Práctica 3: Explorando nuevas herramientas

Alejandro Pérez Blasco

1 PARTE 1: SLAM con Turtlebot 3

`Pregunta 1: Analiza el archivo .yaml y explica que significa cada uno de los campos que muestran.`

Los campos que aparecen en el archivo .yaml son: image, resolution, origin, occupied_thresh, free_thresh y negate [1].

El campo `image` indica la ruta al fichero imagen donde están contenidos los datos de ocupación del mapa.

El campo `resolution` indica la resolución del mapa, representado en metros por píxeles.

El campo `origin` indica la posición en dos dimensiones empleando como referencia el píxel situado en la esquina inferior izquierda. También permite saber la rotación, empleando el valor 0.0 como indicador de no rotación.

El campo `ocuppied_thresh` indica el valor a partir del cual la probabilidad de ocupación indica si un píxel está completamente ocupado o no. Si la probabilidad de ocupación es mayor que el valor, el píxel se considerará completamente ocupado. Por otro lado, el campo `free_thresh` indica el valor a partir del cual la probabilidad de ocupación indica si un píxel está completamente libre o no. Si la probabilidad de ocupación está por debajo de dicho valor, el píxel se considerará completamente libre.

Por último, el campo `negate` puede tener dos valores (0,1) e indica si las interpretaciones de los píxeles al considerarlos libres/ocupados se debe invertir o no. Es decir, dependiendo del valor de negate, los píxeles ocupados y libres invertirán sus valores o no. Esto sirve para que no haya problemas a la hora de considerar las celdas vacías y ocupadas de mapas donde los píxeles de ocupación y no ocupación se consideran con valores opuestos. Por ejemplo, es posible que en un mapa las celdas ocupadas se representen con el color negro, y otro mapa donde el color blanco es el que indica la ocupación.

`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?`

El "recuadro de colores" viene dado por el costmap del mapa local. EL costmap es una estructura que permite almacenar y actualizar información de las celdas ocupadas empleando sensores con el LiDAR o cámaras y sirve para que el robot sepa por dónde puede navegar para evitar colisionar [2].

En cuanto a los colores cálidos/fríos, estos indican si esa celda está ocupada o no, siendo el color cálido el indicativo de obstáculo, y el frío de celda libre.

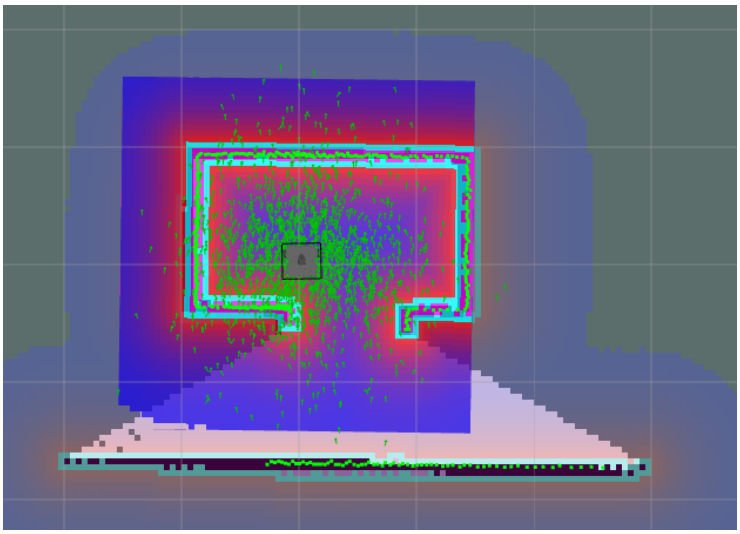

Figure 1: Ejemplo de distribución de colores cálidos/fríos en el mapa.

`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).`

Para la planificación de trayectoria, ROS emplea el paquete navfn, el cual emplea el costmap para trazar la trayectoria de coste mínimo haciendo uso del algoritmo de Dijkstra. Este algoritmo divide el mapa en celdas y trata de comprobar cuál es la manera menos costosa de llegar a cada una de esas celdas. Para ello, comprueba todas las formas de llegar a cada una de las celdas, y le asigna un valor que corresponde a la distancia más corta posible que hay que recorrer para llegar a dicha celda desde la celda inicial. Para obtener estos valores, va recorriendo los vecinos de cada una de las celdas (comenzando en la celda inicial) y asignando la distancia necesaria para llegar a esa celda en caso de que, recorriendo ese camino, se llegue recorriendo una distancia menor a la distancia que le ha sido asignada previamente (si esa celda no ha sido comprobada previamente, tendrá un valor muy alto o un valor booleano que indique si ha sido visitada o no) [3].

`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.`

Para hacer funcionar la navegación, se deben cargar en memoria los nodos: map_server, amcl, robot_state_publisher y move_base [4].

El nodo `map_server` proporciona información del mapa y lo proporciona en forma de servicio.

El nodo `amcl` inicia un filtrado de partículas y determina la posición del robot en el mapa.

El nodo `robot_state_publisher` publica el estado del robot y la posición 3D de los enlaces del robot.

El nodo `move_base` proporciona una acción con la cual, dada una posición destino, intentará alcanzarla empleando la base movible.

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 y responde a la siguiente pregunta:

Pregunta 2: ¿Observas diferencias en el mapeado respecto al primer entorno probado anteriormente?

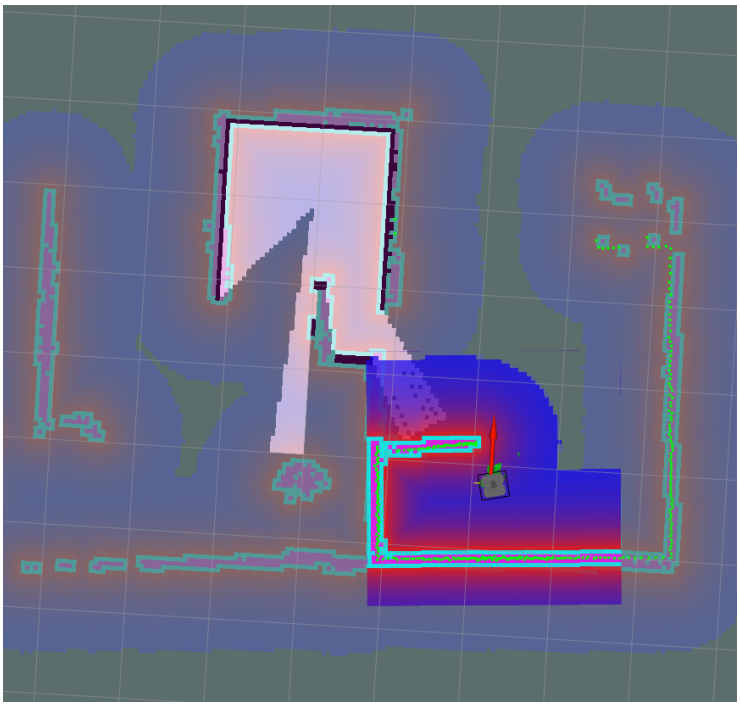

Figure 2: Mapeado mapa muchosObstáculos.

Como se puede apreciar en la Figura 2, el mapa está compuesto de muchos más obstáculos, pero el sistema de mapeado es igual al del primer entorno probado.

Ejercicio 2: Genera un entorno propio con obstáculos y con diferentes configuraciones. Asimismo, responde a la siguiente pregunta:

Pregunta 3: ¿Crees que hay cierto tipo de entornos en los que funciona mejor? (espacios abiertos, espacios pequeños, pasillos,...)

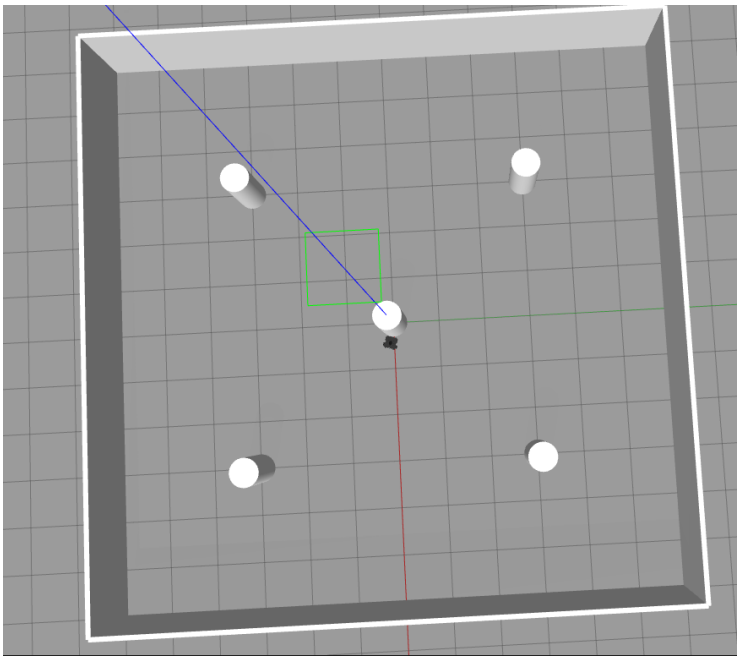

Figure 3: Entorno propio generado con obstáculos.

Opino que los espacios muy abiertos o muy estrechos es donde peores resultados se obtienen, ya que o no hay suficientes referencias (muy abierto), o está todo tan cerca que cuesta distinguir entre los distintos obstáculos o paredes. Por ello, donde mejores resultados se obtienen es en entornos donde los obstáculos están dispersos, pero cercanos entre ellos para poder emplearlos de referencia eficazmente.

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.

Para poder modificar estos parámetros, hay que abrir el archivo gmap-

ping_params.yaml de la carpeta config del paquete turtlebot3_slam.

Pregunta 4: ¿Cómo afectan estos parámetros en la generación del mapa?

El parámetro `particles` indica el número de partículas que emplea el filtro. Cuanto más alto sea este parámetro, más preciso será el algoritmo, pero también más lento. Por otro lado, los parámetros `linearUpdate` y `angularUpdate` indican la distancia y/o rotación que debe realizar el robot para que se realice un nuevo scaneo del mapa. Al igual que con el parámetro de cantidad de partículas, estos dos parámetros, cuanto más altos sean, más detalles se contemplarán, pero más lento y costoso computacionalmente será el proceso de mapeado.

Pregunta 5: ¿Qué información puedo extraer de un fichero bag?

Cuando vas a generar un fichero bag, empleas la herramienta `rqt_bag`, la cual te permite seleccionar los topics que quieras grabar. Una vez seleccionados, comienzas a grabar los mensajes que se publican en todos esos topics, que es la información que, posteriormente, puedes obtener del fichero bag empleando de nuevo rqt_bag [5].

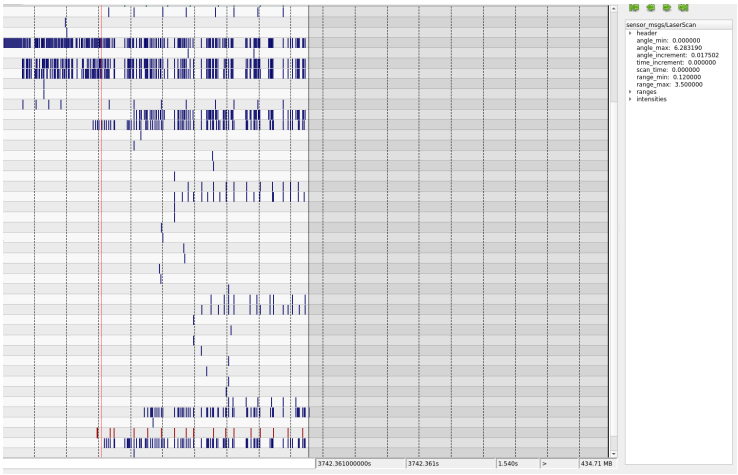

Figure 4: Ejemplo de lectura de mensajes emitidos por el LiDAR en el topic /scan.

Pregunta 6: ¿Se puede modificar la velocidad de reproducción del archivo? ¿Cómo se puede modificar? ¿Afecta a la resolución del mapa generado?

En el propio rqt_bag hay unos símbolos de flechas que te permiten aumentar la velocidad de reproducción. Si aumentas la velocidad de reproducción, la resolución del mapa generado puede verse afectada debido al aumento de frames por segundo que se realizan. Esto es debido al aumento de mensajes publicados

que se han de procesar en poco tiempo.

2 PARTE 2: Migrando a ROS 2 (Humble)

Pregunta 1: Dentro del espacio de trabajo, ¿Qué cambios notas respecto al espacio de trabajo en ROS Noetic?

Los principales cambios son la ausencia de la carpeta `devel`, y la creación de dos nuevas carpetas: `log` e `install`. Además, en vez de emplear catkin, ahora se utiliza colcon.

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?

Los paquetes creados en ROS 2 Humble tienen, en vez de una carpeta src como en ROS Noetic, una carpeta con el nombre del paquete. Esta carpeta tendrá, por defecto, un fichero "__init__.py" y será en esta carpeta donde se crearán los nodos.

Pregunta 3: ¿Qué son los archivos package.xml y setup.py? ¿Tienen alguna similitud con algún archivo de los paquetes de ROS Noetic?

El archivo `package.xml` mantiene el mismo formato que en ROS Noetic. Por otro lado, el archivo `setup.py` sirve, principalmente, para poder seleccionar los nodos ejecutables.

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.

Pregunta 4: ¿Dónde van los archivos .srv en un paquete ROS 2?

Para almacenar de manera organizada los archivos .srv, lo ideal es crear un paquete que contenga el archivo .srv dentro de una carpeta llamada `srv` [6]. Para crear este paquete, llamado `srv_service_temp`, ejecutamos el comando:

```
 ros2 pkg create --build-type
ament\_python --license Apache-2.0 srv\_service\_temp
--dependencies rclpy std\_msgs
```

Una vez creado el paquete, simplemente debemos acceder a él, crear una carpeta `srv`, y crear el fichero.srv (en este caso `Conversion.srv`) con la información proporcionada para la práctica.

Una vez se ha generado este paquete, se genera el paquete `service_temp`, en el cual se creará el servicio y se añadirá como dependencia el paquete generado anteriormente que contiene el fichero .srv.

Para ello, se añadirá al package.xml la línea: <depend>srv_service_temp< /depend>. Asimismo, se deberá añadir al CMakeLists.txt del paquete service_temp la línea: find_package(srv_service_temp REQUIRED).

Pregunta 5: ¿Cuáles son las principales diferencias en cómo se implementan y ejecutan los servicios entre ROS Noetic y ROS 2 Humble?

Para la ejecución, en ROS Noetic, se emplea el comando `rosrun <nombre_del_paquete> <nombre_del_servicio>`, mientras que ROS 2 Humble, emplea el comando `ros2 run <nombre_del_paquete> <nombre_del_servicio>`. Por otro lado, en cuanto a la implementación, la diferencia más notable es la librería princial. En ROS Noetic, se emplea la librería rospy, mientras que en ROS 2 Humble, la librería rclpy. Otra de las principales diferencias es la manera estandarizada de emplear ficheros .msg y ficheros .srv. En ros1, se creaban las carpetas contenedores de estos ficheros en el mismo paquete que, por ejemplo, los servicios que lo emplean. No obstante, en ros2 se crean en un paquete separado que luego debe ser añadido como dependencia.

Haciendo uso de ros2 run, podemos ejecutar el servidor y el cliente de este ejercicio.

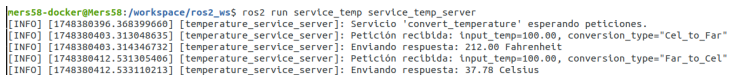

Figure 5: Ejecución del servidor del servicio.

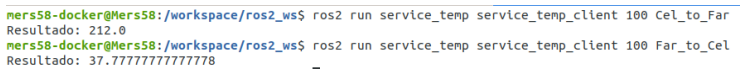

Figure 6: Ejecución del servidor del servicio.

Pregunta 6: ¿En qué se diferencia rclpy en ROS 2 de rospy en cuanto al manejo de servicios?

A diferencia de rospy, rclpy debe heredar de la clase Node. Asimismo, mientras que rospy inicia un nuevo nodo con `rospy.init_node()`, rlcpy emplea el constructo por defecto de Node para crear el nodo. Por otro lado, en cuanto a la creación del servicio, rospy emplea la clase Service para crearlo. En cuanto a rclpy, para crear un servicio empleándolo, se utiliza el método create_service [7].

Pregunta 7: ¿Cuál es el comportamiento del cliente del servicio en ROS 2 cuando el servidor aún no está disponible?

Cuando el servidor no está disponible, el cliente espera a que el servidor esté disponible.

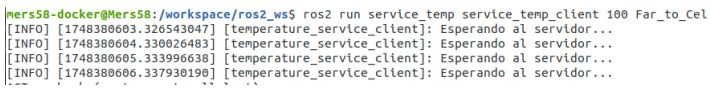

Figure 7: Prueba de ejecución de cliente sin lanzar servidor.

Esto es debido a que mi cliente espera a que el servidor esté disponible añadiendo el siguiente bucle: `while not self.cli.wait_for_service(timeout_sec=1.0)`. En caso de no tener ese bucle agregado, la goal se pierde ya que, aunque no salte error en esta situación, el servidor, al estar disponible, no recibe la goal que emitió el cliente cuando no lo estaba.

Ejercicio 2: Implementa un sistema basado en acciones ROS 2 utilizando Python, para ello deberás crear un nuevo paquete llamado battery_act. 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.

Lo primero que hay que hacer para realizar este ejercicio, es crear los ficheros battery_client.py y battery_charger.py dentro del paquete creado. Por otro lado, también se crea una nueva carpeta llamada `action`, dentro de la cual creamos el fichero `Battery.action`.

Una vez creados los ficheros .py, debemos asegurarnos de que hemos modificado pertinentemente los archivos: CMakeLists.txt y package.xml.

En el archivo `CMakeLists.txt`, debemos asegurarnos de añadir las siguientes líneas (siempre antes del `ament_package()`):

```
find_package(rosidl_default_generators REQUIRED)
rosidl_generate_interfaces(${PROJECT_NAME}
  "action/Battery.action"
)
```

Por otra parte, en el archivo `package.xml`, debemos asegurarnos de tener este fragmento añadido:

```
<buildtool_depend>ament_cmake</buildtool_depend>
<buildtool_depend>rosidl_default_generators</buildtool_depend>
<depend>action_msgs</depend>
<member_of_group>rosidl_interface_packages</member_of_group>
```

Una vez tenemos todos estos archivos correctamente configurados, podemos realizar una prueba ejecutando los siguientes comandos:

Para levantar el servidor, se ejecuta el comando: `python3 battery_charger.py`, y para ejecutar el cliente, el comando: `python3 battery_client.py`

```
mers58-docker@Mers58:/workspace/ros2_ws/src/battery_act$ python3 battery_charger.py
[INFO] [1748013921.362646631] [battery_charger]: Recibido nuevo goal: 20%
[INFO] [1748013921.365361559] [battery_charger]: Ejecutando acción de batería...
[INFO] [1748013921.366444170] [battery_charger]: Batería: 95%
[INFO] [1748013922.369413490] [battery_charger]: Batería: 90%
[INFO] [1748013923.372411925] [battery_charger]: Batería: 85%
[INFO] [1748013924.374646860] [battery_charger]: Batería: 80%
[INFO] [1748013925.377716250] [battery_charger]: Batería: 75%
[INFO] [1748013926.380717432] [battery_charger]: Batería: 70%
[INFO] [1748013927.383697465] [battery_charger]: Batería: 65%
[INFO] [1748013928.386645257] [battery_charger]: Batería: 60%
[INFO] [1748013929.389706925] [battery_charger]: Batería: 55%
[INFO] [1748013930.392689715] [battery_charger]: Batería: 50%
[INFO] [1748013931.395521581] [battery_charger]: Batería: 45%
[INFO] [1748013932.398511636] [battery_charger]: Batería: 40%
[INFO] [1748013933.401396463] [battery_charger]: Batería: 35%
[INFO] [1748013934.404720098] [battery_charger]: Batería: 30%
[INFO] [1748013935.407807267] [battery_charger]: Batería: 25%
[INFO] [1748013936.410844012] [battery_charger]: Batería: 20%
```

Figure 8: Ejemplo de ejecución de la acción. Servidor.

```
mers58-docker@Mers58:/workspace/ros2_ws/src/battery_act$ python3 battery_client.py 20
[INFO] [1748013921.386873393] [battery_client]: Goal aceptado.
[INFO] [1748013921.392345259] [battery_client]: Feedback recibido: Batería = 95%
[INFO] [1748013922.369964028] [battery_client]: Feedback recibido: Batería = 90%
[INFO] [1748013923.373149505] [battery_client]: Feedback recibido: Batería = 85%
[INFO] [1748013924.375221233] [battery_client]: Feedback recibido: Batería = 80%
[INFO] [1748013925.378334716] [battery_client]: Feedback recibido: Batería = 75%
[INFO] [1748013926.381492894] [battery_client]: Feedback recibido: Batería = 70%
[INFO] [1748013927.384277809] [battery_client]: Feedback recibido: Batería = 65%
[INFO] [1748013928.387260207] [battery_client]: Feedback recibido: Batería = 60%
[INFO] [1748013929.390322515] [battery_client]: Feedback recibido: Batería = 55%
[INFO] [1748013930.393358455] [battery_client]: Feedback recibido: Batería = 50%
[INFO] [1748013931.396339353] [battery_client]: Feedback recibido: Batería = 45%
[INFO] [1748013932.399075409] [battery_client]: Feedback recibido: Batería = 40%
[INFO] [1748013933.402492579] [battery_client]: Feedback recibido: Batería = 35%
[INFO] [1748013934.405127215] [battery_client]: Feedback recibido: Batería = 30%
[INFO] [1748013935.408363837] [battery_client]: Feedback recibido: Batería = 25%
[INFO] [1748013936.411361197] [battery_client]: Feedback recibido: Batería = 20%
[INFO] [1748013937.416340606] [battery_client]: Resultado: Batería Baja, por favor cargue el robot!
```

Figure 9: Ejemplo de ejecución de la acción. Cliente.

Pregunta 8: Describe la arquitectura de una acción en ROS 2. ¿En qué se diferencia del sistema basado en actionlib de ROS 1?

La arquitectura de una acción en ROS 2 está basada, principalmente, en 3 partes: goal, feedback y result. El goal es el objetivo de la acción. En el caso del ejercicio, la goal representa el porcentaje de batería límite. El feedback es información sobre la acción que el servidor puede ir emitiendo durante la realización de la acción para que el cliente pueda ir sabiendo cómo está progresando. Por último, el result es el mensaje final, que se emite una vez la acción ha finalizado. En el caso del ejercicio, el result es el mensaje de aviso de que la batería ha llegado al límite objetivo y debe ser cargada.

En cuanto a las diferencias entre ROS 2 y ROS 1 respecto al actionlib, en ROS 1, la librería actionlib gestiona la creación tanto del cliente como del servidor. Por otro lado, en ROS 2, se emplea rclpy.action. Para la creación del cliente, se utiliza la clase ActionClient, y para la creación del servidor, se utiliza la clase ActionServer.

Pregunta 9: ¿Qué ocurre internamente cuando se cancela la acción en ROS 2? ¿en qué se diferencia con ROS Noetic?

Cuando un cliente cancela una acción en ROS 2, el servidor recibe esa petición de cancelación y la goal entra en un estado de cancelación. No obstante, esto no significa que la acción se haya cancelado ya que, dependiendo de la acción, es posible que el servidor tarde algo de tiempo en cancelar la acción completamente [8].

La principal diferencia entre ROS 2 y ROS Noetic en el contexto de la cancelación de una acción es que, en ROS 2, la cancelación se realiza llamando a un servicio dedicado a la cancelación de acciones para saber si el servidor acepta o no la cancelación [9]. Por otro lado, en ROS Noetic, la librería actionlib emplea un topic al cual está suscrito para emitir y recibir peticiones de cancelación [10].

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, o Data Distribution Service, es un framework empleado a modo de middleware que sirve para facilitar la fiabilidad y la escalabilidad de los sistemas distribuidos [11].

Uno de los principales problemas con lo que tenía que trabajar ROS 1 era que necesitaba un ROS Master, lanzado con roscore. Esto implicaba que, si este ROS Master fallaba, todo el sistema se rompía. Por otro lado, con DDS, ROS 2 no requiere de ningún ROS Master que pueda fallar y romper todo. Esto también afecta a la escalabilidad. Como en ROS 2 toda la carga no reside en un punto (ROS Master), se pueden enlazar muchos más nodos sin temor a que un punto central como lo era el ROS Master para ROS 1, se sobrecargue [12].

3 Repositorio Github

Para poder acceder al repositorio de Github con el código generado a lo largo de la realización de esta práctica, acceder al enlace adjuntado como comentario en la entrega, o mediante el enlace adjuntado en la bibliografía [13].

Referencias

[1] Open Source Robotics Foundation. map_server - ros wiki. `http://wiki.ros.org/map_server`. Sitio web oficial del paquete map_server en ROS.

[2] Open Source Robotics Foundation. costmap_2d - ros wiki. `http://wiki.ros.org/costmap_2d`. Sitio web oficial del paquete costmap_2d en ROS.

[3] Open Source Robotics Foundation. navfn - ros wiki. `http://wiki.ros.org/navfn`. Planificador global por defecto en ROS basado en el algoritmo de Dijkstra.

[4] Open Source Robotics Foundation. turtlebot3_navigation - ros wiki. http://wiki.ros.org/turtlebot3_navigation. Paquete que proporciona scripts de lanzamiento para iniciar la navegación con TurtleBot3.

[5] Open Source Robotics Foundation. Bags - ros wiki. http://wiki.ros.org/Bags. Documentación oficial sobre el uso de archivos .bag en ROS.

[6] Open Robotics. Creating custom ros 2 interfaces. https://docs.ros.org/en/crystal/Tutorials/Custom-ROS2-Interfaces.html. Tutorial oficial sobre cómo crear interfaces personalizadas en ROS 2.

[7] Open Robotics. Writing a simple python service and client — ros 2 documentation: Humble. https://docs.ros.org/en/humble/Tutorials/Beginner-Client-Libraries/Writing-A-Simple-Py-Service-And-Client.html#write-the-service-node. Tutorial oficial sobre cómo escribir nodos de servicio y cliente en Python usando rclpy.

[8] user: BTables. What do "accepted" and "canceling" states do in the ros2 action state machine? https://robotics.stackexchange.com/questions/98449/what-does-accepted-and-canceling-states-do-in-the-ros2-action-state-machine, 2022. Consulta en Robotics Stack Exchange sobre la máquina de estados de acciones en ROS 2.

[9] Open Source Robotics Foundation. Understanding ros2 actions — ros 2 documentation: Foxy. https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html, 2020. Tutorial oficial sobre acciones en ROS 2 (Foxy).

[10] Open Source Robotics Foundation. actionlib/detaileddescription - ros wiki. http://wiki.ros.org/actionlib/DetailedDescription. Documentación detallada del sistema de acciones en ROS 1.

[11] Real-Time Innovations (RTI). Ros 2 dds – what is dds? https://community.rti.com/page/ros-2-what-dds, 2021. Explicación del papel de DDS como middleware en ROS 2 por RTI.

[12] Open Source Robotics Foundation. Ros on dds: Design article. https://design.ros2.org/articles/ros_on_dds.html, 2019. Artículo de diseño oficial que describe cómo ROS 2 se implementa sobre DDS.

[13] Alejandro Pérez Blasco. Repositorio del proyecto tar p3. https://github.com/apbp4-ua/TAR_P3.git, 2025. Repositorio en GitHub con el código del proyecto.