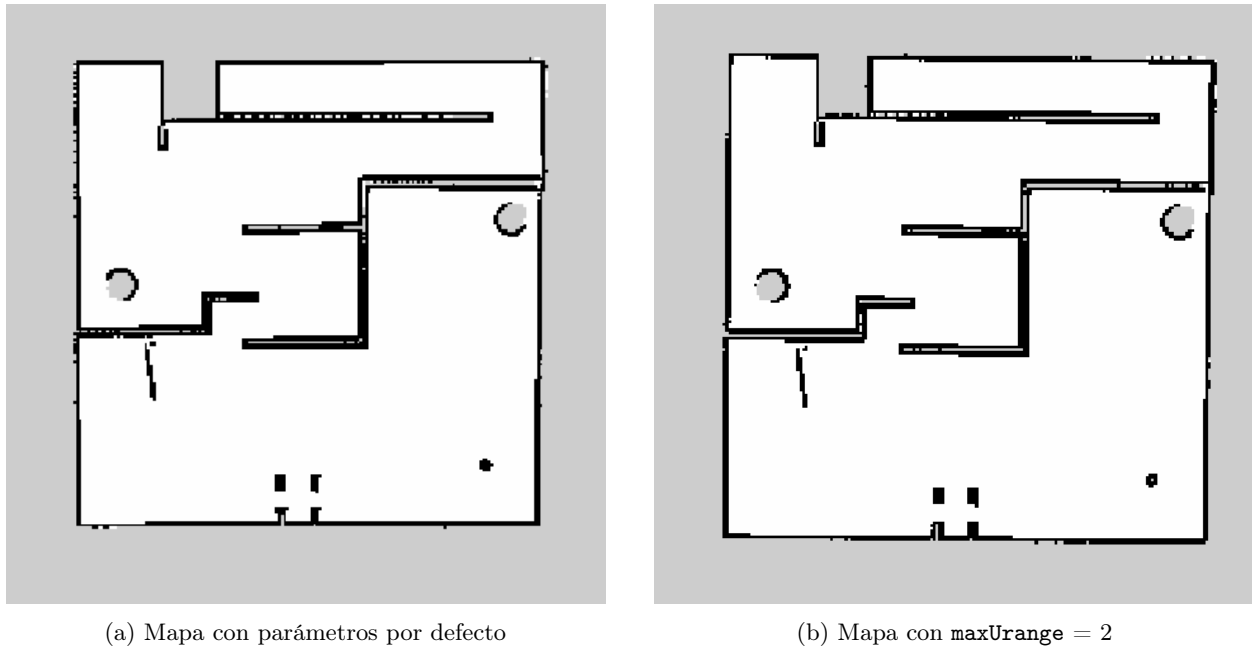


Figure 21: Comparación del mapa resultado cambiando **maxUrange**

Al reducir el parámetro **maxUrange** de su valor por defecto (3.5) a 2, no se observa una mejora clara en la reducción de ruido. Al limitar el alcance máximo del sensor láser, se descartan lecturas más lejanas que podrían haber contribuido a completar el entorno, lo que en este caso no parece haber tenido un impacto positivo significativo en la limpieza o precisión del mapa. Simplemente hay zonas en las que ha mejorado ligeramente, como en detalles pequeños como los obstáculos de la sala de abajo, y hay zonas en las que ha empeorado ligeramente, como algunas esquinas. Por tanto, la reducción de **maxUrange** puede no ser recomendable en entornos grandes o con obstáculos relevantes a media distancia.

En resumen, aunque las diferencias entre mapas generados con parámetros por defecto y aquellos obtenidos tras modificar los valores de configuración pueden no parecer muy grandes, sí se han observado efectos concretos en distintos aspectos de la calidad del mapeado. En general, parámetros como **particles**, **linearUpdate** o **minimumScore** han mostrado mejoras notables en la definición de contornos y la estabilidad de la localización. Otros como **angularUpdate** o **resampleThreshold** han tenido un impacto más variable. Es importante destacar que el diseño del entorno utilizado probablemente ha influido en que las diferencias no hayan sido más marcadas.

Pregunta 5.

¿Qué información puedo extraer de un fichero bag?

Un fichero **.bag** de ROS permite registrar y almacenar mensajes intercambiados entre nodos,

contienen una representación exacta de lo ocurrido en tiempo real. Entre la información que se puede extraer de un fichero bag se encuentran los datos de sensores como el escáner láser (mensajes de tipo `sensor_msgs/LaserScan`), odometría del robot (`nav_msgs/Odometry`) y las transformaciones entre marcos de referencia (`tf2_msgs/TFMessage`), esenciales para interpretar correctamente las posiciones y orientaciones a lo largo del tiempo.

Además, un bag permite analizar con detalle la frecuencia de publicación de los mensajes, su secuencia temporal, y facilita la reproducción controlada de escenarios para validar algoritmos como SLAM o navegación.

Utilizando el comando `rosv bag info`, se puede obtener un resumen detallado del contenido del archivo, incluyendo:

- Ruta del fichero (path) y versión del formato.
- Duración total de la grabación y marcas de tiempo de inicio y fin.
- Tamaño del archivo y número total de mensajes registrados.
- Método de compresión aplicado, si lo hubiera.
- Tipos de mensajes almacenados, como por ejemplo `sensor_msgs/LaserScan`, `nav_msgs/Odometry` o `tf2_msgs/TFMessage`, con sus correspondientes identificadores MD5.
- Topics registrados, el número de mensajes en cada uno y su tipo de mensaje.
- Número de conexiones activas por topic, lo que indica cuántos nodos publicaban simultáneamente en un mismo canal.

```
monica@ubuntu-monica:~/Desktop/TAR/practica_3_2425/catkin_ws$ rosv bag info src/slam_pkg/bagfiles/slam_experimento_lento.bag
path:          src/slam_pkg/bagfiles/slam_experimento_lento.bag
version:       2.0
duration:      47:05s (2825s)
start:         Jan 01 1970 01:05:12.13 (312.13)
end:           Jan 01 1970 01:52:17.99 (3137.99)
size:          145.0 MB
messages:      324977
compression:   none [189/189 chunks]
types:
  nav_msgs/Odometry      [cd5e73d190d741a2f92e81eda573aca7]
  sensor_msgs/LaserScan  [90c7ef2dc6895d81024acba2ac42f369]
  tf2_msgs/TFMessage     [94810edda583a504dfda3829e70d7eec]
topics:
  /odom      84777 msgs : nav_msgs/Odometry
  /scan      14129 msgs : sensor_msgs/LaserScan
  /tf        226071 msgs : tf2_msgs/TFMessage (3 connections)
```

Figure 22: Información general del archivo `.bag`

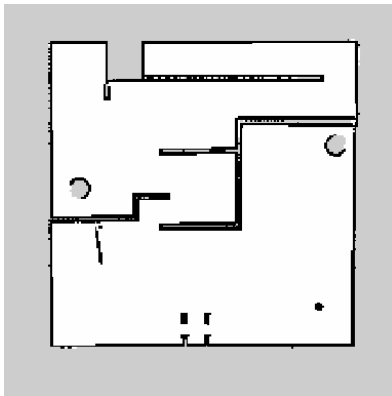
En la imagen 22 se puede ver que en el caso concreto del fichero `slam_experimento_lento.bag`, la grabación tiene una duración total de 47 minutos y 5 segundos, abarcando desde el segundo 312 hasta el 3137 desde el inicio de la grabación. El fichero tiene un tamaño de 145 MB y contiene un total de 324.977 mensajes distribuidos en tres topics principales: `/odom` con 84.777 mensajes del tipo `nav_msgs/Odometry`, `/scan` con 14.129 mensajes de tipo `sensor_msgs/LaserScan`, y `/tf` con 226.071 mensajes de tipo `tf2_msgs/TFMessage`, provenientes de tres conexiones distintas. La ausencia de compresión indica que todos los datos están almacenados en crudo.

Pregunta 6.

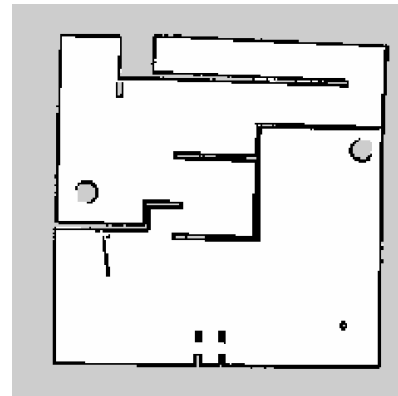
¿Se puede modificar la velocidad de reproducción del archivo? ¿Cómo se puede modificar? ¿Afecta a la resolución del mapa generado?

Es posible modificar la velocidad de reproducción de un archivo `.bag` utilizando la opción `-r` del comando `rosbag play`. Esta opción permite ajustar la tasa de reproducción relativa respecto al tiempo real. Modificar la velocidad de reproducción puede afectar indirectamente a la resolución del mapa generado por SLAM si el algoritmo no dispone de tiempo suficiente para procesar correctamente los datos entre escaneos, especialmente cuando se reproduce demasiado rápido. Una velocidad muy alta puede provocar pérdidas de datos o errores de sincronización, bajando la calidad del mapa. En cambio, reproducir más lento facilita una mejor interpretación y asimilación de los datos, lo cual puede beneficiar la calidad del mapa en algunos casos.

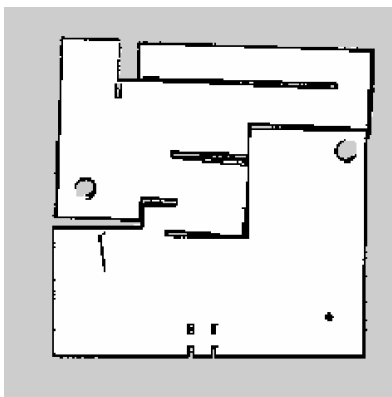
Se ha probado a generar el mapa con una velocidad de reproducción del bag de 2.0, 4.0 y 8.0, todas con los parámetros por defecto. Los resultados se pueden ver en la siguiente figura.



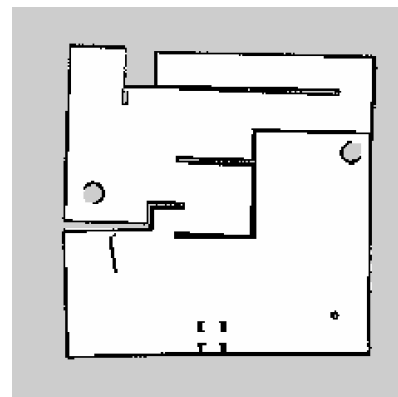
(a) Mapa generado a velocidad 1.0 (original)



(b) Mapa generado a velocidad 2.0



(c) Mapa generado a velocidad 4.0



(d) Mapa generado a velocidad 8.0

Figure 23: Comparación de mapas generados a diferentes velocidades

A medida que la velocidad aumenta, se puede observar una pérdida de precisión en el mapa proporcional al aumento de velocidad. En particular, las imágenes (c) y (d) muestran inconsistencias en algunas paredes y bordes que aparecen deformados o mal alineados, especialmente en zonas con giros o estructuras más estrechas. Además, se aprecia que la estancia superior del mapa aparece desplazada o ligeramente girada respecto a su posición real. Esto ocurre porque al reproducir el archivo más rápido, los mensajes llegan con mayor frecuencia relativa, pero el algoritmo de SLAM puede no procesarlos a tiempo o perder datos clave entre actualizaciones. Por tanto, aunque la reproducción acelerada es posible, puede afectar negativamente la calidad del mapa generado, especialmente a velocidades muy elevadas.

3 Parte 2 - Migrando a ROS 2

Pregunta 1.

Dentro del espacio de trabajo, ¿Qué cambios notas respecto al espacio de trabajo en ROS Noetic?

En el espacio de trabajo de ROS Noetic encontrábamos la siguiente distribución en carpetas:

- **src:** contiene el código fuente de los paquetes desarrollados.
- **build:** carpeta generada tras la compilación, contiene los archivos intermedios creados por el sistema de construcción.
- **devel:** incluye los ejecutables, scripts y configuraciones listas para ser utilizadas directamente sin necesidad de instalación.

En cambio, en el espacio de trabajo de ROS2 (Humble), podemos encontrar variaciones en la distribución:

- **src:** sigue conteniendo el código fuente de los paquetes.
- **build:** contiene los archivos intermedios generados durante la compilación, pero ahora se organiza internamente por paquete.
- **install:** reemplaza completamente a **devel**; todos los ejecutables, bibliotecas y archivos de configuración se ubican aquí tras la compilación.
- **log:** almacena los registros de compilación y ejecución, facilitando la depuración de errores.

Por tanto, podemos observar que tanto en ROS como en ROS2 tenemos la carpeta `src`, donde vamos a encontrar todo el código fuente de los paquetes creados, así como la carpeta `build`, donde, tras compilar, podemos encontrar los archivos intermedios de la compilación. Sin embargo, en ROS2 la carpeta `install` sustituye a la carpeta `devel`, ya que es la carpeta que contiene ahora todos los ejecutables. Además, aparece la carpeta `log`, donde podemos encontrar todos los registros de compilación y ejecución, suponiendo una mejora a la hora de querer depurar los errores.

Estos cambios son debidos al uso de *colcon* para realizar la construcción en vez de *catkin*, lo cual introduce un modelo más modular y alineado con buenas prácticas de empaquetado e instalación.

Pregunta 2.

De nuevo, los archivos que se crean dentro del paquete son diferentes a los que se creaban en ROS Noetic, ¿Donde se ubicarán ahora los nodos que creemos en el paquete?

Si analizamos el interior de `src` para ver la estructura del paquete creado, podemos observar la siguiente distribución:

```
ros2_ws/
|- src/
    |- <package_name>/
        |- <package_name>/          <- Aquí van los scripts/nodos en Python
        |   |- my_node.py          <- Nodo de ejemplo
        |- package.xml
        |- setup.py
        |- resource/
            |- <package_name>
```

Al compararlo con la estructura de ROS 1, la cual era la siguiente:

```
catkin_ws/
|- src/
    |- <package_name>/
        |- src/                    <- Aquí van los scripts o nodos en C++ o Python
        |- CMakeLists.txt         <- Archivo de configuración para catkin
        |- package.xml
```

Podemos observar que dentro de `src` se sigue creando la carpeta con el nombre del paquete, sin embargo, mientras que en ROS1 teníamos dentro una carpeta `src` donde se contenía el código de los nodos, en ROS2, dentro de la carpeta del paquete no se crea una carpeta `src`, sino una carpeta con el mismo nombre del paquete, donde están contenidos los nodos que se creen.

Pregunta 3.

¿Qué son los archivos `package.xml` y `setup.py`? ¿Tienen alguna similitud con algún archivo de los paquetes de ROS Noetic?

Empezando por *package.xml*. Este archivo contiene los metadatos del paquete, tales como el nombre, la versión, descripción, mantenedores, licencias y dependencias del paquete. Su función principal es informar al sistema de compilación sobre las características del paquete y sus relaciones con otros paquetes. Al compararlo con el archivo de ROS, podemos decir que cumplen el mismo propósito. Sin embargo, en ROS 2 el formato ha sido actualizado para adaptarse al sistema de construcción *ament*, incluyendo nuevas etiquetas como `<build_type>`.

Por otro lado, el archivo *setup.py* es un archivo específico para paquetes escrito en Python. Define cómo se instala el paquete y qué módulos o scripts se deben exponer como ejecutables. Se basa en las herramientas estándar de distribución de Python (setuptools) y permite registrar los nodos como entradas ejecutables.

En ROS Noetic no existe un equivalente directo en los paquetes construidos exclusivamente con CMake. No obstante, si se utiliza `catkin_python_setup()` en un paquete Python, es posible incorporar un archivo `setup.py` con una función similar, aunque esta práctica es menos común que en ROS 2.

En conclusión, `package.xml` mantiene su rol fundamental en ambos sistemas, asegurando la correcta gestión de metadatos y dependencias, mientras que `setup.py` representa una evolución necesaria para la instalación y gestión de paquetes Python en ROS 2, facilitando la integración y distribución de nodos escritos en este lenguaje.

Ejercicio 1.

Crea un paquete llamado `service_temp`, este paquete deberá albergar un servicio de ROS 2. Este servicio se encargará de realizar conversiones de temperatura, concretamente de Grados Celcius a Farenheit y viceversa. El funcionamiento es el siguiente: Servidor: Recibe el valor de temperatura y realiza la conversión. Cliente: Envía el valor de temperatura y el tipo de conversión a realizar y muestra el resultado obtenido desde el servidor. Ten en cuenta que el valor de temperatura deberá especificarse desde la línea de comandos del terminal al lanzar el nodo correspondiente.

Para crear el paquete, lo primero que tenemos que hacer es ejecutar en una terminal los comandos:

```
cd /ros2_ws/src
ros2 pkg create --build-type ament_cmake --license Apache-2.0 service_temp --dependencies
rclcpp std_msgs
```

Podemos observar que se va a utilizar `ament_cmake` como método de compilación. Esto es debido a que el servicio va a ser implementado en C++ y presenta una integración más directa con las herramientas y convenciones de compilación de ROS 2 para código nativo, permitiendo gestionar fácilmente dependencias, generación de interfaces y construcción de ejecutables optimizados. Podemos observar también que el paquete va a depender de las bibliotecas `rclcpp` y `std_msgs`, ya que `rclcpp` proporciona las funcionalidades esenciales para programar nodos en C++ dentro de ROS 2, incluyendo la creación y manejo de servicios, y `std_msgs` provee tipos de mensajes estándar que facilitan la comunicación entre nodos.

```
cd /workspace/ros2_ws/src/service_temp
mkdir srv
```

De esta manera ya tenemos la carpeta de servicios creada, y solo tenemos que crear el archivo con la información del servicio presente en la práctica, para ello, dentro de la

carpeta que acabamos de crear ejecutamos *touch ConvertTemp.srv* y le añadimos el contenido mencionado:

```
1 float64 input_temp
2 string conversion_type # Cel_to_Far o Far_to_Cel
3 ---
4 float64 converted_temp
```

Con el archivo del servicio creado, el siguiente paso es la configuración de los archivos de package.xml y CMakeLists.txt para que el paquete esté bien configurado. De esta manera, en CMakeLists.txt, vamos a añadir las siguientes líneas para que se encarguen de generar automáticamente el código necesario para que ROS 2 pueda usar el servicio personalizado en los distintos lenguajes soportados (C++, Python, etc.). Esto incluye la creación de los archivos de cabecera, mensajes y soporte para los tipos de servicio definidos en el archivo .srv.

Además, debemos asegurarnos de incluir las dependencias necesarias para que los ejecutables del cliente y servidor puedan enlazar correctamente con el código generado del servicio:

```
1 find_package(rosidl_default_generators REQUIRED)
2
3 # Generar el servicio personalizado
4 rosidl_generate_interfaces(${PROJECT_NAME}
5     "srv/ConvertTemp.srv"
6 )
```

Por otro lado, en el package.xml vamos a añadir las siguientes líneas para declarar las dependencias necesarias para la generación y ejecución del servicio personalizado:

```
1 <build_depend>rosidl_default_generators</build_depend>
2 <exec_depend>rosidl_default_runtime</exec_depend>
3
4 <member_of_group>rosidl_interface_packages</member_of_group>
```

Estas líneas aseguran que durante la compilación se incluya el paquete *rosidl_default_generators* que genera el código de interfaz para los servicios, y que en tiempo de ejecución esté disponible el paquete *rosidl_default_runtime*. Además, la etiqueta `<member_of_group>` indica que este paquete forma parte del grupo de paquetes que definen interfaces ROS (mensajes y servicios), lo cual facilita su manejo dentro del ecosistema ROS 2.

Con el servicio correctamente configurado, podemos pasar a la creación del servidor y del cliente que utilicen el contenido del .srv creado como comunicación y realicen la conversión. Para ello, empezando por el servidor, se ha creado dentro de la carpeta src un fichero llamado temperature_server.cpp. Este archivo contiene una clase llamada TemperatureServer que hereda de Node, por lo que es un nodo ROS2.

```
1 #include "rclcpp/rclcpp.hpp"
2 #include "service_temp/srv/convert_temp.hpp"
3 #include <memory>
```

```

4
5 using std::placeholders::_1;
6 using std::placeholders::_2;
7 class TemperatureServer : public rclcpp::Node
8 {
9 public:
10 TemperatureServer() : Node("temperature_server")
11 {
12     service_ = this->create_service<service_temp::srv::ConvertTemp>(
13         "convert_temperature",
14         std::bind(&TemperatureServer::convert_temperature, this, _1, _2));
15 }

```

Podemos observar en el constructor que inicializamos el nodo con el nombre de "temperature_server", además, asociamos la función convert_temperature para que maneje las solicitudes de este servicio, usando std::bind para pasar el puntero this y los argumentos del servicio, utilizando placeholders para indicar que cuando se llame a la función enlazada, los argumentos que reciba deben de ir en esas posiciones. Podemos observar también en los includes que se ha incluido el servicio generado a partir del archivo .srv

Continuando con la función que procesa las solicitudes, podemos ver su implementación a continuación:

```

1 private:
2 void convert_temperature(
3     const std::shared_ptr<service_temp::srv::ConvertTemp::Request> request,
4     std::shared_ptr<service_temp::srv::ConvertTemp::Response> response)
5 {
6     if (request->conversion_type == "Cel_to_Far") {
7         response->converted_temp = (request->input_temp * 9.0 / 5.0) + 32.0;
8     } else if (request->conversion_type == "Far_to_Cel") {
9         response->converted_temp = (request->input_temp - 32.0) * 5.0 / 9.0;
10    } else {
11        RCLCPP_WARN(this->get_logger(), "Tipo de conversión no válido");
12        response->converted_temp = request->input_temp;
13    }
14    RCLCPP_INFO(this->get_logger(), "Solicitud: %.2f %s -> %.2f",
15        request->input_temp,
16        request->conversion_type.c_str(),
17        response->converted_temp);
18 }
19
20 rclcpp::Service<service_temp::srv::ConvertTemp>::SharedPtr service_;
21 };

```

Cómo se ha explicado, esta función se llama cuando el cliente hace una solicitud, recibiendo dos punteros, uno con la petición y otra con la respuesta que debe devolver. Según el valor de conversion_type, se realiza la conversión correspondiente. De igual manera, si el tipo no es válido se avisa con un warning y se devuelve la temperatura sin modificar. Por último, se imprime un mensaje informativo con el valor recibido, el tipo de conversión y el resultado obtenido. De igual manera, se mantiene un puntero inteligente a la instancia del servicio para que viva mientras el nodo esté activo.

Por último, se ha creado la función main para inicializar y lanzar el nodo. Este pequeño

método crea y ejecuta el nodo que se ha explicado anteriormente hasta que se detenga (con CTRL+C por ejemplo), realizando el cierre ordenado de ROS2.

```
1 private:
2 int main(int argc, char **argv)
3 {
4     rclcpp::init(argc, argv);
5     rclcpp::spin(std::make_shared<TemperatureServer>());
6     rclcpp::shutdown();
7     return 0;
8 }
```

Con el servidor ya creado, se ha programado también un cliente. Para ello, se ha creado un archivo llamado `temperature_client.cpp`. Este archivo contiene un método `main` que contiene toda la lógica. Se ha incluido el servicio creado de la misma forma que en el servidor. Empezando con la lógica del código, podemos observar que lo primero que hace es comprobar que se haya invocado con dos 2 argumentos exactamente, siendo el número a convertir y el tipo de conversión. En caso contrario, produce un error y termina la ejecución.

```
1 int main(int argc, char **argv)
2 {
3     rclcpp::init(argc, argv);
4
5     if (argc != 3) {
6         RCLCPP_ERROR(rclcpp::get_logger("rclcpp"), "Uso: cliente <temperatura> <tipo_conversion>");
7         return 1;
8     }
9 }
```

Si está todo correcto, crea un nodo llamado `temperature_client` y un cliente para el servicio `convert_temperature` usando la definición del servicio `ConvertTemp`, creado por nosotros:

```
1 auto node = rclcpp::Node::make_shared("temperature_client");
2 auto client = node->create_client<service_temp::srv::ConvertTemp>("convert_temperature");
3
```

Después, espera a que el servidor esté disponible y después crea un objeto request del servicio, asignando la temperatura indicada por parámetro y el tipo de conversión. Por último, envía la solicitud de forma asíncrona (no bloquea la ejecución), ya que después se utiliza `spin_until_future_complete` para realizar dicho bloqueo hasta que llegue la respuesta o falle, imprimiendo el resultado y finalizando su ejecución.

```
1 while (!client->wait_for_service(1s)) {
2     RCLCPP_INFO(node->get_logger(), "Esperando al servicio...");
3 }
4 auto request = std::make_shared<service_temp::srv::ConvertTemp::Request>();
5 request->input_temp = std::stod(argv[1]);
6 request->conversion_type = argv[2];
7 auto result = client->async_send_request(request);
8
```

```

9   if (rclcpp::spin_until_future_complete(node, result) ==
10       rclcpp::FutureReturnCode::SUCCESS)
11   {
12       RCLCPP_INFO(node->get_logger(), "Resultado: %.2f", result.get()->converted_temp);
13   } else {
14       RCLCPP_ERROR(node->get_logger(), "Fallo al llamar al servicio");
15   }
16   rclcpp::shutdown();
17   return 0;
18 }

```

Una vez se han creado los nodos de cliente y servidor, hay que realizar otra vez unos pequeños cambios en el fichero CMakeLists.txt, para añadir los ejecutables y la instalación de los nodos creados anteriormente. Para ello, se han añadido las siguientes líneas:

```

1  # Ejecutables
2  add_executable(temperature_server src/temperature_server.cpp)
3  ament_target_dependencies(temperature_server rclcpp)
4  # Vinculo con las interfaces generadas
5  rosidl_target_interfaces(temperature_server ${PROJECT_NAME} "rosidl_typesupport_cpp")
6
7  add_executable(temperature_client src/temperature_client.cpp)
8  ament_target_dependencies(temperature_client rclcpp)
9  rosidl_target_interfaces(temperature_client ${PROJECT_NAME} "rosidl_typesupport_cpp")
10
11 # Instalación
12 install(TARGETS
13     temperature_server
14     temperature_client
15     DESTINATION lib/${PROJECT_NAME}
16 )
17
18 ament_export_dependencies(rosidl_default_runtime)

```

Con esto correctamente configurado, ya podemos probar el servicio implementado, para ello, lo primero que hay que hacer es realizar la compilación utilizando *colcon build --packages-select service_temp* estando en la carpeta *ros2_ws*. Cuando haya compilado, en cualquier terminal que tengamos o que abramos hay que ejecutar *source install/setup.bash*. Y por último, solo hay que ejecutar los nodos, para ello:

Terminal 1:

```
ros2 run service_temp temperature_server
```

Terminal 2:

```
ros2 run service_temp temperature_client <Temperatura> <Conversión>
```

En la figura 24 se puede visualizar un ejemplo de ejecución realizando dos peticiones distintas al servidor, una para cada tipo de conversión.


```
jairo-docker@jairubuntu:/workspace/ros2_ws$ ros2 run service_temp temperature_server
[INFO] [1748353275.011073231] [temperature_server]: Solicitud: 212.00 Far_to_Cel
-> 100.00
[INFO] [1748353300.410131887] [temperature_server]: Solicitud: 100.00 Cel_to_Far
-> 212.00
```

(a) Funcionamiento servidor

```
jairo-docker@jairubuntu:/workspace/ros2_ws$ ros2 run service_temp temperature_client 212 Far_to_Cel
[INFO] [1748353275.011764072] [temperature_client]: Resultado: 100.00
jairo-docker@jairubuntu:/workspace/ros2_ws$ ros2 run service_temp temperature_client 100 Cel_to_Far
[INFO] [1748353300.410267155] [temperature_client]: Resultado: 212.00
jairo-docker@jairubuntu:/workspace/ros2_ws$
```

(b) Funcionamiento cliente

Figure 24: Ejemplo ejecución servicio conversor de temperaturas.

Pregunta 4.

¿Dónde van los archivos .srv en un paquete ROS 2?

Como se ha podido observar durante la realización del ejercicio 1, estos ficheros se añaden dentro de una carpeta llamada `srv`, situada dentro del paquete. Este directorio se declara automáticamente como parte de las interfaces generadas al configurarse correctamente en el `CMakeLists.txt` y `package.xml` usando `rosidl_generate_interfaces()` y las dependencias `rosidl_default_generators` y `rosidl_default_runtime`.

Pregunta 5.

¿Cuáles son las principales diferencias en cómo se implementan y ejecutan los servicios entre ROS Noetic y ROS 2 Humble?

En la siguiente tabla se muestran algunos de las diferencias más importantes a la hora del uso y la implementación de servicios entre ROS 1 y ROS 2.

Aspecto	ROS Noetic (ROS 1)	ROS 2 Humble
Creación del servicio	Se usa <code>rospy.Service</code> o <code>ros::ServiceServer</code> directamente	Se usa <code>create_service()</code> desde una clase que hereda de <code>Node</code>
Cliente del servicio	<code>rospy.ServiceProxy</code> o <code>ros::ServiceClient</code>	<code>create_client()</code> y uso de llamadas asíncronas
Manejo del nodo	Nodo global con <code>ros::init()</code> , sin clases por defecto	Enfoque orientado a objetos: los nodos se definen como clases derivadas de <code>Node</code>
Ejecución del servicio	Se usa <code>spin()</code> global que bloquea todo el hilo principal	Se puede usar <code>rclcpp::spin()</code> con múltiples nodos o en hilos separados
Concurrencia	Limitada, ejecución monohilo por defecto	Soporte para múltiples hilos mediante <code>MultiThreadedExecutor</code>

Table 2: Comparativa entre implementación y ejecución de servicios en ROS Noetic y ROS 2 Humble

En resumen, ROS 2 introduce un enfoque más moderno y estructurado, basado en programación orientada a objetos y con mejor soporte para la concurrencia y ejecución asíncrona. Mientras que en ROS Noetic se trabaja con funciones globales y estructuras más simples, ROS 2 permite diseñar nodos como clases y manejar servicios de forma no bloqueante, facilitando arquitecturas más escalables y robustas.

Pregunta 6.

¿En qué se diferencia rclpy en ROS 2 de rospy en cuanto al manejo de servicios?

Entre las principales diferencias podemos encontrar que rclpy permite un uso basado en clases y objetos, además de ser mucho más flexible y modular que rospy, siendo este más procedural. Además, rclpy presenta una comunicación basada en DDS (Data Distribution Service), lo que proporciona mayor robustez, tolerancia a fallos y un sistema de Calidad de Servicio (QoS) configurable, permitiendo adaptar la comunicación a las necesidades del sistema.

Otra ventaja de rclpy es que permite servicios asíncronos de forma nativa utilizando `async_send_request`, lo cual es muy útil en sistemas concurrentes o cuando se desea evitar bloqueos del nodo.

Por otro lado, rospy está basado en `ros::init` y hace uso extensivo de variables y funciones globales, como `rospy.Service`, lo que puede dificultar la escalabilidad del sistema y su reutilización. Además, la comunicación en rospy se basa en un middleware más antiguo (ROS Master + TCPROS), que no ofrece las capacidades de configuración ni la robustez de DDS.

En resumen, rclpy está diseñado para sistemas modernos y más complejos, mientras que rospy es más sencillo y adecuado para aplicaciones pequeñas o de propósito educativo.

Pregunta 7.

¿Cuál es el comportamiento del cliente del servicio en ROS 2 cuando el servidor aún no está disponible?

Cómo se ha explicado en el código que hemos realizado, se queda en bucle esperando a que el servicio esté disponible para realizar la petición. Hasta que no pueda enviarlo no va a realizar ninguna otra acción, solo imprimirá por pantalla, tal y como se puede ver en la Figura 25.

```
jairo-docker@jairubuntu:/workspace/ros2_ws$ ros2 run service_temp temperature_client 100 a
[INFO] [1748370412.141396346] [temperature_client]: Esperando al servicio...
[INFO] [1748370413.141900477] [temperature_client]: Esperando al servicio...
[INFO] [1748370414.142143563] [temperature_client]: Esperando al servicio...
[INFO] [1748370415.142438417] [temperature_client]: Esperando al servicio...
[INFO] [1748370416.142684041] [temperature_client]: Esperando al servicio...
[INFO] [1748370417.143210475] [temperature_client]: Esperando al servicio...
[INFO] [1748370418.143837893] [temperature_client]: Esperando al servicio...
[INFO] [1748370419.144395896] [temperature_client]: Esperando al servicio...
```

Figure 25: Cliente esperando a que el servidor esté disponible.

Ejercicio 2.

Implementa un sistema basado en acciones ROS 2 utilizando Python, para ello deberás crear un nuevo paquete llamado `battery_act`, puedes encontrar la info necesaria en el siguiente enlace y en el siguiente. Esta acción se va a encargar de simular un proceso de descarga de la batería de un robot, enviando información sobre el progreso y devolviendo

un resultado al finalizar. El objetivo es establecer un valor de batería en el cual el robot debe enviar un aviso de "batería baja" para proceder a su carga. El funcionamiento de la acción sería el siguiente:

El nodo Servidor llamado `battery_charger`, recibirá el Goal que será el valor de batería, en tanto porcentual, en el cual el robot deberá mandar el aviso (Ej: 20%). Por tanto, el robot partirá de un 100% de batería y esta se irá reduciendo en un 5% por cada segundo que pase. El Servidor deberá ir actualizando el valor del Feedback mostrando el valor actual de la batería del robot. Asimismo, el Servidor devolverá como Result un aviso (Ej: "Batería Baja, por favor cargue el robot!"). Ten en cuenta que la acción debe poder cancelar si se manda la petición de cancelación. El nodo Cliente llamado `battery_client`, enviará el Goal al servidor teniendo en cuenta que este valor se recogerá desde la línea de comandos al lanzar el nodo. El nodo Cliente deberá ir publicando el Feedback de la acción, así como el mensaje del Result una vez haya finalizado la acción.

La creación y configuración del paquete va a ser muy similar a la realizada durante el ejercicio 1. La primera diferencia viene dada en las dependencias, ya que al crear el paquete con el comando `ros2 pkg create -build-type ament_cmake -license Apache-2.0 battery_act -dependencies rclpy action_msgs` podemos observar que mantenemos el build de cmake, por la facilidad a la hora de crear las acciones, pero en las dependencias tenemos rclpy ya que el sistema va a estar implementado en python. De igual manera, también se ha añadido la dependencia de `action_msgs`, para poder crear las acciones pertinentes.

Con el paquete ya creado, el siguiente punto es configurar el paquete para poder crear la acción sin problemas. El primer paso es crear una carpeta llamada `action` dentro de la carpeta del paquete, y dentro se ha creado un archivo llamado `Battery.action`. Su contenido indica la estructura de la que va a disponer la acción, estando formada por el goal, que es el objetivo a cumplir, el resultado, que es un string donde se indicará que la batería está baja, y el feedback, que irá indicando el porcentaje de batería que va quedando en cada instante de tiempo.

```
1  # Goal
2  int32 target_percentage
3  ---
4  # Result
5  string warning
6  ---
7  # Feedback
8  int32 current_percentage
9
```

Con la acción ya creada, solo hay que configurar los archivos de configuración pertinentes. Al igual que en el ejercicio anterior, estos son el `CMakeLists.txt` y el `package.xml`. Empezando por el `package.xml`, tienen que añadirse las mismas dependencias que en el ejercicio anterior, además de las de `action_msgs`, ya que son las que permiten el envío de los mensajes de las acciones.

```

1  <!-- Python dependencies -->
2  <buildtool_depend>ament_cmake</buildtool_depend>
3  <exec_depend>rclpy</exec_depend>
4
5  <!-- Action interface dependencies -->
6  <build_depend>roslaunch</build_depend>
7  <exec_depend>roslaunch</exec_depend>
8  <build_depend>action_msgs</build_depend>
9  <exec_depend>action_msgs</exec_depend>
10 <build_depend>builtin_interfaces</build_depend>
11 <exec_depend>builtin_interfaces</exec_depend>
12
13 <member_of_group>roslaunch_interface_packages</member_of_group>

```

Podemos observar en el fragmento de código las líneas que hay que añadir para configurar correctamente las dependencias. Continuando con el CMakeLists.txt, tiene que presentar los elementos mostrados a continuación:

Podemos observar que se ha añadido la búsqueda del paquete de action_msgs. Como hemos indicado anteriormente, este paquete es necesario para poder enviar los mensajes de la acción sin tener ningún problema. De igual manera, se ha añadido la generación de la interfaz de la acción que hemos creado, para poder utilizarla en nuestro código. Por último, se han definido los scripts que van a contener el servidor y el cliente, para que se instalen correctamente al realizar la compilación.

Con estos archivos configurados correctamente, ya podemos escribir el código del servidor y del cliente. Empezando con el servidor, se ha creado un archivo llamado battery_charger.py, que contiene toda la lógica del servidor. El servidor funciona de tal manera que se puede cancelar la acción en tiempo real. Si la batería final es 20% o menor, se envía un mensaje crítico y sino, una advertencia. También se ha configurado el mismo para que solo se pueda ejecutar una acción, para simplificar el problema. Para lograr este objetivo vamos a centrarnos en el constructor, cuyo código es el siguiente:

```

1  from battery_act.action import Battery
2
3
4  class BatteryChargerServer(Node):
5      def __init__(self):
6          super().__init__('battery_charger')
7          self._goal_handle = None
8          self._goal_lock = threading.Lock()
9          self._action_server = ActionServer(
10             self,
11             Battery,
12             'battery_monitor',
13             execute_callback=self.execute_callback,
14             goal_callback=self.goal_callback,
15             handle_accepted_callback=self.handle_accepted_callback,
16             cancel_callback=self.cancel_callback,
17             callback_group=ReentrantCallbackGroup()

```

```

18         )
19         self._current_battery = 100

```

Podemos ver al principio cómo importar la acción que hemos creado nosotros para utilizarla. De igual manera, podemos observar que el constructor se inicia un `action_server` relacionado con la acción `Battery` y cuyo nombre va a ser `battery_monitor`. Este servidor está configurado por diversos métodos que van a encargarse de la lógica del mismo. El `execute_callback` va a encargarse de procesar y realizar toda la petición, comprobando si se cancela o no la acción. El `goal_callback` se va a encargar de aceptar la petición y enviar la aceptación al cliente. El `handle_accpeted_callbnack` se va a encargar de comprobar si se está realizando ya una petición, cancelarla y aceptar la nueva. Por último, el `cancel_callback` se va a encargar de procesar las solicitudes de cancelación. Ahora vamos a analizar cómo se han implementado los métodos para realizar dichas anteriormente.

```

1     def handle_accepted_callback(self, goal_handle):
2         with self._goal_lock:
3             if self._goal_handle is not None and self._goal_handle.is_active:
4                 self.get_logger().info('Cancelando el objetivo anterior antes de aceptar uno nuevo.')
5                 self._goal_handle.abort()
6                 self._goal_handle = goal_handle
7             self.get_logger().info('Nuevo objetivo aceptado.')
8             goal_handle.execute()
9
10
11     def cancel_callback(self, goal_handle):
12         self.get_logger().info('Cancelación solicitada.')
13         return CancelResponse.ACCEPT
14
15     def goal_callback(self, goal_request):
16         self.get_logger().info(f'Recibido objetivo: {goal_request.target_percentage}%')
17         return GoalResponse.ACCEPT

```

Podemos observar en el código que para comprobar si se está realizando una petición se comprueba el `goal_handle` y si está activo se aborta el objetivo y se acepta el que está entrando. Por otro lado, la cancelación del objetivo se encarga de enviar la aceptación de cancelación del objetivo. De igual manera, aceptar un objetivo consiste en enviar el mensaje de aceptación del objetivo antes de comenzar a realizar la lógica necesaria para obtener dicho objetivo.

```

1     def execute_callback(self, goal_handle):
2         self._current_battery = 100
3         target = goal_handle.request.target_percentage
4         self.get_logger().info(f'Comenzando descarga hacia {target}%')
5
6         feedback_msg = Battery.Feedback()
7         while self._current_battery > target:
8             if not goal_handle.is_active:
9                 self.get_logger().info('Abortando ejecución, el objetivo ya no está activo.')
10                return Battery.Result()
11
12         if goal_handle.is_cancel_requested:

```

```

13         goal_handle.canceled()
14         self.get_logger().info('Request de cancelación recibido, abortando ejecución.')
15         return Battery.Result()
16
17         time.sleep(1)
18         self._current_battery -= 5
19         feedback_msg.current_percentage = self._current_battery
20         self.get_logger().info(f'Batería actual: {self._current_battery}%')
21         goal_handle.publish_feedback(feedback_msg)
22
23
24         result = Battery.Result()
25         if self._current_battery <= 20:
26             result.warning = '¡Advertencia! Bateria críticamente baja.'
27         else:
28             result.warning = 'Se está agotando la batería. Conecte el cargador.'
29         self.get_logger().info('Objetivo alcanzado.')
30         with self._goal_lock:
31             if not goal_handle.is_active:
32                 self.get_logger().info('Goal aborted')
33                 return Battery.Result()
34
35         goal_handle.succeed()
36
37         return result

```

El método que se encarga de llegar a dicho objetivo es el más largo, sin embargo, es muy simple. Consiste en un bucle principal, en el que al principio de cada iteración se comprueba si el nodo cliente está activo o si ha enviado un mensaje de cancelación. En cualquiera de estos dos casos se cancela la acción y se queda el servidor a la espera de otra. Si está todo correcto, espera 1 segundo, resta 5 a la batería y envía el porcentaje actual de batería. Cuando se llega al objetivo, se envía el mensaje con la alerta de la batería, se marca el goal como completado y el servidor pasa a esperar otra solicitud.

Una vez analizado el código relacionado con el servidor, vamos a pasar al código del cliente implementado. Su lógica es muy simple, se invoca indicándole por parámetro la batería límite, se pone en contacto con el servidor para que le monitoree la batería que le queda y va indicando el porcentaje de batería restante, finalizando con el mensaje de warning pertinente. Para entender mejor cómo se ha implementado, vamos a ir explicando el código relacionado con esta lógica, así como analizando su significado.

```

1 def __init__(self):
2     super().__init__('battery_client')
3     self._action_client = ActionClient(self, Battery, 'battery_monitor')
4     self._goal_handle = None
5     self._shutdown_requested = False
6
7     def cancel_done(self, future):
8         cancel_response = future.result()
9         if len(cancel_response.goals_canceling) > 0:
10             self.get_logger().info('Goal successfully canceled')
11         else:

```

```

12         self.get_logger().info('Goal failed to cancel')
13
14     rclpy.shutdown()
15
16     def goal_response_callback(self, future):
17         goal_handle = future.result()
18         if not goal_handle.accepted:
19             self.get_logger().info('Objetivo rechazado.')
20             return
21
22         self.get_logger().info('Objetivo aceptado')
23         self._goal_handle = goal_handle
24         get_result_future = goal_handle.get_result_async()
25         get_result_future.add_done_callback(self.get_result_callback)

```

Empezando con el constructor, podemos observar que se crea un `action_client` que va vinculado al servidor creado anteriormente. De igual manera, podemos observar en el método `cancel_done`, que este método va a monitorear cuando se solicita la cancelación si esta ha sido completada con éxito o no. En cuanto al `goal_response_callback`. Este método se va a encargar de confirmar que el objetivo ha sido aceptado por el servidor, asignar el `goal_handle` y preparar la obtención del resultado final una vez se haya completado el objetivo.

```

1     def feedback_callback(self, feedback_msg):
2         porcentaje_actual = feedback_msg.feedback.current_percentage
3         self.get_logger().info(f'Feedback: batería actual {porcentaje_actual}%')
4
5
6     def send_goal(self, porcentaje_objetivo):
7         self.get_logger().info('Esperando al servidor de acción...')
8         self._action_client.wait_for_server()
9
10        goal_msg = Battery.Goal()
11        goal_msg.target_percentage = porcentaje_objetivo
12
13        self.get_logger().info(f'Enviando objetivo: {porcentaje_objetivo}%')
14        self._send_goal_future = self._action_client.send_goal_async(
15            goal_msg,
16            feedback_callback=self.feedback_callback
17        )
18        self._send_goal_future.add_done_callback(self.goal_response_callback)
19
20    def cancel_goal(self):
21        self.get_logger().info('Solicitando cancelación del objetivo...')
22        future = self._goal_handle.cancel_goal_async()
23        future.add_done_callback(self.cancel_done)
24
25
26    def get_result_callback(self, future):
27        result = future.result().result
28        self.get_logger().info(f'Resultado recibido: {result.warning}')
29        self._shutdown_requested = True

```

Los siguientes métodos completan la lógica del cliente. En `feedback_callback` podemos observar que se obtiene el porcentaje actual de batería y se muestra su contenido. El método

send_goal se encarga de enviar el objetivo. Podemos observar que esperamos a que el servidor esté disponible, creamos un mensaje goal y le añadimos el porcentaje que se ha pasado por parámetro, a la vez que preparamos los métodos de callback para ir obteniendo la información referente al servidor. Por último, el método cancel_goal se va a llamar cuando se pulse CTRL+C en el teclado y se encarga de enviar el mensaje de petición de cancelación del objetivo. El método get_result_callback se encarga de obtener el resultado una vez se ha completado el objetivo y mostrarlo.

Con la lógica mencionada anteriormente ya están preparados tanto el cliente como el servidor, solo queda realizar la compilación y la ejecución de los mismos. Es importante mencionar que una vez compilado con *colcon build --packages-select battery_act* es necesario realizar el *source install/setup.bash* en todas las terminales, estén abiertas o se abran después. Con la compilación hecha ya solo queda probarlo ejecutando los siguientes comandos:

Terminal 1:

```
ros2 run battery_act battery_charger.py
```

Terminal 2:

```
ros2 run battery_act battery_client.py <Límite de batería>
```

En la figura 26 y 27 se pueden visualizar ejemplos de ejecución, una realizando la ejecución completa hasta llegar al objetivo y la otra con una cancelación por parte del cliente.

```

jairo-docker@jairoubuntu:/workspace/ros2_ws$ ros2 run battery_act battery_charger.py
[INFO] [1748510172.808972865] [battery_charger]: Recibido objetivo: 20%
[INFO] [1748510172.810258680] [battery_charger]: Nuevo objetivo aceptado.
[INFO] [1748510172.813017923] [battery_charger]: Comenzando descarga hacia 20%
[INFO] [1748510173.814754281] [battery_charger]: Batería actual: 95%
[INFO] [1748510174.816965991] [battery_charger]: Batería actual: 90%
[INFO] [1748510175.819487911] [battery_charger]: Batería actual: 85%
[INFO] [1748510176.820935512] [battery_charger]: Batería actual: 80%
[INFO] [1748510177.822801226] [battery_charger]: Batería actual: 75%
[INFO] [1748510178.825052102] [battery_charger]: Batería actual: 70%
[INFO] [1748510179.827068047] [battery_charger]: Batería actual: 65%
[INFO] [1748510180.829386006] [battery_charger]: Batería actual: 60%
[INFO] [1748510181.832303921] [battery_charger]: Batería actual: 55%
[INFO] [1748510182.835197191] [battery_charger]: Batería actual: 50%
[INFO] [1748510183.837227710] [battery_charger]: Batería actual: 45%
[INFO] [1748510184.839823281] [battery_charger]: Batería actual: 40%
[INFO] [1748510185.842767350] [battery_charger]: Batería actual: 35%
[INFO] [1748510186.844537585] [battery_charger]: Batería actual: 30%
[INFO] [1748510187.847128804] [battery_charger]: Batería actual: 25%
[INFO] [1748510188.849264418] [battery_charger]: Batería actual: 20%
[INFO] [1748510188.849878126] [battery_charger]: Objetivo alcanzado.
[INFO] [1748510193.390354147] [battery_charger]: Recibido objetivo: 20%
jairo-docker@jairoubuntu:/workspace/ros2_ws$ ros2 run battery_act battery_client
.py 20
[INFO] [1748510172.782198715] [battery_client]: Esperando al servidor de acción.
[INFO] [1748510172.782763356] [battery_client]: Enviando objetivo: 20%
[INFO] [1748510172.810579080] [battery_client]: Objetivo aceptado
[INFO] [1748510173.816072100] [battery_client]: Feedback: batería actual 95%
[INFO] [1748510174.818687081] [battery_client]: Feedback: batería actual 90%
[INFO] [1748510175.821427650] [battery_client]: Feedback: batería actual 85%
[INFO] [1748510176.821907701] [battery_client]: Feedback: batería actual 80%
[INFO] [1748510177.824237744] [battery_client]: Feedback: batería actual 75%
[INFO] [1748510178.825919675] [battery_client]: Feedback: batería actual 70%
[INFO] [1748510179.828238938] [battery_client]: Feedback: batería actual 65%
[INFO] [1748510180.831748670] [battery_client]: Feedback: batería actual 60%
[INFO] [1748510181.834599387] [battery_client]: Feedback: batería actual 55%
[INFO] [1748510182.837691318] [battery_client]: Feedback: batería actual 50%
[INFO] [1748510183.838411234] [battery_client]: Feedback: batería actual 45%
[INFO] [1748510184.841337524] [battery_client]: Feedback: batería actual 40%
[INFO] [1748510185.844598306] [battery_client]: Feedback: batería actual 35%
[INFO] [1748510186.845571270] [battery_client]: Feedback: batería actual 30%
[INFO] [1748510187.848269719] [battery_client]: Feedback: batería actual 25%
[INFO] [1748510188.850384003] [battery_client]: Feedback: batería actual 20%
[INFO] [1748510188.852399053] [battery_client]: Resultado recibido: ¡Advertencia
! Batería críticamente baja.

```

Figure 26: Ejecución completa de monitorización de batería.

```

jairo-docker@jairoubuntu:/workspace/ros2_ws$ ros2 run battery_act battery_charger.py
[INFO] [1748510187.847128804] [battery_charger]: Batería actual: 25%
[INFO] [1748510188.849264418] [battery_charger]: Batería actual: 20%
[INFO] [1748510188.849878126] [battery_charger]: Objetivo alcanzado.
[INFO] [1748510193.390354147] [battery_charger]: Recibido objetivo: 20%
[INFO] [1748510193.390993324] [battery_charger]: Nuevo objetivo aceptado.
[INFO] [1748510193.392379717] [battery_charger]: Comenzando descarga hacia 20%
[INFO] [1748510194.394993854] [battery_charger]: Batería actual: 95%
[INFO] [1748510195.396525858] [battery_charger]: Batería actual: 90%
[INFO] [1748510196.398776227] [battery_charger]: Batería actual: 85%
[INFO] [1748510197.401273920] [battery_charger]: Batería actual: 80%
[INFO] [1748510198.404235740] [battery_charger]: Batería actual: 75%
[INFO] [1748510198.720806333] [battery_charger]: Cancelación solicitada.
[INFO] [1748510199.406342955] [battery_charger]: Batería actual: 70%
[INFO] [1748510199.407301178] [battery_charger]: Request de cancelación recibido, abortando ejecución.
jairo-docker@jairoubuntu:/workspace/ros2_ws$ ros2 run battery_act battery_client
.py 20
[INFO] [1748510193.388326192] [battery_client]: Esperando al servidor de acción.
[INFO] [1748510193.388887783] [battery_client]: Enviando objetivo: 20%
[INFO] [1748510193.391246366] [battery_client]: Objetivo aceptado
[INFO] [1748510194.396238804] [battery_client]: Feedback: batería actual 95%
[INFO] [1748510195.397644311] [battery_client]: Feedback: batería actual 90%
[INFO] [1748510196.401338883] [battery_client]: Feedback: batería actual 85%
[INFO] [1748510197.402572590] [battery_client]: Feedback: batería actual 80%
[INFO] [1748510198.405936592] [battery_client]: Feedback: batería actual 75%
^C[INFO] [1748510198.725358365] [battery_client]: Ctrl+C pulsado: solicitando cancelación del objetivo...
Failed to publish log message to rosout: publisher's context is invalid, at ./src/crc/publisher.c:389
[INFO] [1748510198.725800523] [battery_client]: Solicitando cancelación del obje

```

Figure 27: Cancelación de monitorización de batería.

Pregunta 8.

Describe la arquitectura de una acción en ROS 2. ¿En qué se diferencia del sistema basado en `actionlib` de ROS 1?

Una acción en ROS 2 es una abstracción construida sobre los mecanismos de DDS que permite ejecutar tareas que llevan tiempo y reciben actualizaciones intermedias. Utiliza los siguientes tipos de comunicación:

- **Goal (SendGoal).** El cliente envía un objetivo al servidor.
- **Feedback.** El servidor envía actualizaciones periódicas al cliente.
- **Result (GetResult).** El cliente recibe el resultado final.
- **Cancel (CancelGoal).** El cliente puede solicitar la cancelación de un objetivo en ejecución.

De igual manera, como se ha podido ver en el desarrollo del ejercicio, podemos encontrar diferentes componentes principales, como el `ActionClient` o `ActionServer` que nos permiten crear nodos que ejecutan y crean una acción. También es importante mencionar los mensajes generados a partir del archivo `.action`, que crea los mensajes `Goal`, `Result` y `Feedback`.

Comparándolo con ROS 1, podemos encontrar las siguientes diferencias:

Característica	ROS 1 (<code>actionlib</code>)	ROS 2 (<code>rclpy</code> , <code>rclcpp</code>)
Transporte	TCPROS	DDS
Integración	Separada del sistema de ROS	Integrada nativamente
QoS (Calidad de Servicio)	No configurable	Configurable (QoS por DDS)
Concurrencia	Limitada	Mejor con <code>MultiThreadedExecutor</code>
Seguridad, escalabilidad	Limitada	Mejora gracias a DDS
Simplicidad	Menos estructurado	Más estructurado y robusto

Table 3: Comparación entre acciones en ROS 1 y ROS 2

Podemos observar que una de las diferencias clave es el mecanismo de transporte: mientras que ROS 1 utiliza TCPROS, ROS 2 se basa en DDS (Data Distribution Service), lo que permite una comunicación más fiable y flexible. ROS 2 también ofrece integración nativa del sistema de acciones dentro del middleware, en contraste con ROS 1, donde `actionlib` es una librería separada.

Además, ROS 2 permite configurar políticas de Calidad de Servicio (QoS), lo que es fundamental para ajustar la fiabilidad, latencia y comportamiento de la red según las necesidades de la aplicación. En cuanto a la concurrencia, ROS 2 proporciona herramientas como el `MultiThreadedExecutor` para manejar múltiples hilos de ejecución, mejorando la capacidad de respuesta frente a múltiples eventos concurrentes.

Por último, la arquitectura basada en DDS en ROS 2 aporta mejoras significativas en términos de escalabilidad, seguridad y robustez, haciendo que el sistema de acciones sea más adecuado para aplicaciones industriales y distribuidas.

Pregunta 9.

¿Qué ocurre internamente cuando se cancela la acción en ROS 2? ¿en qué se diferencia con ROS Noetic?

En ROS 2, cuando el cliente llama a `cancel_goal_async()`, es decir, realiza una petición de cancelación:

- Se envía una solicitud de cancelación al servidor a través del tópico de cancelación `/action_name/_action/cancel_request`.
- El servidor recibe esta solicitud y verifica si el objetivo aún puede ser cancelado.
- Si es cancelable, el estado del objetivo cambia a `CANCELED`.
- El servidor debe terminar manualmente la ejecución del objetivo (el callback debe ser interrumpible).
- Se envía una respuesta de cancelación al cliente, y posteriormente un `Result` con el estado final.

De esta manera, una vez que sabemos cómo funciona internamente, podemos identificar las principales diferencias respecto a cómo funcionaba en ROS 1:

- En ROS 1, la cancelación se realiza mediante un mensaje *GoalID* publicado en el tópico `/goal_cancel`, y el servidor es responsable de comprobar periódicamente si debe cancelar la ejecución, mientras que en ROS 2 el proceso está formalizado mediante un servicio de cancelación explícito, con validación del estado del objetivo y una transición clara a *CANCELED*.
- ROS 2 utiliza una arquitectura más formal, con servicios separados y estados bien definidos, similar a una máquina de estados finitos (FSM).
- Gracias al uso de DDS y soporte para políticas de Calidad de Servicio (QoS), ROS 2 ofrece una gestión de cancelación más fiable y controlada.

Pregunta 10.

¿Cómo mejora el uso de DDS en ROS 2 la fiabilidad y escalabilidad de la comunicación basada en acciones en comparación con ROS 1?

DDS (Data Distribution Service) mejora significativamente la arquitectura de ROS 2 respecto a ROS 1 en varios aspectos clave:

Aspecto	Mejora con DDS (ROS 2)
Fiabilidad	DDS permite configurar políticas QoS como <i>Reliable</i> , garantizando la entrega de mensajes críticos.
Escalabilidad	DDS permite múltiples suscriptores y publicadores distribuidos sin depender de un nodo maestro.
Seguridad	DDS soporta mecanismos de seguridad nativos, como cifrado de mensajes y autenticación entre nodos.
Desacoplamiento	Comunicación completamente descentralizada: los nodos se descubren automáticamente en la red.
Tolerancia a fallos	Al eliminar el nodo maestro presente en ROS 1, se reducen los puntos únicos de fallo del sistema.
Latencia predecible	DDS permite configuraciones en tiempo real y control fino de latencia para aplicaciones críticas.

Table 4: Mejoras aportadas por DDS a la arquitectura de ROS 2

En resumen, DDS dota a ROS 2 de una base mucho más robusta y configurable que los protocolos TCPROS/UDPROS de ROS 1. Esto resulta especialmente útil para sistemas distribuidos o de tiempo real, donde las acciones requieren una comunicación fiable, feedback constante y capacidades de cancelación o resultados bajo condiciones de carga.