Universidad Zaragoza
1542

Trabajo Fin de Grado

Simulación en Gazebo de robots móviles para tareas de transporte y manipulación.

Simulation of mobile robots in Gazebo for transport and manipulation tasks.

Autor/es

Gloria del Olmo Hernández

Director/es

Rosario Aragüés Muñoz
Gonzalo López Nicolás

Escuela de Ingeniería y Arquitectura

2019

Simulación con Gazebo de robots móviles para tareas de transporte y manipulación.

Gloria del Olmo Hernández.

Agradecimientos

“Quiero aprovechar la ocasión para agradecer el apoyo que he recibido de mi familia y amigos, que han estado presentes en cada momento que se presentaba un obstáculo para ayudarme a superarlo.

A Javi, que no ha cesado de darme ánimos repitiendo que podía con todo.

A mis tutores, con su gran positividad han conseguido que sobrepase cualquier problema o error con optimismo y han conseguido que esto se vea como algo fácil.

A todos, gracias.”

Resumen

Hoy en día desechamos toneladas de basura reciclable que acaba sin ser separada. Es una tarea tediosa, sobre todo cuando se trata de acumulaciones de basura en espacios públicos como los pequeños plásticos de las playas o los macrobotellones. Si una tarea es costosa y aburrida, es tarea para un robot. En este proyecto, se va a presentar la base de una implementación motivada por la recogida de basura y el reciclaje de la misma.

Se ha usado el entorno de trabajo ROS. A pesar de la importancia que tiene ROS, una de las limitaciones para nuevos usuarios que quieran usar este entorno de trabajo es que los tutoriales existentes están dirigidos a la puesta en marcha de cada funcionalidad. Saber utilizar fácilmente las funcionalidades ya implementadas a alto nivel no es lo más útil para aprender a desarrollar una nueva aplicación, por lo que en este proyecto se ha recopilado la información necesaria para explicar cada funcionalidad de forma que se sepa aprovechar la información para desarrollar otras funcionalidades.

Además, este trabajo se ha centrado en la base móvil de TurtleBot 3, el brazo robótico de OpenManipulator y la unión de ambos. Se ha usado el conjunto de software, librerías y paquetes de ROS asociados a estos robots para llevar a cabo la tarea propuesta.

En primer lugar, se ha estudiado el entorno de trabajo ROS y algunas de las posibilidades que ofrece. Posteriormente, se han visto, explicado y extendido las funcionalidades que ROS ofrece para las plataformas de TurtleBot 3 y el brazo robótico de OpenManipulator, además de analizarlas para trabajar con el conjunto de forma coordinada. Para ello, se han seguido varios tutoriales y se ha recopilado información para aprender la forma en la que trabaja ROS para llevar a cabo cada funcionalidad.

Finalmente, recopilando los conocimientos adquiridos y estudiando el código desarrollado para implementar futuras aplicaciones, se llega a la conclusión de que los objetivos de este trabajo se han cumplido satisfactoriamente.

Índice

Índice de figuras:

1. Introducción

Hoy en día, se está dando una gran importancia al medio ambiente y al problema existente de la gran cantidad de basura que se desperdicia en el mar, playas e incluso en espacios públicos en momentos puntuales, como plazas, calles o descampados. Hay toneladas de pequeños plásticos o miles de colillas en la arena de las playas que repercuten en la salud animal, por lo que se reúnen voluntarios para retirarlos de playas contaminadas. Además, el reciclaje y la reutilización de materiales está tomando gran importancia, por lo que en este proyecto se va a presentar la base de una futura idea de pequeños robots manipuladores, autónomos y capaces de clasificar basura o hacer una recogida de pequeños plásticos y colillas. Interesa un robot pequeño, manejable, que sea capaz de posicionarse en un lugar determinado del espacio, que sea capaz de navegar por el entorno sin chocarse y pueda manejar pequeños objetos de poco peso. Un trabajo que a las personas les cuesta tiempo y esfuerzo y que a los pequeños robots les cuesta poco y podrían estar haciéndolo constantemente, consiguiendo una limpieza más efectiva y eficiente. Esto se podría aplicar desde un entorno amplio: como playas o plazas en las que se hacen macrobotellones y se retira la basura sin clasificar o reciclar, cuando la mayoría son plásticos y vidrios fácilmente seleccionables, hasta un pequeño entorno como la habitación de casa o un local en el que se hace un cumpleaños infantil.

Se va a usar el entorno de trabajo ROS y se va a utilizar un pequeño robot con brazo móvil con algunas funcionalidades ya desarrolladas. Este conjunto está formado por la base móvil de TurtleBot 3 (Fig. 1) y el brazo robótico de Open Manipulator (Fig. 1). Se analizan y se aprende el funcionamiento de las funcionalidades que ofrecen ambas plataformas para poder utilizarlas cuando se use el conjunto de ambas.

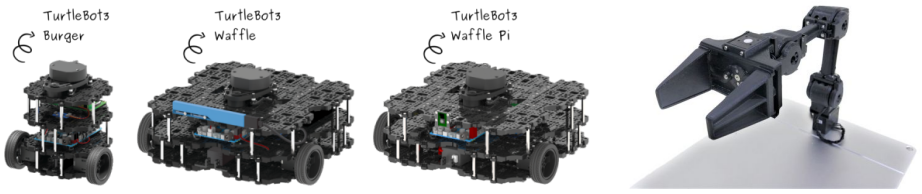

Fig. 1: Los tres modelos de TurtleBot 3 (izq.) [3] y el brazo de OpenManipulator (dcha) [4]

OpenManipulator con TurtleBot 3 (Fig. 2) es un conjunto manejable y se va a utilizar para dirigirlo a un punto determinado del entorno para coger un objeto y llevarlo a otro punto determinado. Esta funcionalidad, posteriormente, se podrá ampliar con reconocimiento de objetos y comunicación entre varios robots.

Fig. 2: OpenManipulator con la base móvil de TurtleBot 3 [30]

Se va a introducir el presente proyecto hablando brevemente sobre los antecedentes de los brazos robóticos. Se va a explica el contexto del que se parte, así como de las herramientas que se van a utilizar y los objetivos y las tareas a llevar a cabo en el mismo. Para finalizar esta sección, se encuentra una breve descripción de los contenidos de esta memoria.

1.1. Antecedentes

Uno de los primeros robots inteligentes surgió en 1966. Shakey (Fig. 3) era capaz de moverse lentamente por una sala planeando su propia ruta, navegando autónomamente y evitando los obstáculos que se encontraba por el camino [1]. Shakey fue un adelantado a su tiempo.

Fig. 3: Shakey con sus creadores. [31]

Al comienzo del auge de la robótica industrial (finales 1970), los robots manipuladores eran poco inteligentes, telemanipulados y realizaban tareas o funciones repetitivas. Además, eran de grandes tamaños y poco manejables, ya que uso estaba destinado principalmente a la industria.

La robótica industrial comenzó a evolucionar a la robótica de servicio (1990), tal y como la conocemos actualmente. Esta evolución fue surgiendo a medida que cambiaban las necesidades del ser humano. Por ejemplo, los primeros brazos robóticos surgieron en los años 60 después de la guerra, cuando las personas necesitaban suplir el miembro que habían perdido. En los años 80, con el auge de la robótica y la inteligencia

artificial, se comenzaron a diseñar estos brazos y demás robots de servicio en Estados Unidos [2].

Los robots de servicio (Fig. 4) disponen de inteligencia artificial, capacidad de adaptación y capacidad de aprendizaje, además de ser muy manejables y realizar tareas versátiles y variadas. Todo esto es posible gracias a los diferentes sensores que llevan integrados y la capacidad que se tiene actualmente de construirlos con varios pesos y tamaños. Hoy en día, estos robots, desempeñan tanto tareas personales como pueden ser tareas domésticas, de entrenamiento personal, de apoyo a discapacitados, etc, como tareas profesionales como tareas agrícolas, de construcción, de vigilancia y rescate, en medicina, etc.

Fig. 4: Robots de servicio. Medicina (izq.), rehabilitación(centro) y deporte (dcha.). [29]

1.2. Entorno de trabajo y herramientas

En esta sección se va a introducir el entorno de trabajo en el que se va a desarrollar el presente proyecto. Se van a dar a conocer las herramientas y algunos conceptos básicos que se usarán más adelante. Todo ello se irá viendo con más detalle durante el desarrollo de esta memoria.

Se parte del sistema operativo robótico ROS (Robot Operating System) y de las plataformas estándar de este sistema: TurtleBot 3 (TB3) y OpenManipulator (OM). Este

sistema incluye y ofrece varias funcionalidades. Algunas de estas, se van a estudiar para el desarrollo de este proyecto.

El entorno de trabajo ROS funciona como un sistema operativo y se basa en la comunicación entre nodos a través de topics y/o servicios dentro de un máster. Este framework, contiene un conjunto de paquetes aportados por los usuarios con diferentes funcionalidades como localización, mapeo, simulación, navegación, etc.

La plataforma móvil TurtleBot es un robot de plataforma estándar ROS. Deriva del robot Turtle, el cual, fue impulsado por el lenguaje educativo de programación de computadoras Logo en 1967. Desde entonces, Turtlebot se ha convertido en la plataforma más popular entre desarrolladores y estudiantes. Hay tres versiones de la serie TurtleBot (Fig. 5): TurtleBot1 (2010) que fue desarrollada al mismo tiempo que el robot iRobot Roomba, TurtleBot2 (2012) y TurtleBot3 (2017) que se desarrolló con características que completan las funciones de las versiones anteriores y cumplen las demandas de los usuarios [3].

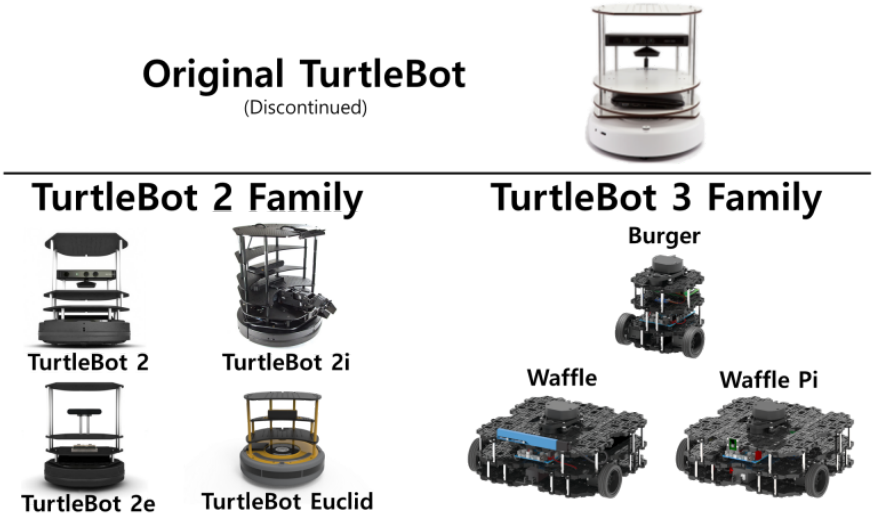

Fig. 5: Los modelos de TurtleBot, TurtleBot 2 y TurtleBot 3. [32]

TurtleBot3 es un robot móvil pequeño, asequible, programable y basado en ROS. Sus funcionalidades principales son SLAM, Navegación y Manipulación y se puede utilizar como un manipulador móvil capaz de manipular un objeto al conectar un manipulador como OpenManipulator [3].

El brazo robótico OpenManipulator está habilitado para ROS y es una plataforma de robot completamente abierta. Además, permite a los usuarios controlarlo más fácilmente mediante la vinculación con el paquete de MoveIt! [4].

El proyecto se va a desarrollar gracias a varias GUI (Graphical User Interface) y demás herramientas que ofrece ROS, alguna de estas son:

- Gazebo: es un simulador para cuerpos rígidos 3D y se integra con ROS utilizando mensajes ROS, servicios y reconfiguración dinámica [5]. Gracias a esta herramienta se van a poder observar las diferentes funcionalidades de la plataforma móvil (TB3) y el brazo robótico (OM) en un espacio simulado.

- RViz: es una herramienta GUI o un visualizador de ROS.

- MoveIt!: es un software de código abierto para ROS basado en la técnica de software para la manipulación móvil [6]. Esta herramienta se va a utilizar a través de su complemento Motion Planning Plugin (MPP) en RViz.

- RQT: es un sistema software que implementa las diversas herramientas GUI en forma de complementos, se pueden ejecutar todas las herramientas como ventanas acoplables dentro de rqt [7]. Dicho de otra forma, es un administrador de interfaces gráficas de usuario de ROS. Gracias a esta herramienta va a ser posible extraer los diagramas en los que quedan reflejados los nodos activos y los topics utilizados en cada *launch* o ejecutable. Además, se podrá comprobar el funcionamiento de los topics y servicios mandando mensajes a través de rqt.

Estas herramientas van a ser ejecutadas por archivos de extensión *launch*, archivos que ejecutan de forma ordenada los nodos implicados en la funcionalidad a realizar y donde se hace la parametrización de esos nodos. Se clasifican como archivos de configuración [9]. En este proyecto, este tipo de archivos se van a utilizar para ejecutar las funcionalidades que se van a describir en la presente memoria.

A continuación, se van a explicar brevemente algunos términos que se van a usar durante el proyecto:

- Archivo con extensión .yaml: es un archivo de desarrollo. YAML fue creado bajo la creencia de que todos los datos pueden ser representados adecuadamente como combinaciones de listas, mapeos y datos escalares (valores simples). La sintaxis es relativamente sencilla y fue diseñada teniendo en cuenta que fuera muy legible pero que a la vez fuese fácilmente mapeable a los tipos de datos más comunes en la mayoría de los lenguajes de alto nivel [10].

- Archivo con extensión .world: archivos en lenguaje XML. En este proyecto se van a definir los diferentes entornos de simulación en Gazebo con esta extensión.

- URDF: Unified Robot Description Format o Formato de descripción de robot unificado, es un formato XML para representar el modelo de un robot.[12]

- Archivo con extensión .xacro: xacro es un lenguaje macro XML, se pueden construir archivos XML más cortos y legibles utilizando macros que se expanden a otras expresiones XML [11]. En este tipo de archivos se encontrarán los modelos URDF de los modelos robot.

- Archivo con extensión .rviz: archivos de configuración de interfaz de la herramienta RViz.

- Terminal: también llamado terminal de texto o consola de texto. Los ordenadores actualmente tienen incorporado la consola del sistema. La terminal es una interfaz para comunicarse con un computador a través de un teclado para entrada de datos y una pantalla para salida de datos. Estos datos son únicamente caracteres alfanuméricos (sin gráficos). Las consolas de texto son apropiadas para la interfaz de línea de comandos y las interfaces de texto de usuario. [13] Para abrir una terminal de texto en Ubuntu se presiona: ctrl + t.

Tras esta pequeña introducción al entorno de trabajo, herramientas y algunos términos que se van a utilizar, se van a ver los diferentes objetivos y tareas del presente proyecto.

1.3. Objetivos y tareas

El objetivo principal de este proyecto consiste en proponer una implementación de una aplicación que sirva de base para una futura aplicación completa de recogida de basura y reciclaje. Esta funcionalidad básica va a consistir en la búsqueda de basura por parte de una base móvil y la recogida de la misma por parte de un brazo robot. Esta aplicación se basa en la programación en ROS de uno o varios archivos *launch* y en la modificación y/o creación de nodos que sean pertinentes.

Además, como objetivos secundarios y unidos al objetivo principal, se van a estudiar las posibilidades que ofrece el entorno de trabajo ROS para la base móvil de TurtleBot 3 y el brazo robótico de Open Manipulator. ROS es uno de los entornos de trabajo más importantes en robótica.

Los tutoriales existentes de ROS suelen estar dirigidos a la explicación esquemática de la puesta en marcha de una funcionalidad y suelen describir de forma secuencial las acciones a seguir. Indican paso a paso los comandos a escribir en la terminal para la instalación del software, la ejecución de un *launch* concreto y, sin

explicar demasiado cómo se usan los programas que se ejecutan para llevar a cabo la aplicación, indican lo que debería de hacer la funcionalidad ejecutada. Debido a esto, se va a recopilar información de varias fuentes y se va a exponer ordenada y detallada para la ejecución de cada funcionalidad, aprendiendo y entendiendo el funcionamiento de ROS a más bajo nivel. Se van a explicar detalladamente los pasos a seguir para completar con éxito diferentes tareas, aprovechando cada aplicación que nos ofrece ROS para las plataformas descritas en capítulo 1. Además, se van a ampliar las funcionalidades asociadas a TB3 y a OM, sobre todo, las aplicaciones para el conjunto de la base y el brazo, indicando modificaciones o aclarando cómo parametrizar cada aplicación para conseguir los comportamientos deseados de cada una.

A lo largo del proyecto se van a llevar a cabo diferentes tareas para llegar a la resolución del objetivo final. Esto lleva una planificación temporal que se muestra en la siguiente tabla (Tabla 1):

	Julio	Agosto	Septiembre	Octubre	Noviembre
Instalación del Software necesario.					
Introducción a ROS.					
Realización de tutoriales de las herramientas.					
Realización de tutoriales de Turtlebot3.					

Realización de tutoriales de OpenManipulator.					
Realización de tutoriales de OM con TB3					
Estudio del objetivo.					
Explicación del programa objetivo.					
Realización de la memoria.					

Tabla 1: Distribución de tareas en el tiempo

Esta planificación (Tabla 1) incluye tareas de preparación del desarrollo del programa, así como tareas de inicialización y finalización del proyecto.

La presente memoria se ha ido desarrollando a medida que se han ido realizando los tutoriales y a medida que se ha ido reuniendo la información necesaria sobre las herramientas a utilizar, así como, a la vez que se ha ido avanzando en el desarrollo del proyecto y la implementación de la aplicación básica para la recogida de basura.

Las tareas de inicio del proyecto han consistido en la instalación del software necesario y la realización de los tutoriales de ambas plataformas de ROS, TurtleBot 3 y OpenManipulator, totalmente necesarios para la comprensión del sistema, de las diferentes herramientas y del robot a usar, además de posibilitar la reutilización de

código y de diferentes funcionalidades. Una vez se han adquirido los conocimientos necesarios para saber utilizar las herramientas comentadas en la sección 1.2., se fija un objetivo más definido y detallado partiendo de la idea impulsora a realizar en este proyecto. Tras tener la base fijada y clara, se da comienzo al estudio de los diferentes caminos para llegar a cumplir el objetivo estimado, así como, las diferentes posibilidades y desarrollos a hacer, tras esto, se procede a la creación de un programa que cumpla los objetivos de la mejor forma prevista.

1.4. Contenido de la memoria

La memoria consta de varios apartados que explican el desarrollo y las conclusiones del presente proyecto.

Dentro del desarrollo se van a encontrar diferentes capítulos explicando y analizando el trabajo llevado a cabo. Así como los experimentos realizados, el diseño y la propuesta de implementación del programa base.

En el capítulo 2 se va a realizar una introducción al entorno de trabajo ROS. En los capítulos 3, 4 y 5, se van a explicar las diferentes funcionalidades que ofrece ROS para el brazo móvil de TurtleBot 3, el brazo robótico de OpenManipulator y las funcionalidades de la integración de ambas plataformas (brazo montado sobre la plataforma móvil). En el capítulo 6 se van a describir las diferentes funcionalidades extendidas para las plataformas descritas.

Como conclusión, en el capítulo 7, se van a hacer varias conclusiones con diferentes puntos de vista, por un lado, se va a hablar de lo llevado a cabo en este proyecto. Por otro lado, se va a hablar de las posibles mejoras y avances en un futuro del presente proyecto y, finalmente, se va a hacer una breve valoración personal del mismo.

2. ROS

El sistema ROS, como ya se ha introducido en la sección 1.2. Entorno de trabajo y Herramientas, se basa en la comunicación entre nodos a través de la publicación de mensajes en topics y, esta comunicación, es posible gracias al nodo master. Para comenzar, se van a explicar estos cinco términos que se van a nombrar repetidamente en esta memoria:

- Master: es un nodo principal que funciona como núcleo del sistema proporcionando una plataforma que hace posible la comunicación entre los diferentes nodos del sistema, en nuestro caso es roscore [8].

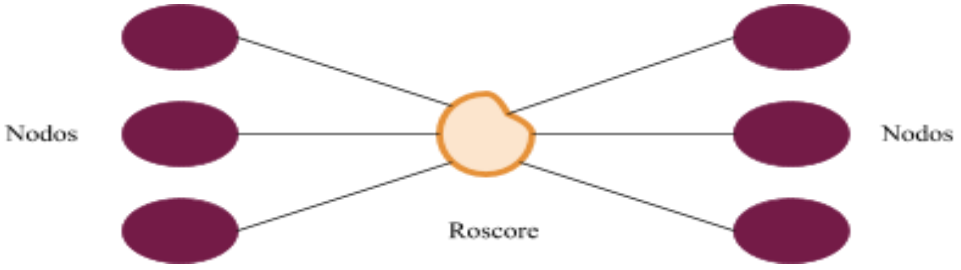

Fig. 6: Master o Roscore

La imagen (Fig. 6) es una forma visual de explicar el *master*. Roscore es el capitán del equipo y se encarga de tener el registro de nombres, servicios y parámetros, además del registro de salida al que todos los nodos envían información (/rosout) [8].

- Nodo: es un programa o aplicación. Puede enviar y/o recibir información de otros nodos a través de un topic y puede usar y/u ofrecer servicios a otros nodos [8]. En la siguiente imagen (Fig. 7) se ve una forma esquematizada de la comunicación entre dos nodos a través de un topic y de la oferta y uso de un servicio de un nodo a otro.

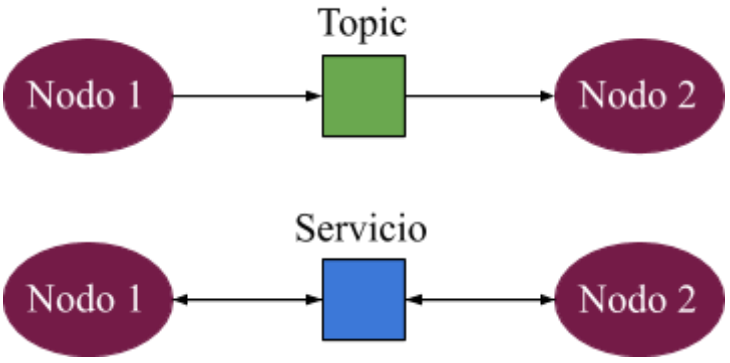

Fig. 7: Comunicación entre nodos.

- Topic: es un bus o canal de comunicación por el cual los nodos se envían o reciben mensajes. Estos, son unidireccionales, lo que significa que el nodo no va a recibir respuesta del *topic* al que ha publicado ni va a saber si ha llegado la información [8]. Un nodo publica datos continuamente a través de los *topics* y cualquier otro nodo puede usar esa información.

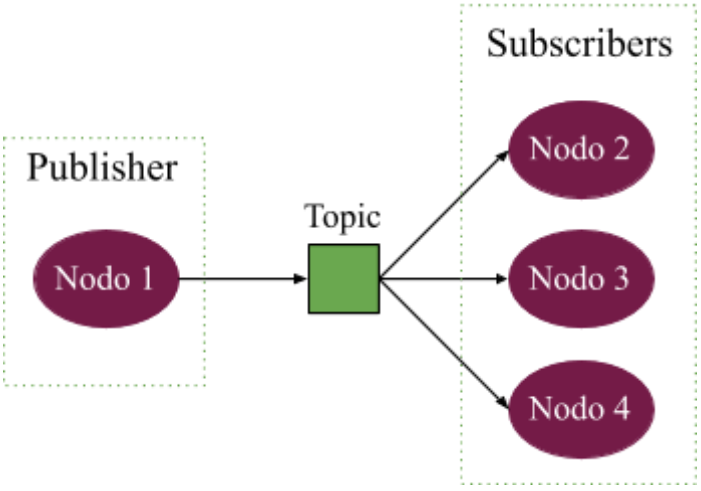

Fig. 8: Envío y recibimiento de información a través de un topic.

La imagen (Fig. 8) representa la comunicación unidireccional entre nodos a través de un topic. El nodo que envía o publica la información recibe el

nombre de *publisher* (o editor) y los nodos que recogen o reciben la información reciben el nombre de *subscribers* (o suscriptores).

- Servicio: un nodo puede ofrecer un servicio, este nodo se denomina server (o servidor) y es el que se encarga de ejecutar la tarea deseada. Cualquier otro nodo puede usar este servicio y se denomina client (o cliente), es el que demanda la tarea [14].

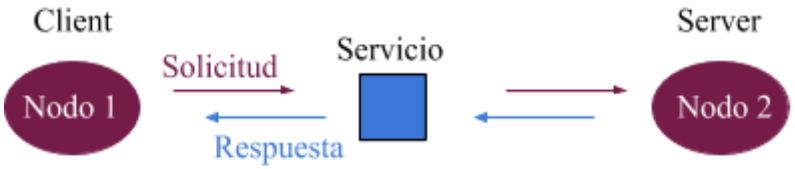

Fig. 9: Petición de un servicio por un nodo y respuesta del nodo servidor.

En la imagen (Fig. 9) se ve el esquemático de la comunicación entre dos nodos a través de un servicio: el cliente, en este caso el nodo 1, solicita el servicio del servidor, en este caso el nodo 2. El servidor recibe la petición, realiza el servicio y devuelve una respuesta al cliente. Por ejemplo, el cliente solicita el servicio de una suma con los datos 3 y 5 y el servidor le devuelve la solución de la operación; 8.

- Mensajes: son estructuras de datos simples y pueden incluir estructuras y matrices anidadas, se definen en archivos msg. Los tipos de mensaje primitivos estándar son entero, coma flotante, booleano, etc. Los nodos también pueden intercambiar un mensaje de solicitud y respuesta como parte de una llamada de servicio ROS. Estos mensajes de solicitud y respuesta se definen en archivos srv [15].

Para ejecutar las diferentes funcionalidades proporcionadas por ROS se necesitará tener en proceso el master o *roscore*, ya que, como se ha visto en este apartado, el *master* es el que hace posible la comunicación entre los diferentes nodos. También es necesario que cada proceso se ejecute en una terminal diferente.

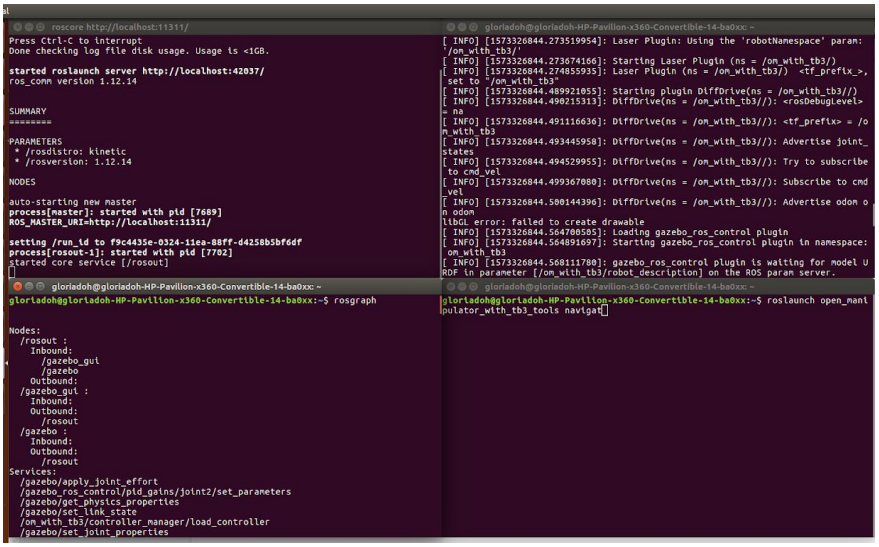

Fig. 10: Diferentes terminales con diferentes procesos en ejecución.

La imagen (Fig. 10) es un ejemplo de diferentes procesos abiertos en diferentes terminales, en este caso, hay cuatro terminales abiertas. La primera, arriba a la izquierda, se le ha pasado el comando *roscore*, el cual, ejecuta el *master*. Las otras tres, son diferentes procesos como la ejecución de un *launch,* la visualización en formato texto del gráfico de comunicación entre nodos con el comando *rosgraph* y el comienzo de un comando elegido aleatoriamente para el ejemplo.

Se van a ver algunos de los comandos de terminal que se van a utilizar en el proyecto para el análisis de las diferentes funciones y el desarrollo del mismo:

- Roscore: ejecuta el proceso del master o servicio principal, lo que incluye: master (Servicio de nombre ROS) + rosout (registro de registros) + parameter server (gestión de parámetros) [16].

- Rosrun: ejecuta un nodo [16].

- Roslaunch: ejecuta los archivos con extensión .launch y permite añadir parámetros del *launch* en la misma línea de comando [16].

- Rosnode: da información de los nodos de ROS [16].

- Rostopic: da información de los topics ROS [16].

- Rosservice: da información de los servicios ROS [16].

- Rosgraph: da la información en formato de texto del gráfico ROS de nodos, servicios y *topics* activos, así como de la comunicación entre ellos [17]. También se le llama así a la representación gráfica de la comunicación entre nodos (representados con círculos) a través de los diferentes *topics* (representados con flechas) implicados en esa comunicación.

- ROS_NAMESPACE: variable de ROS en el que guarda su espacio de nombres. Todos los nombres en el nodo se resolverán en función de este valor de nombre. En este proyecto se va a usar para elegir el robot al que aplicarle la funcionalidad a ejecutar [18].

- Export: se le da valor a una variable a utilizar o necesaria en el ejecutable. En este caso, se va a utilizar para darle el modelo de robot deseado a cargar en cada ejecutable. Se usa cuando se necesitan nombres de etiqueta que no se especifican en ROS.

- Ctrl+c: exist, cierra o sale del proceso.

3. Turtlebot 3

En este capítulo se van a analizar las principales funciones que nos ofrece ROS con TurtleBot 3, así como los nodos y topics que juegan un papel importante en cada uno de ellos.

Como se ha hablado en la sección 1.2. Entorno de trabajo y herramientas, TurtleBot 3 tiene tres modelos diferentes llamados: Waffle, Waffle PI y Burger (Fig. 1 imagen de la izquierda). En este proyecto se va a trabajar con el modelo waffle continuamente.

Por otro lado, se va a dar por hecho que el master está en proceso para la ejecución de cualquier *launch*, tal y como se ha comentado en el apartado 2. ROS. El *master* se ejecuta escribiendo *roscore* en una terminal y dándole a enter, es decir, ejecutando el comando *roscore* en la terminal.

3.1. Simulación en Gazebo

Con Gazebo es posible simular el robot en un entorno determinado, ya que Gazebo recoge datos de los diferentes sensores integrados en la TurtleBot 3. Es similar a que la base móvil estuviera en una habitación real. En general, es un proceso que tiene que estar en ejecución cuando se ejecuten otros nodos para tener datos necesarios del entorno y del robot, como por ejemplo: la distancia a una pared a través del sensor láser o la posición del robot en el entorno con el sensor de odometría.

Para ejecutar la aplicación de Gazebo se van a utilizar archivos *launch*. Todos estos archivos que se van a utilizar para simular el robot en un entorno tienen tres cosas en común: abren el simulador Gazebo, cargan el modelo del robot (archivo *xacro*,

donde se describe el URDF del robot) y cargan un entorno (archivo *world*). El modelo URDF y los entornos pueden ser creados y/o editados desde la aplicación de Gazebo o desde los propio archivos escritos en XML.

Se van a utilizar en este proyecto archivos *launch* para simular la situación en la que se encuentra el robot:

- turtlebot3_empty_world.launch, carga el modelo de la TB3 en un mundo vacío (Fig. 11: imagen de la izquierda).

- turtlebot3_world.launch, carga el modelo de la TB3 en un entorno ya configurado, este entorno tiene una forma de tortuga (Fig. 11: imagen central).

- turtlebot3_house.launch, carga el modelo de la TB3 en un entorno ya configurado, este entorno contiene diferentes habitáculos amueblados, como si de una casa se tratase (Fig. 11: imagen de la derecha).

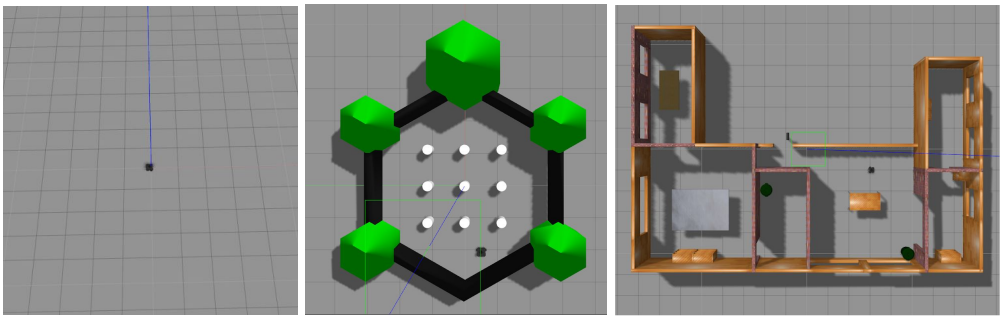

Fig. 11: Diferentes entornos de Gazebo

Existen otros entornos ya configurados, pero no se van a usar en el desarrollo de este proyecto.

Estos archivos se van a ejecutar en una nueva terminal con el comando roslaunch. En el caso de ejecutar el *launch* que posiciona el robot en un entorno vacío la línea de comando completa es:

roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch

Siendo turtlebot3_gazebo el directorio principal donde se encuentra el *launch* y turtlebot3_empty_world.launch el nombre del ejecutable.

Los otros dos entornos se sitúan en el mismo directorio que este último, por lo que el comando para cargar el entorno con forma de tortuga sería:

roslaunch turtlebot3_gazebo turtlebot3_world.launch

Y para el de las habitaciones o la casa:

roslaunch turtlebot3_gazebo turtlebot3_house.launch

En el caso de los *launch* comentados en este apartado, se da la opción de seleccionar el modelo URDF deseado entre Waffle, Waffle Pi y Burguer, los tres modelos predeterminados que ya se han visto en la introducción de esta sección, capítulo 3.. Esta selección se hace a través del comando *export* de ROS en la misma terminal en la que se va a ejecutar el archivo *launch*. La línea de comando completa a escribir en la terminal es:

export TURTLEBOT3_MODEL=waffle

Siendo TURTLEBOT3_MODEL la variable de entorno pedida por el *launch*, es decir, la cajita donde guarda el nombre del modelo del robot, y Waffle el nombre del modelo robot elegido.

Estos *launch* se podrían modificar fácil y rápidamente para dejar un modelo predefinido y no tener que indicarle al *launch*, cada vez que se ejecuta, el modelo de TB3 deseado.

Una vez se tiene simulado el robot en un entorno se va a explicar cómo se va a mover el robot por dicho entorno en el siguiente apartado.

3.2. Movimiento

Una vez abierto Gazebo se procede a mover la base. Es posible mover la base cambiando la velocidad manualmente o hacer que la TurtleBot3 se mueva automáticamente por el entorno.

Los dos *launch* que consiguen llevar esto a cabo son:

- turtlebot3_teleop_key.launch: es una forma telemanipulada de mover la base móvil. Permite al usuario elegir a través de teclado las velocidades lineal y angular para mover la base por el entorno.

- turtlebot3_simulation.launch: es una forma de movimiento automático del robot. Permite mover la base móvil automáticamente por el entorno sin chocar con objetos ni con paredes.

Para ejecutar ambos *launch* se tiene que abrir una nueva terminal e indicar el modelo robot con el comando:

```
export TURTLEBOT3_MODEL=waffle
```

Al igual que en la sección 3.1. Simulación en Gazebo, se tiene que tener *roscore* y el robot en el entorno simulado en Gazebo en proceso.

Primero se va a explicar el movimiento telemanipulado y los pasos a seguir para llevarlo a cabo. Para ejecutar el *launch* de telemanipulación, en la misma terminal en la que se ha elegido el modelo robot, se escribe el comando:

roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch

Una vez se ha ejecutado, aparecerá en la terminal un mensaje con las teclas que manejan el robot y las velocidades máximas y mínimas de cada velocidad en cada modelo. (Fig. 12)

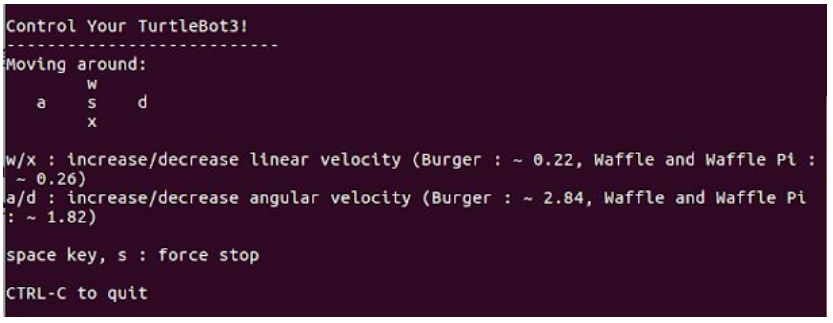

Fig. 12: Mensaje de la terminal al ejecutar la aplicación de telemanipulación.

Los valores de velocidades máximas que indica el nodo teleop en la ventana de la terminal son:

- Velocidad máxima lineal del modelo Burger: 0.22 m/s
- Velocidad máxima angular del modelo Burger: 2.84 rad/s
- Velocidad máxima lineal de los modelos Waffle: 0.26 m/s
- Velocidad máxima angular de los modelos : 1.82 rad/s

Además, el mensaje informa de qué teclas están implicadas en el movimiento del robot y de qué manera está cada una. Estas teclas son: w, a, s, d y x, cada una de estas da una orden diferente a la base para su movimiento:

- w: aumenta 0.01 m/s la velocidad lineal (se avanza)
- x: disminuye 0.01 m/s la velocidad lineal (se retrocede)
- d: aumenta 0.1 rad/s la velocidad angular (se gira a la derecha)
- a: disminuye 0.1 rad/s la velocidad angular (se gira a la izquierda)
- s: anula ambas velocidades (inicializa los valores a cero)

Una vez se tienen los datos suficientes como para saber manejar el robot por el entorno, se van a explicar los nodos y *topics* que están implicados en este movimiento. Para ver qué nodos y *topics* están implicados en esta funcionalidad, como se ha hablado en el capítulo 2. ROS, hay dos opciones:

- En una nueva terminal escribir el comando: *rosgraph*. Este comando mostrará en la misma terminal los nodos y servicios que se encuentran activos en cada instante de tiempo (Fig. 13).

- En una nueva terminal escribir el comando: *rqt_graph*. Este comando mostrará de una forma visual los nodos y *topics* implicados en la aplicación a través de la aplicación de rqt (Fig. 14).

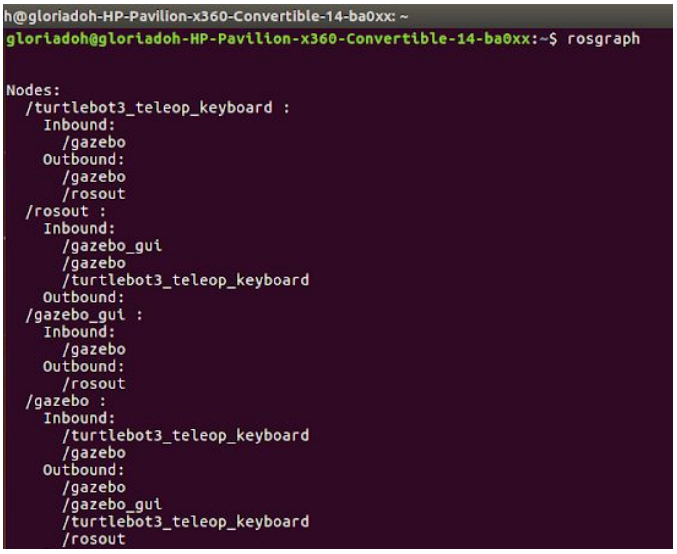

Fig. 13: rosgraph: lista de nodos activos.

En la imagen (Fig. 13) se ve la lista en la terminal de los nodos activos y la comunicación entre ellos, por ejemplo, el nodo turtlebot3_teleop_keyboard proporciona información a Gazebo (es inbound o nodo entrante) y Gazebo y rosout le proporcionan información a turtlebot3_teleop_keyboard (es outbound o nodo saliente).

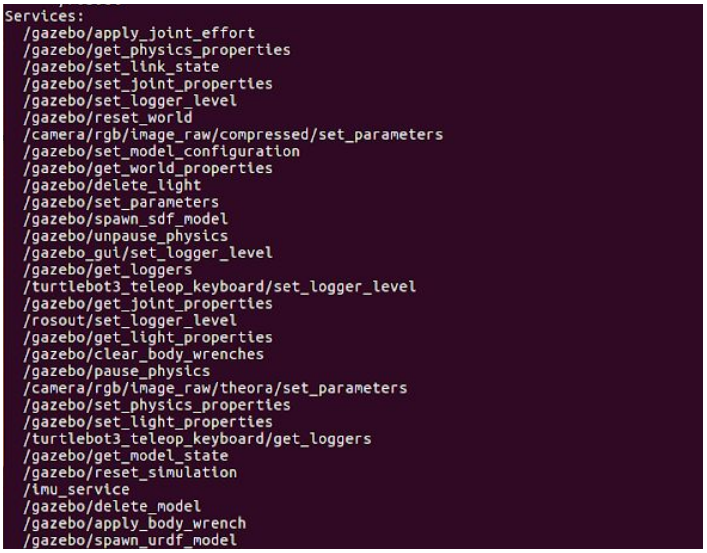

Fig. 14: rosgraph: lista de servicios activos.

En la imagen anterior (Fig. 14) se muestra la lista de todos los servicios activos.

Una forma más visual de la comunicación entre nodos se muestra en la siguiente imagen (Fig. 15):

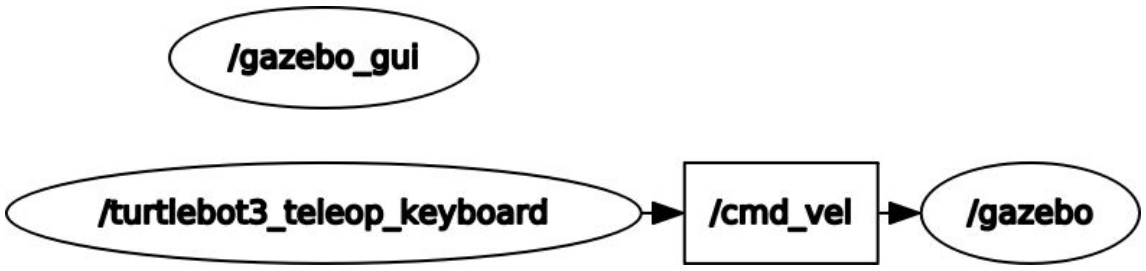

Fig. 15: Rosgraph de los nodos y topics activos en la telemanipulación del robot.

Como se puede observar en el *rosgraph* exportado de la herramienta rqt (Fig. 15), los nodos activos son gazebo y turtlebot3_teleop_keyboard y se comunican a través del *topic* cmd_vel.

El nodo Gazebo es el encargado de enviar y recoger los datos necesarios para la herramienta Gazebo.

El nodo turtlebot3_teleop_keyboard se encarga de la telemanipulación, es decir, recoge datos de teclado y los publica en cmd_vel.

El topic cmd_vel es el encargado del valor de las velocidades lineal y angular. Por lo que el nodo de teleoperación envía a través de cmd_vel las velocidades a Gazebo, quien simula el robot con esos datos (hace que se efectúa el movimiento de la base).

En segundo lugar y ya vista la forma telemanipulada de mover la TB3, se va a explicar el movimiento automático de la base móvil.

Para ejecutar el *launch* de simulación o movimiento automático, en la misma terminal en la que se ha elegido el modelo robot, escribimos el comando:

```
roslaunch turtlebot3_gazebo turtlebot3_simulation.launch
```

Una vez se ha ejecutado, la base móvil comenzará a moverse libremente por el entorno sin chocar.

Los nodos y *topics* activos en esta funcionalidad de TB3 se ven ejecutando en una nueva terminal el comando: *rqt_graph*.

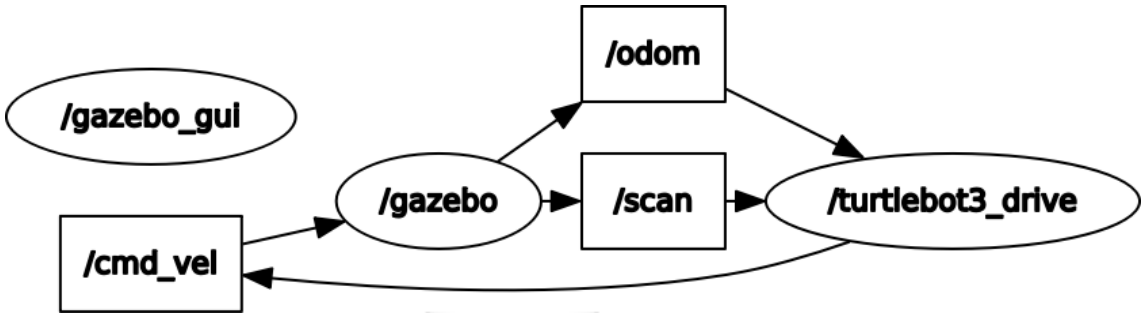

Fig. 16: Rosgraph de nodos y topics activos en la simulación del movimiento de la TB3.

Se observa en el *rosgraph* proporcionado por rqt (Fig. 16) que los nodos activos en esta funcionalidad son gazebo y turtlebot3_drive y los *topics* implicados en la comunicación entre ellos son cmd_vel, scan y odom.

Aquí también aparece el nodo gazebo, ya que el programa está en proceso. Se puede ver que gazebo es suscriptor de cmd_vel y publicador de odom y scan. Es decir, recoge datos de velocidad para simular el movimiento del robot y proporciona los datos de los sensores de odometría o posición del robot (información en el topic odom) y escáner (información en el topic scan).

Turtlebot3_drive es el nodo encargado de calcular, a partir de los datos de escáner y odometría, las velocidades que tiene que llevar el robot. Este nodo se basa en una máquina de estados que sigue el siguiente diagrama de flujo (Fig. 17):

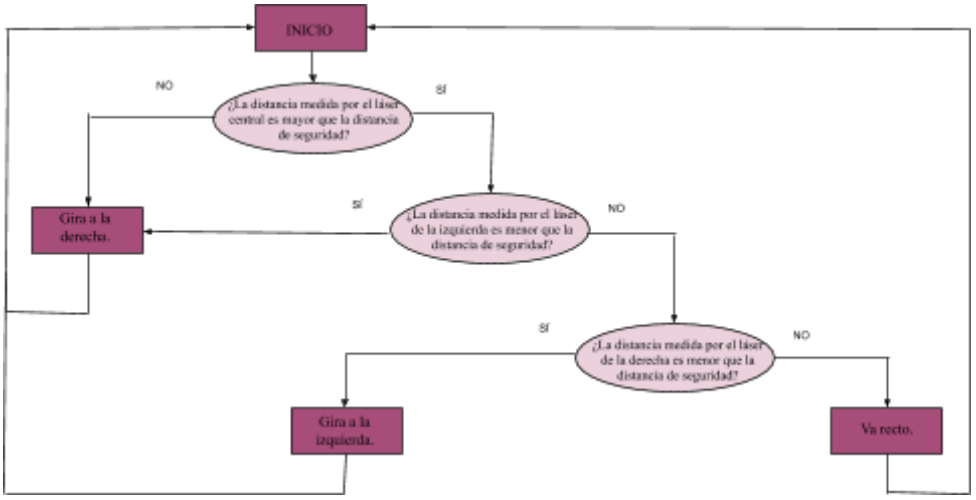

Fig. 17: Diagrama de flujo del funcionamiento del nodo turtlebot3_drive.

En el estado inicial comprueba si tiene un objeto delante, en ese caso, gira a la derecha. En caso contrario, el robot comprueba si tiene algo a su izquierda o a su derecha y, si es así, gira a la derecha o a la izquierda respectivamente. Si no tiene nada alrededor, avanza recto. Con este proceso, la base móvil, evita chocar con objetos o paredes.

Por otro lado, el nodo impone unas velocidades fijas para girar e ir recto. Las velocidades lineal y angular a las que se mueve la TB3 son en cada caso:

- Para girar a la derecha: velocidad lineal 0 m/s y velocidad angular -1.5 rad/s.
- Para girar a la izquierda: velocidad lineal 0 m/s y velocidad angular 1.5 rad/s.
- Para ir recto: velocidad lineal 0.3 m/s y velocidad angular 0 rad/s.

Y las distancias de seguridad a los objetos son:

- Para el sensor láser central: 0.7 m.
- Para el sensor láser izquierdo: 0.6 m.
- Para el sensor láser derecho: 0.6 m.

Una vez explicado las posibilidades que ofrece ROS para el movimiento del robot, se van a explicar otras funcionalidades ya implementadas en las que se va a necesitar mover el robot.

3.3. SLAM

La técnica SLAM consiste en construir un mapa de un entorno desconocido a través de lo datos recibidos de diferentes sensores. En este proyecto, Gazebo recoge los datos de los sensores y se gestionan para conseguir un mapa del entorno y, más tarde, guardarlo.

Como definición más formal, SLAM en robótica es "localización y mapeado Simultáneo" [19]. El robot a medida que se mueve por el entorno va creando el mapa calculando distancias a los objetos con el láser y se va localizando en el mismo a través del sensor de odometría. Además, va corrigiendo el error que se produce.

Como se ha dicho al comienzo del capítulo 3. el *master* tiene que estar en proceso para hacer posible la comunicación entre los demás nodos. Además, se tiene que tener simulado el robot en un entorno como se ha indicado en la sección 3.1.

El *launch* que se ocupa de ejecutar la funcionalidad de SLAM se llama turtlebot3_slam.launch, antes de ejecutarlo, en una nueva terminal se necesita indicar el modelo de la base con el comando:

```
export TURTLEBOT3_MODEL=waffle
```

El *launch* que se ocupa de la funcionalidad de SLAM tiene la posibilidad de escoger entre varias opciones de slam: *cartographer, frontier exploration, gmapping, hector* y *karto*. Se pasa por parámetro el método deseado y se escribe al final de línea de comando para ejecutarlo en la misma terminal en la que se ha elegido el modelo de la base:

roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping

Este *launch* se encarga de ejecutar otro archivo que es el encargado real de ejecutar el método seleccionado, por lo que se puede ejecutar directamente el *launch* de la técnica SLAM deseada, en nuestro caso *gmapping,* de la forma:

roslaunch turtlebot3_slam turtlebot3_gmapping.launch

En la siguiente imagen (Fig. 18) se muestra el *rosgraph* de nodos y topics activos para esta aplicación.

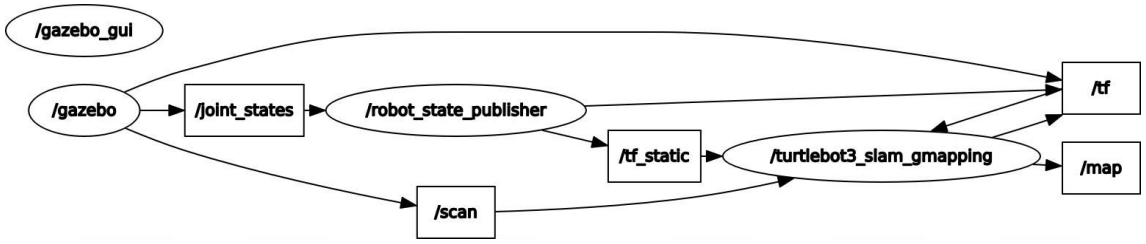

Fig. 18: Rosgraph de nodos y topics activos ejecutando SLAM.

Gazebo, robot_state_publisher y turtlebot3_slam_gmapping son los nodos activos para llevar a cabo la opción de *gmapping* de la técnica SLAM. Se tiene en cuenta que este *rosgraph* no incluye los nodos y topics activos durante el movimiento del robot, necesario para esta función.

El *rosgraph* completo de nodos y *topics* activos durante la realización de esta funcionalidad es la unión del *rosgraph* del movimiento manipulado (Fig. 15) o del movimiento automático (Fig. 16) con el de la funcionalidad SLAM (Fig. 18)

Gazebo es *publisher* o *editor* de los *topics* joint_states, tf y scan. robot_state_publisher es suscriptor (recoge información) de joint_states y editor (proporciona información) de tf y tf_static y, por último, turtlebot3_slam_gmapping recoge los datos de tf, tf_static y scan y actualiza tf y map publicando en ellos.

En la sección 3.2. ya se ha hablado del nodo Gazebo y del *topic* scan presentes y activos en el SLAM.

El nodo robot_state_publisher se ocupa de recoger los datos del *topic* joint_states [20], en el que se encuentra la información del estado de cada unión del robot recogida previamente del modelo URDF [21], y calcula la cinemática del robot para publicar el estado del robot en un árbol de coordenadas tridimensionales cinemáticas en los *topics* tf y tf_static [20]. Éstos suelen trabajar unidos [21].

Los *topics* tf y tf_static permiten tener un seguimiento de los múltiples sistemas de coordenadas 3D. Estos sistemas pueden ser: el sistema de referencia del entorno, un sistema de referencia de la base, un sistema de referencia del agarre, etc [22].

El *topic* tf recoge la información del seguimiento de todos los sistemas de coordenadas en una estructura de árbol almacenada en el tiempo. De esta forma, mantiene la relación entre dos partes del robot o de una parte del robot y una parte del entorno a lo largo del tiempo. Gracias a esto se pueden responder preguntas como: ¿dónde estaba la cabeza del robot en relación con el marco del mundo hace 5 segundos?, ¿cuál es la posición del objeto situado en la pinza en relación con la base del robot?, ¿cuál es la posición actual del marco base en el marco del mapa? [22].

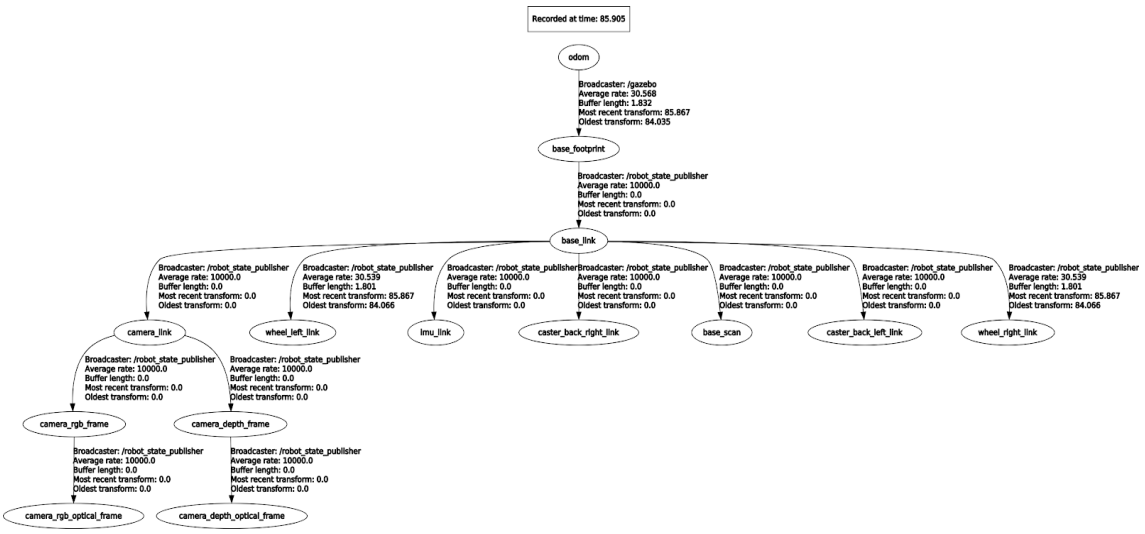

Fig. 19: Árbol de coordenadas o TF.

El árbol de coordenadas (Fig. 19) es un árbol en el que se encuentra la relación de transformación entre las diferentes uniones del robot, como pueden ser las ruedas, la base o incluso la cámara. En este caso, se puede ver que el sensor de odometría está relacionado con la base y la base con las ruedas, la cámara y otros sensores. En cada uno de estas relaciones se encuentra un valor de tiempo de la última modificación y de la antepenúltima.

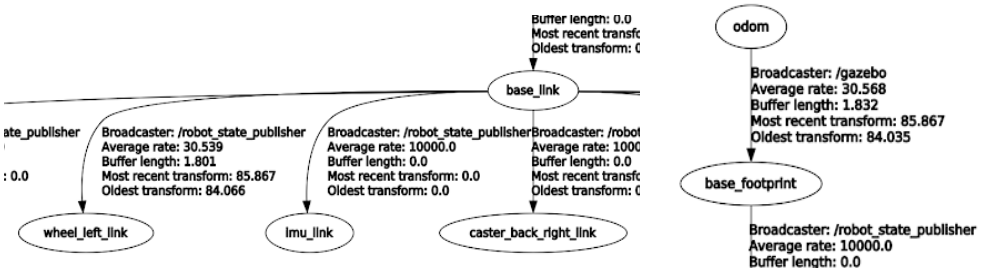

Fig. 20: Porciones del árbol de coordenadas aumentadas.

Por ejemplo, en la figura 20, se puede observar con más detalle este árbol. Se puede observar que la rueda izquierda de la base del robot ha tenido una transformación respecto la propia base. En la línea del tiempo ha sido a los 85,867 segundos y la

transformación anterior ha sido de 84,066 segundos, por lo que la frecuencia de muestreo para las transformaciones es de 1,801 segundos aproximadamente, tal y como indica el campo "Buffer length". La odometría cambia a la vez que la rueda en el tiempo. Dentro de este árbol, también se guardan las transformaciones en los ejes de coordenadas (x, y, z) aunque no se vean directamente. Por lo que se puede utilizar para movimientos relativos.

El topic tf_static, sin embargo, recoge las transformaciones fijas y que no cambian en el tiempo, por lo que son válidas en cualquier momento [23].

El nodo turtlebot3_slam_gmapping recoge los datos de las coordenadas completas del robot (las relaciones de transformación cambiantes en el tiempo tf y las fijas tf_static) y del escáner, para tener los datos del robot sobre el entorno. A su vez, es editor del topic map y del *topic* tf, por lo que actualiza los datos de mapa y de coordenadas de las partes del robot. El *topic* map transporta entre nodos la información y los datos de mapa para poder formar un mapa del entorno.

Ahora bien, una vez iniciado el ejecutable, se tiene que mover el robot para recorrer el entorno y poder guardar todos los datos creando un mapa completo. Hay dos posibilidades para mover el robot como ya se ha indicado en la sección 3.2.: telemanipulación o simulación automática.

Una vez se ha recorrido todo el entorno, se puede guardar en formato mapa lo que se ha mapeado escribiendo en una nueva terminal la línea de comando:

```
rosrun map_server map_saver -f ~/NombreMapa
```

El nombre con el que se va a guardar el mapa es "NombreMapa", donde se escribe el nombre deseado.

Los mapas se almacenan en dos archivos: un archivo de imagen (.png) que codifica los datos de ocupación. Con la configuración estándar, los píxeles más blancos son libres, los píxeles más negros están ocupados y los píxeles intermedios o grises son desconocidos (Fig. 21). Y un archivo de configuración (.YAML) que describe los metadatos del mapa y nombra el archivo de imagen .png. Este archivo tiene unos campos requeridos, los cuales son [24]:

- Image (imagen): ruta al archivo de imagen que contiene los datos de ocupación; puede ser absoluto o relativo a la ubicación del archivo YAML [24].

- Resolution (resolución): resolución del mapa, metros / píxel [24].

- Origin (origen): La posición 2D del píxel inferior izquierdo del robot en el mapa, con el formato: x, y, ψ, con yaw como rotación en sentido antihorario (ψ = 0 significa que no hay rotación) [24].

- Occupied_thresh: es un umbral en el que los píxeles con una probabilidad de ocupación superior a este se consideran completamente ocupados [24].

- Free_thresh: es un umbral en el que los píxeles con una probabilidad de ocupación inferior a este se consideran completamente libres [24].

- Negate (negativo): indica si la codificación libre/ocupada o blanca/negra debe invertirse, la interpretación de los umbrales no se ve afectada [24].

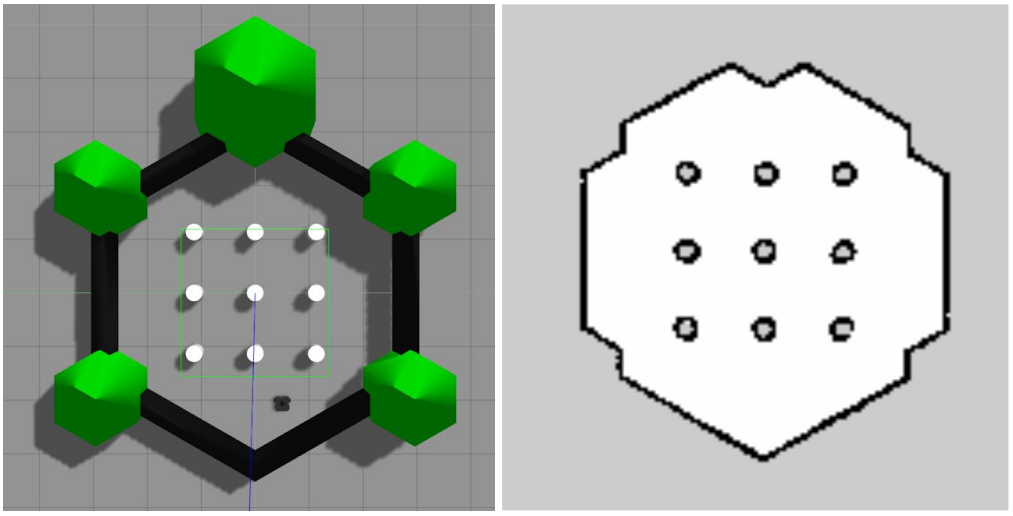

Fig. 21: Entorno original (izq.) e imagen del mapa creado (dcha.)

Con esto se consigue obtener un mapa del entorno que se puede usar posteriormente para otras funcionalidades. También se obtiene la información del espacio ocupado y del espacio libre que se puede recorrer del entorno pertinente en cada caso.

3.4. Navegación

La funcionalidad de navegación consiste en ir a un punto determinado del mapa autónomamente sin chocar con ningún objeto ni ninguna pared presente. Esto se lleva a cabo a través del ejecutable turtlebot3_navigation.launch, el cual, carga el mapa por defecto map y abre RViz, además de llamar a otros nodos. El mapa que se ha creado tras hacer SLAM es un mapa que se puede utilizar en este momento.

Para esto se necesita, como en todos los casos, el *master* en proceso y el simulador Gazebo. En una nueva terminal se indica el modelo robot a seleccionar antes de ejecutar el archivo que abre Gazebo con el comando:

```
export TURTLEBOT3_MODEL=waffle
```

Además, en otra nueva terminal, se exportará el modelo robot y se ejecutará el archivo indicado escribiendo en la terminal el comando:

roslaunch turtlebot3_navigation turtlebot3_navigation.launch

Se debe tener en cuenta que se tiene que indicar primero el mismo modelo robot que se ha indicado en Gazebo con el comando export. En este caso, también hay que tener en cuenta que el mapa elegido en el archivo de navigation.launch tiene que corresponder al entorno ejecutado en Gazebo. Este archivo se puede ejecutar con el mapa por defecto map o añadir a la línea de comando anterior: map_file:=$HOME/NombreMapa.yaml para pasar como parámetro el mapa deseado para abrir en RViz, indicando el directorio en el que se encuentra y el nombre del mismo.

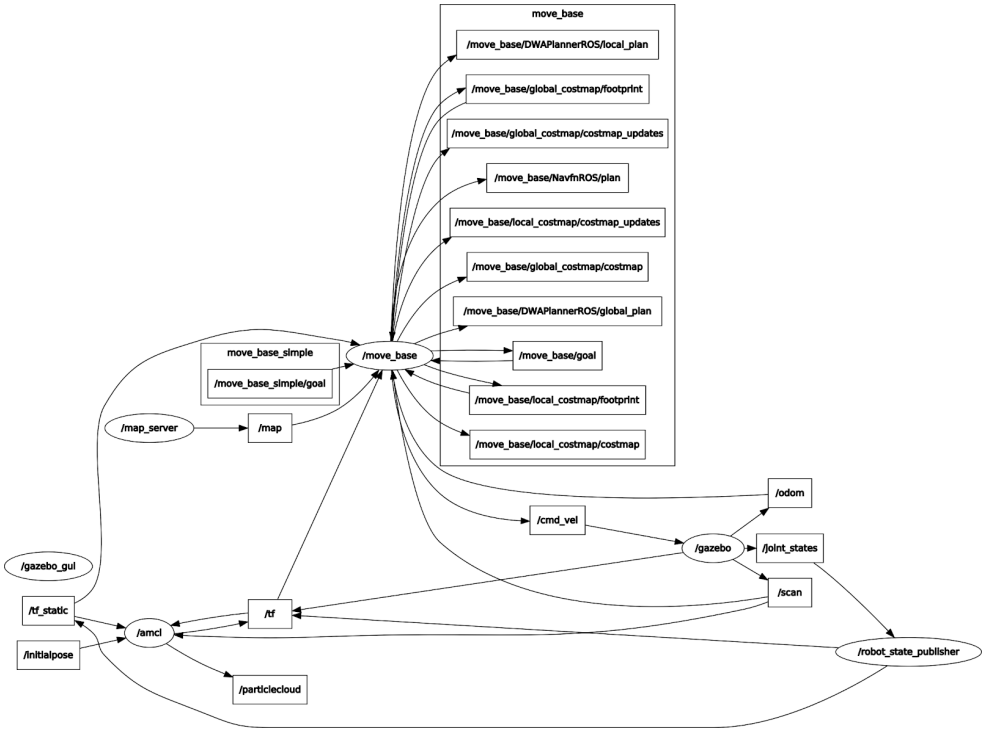

Fig. 22: Rosgraph del ejecutable turtlebot3_navigation.launch en proceso.

Durante la aplicación de navegación se encuentran activos más nodos y topics (Fig. 22) que en las funcionalidades de las secciones 3.2. y 3.3., ya que es una función más compleja. El rosgraph que se muestra (Fig. 22) no contiene los nodos y *topics* activos durante el movimiento del robot, por lo que el *rosgraph* completo con el movimiento de la base es más complejo aún. Como ya se ha señalado durante el desarrollo de este proyecto, hay dos posibilidades para mover el robot y el flujo completo sería la unión del flujo de navegación (Fig. 22) y el flujo de movimiento del robot por telemanipulación (Fig. 15) o el de navegación con el de simulación automática (Fig. 16).

En el *rosgraph* hay nodos y *topics* ya conocidos como los nodos: gazebo y robot_state_publisher, y los *topics*: scan, odom, joint_states, tf, tf_static, cmd_vel y map, pero el archivo turtlebot3_navigation.launch ejecuta también los nodos: map_server, amcl y move_base.

El nodo map_server ofrece datos de mapas como un servicio ROS y proporciona la utilidad de línea de comandos que permite guardar mapas generados dinámicamente en un archivo [24]. Por lo que se encarga de todo lo relacionado con el mapa y de enviar la información necesaria a través del *topic* map visto en SLAM.

Otro nodo clave para está función es amcl. Amcl es un sistema de localización probabilístico para un robot que se mueve en un espacio bidimensional (2D) e implementa el enfoque adaptativo o muestreo KLD de localización de Monte Carlo, el cual, utiliza un filtro de partículas para rastrear la posición de un robot en un mapa conocido. Por lo que el nodo amcl toma un mapa basado en láser e inicializa su filtro de partículas de acuerdo con los parámetros proporcionados, escanea con láser el entorno en el que se encuentra y manda las posiciones estimadas del robot a través de mensajes en forma de árbol de coordenadas por el topic tf. Hay que tener en cuenta que si no se establecen parámetros, el estado inicial del filtro será una nube de partículas de tamaño moderado centrada alrededor de (0,0,0) [25].

Este nodo funciona como suscriptor del sensor láser (*topic* scan), del árbol de coordenadas TF (*topics* tf y tf_static) y del *topic* initialpose (este *topic* transporta los datos de media y covarianza con los que reinicializar el filtro de partículas cada vez que se avanza en el mapa). Además, funciona como editor del *topic* particlecloud (el cual, lleva la información del conjunto de posiciones estimadas que mantienen el filtro) y del *topic* tf donde publica las transformaciones actualizadas [25].

El nodo move_base implementa las funciones y comunicaciones necesarias para que el robot consiga alcanzar la posición objetivo. En ausencia de obstáculos dinámicos, el nodo move_base conduce al robot a la posición objetivo indicada. Por defecto, el nodo move_base toma diferentes acciones para llegar al objetivo. Primero intenta encontrar el espacio libre de obstáculos y, más adelante, mira a su alrededor para comprobar si puede avanzar. Con esto calcula la trayectoria por un espacio libre de obstáculos hasta la posición final indicada. Si no consigue llegar al objetivo bloqueado por obstáculos, el robot considerará que su objetivo no es factible y notificará al usuario que ha abortado [26].

El nodo publica en cmd_vel, *topic* del que ya se ha hablado, comunicando a través de él una secuencia de comandos de velocidad destinados a ser ejecutados por una base móvil. Gazebo es el suscriptor de este *topic* y, por lo tanto, el que utilizará estos datos para proporcionarlos a la base móvil. También es editor de muchos nodos propios del paquete move_base, pero uno de los más característicos es move_base/goal, donde se encuentra la posición objetivo en el entorno [26].

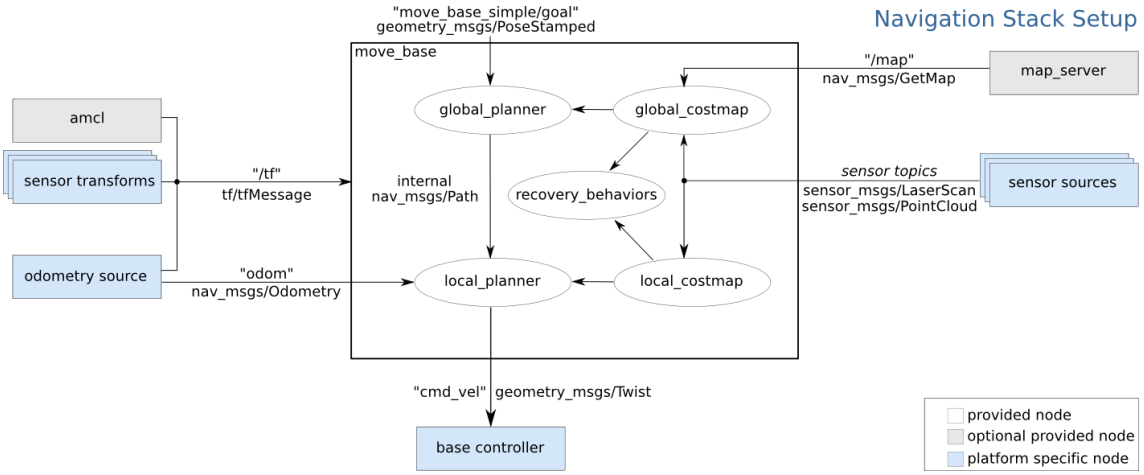

Fig. 23: Esquema de funcionamiento del paquete move_base.

En la imagen de arriba (Fig. 23) se muestra una vista esquemática del nodo move_base y su interacción con otros componentes. Los cuadrados en azul varían según la plataforma del robot, los grises son opcionales pero se proporcionan para todos los sistemas, y los nodos blancos son obligatorios pero también se proporcionan para todos los sistemas. Aquí se observa como el nodo move_base tiene dos planificadores, uno local y otro global, y dos mapas, uno local y otro global también. Los locales son los que se llevan a cabo con la navegación tras recibir los datos en tiempo real del robot, por lo que partiendo de los globales, se pueden corregir y completar los locales con los datos recibidos de Gazebo. Es decir, partiendo de un mapa global, se hace una planificación hasta la posición objetivo pero, con el mapa local que se va creando a medida que el robot va avanzando, se corrige el mapa global y se corrige la planificación inicial, ya que se ha podido añadir un objeto a entorno [26].

Con todos estos nodos clave y la comunicación entre ellos a través de los *topics* vistos se consigue navegar por un mapa dado sin que el robot se choque con ningún obstáculo, siendo capaz de calcular una ruta y recalcular esta si aparece un obstáculo no previsto.

Una vez se han ejecutado todos los archivos necesarios para llevar a cabo la navegación, cabe la posibilidad de que se tenga que ajustar el mapa, es decir, en el mapa

cargado, el robot se encuentra en un punto del mapa que puede diferir de la posición actual del robot en el mapa simulado de Gazebo. Si pasa esto, la información recogida del robot, no coincidirá con el mapa y comenzará con el plan de recuperación comentado anteriormente. En la imagen (Fig. 24) se ve como la pared del mapa (líneas formadas por los puntos negros y azules) no coincide con la información del láser enviada por el robot (línea formada por los puntos azules, verdes y rojos). Por ello, se tiene que ajustar el mapa cargado con el formado por la información a tiempo real proporcionada por Gazebo.

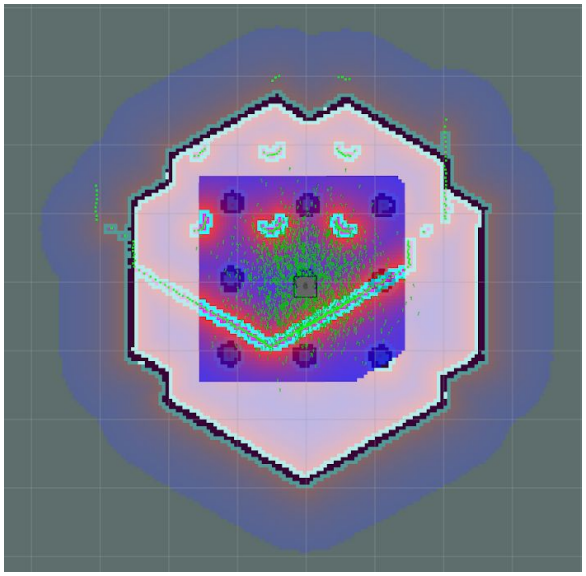

Fig. 24: Programa RViz con el mapa cargado no coincidente con el que ve el robot.

El ajuste del mapa se puede llevar a cabo de dos formas diferentes:

- Ajuste del mapa a través de la herramienta RViz: con la opción de "2D pose estimate" de RViz. Se indican la posición y dirección aproximadas del robot en el entorno real en el mapa cargado con el ratón. Esto se lleva a cabo presionando el botón izquierdo del ratón en la posición deseada para indicar la posición y arrastrando el ratón en la dirección deseada antes de soltar el botón para indicar la dirección. Así se consigue que coincida la posición real del robot con la estimada en el mapa cargado.

- Ajuste del mapa a través del archivo .YAML de configuración del mapa: este archivo, como se ha descrito en la sección 3.3., contiene el origen del robot. Simplemente hay que indicar en el archivo las coordenadas iniciales del robot en el entorno real (simulado en Gazebo en tiempo real).

Si el mapa no es ajustado, es decir, el robot se encuentra en un punto diferente del mapa en el mapa cargado y el mapa simulado, puede haber errores al calcular la trayectoria, ya que parte de una posición que no es la correcta. Por ejemplo, en el caso de la imagen de los mapas no coincidentes (Fig. 24), el robot va a completar el mapa cargado con lo que está viendo actualmente, por lo que va a añadir al mapa los objetos y paredes que está recibiendo y, al recalcular la trayectoria a la posición de destino, lo más seguro es que se encuentre encerrado entre tanto obstáculo o la trayectoria que consiga calcular no sea la más acertada. En definitiva, no estará navegando por el entorno que es realmente, sino con el mismo entorno solapado dos veces.

Una vez se tienen ajustados los mapas, se puede proceder a la navegación. Primero se indica posición y dirección final u objetivo al robot a través de la herramienta RViz. En RViz existe una opción en la que se le puede marcar, al igual que en el caso de indicar la posición estimada real del robot, la posición y dirección finales del robot. Esta opción es “2d nav goal”. Una vez se selecciona la opción indicada, se selecciona sobre el mapa la posición y se arrastra antes de soltar para marcar la dirección. Tras proporcionarle al robot una posición objetivo, TB3 calcula la trayectoria más corta. Una vez que tiene la trayectoria calculada, comienza a dirigirse hacia la posición destino sin chocar. En el caso de que haya un objeto no previsto para la base, TB3 recalcula la trayectoria a partir del mapa corregido (añadiendo el objeto que ha visto en el mapa cargado) para llegar a la posición objetivo por el camino más corto.

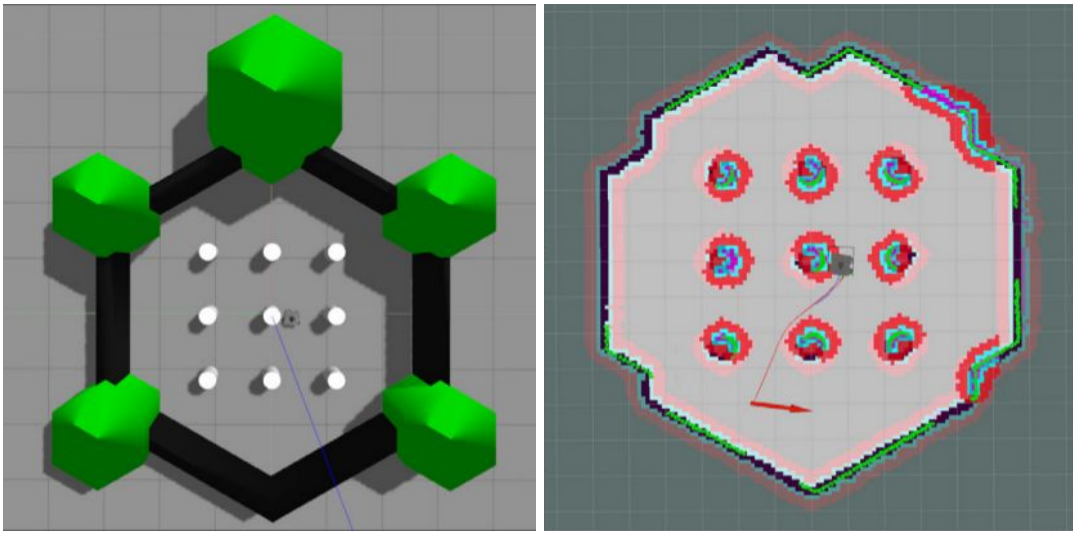

Fig. 25: Entorno Gazebo (izq.) y mapa en RViz con la posición objetivo marcada (dcha.)

En la imagen anterior (Fig. 25) se observa el entorno real en Gazebo comparado con el entorno en RViz. La imagen izquierda de la figura 25 representa el robot en el entorno en Gazebo simulando el mundo real y la imagen derecha de la figura 25 representa al robot posicionado en el mapa cargado con la trayectoria a seguir y la posición objetivo. La flecha roja indica la dirección (ψ) en la que debe quedarse el robot para finalizar la tarea de navegación y, el punto inicial de esa flecha, indica la posición 2D (x, y) objetivo. La línea roja desde el robot hasta la flecha roja es la trayectoria calculada hasta la posición objetivo.

Hay parámetros configurables en la aplicación de navegación, como por ejemplo el valor de ensanchamiento de los bordes de los objetos o paredes. Este parámetro se basa en aumentar o disminuir, de alguna forma, el tamaño de los objetos. Con la configuración de este parámetro, el robot pasa a mayor o menor distancia evitando choques de alguna parte del robot con los diferentes obstáculos, ya que desde el sensor de posición hasta otra parte del robot hay una distancia no calculada, por lo que la base puede colisionar con un obstáculo al ir por la trayectoria calculada si no se tienen en cuenta estas distancias. Si este parámetro se configura con un valor demasiado alto, puede que no quede espacio entre objetos y el robot no sea capaz de calcular una trayectoria, ya que no cabe y esa "distancia de seguridad" el robot lo entiende como

obstáculo. Este es un mero ejemplo de las diferentes configuraciones que se pueden llevar a cabo en los mapas de navegación o en la aplicación de navegación.

Con esto, se puede navegar por cualquier entorno del que se tenga un mapa previo evitando obstáculos.

3.5. Multirobot

Se han visto varias aplicaciones básicas para una sola TB3, pero también cabe la posibilidad de tener varias bases móviles en un mismo entorno. Se van a comparar algunas de las funcionalidades de las que se ha hablado entre una única base y varias bases en el entorno.

La posibilidad de multirobot abre un gran abanico de posibles aplicaciones o tareas a realizar con la base móvil de TB3. Al disponer de un mayor número de robots, por ejemplo, se podría completar la búsqueda de basura en mucho menos tiempo, pudiendo informar a otros por dónde han pasado o qué han encontrado.

Para llevar a cabo las funcionalidades multirobot, al igual que con una TB3, tiene que estar en proceso *roscore* y en una nueva terminal ir abriendo o ejecutando cada funcionalidad o *launch* deseado.

Al igual que en el capítulo 3., va a ser necesario tener el simulador Gazebo ejecutado para tener los datos a tiempo real de los sensores (escáner, cámara, posición, etc) del robot. Por lo tanto, se va a ejecutar el archivo multi_turtlebot3.launch para abrir Gazebo con varios robots en un entorno. En una nueva terminal se va a escribir el comando:

```
export TURTLEBOT3_MODEL=waffle
```

Comando con el que se elige el modelo robot para los tres robots del entorno. En otra terminal se escribe el comando:

roslaunch turtlebot3_gazebo multi_turtlebot3.launch

Este *launch* abre el entorno house.world, un entorno con varias habitaciones, y carga tres modelos robot (archivos URDF) posicionando cada uno en una parte del entorno (Fig. 26). El modelo robot se sigue eligiendo con el comando export, pero también se puede predefinir un modelo para los tres robots o el modelo robot deseado para cada uno de ellos, al igual que se puede posicionar la cantidad de TB3 haciendo una modificación del ejecutable. También se puede configurar en el propio ejecutable la posición en el entorno de cada robot y el propio entorno.

En la imagen (Fig. 26) se observan los robots posicionados en el mapa, cada uno se encuentra en una habitación diferente. El robot de la habitación de la izquierda es tb3_1, el de la habitación del centro tb3_2 y el de la habitación de la derecha tb3_0.

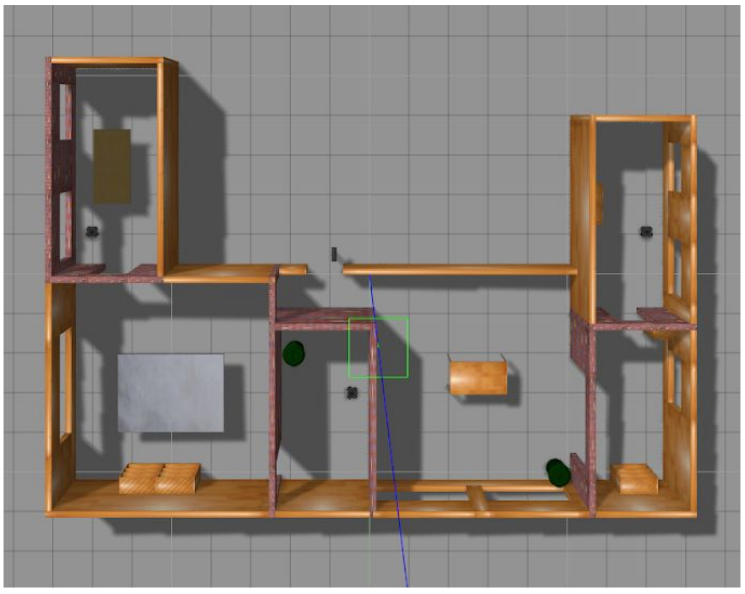

Fig. 26: Tres TB3 distribuidas en el entorno house.world.

En la sección 3.4. se ha visto que la navegación autónoma por un entorno es la funcionalidad más compleja y, para su realización, es necesario aplicar todas las funciones vistas con anterioridad. Esta aplicación no está incluida para multirobot en el paquete de ROS, por ese motivo, se van a explicar todas las funcionalidades útiles para llevar a cabo la funcionalidad de navegación sin explicar la funcionalidad de navegación en multirobot, ya que no está implementada.

Para la navegación es necesario crear un mapa con la función SLAM. La ventaja de hacer SLAM con varios robots es que se recorre el entorno por completo más rápidamente, ya que no solo es un robot el que obtiene datos del entorno.

El mapa para la navegación puede ser cualquier mapa ya creado, por ejemplo, se podría usar el mapa creado en la sección 3.3. mientras que en Gazebo se ejecutase el entorno del mapa, no tiene sentido si no coinciden entorno y mapa.

En este caso, para ver la función de SLAM multirobot, se va a crear un nuevo mapa del entorno turtlebot3_house.world. Además, es el entorno que carga el *launch* propuesto para multirobot en esta sección (sección 3.5.) y, así, no es necesario hacer modificaciones en el ejecutable.

Tras tener *roscore* y Gazebo ejecutados, en una nueva terminal, se va a ejecutar la función SLAM para el mapeo del entorno. Para conseguir el mapa con multirobot se van a tener que abrir muchos procesos, además de *roscore* y Gazebo, y cada uno de ellos en una terminal diferente (Fig. 27).

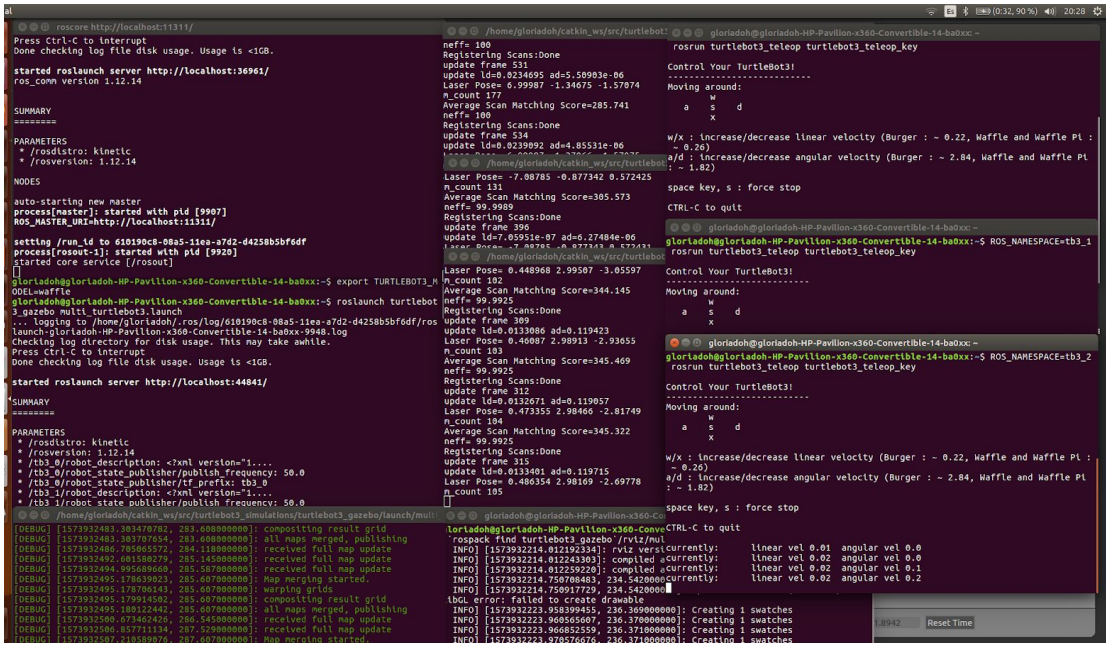

Fig. 27: Todos los procesos activos para llevar a cabo SLAM multirobot.

El *launch* encargado de ejecutar esta aplicación es turtlebot3_gmapping.launch, el mismo que se ha ejecutado en la sección 3.3. para que una sola TB3 realice la funcionalidad de SLAM. La diferencia entre ejecutar el *launch* en este caso multirobot y ejecutar el *launch* con una solo TB3 es que en este caso se va a asignar a cada robot esa funcionalidad. En el caso de esta funcionalidad los robots son independientes, por lo que el *launch* no ejecuta la funcionalidad para todos los robots sino que ejecuta la función para cada robot. Por ello, hay que indicarle al *launch* el robot con el que se va a realizar el SLAM. Para esto, se va a usar el comando de ROS ROS_NAMESPACE, variable que guarda el espacio de nombres como se ha introducido al principio de este apartado en la capítulo 2.

Antes de ejecutar en la terminal el *launch* que lleva a cabo la funcionalidad de SLAM, se debe de elegir el modelo robot con el comando:

export TURTLEBOT3_MODEL=waffle

Tienen que coincidir el modelo robot en Gazebo y el que se proporciona en la misma terminal para realizar SLAM. Si no fuera el mismo modelo de robot, habría inconsistencias.

En este caso, los robots se llaman tb3_0, tb3_1 y tb3_2. Nombres por defecto que pueden ser editables, como todo. Por lo que si se quiere que, por ejemplo, la base tb3_1 haga mapeo, se asignará la funcionalidad de mapeo a ese robot de la siguiente forma:

ROS_NAMESPACE=tb3_1 roslaunch turtlebot3_slam turtlebot3_gmapping.launch.

Se indica el nombre deseado con ROS_NAMESPACE. Tras elegir la variable de nombre adecuada, se escribe el *launch* a ejecutar con el comando dedicado (*roslaunch*). En la misma línea de comando, se tienen que añadir algunos parámetros para que el nodo implicado recoja y publique los datos a los *topics* correctos, asociados a cada robot. Por defecto están los datos de un robot en general y, por supuesto, cuando sólo hay una TB3 no hay confusión, pero cuando existen varias bases móviles hay que indicar claramente de cuál de ellas se escogen los datos. En este caso, en la función *gmapping*, los parámetros que son importantes en el archivo son:

- set_base_frame
- set_odom_frame
- set_map_frame.

A estos parámetros se les da el valor deseado y se añaden a la línea de comando anterior en la que se ejecuta turtlebot3_gmapping.launch:

- set_base_frame:=tb3_1/base_footprint
- set_odom_frame:=tb3_1/odom
- set_map_frame:=tb3_1/map

Como sus propios nombres indican, se proporciona la base asociada al robot tb3_1, el sensor de posición del tb3_1 y el mapa asociado a ese robot respectivamente.

En definitiva, la línea de comando a escribir en la nueva terminal para llevar a cabo la funcionalidad de SLAM para el robot llamado tb3_1 es:

```
ROS_NAMESPACE=tb3_1 roslaunch turtlebot3_slam turtlebot3_gmapping.launch
set_base_frame:=tb3_1/base_footprint set_odom_frame:=tb3_1/odom
set_map_frame:=tb3_1/map
```

Para realizar el mapeo con los tres robots se va a ejecutar para cada robot el *launch* en una nueva terminal, ya que es un proceso diferente asociado a cada TB3. Es decir, se abre una terminal, se escribe la línea de comando:

```
ROS_NAMESPACE=tb3_0 roslaunch turtlebot3_slam turtlebot3_gmapping.launch
set_base_frame:=tb3_0/base_footprint set_odom_frame:=tb3_0/odom
set_map_frame:=tb3_0/map
```

En otra terminal se escribe:

```
ROS_NAMESPACE=tb3_1 roslaunch turtlebot3_slam turtlebot3_gmapping.launch
set_base_frame:=tb3_1/base_footprint set_odom_frame:=tb3_1/odom
set_map_frame:=tb3_1/map
```

Y, por último, en una nueva terminal se escribe:

```
ROS_NAMESPACE=tb3_2 roslaunch turtlebot3_slam turtlebot3_gmapping.launch
set_base_frame:=tb3_2/base_footprint set_odom_frame:=tb3_2/odom
set_map_frame:=tb3_2/map
```

Así se tienen a los tres robots realizando SLAM. Es importante ejecutar esta función con cada robot que está en simulación en el entorno porque, si no es así, se encuentran árboles TF descolgados y sin unir, ya que usan el mismo mapa de referencia. Esto da problemas porque, al existir robots sin referenciar en el mapa pero existiendo en el mismo, las transformaciones del robot o robots descolgados no pasan correctamente al mapa. Es decir, si en el entorno simulado con Gazebo hay tres robots, esta función se va a ejecutar tres veces (una vez por robot) para que el árbol TF esté bien unido (Fig. 28).

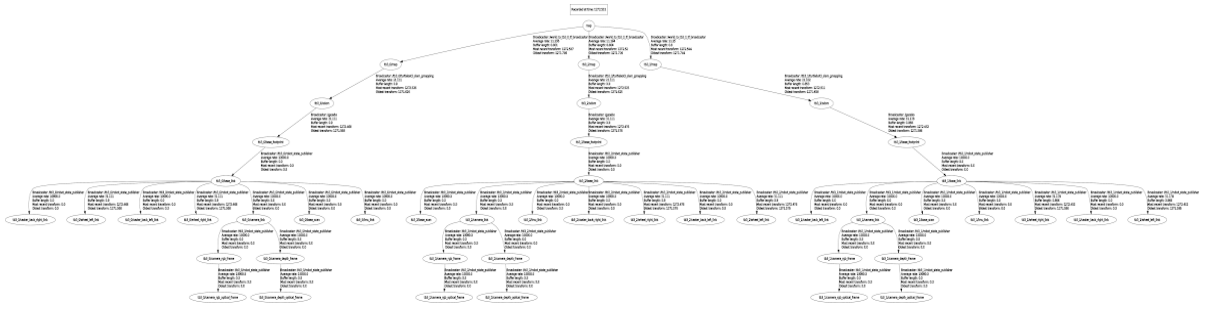

Fig. 28: Árbol de coordenadas TF completo con los 3 robots y el mapa.

Esta funcionalidad construye un mapa a través de los datos recibidos de cada robot a través de Gazebo. Si son tres robots, se obtienen tres mapas diferentes, cada uno proporcionado por los datos recibidos de cada base móvil, por lo que para unir esos tres mapas y solo obtener uno, que es lo interesante, se usa el archivo multi_map_merge.launch, el cual realiza la unión de los tres mapas (Fig. 29). Esto se hace a través del comando en una nueva terminal:

```
roslaunch turtlebot3_gazebo multi_map_merge.launch.
```

Este *launch* consigue unir los tres mapas creados, uno por cada robot, en un solo mapa.

Una vez tenemos esto, se necesita ejecutar la herramienta GUI que facilita la visualización del mapeo: RViz (Fig. 29). En una nueva terminal se escribe el comando:

```
rosrun rviz rviz -d `rospack find turtlebot3_gazebo`/rviz/multi_turtlebot3_slam.rviz
```

Si no se ejecuta esta herramienta, no se verá por pantalla el mapa que van creando los robots, solo se podrán ver los robots moviéndose en Gazebo pero, en Gazebo, no se refleja visualmente lo que ha visto y ve el robot a través del sensor láser.

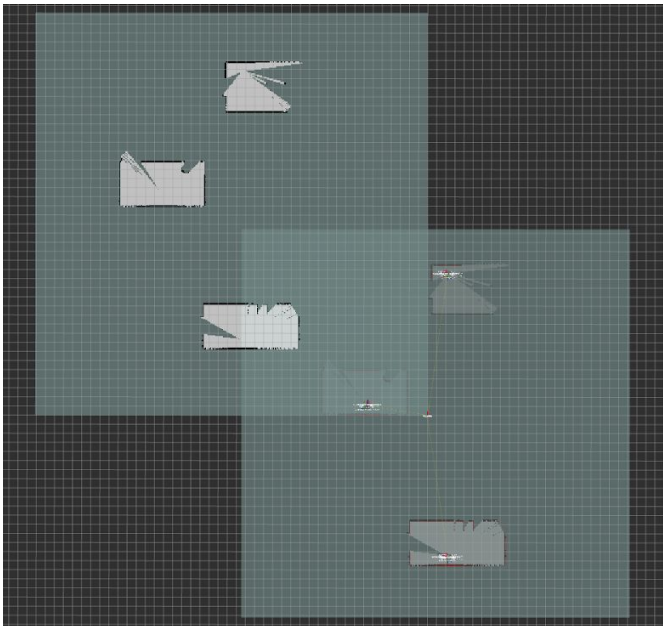

Fig. 29: RViz. Mapa unido (arriba izq) y mapas independientes (abajo dcha)

Si se hiciera de igual manera que en la sección 3.3., con el *launch* turtlebo3_slam.launch, se debería de configurar el *launch* para que el argumento 'open_rviz' tuviera el valor '*false'* en vez de '*true'* como está por defecto, además de seleccionar por parámetro la técnica SLAM a llevar a cabo. Si se selecciona abrir RViz en cada comando ejecutado, se abriría una ventana de RViz asociada a cada robot y no se conseguiría visualizar el conjunto del mapa con los datos de las tres bases móviles. Por ello, se ejecuta en una nueva terminal la herramienta RViz a través del archivo

multi_turtlebot3_slam.rviz, preparado para reflejar los datos recogidos de los tres robots y presentarlos por pantalla.

Para registrar todos los datos del entorno para crear el mapa del mismo, al igual que en la sección 3.3., el robot necesita pasearse por el entorno. Para ello, se va a utilizar la telemanipulación de las bases a través del nodo turtlebot3_teleop_key. Al igual que con la función SLAM, cada robot se mueve independiente del otro, por lo que se necesita ejecutar la telemanipulación para cada robot a través de una terminal diferente para cada TB3, ya que son tres procesos diferentes (Fig. 30). Esto se lleva a cabo abriendo una nueva terminal (ctrl + t) a través del comando:

```
ROS_NAMESPACE=tb3_0 rosrun turtlebot3_teleop turtlebot3_teleop_key
```

Para cada robot se ejecuta el *launch* cambiando la variable de nombre. Para manejar los tres robots, en una nueva terminal se ejecuta el *launch* para el movimiento del segundo robot y se escribe el comando:

```
ROS_NAMESPACE=tb3_1 rosrun turtlebot3_teleop turtlebot3_teleop_key
```

Y, por último, en otra terminal se ejecuta la aplicación para el movimiento del tercer robot y se escribe el comando:

```
ROS_NAMESPACE=tb3_2 rosrun turtlebot3_teleop turtlebot3_teleop_key
```

Teniendo estos *launch* en proceso, se pueden manejar los tres robots escribiendo en cada terminal los controles correspondientes (Fig. 30). Siendo estos los mismos comentados en la sección 3.2.:

- w: aumento de velocidad lineal.
- x: disminución de velocidad lineal.
- a: aumento de velocidad angular.

- d: disminución de velocidad angular.
- s: puesta a cero.

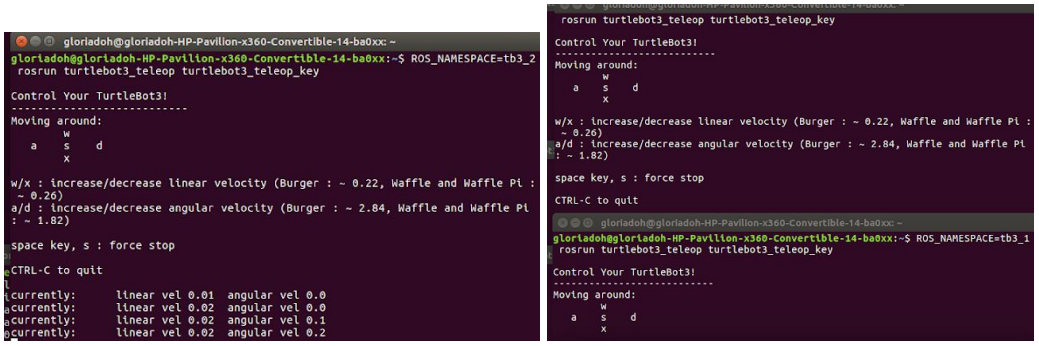

Fig. 30: Terminales con el nodo turtlebot3_teleop_key en proceso. Una por robot.

Una vez recorrido el mapa, se puede guardar gracias al paquete map_server para poder utilizarlo más adelante. En una nueva terminal, se escribe el comando:

rosrun map_server map_saver -f ~/NombreMapa

En las imágenes que vienen a continuación (Fig. 31), se muestra el entorno simulado en Gazebo (house.world) y el mapa creado tras realizar el mapeo con la aplicación de SLAM.

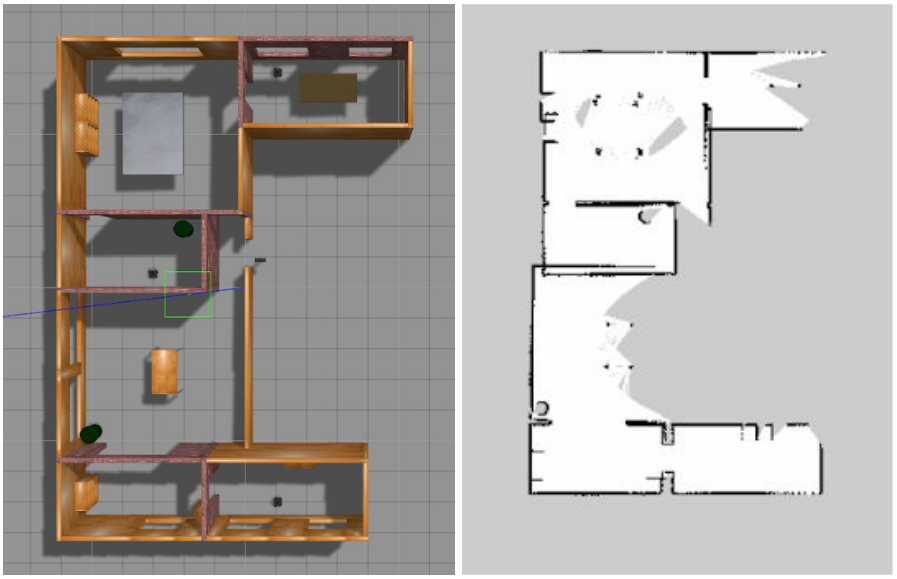

Fig. 31: Entorno original (izq.) e imagen del mapa creado (dcha.)

Los nodos y *topics* implicados en esta funcionalidad (Fig. 32) son todos los implicados en la sección 3.3. triplicados y referenciados a cada robot. Es decir, se ven tres grupos de nodos y topics, uno por robot, que tienen activos los mismos nodos y topics que los de la funcionalidad SLAM con una sola base móvil.

Aparece el nodo map_merge, encargado de juntar los tres mapas proporcionados por los robots (tb3_0/map, tb3_1/map y tb3_2/map) en uno (map).

Se puede observar (Fig. 32) que los tres robots comparten árbol de coordenadas, no hay un árbol tf por cada robot, sino que hay un topic tf y otro tf_static para los tres grupos. Estos se forman a través de los nodos world_to_tb3_0_tf_broadcaster, world_to_tb3_1_tf_broadcaster y world_to_tb3_2_tf_broadcaster que se encargan de hacer la transformación a coordenadas tf para formar un único árbol de coordenadas para las tres TB3.

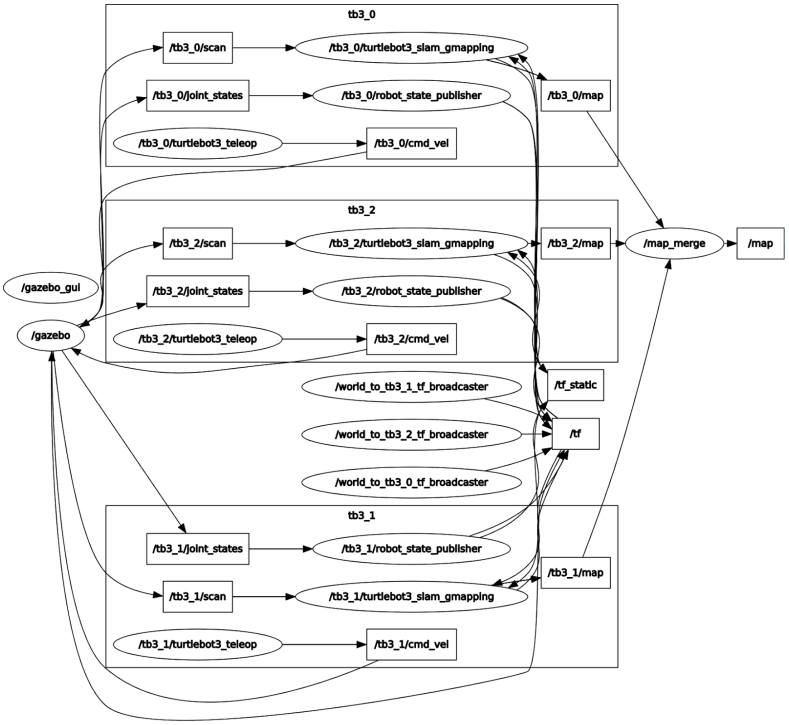

Fig. 32: Rosgraph de nodos y topics activos en multirobot.

Se han visto todas las posibilidades ya implementadas que nos ofrece ROS para aplicaciones multirobot. Gracias a esto, en este caso, se consigue crear un mapa mucho más rápido que con una sola base móvil.

4. OpenManipulator

Tras haber visto la base móvil, se va a ver el brazo robot de OpenManipulator. Este manipulador se puede manejar a través de mensajes ROS y se puede simular de la misma forma que la base en Gazebo.

Para la simulación de OpenManipulator se necesita cargar el brazo en Gazebo (Fig. 33), así se verá la respuesta que tendría el robot real al mandarle diferentes órdenes. En la imagen (Fig. 33) se muestra el brazo de OM simulado en Gazebo en un entorno vacío.

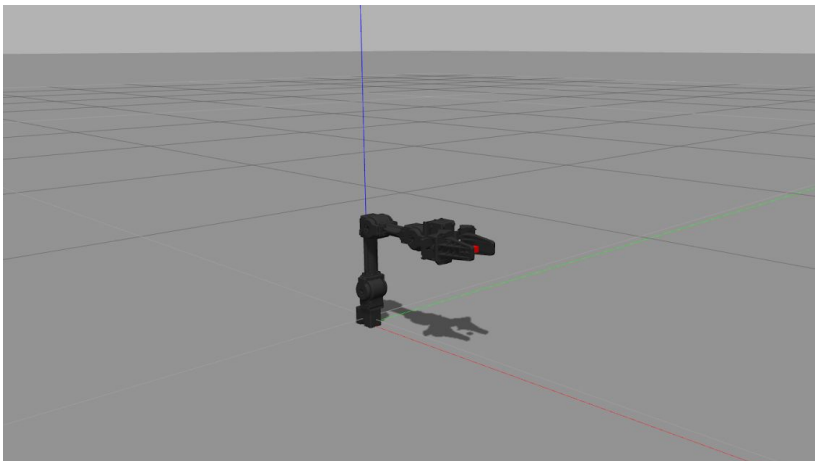

Fig. 33: Brazo de OM simulado en un entorno vacío de Gazebo.

Se va a tener el master en proceso y en una nueva terminal se va a ejecutar el *launch* que simula el brazo de OM en un entorno vacío en Gazebo, el comando a escribir en la terminal es:

```
roslaunch open_manipulator_gazebo open_manipulator_gazebo.launch
```

Una vez se tiene el brazo en simulación, se le da a la opción de *play* en Gazebo para transmitir los datos y se va a poder mover el brazo de diferentes formas: se pueden mandar mensajes a OpenManipulator por comandos ROS o se puede enviar información a través de topics y servicios implicados en el movimiento del brazo manipulador a través de la herramienta rqt o, más intuitivo, se puede manejar con una herramienta GUI propia de OpenManipulator o la aplicación de MoveIt!.

Se van a dar a conocer algunos topics y servicios implicados en el movimiento del brazo, los cuales, podrán ser usados y modificados por mensajes ROS o en rqt o los utilizan las herramientas gráficas como MoveIt!. Estos topics y servicios son los utilizados por el nodo open_manipulator_controller, el nodo principal encargado en el movimiento del brazo robot.

Topics en los que open_manipulator_controller es publicador o editor (publica información) [27]:

- /open_manipulator/states: recoge la información sobre el estado del brazo. Contiene la información de si los actuadores están habilitados o deshabilitados, si el brazo está en movimiento o se encuentra parado...

- /open_manipulator/joint_states: recoge la información sobre los estados de las articulaciones. Esta información consiste en el nombre de la articulación, la posición y/o la velocidad del mismo.

- /open_manipulator/gripper/kinematics_pose: recoge la información sobre la posición y la orientación de la pinza, herramienta o efector final.

- /open_manipulator/*joint_name*_position/command: recoge la información de la posición objetivo elegida para cada articulación. En la parte *joint_name* se indica el nombre de la articulación a la que se indica la posición final.

Topics de los que open_manipulator_controller es suscriptor (recoge información) [27]:

- /open_manipulator/option: da información sobre la configuración de las diferentes opciones de OM, como que muestre la descripción del brazo.

- /open_manipulator/move_group/display_planned_path: da la información de la trayectoria planificada, planificada desde moveIt!

- /open_manipulator/move_group/goal: da la información de la opción elegida para la planificación de la trayectoria, publicada desde moveIt!

- /open_manipulator/execute_trajectory/goal: da la información sobre los estados de ejecución de la trayectoria planificada, publicada desde moveIt!

Algunos de los servicios que ofrece el nodo open_manipulator_controller [27]:

- /open_manipulator/goal_task_space_path_from_present: Crea una trayectoria desde la posición cinemática actual dándole la posición cinemática (posición completa: posición y orientación) del efector final (herramienta) y el tiempo total de la trayectoria.

- /open_manipulator/goal_task_space_path_from_present_position_only: Crea una trayectoria desde la posición cinemática actual dándole la posición del efector final (herramienta) y el tiempo total de la trayectoria.

- /open_manipulator/goal_task_space_path_from_present_orientation_only: Crea una trayectoria desde la posición cinemática actual dándole la orientación del efector final (herramienta) y el tiempo total de la trayectoria.

- /open_manipulator/goal_tool_control: Mueve la herramienta de OM.

- /open_manipulator/set_actuator_state: Controla el estado de los actuadores. Si el valor es verdadero los actuadores están activados, si es falso están desactivados.

- /open_manipulator/goal_drawing_trajectory: Crear una trayectoria que sigue un dibujo determinado. Se puede crear un círculo, un rombo, un corazón y una trayectoria en línea recta.

Tras haber visto los diferentes topics y servicios para open_manipulator_controller, se van a ver las diferentes opciones para mover el brazo, las cuales, utilizan algunos de esos topics y servicios.

En el caso de utilizar la GUI específica para OpenManipulator, se necesita ejecutar el archivo open_manipulator_controller.launch, el cual, ejecuta el nodo open_manipulator en el paquete de open_manipulator_controller. Esto se lleva a cabo en una nueva terminal con el comando:

```
roslaunch open_manipulator_controller open_manipulator_controller.launch
use_platform:=false
```

Se establece el parámetro use_platform=false para que los mensajes enviados a través del controlador OpenManipulator - X lleguen a Gazebo en vez de enviarlos a la plataforma real [28].

Una vez se tiene el controlador en proceso, se ejecuta el archivo que ejecuta la herramienta GUI de OM (Fig. 34) en una nueva terminal escribiendo el comando:

```
roslaunch open_manipulator_control_gui open_manipulator_control_gui.launch
```

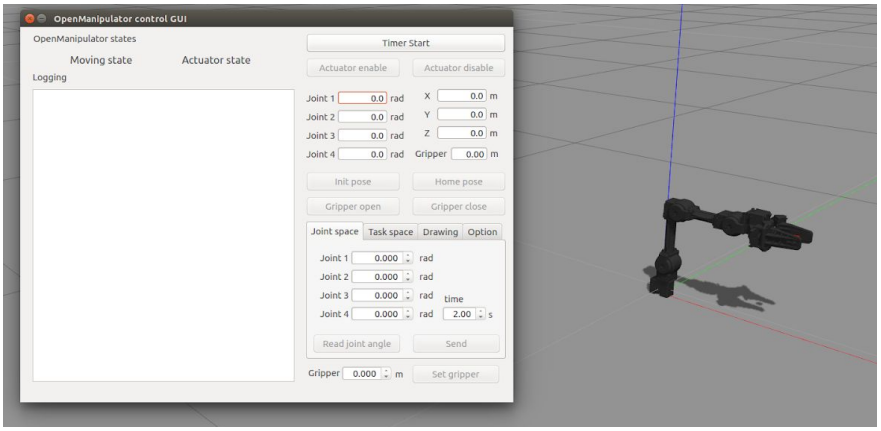

Fig. 34: GUI de OM con el brazo simulado en Gazebo.

Esta herramienta permite mover el brazo modificando los ángulos de cada articulación. También da la posibilidad de seguir diferentes tipos de trayectorias (lineal, circular, …) proporcionándole diferentes datos para poder realizar la trayectoria deseada. El brazo sigue la trayectoria con los datos introducidos, por ejemplo: la trayectoria circular con radio 0.3 m y 1 revolución (Fig. 35). En definitiva, tiene todos los parámetros necesarios para poder manejar el brazo como se quiera, abrir y cerrar la pinza, mandar trayectorias en el espacio de articulaciones, mandar trayectorias en el espacio de tareas, mandar diferentes trayectorias haciendo un dibujo, etc.

En las imágenes (Fig. 35) se muestra la trayectoria circular. La imagen de la izquierda es el brazo en posición inicial, en el centro, una posición intermedia de la trayectoria circular con la herramienta GUI donde se indica en la pestaña "*Drawing*" los datos de la trayectoria y, a la derecha, la imagen muestra la posición final de la trayectoria circular. En la imagen central, también se ve el botón "*Send*", el cual, se usa para enviar los datos de la trayectoria deseada.

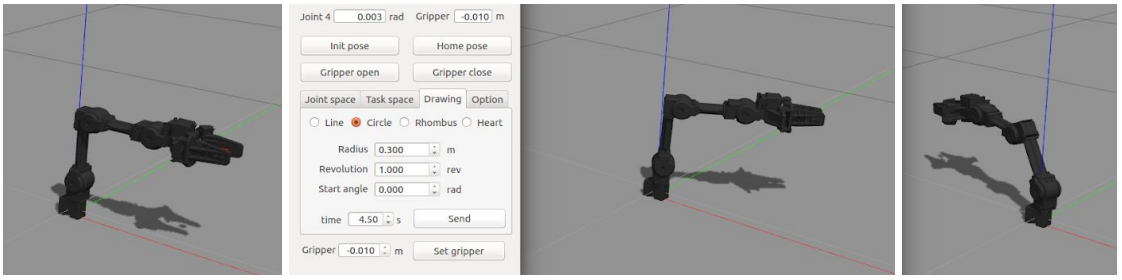

Fig. 35: GUI de OM con trayectoria circular y el brazo en Gazebo

Los nodos y topics activos de este caso se muestran en la imagen que sigue (Fig. 36). Con esto, se ve una idea de cómo es la comunicación entre el brazo y el simulador. Hay nodos asociados a un grupo o robot, como open_manipulator/control_gui, open_manipulator/gripper_sub_publisher y open_manipulator/controller_spawner asociados al brazo open_manipulator. Si tuviéramos dos brazos, estos tres nodos estarían repetidos de la forma: nombre_robot/nodo.

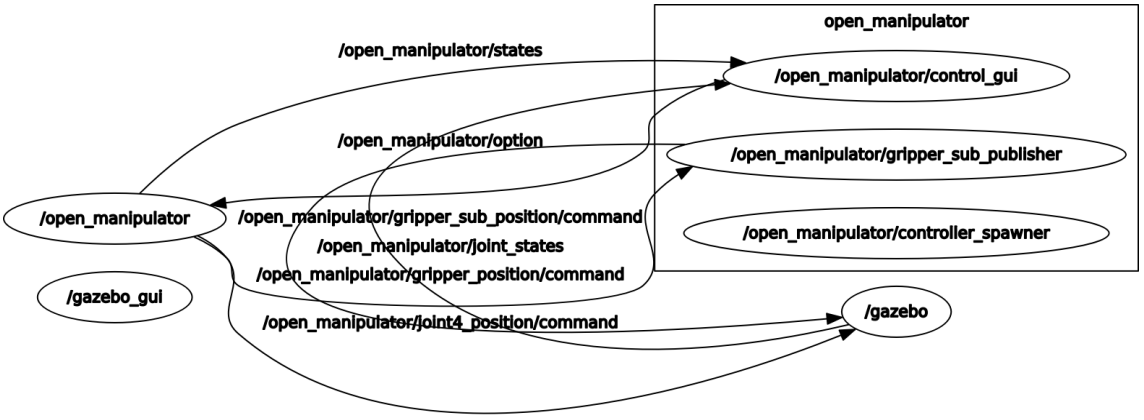

Fig. 36:Rosgraph de nodos y topics activos ejecutando la GUI de OM

Los *topics* que se ven activos (Fig. 36), son los mismos ya comentados en este apartado (capítulo 4.), siendo todos ellos topics publisher y subscriber del nodo de open_manipulator del paquete open_manipulator_controller.

Los nodos activos en esta funcionalidad (Fig. 36) son:

- Gazebo: nodo que controla la interfaz de Gazebo para la simulación del brazo robot.

- open_manipulator: es el brazo open_manipulator, quien proporciona los datos de su estado general o de cada articulación.

- open_manipulator/control_gui: proporciona una interfaz gráfica de usuario basada en QT. Es suscriptor y editor de los topics vistos en esta sección (suscriptor de los topics states, joint_states, kinematics_pose y editor del topic option) además de ofrecer algunos de los servicios también vistos. Es el encargado de mandar a Gazebo las trayectorias solicitadas desde la GUI de OM.

- open_manipulator/gripper_sub_publisher: se encarga de publicar el estado de la pinza.

- open_manipulator/controller_spawner: es el que se encarga de reproducir el control del brazo.

Otra opción, es usar la herramienta MoveIt!, la cual, ofrece infinidad de posibilidades. En nuestro caso, se va a ejecutar el paquete de MoveIt! a través de la herramienta RViz.

Se va a utilizar MoveIt! ejecutando el paquete de control que usa esta herramienta. Este es el mismo archivo que se ha ejecutado para llevar a cabo el movimiento del brazo a través de la GUI propia del paquete de OpenManipulator, pero con una parametrización diferente. A esta línea de comando se le tiene que añadir el parámetro use_moveit y asignarle un valor *true*, de este modo, se ejecutará RViz con la herramienta de MPP para RViz de MoveIt! (Fig.37). Por defecto tiene un valor *false*, lo que significa que no ejecuta MoveIt!.

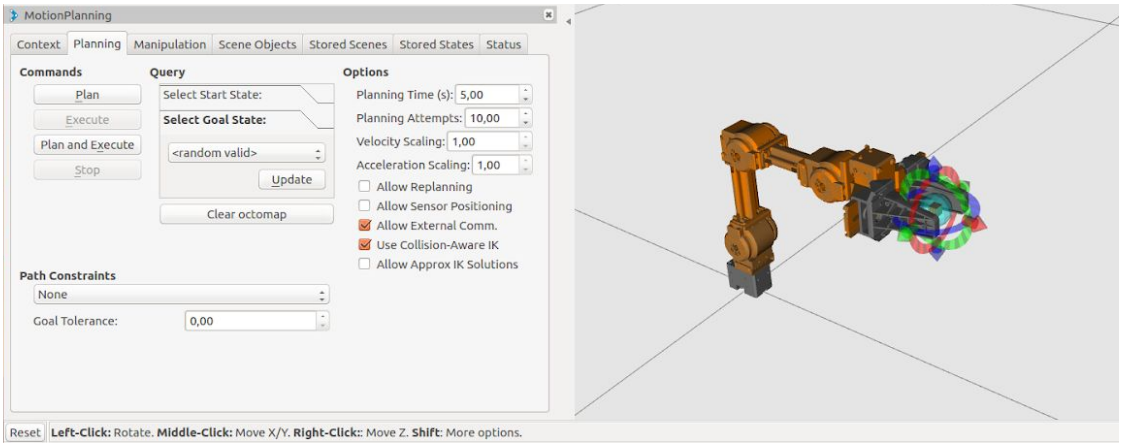

Fig. 37: MPP con el brazo de OM en RViz.

En una nueva terminal se ejecuta el controlador de OM y MoveIt! (Fig. 37) escribiendo el comando:

roslaunch open_manipulator_controller open_manipulator_controller.launch use_moveit:=true use_platform:=false

También se pueden añadir otros parámetros como el nombre del robot, el nombre del grupo a planificar (planning_group_name = arm, por defecto) y/o el tiempo de muestreo al hacer la trayectoria (moveit_sample_duration).

Gracias a la herramienta de Motion Planning Plugin se puede mover el brazo de una posición a otra. La diferencia con la GUI propia de OpenManipulator es que en vez de proporcionarle los datos individuales de cada articulación, se le da una posición inicial y final y MoveIt! te planea y ejecuta la trayectoria de la posición inicial a la posición final dadas.

Los nodos y topics activos en este caso se muestran en la siguiente imagen (Fig. 38):

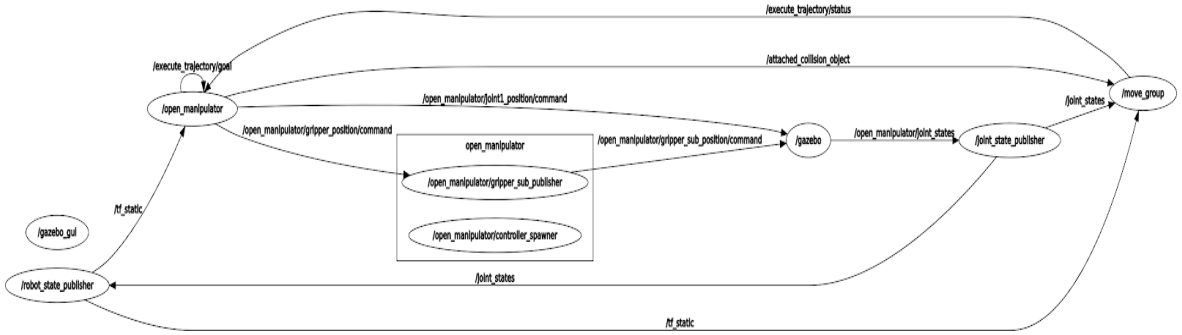

Fig. 38: Rosgraph de nodos y topics activos en el movimiento del brazo con MPP.

Los nodos activos son gazebo, robot_state_publisher, open_manipulator y los dos nodos de open_manipulator: gripper_sub_publisher y controller_spawner. De todos ellos se ha hablado en este capítulo 4. y otras secciones anteriores durante todo el capítulo 3.. También se encuentran los nodos joint_state_publisher y move_group:

- joint_state_publisher: está unido al nodo robot_state_publisher, le manda información del estado de las articulaciones.

- move_group: es el nodo principal de MoveIt!. Es el encargado de MPP en este caso.

Además de los nodos, los topics también se han visto en los capítulos 3 y 4. Como tf_static, execute_trajectory/goal, *joint_name*/command y joint_states. Pero todavía no se ha hablado de attached_collision_object y execute_trajectory/status:

- attached_collision_object: transporta la información sobre algunos de los objetos presentes para no colisionar con ellos al mover el brazo.

- execute_trajectory/status: transporta la información sobre el estado de la trayectoria que se está ejecutando.

La herramienta de MoveIt! tiene muchas posibilidades por sí misma. Además, tiene un asistente en el que se le pueden añadir las posiciones que se quiera para, más tardes, con el MPP y RViz, el robot pueda planificar y ejecutar las trayectorias a todas las posiciones configuradas. El tutorial del asistente es muy sencillo de seguir, está en el enlace: http://docs.ros.org/kinetic/api/moveit_tutorials/

Además de estas dos formas de mover el brazo, hay otras tres más. Una de ellas es enviar a través de la terminal de comandos ROS mensajes a través de los topics dedicados a ello, estos topics son:

- open_manipulator/joint1_position/command
- open_manipulator/joint2_position/command
- open_manipulator/joint3_position/command
- open_manipulator/joint4_position/command
- open_manipulator/gripper_position/command

Para llevar a cabo el movimiento de las diferentes articulaciones del brazo, en una nueva terminal se escribe la línea de comando:

```
rostopic pub /open_manipulator/joint2_position/command std_msgs/Float64
"data: -1" --once
```

Así se publica un mensaje estándar de tipo Float64 en el topic open_manipulator/joint2_position/command, en este caso, se está enviando el valor de -1 a la articulación 2 del robot llamado open_manipulator. Con los cinco topics mencionados se puede posicionar el brazo, posicionando cada articulación y la pinza de una forma determinada.

En la siguiente imagen (Fig. 39) se ven los nodos y topics activos en este caso del movimiento del brazo. Se puede observar que aparece un nodo llamado rostopic_30526_1574012778481. Este nodo es la terminal de ROS que publica el

mensaje al *topic* pertinente (joint2_position/command). Los demás nodos y *topics* son ya conocidos.

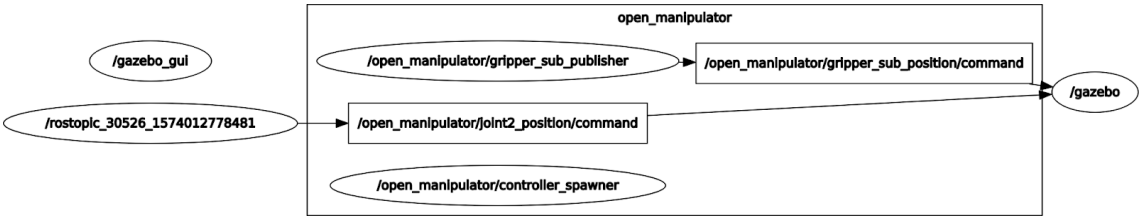

Fig. 39: Rosgraph de nodos y topics activos en el movimiento de la segunda articulación de OM.

Otra forma es a través de la herramienta rqt, por la cual se envían los mismos mensajes a través de los topics, pero en vez de publicarlo a través de la terminal, se publica a través de rqt (Fig. 40). Se modifica el valor de la variable, en este caso el campo *expression*. Cuando se tienen los valores deseados, se le da a publicar mensaje y, así, le llega el mensaje al brazo.

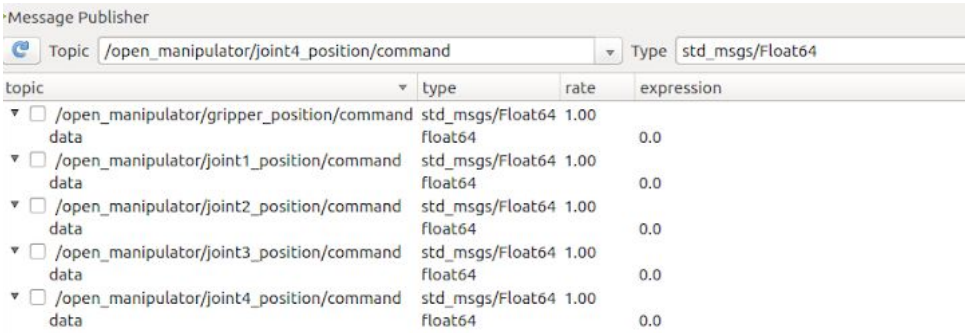

Fig. 40: Ventana de rqt con topics para publicar mensajes a través de ellos.

Para ello, se ejecuta rqt en una nueva terminal. En rqt, se sigue la ruta plugins>Topics>Message Publisher y se buscan los topics deseados en los que publicar cierto mensaje (Fig. 41).

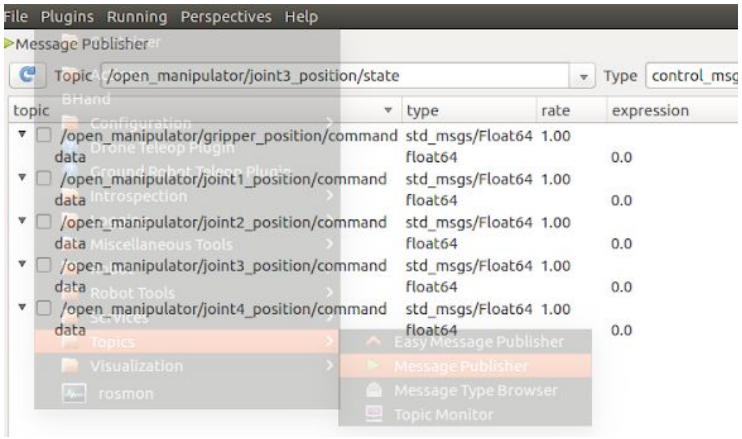

Fig. 41: Ventana de rqt escogiendo el plugin de Message Publisher.

Se cambian los valores deseados de la ventana de rqt y, seleccionando con el botón derecho del ratón el topic para publicar el mensaje, se le da a 'Publish Selected Once', para publicar el mensaje del topic seleccionado.

Se van a publicar diferentes valores para las articulaciones y la pinza y cambiar la posición del brazo robot. Se van a mostrar dos posiciones ejemplo con diferentes valores de la pinza y las articulaciones.

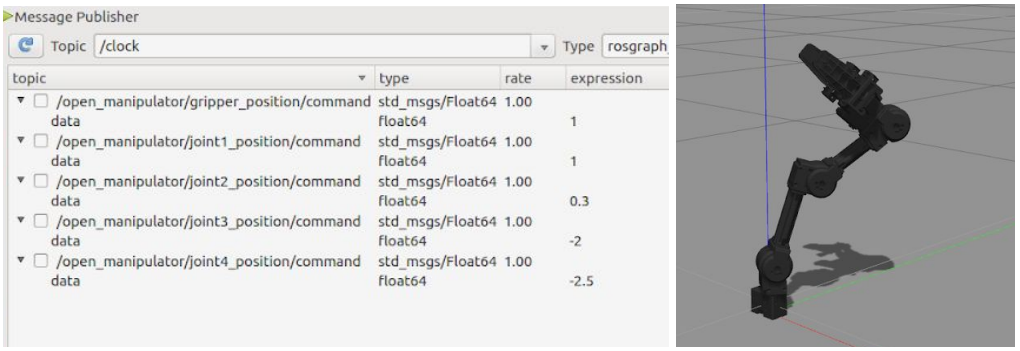

Fig. 42: Ejemplo de movimiento del brazo robot con rqt.

En ambos ejemplos (Fig. 42 y Fig. 43), en la parte izquierda de la imagen, se muestra rqt con los cinco topics implicados directamente en el movimiento del robot y

el valor que se le ha dado a cada uno. A la derecha se ve la posición del brazo robot que corresponde a esos valores.

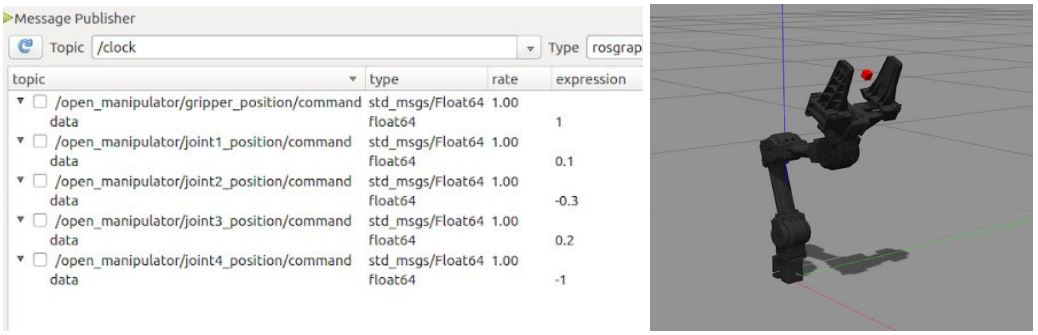

Fig. 43: Segundo ejemplo de movimiento del brazo robot con rqt.

Si miramos en la misma herramienta el rosgraph de nodos y topics activos (Fig. 44), comprobamos efectivamente que son todos los topics que se han usado o seleccionado en rqt, además de otros implicados en el movimiento, como controller_spawner (sin relacionar porque no se estaba efectuando ningún movimiento en el momento en el que se ha imprimido el gráfico), gripper_sub_publisher y gripper_sub_position/command (van unidos a gripper_position/command, ya que al mover la pinza se debe de mover esa parte de la pinza también) y gazebo, donde se tiene simulado el brazo.

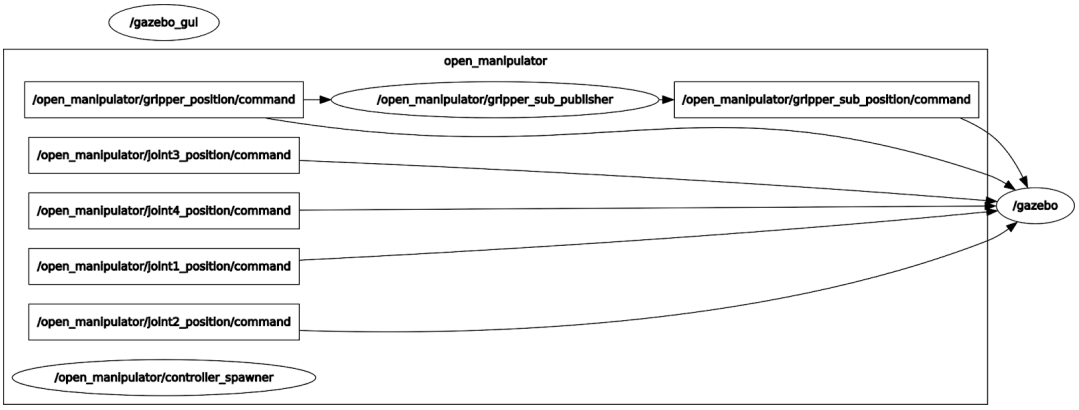

Fig. 44: Rosgraph de nodos y topics activos en el movimiento del brazo con rqt.

El brazo de OpenManipulator también se puede manejar telemanipuladamente con el *launch* llamado open_manipulator_teleop_keyboard. Teniendo el nodo *roscore* en proceso y Gazebo ejecutado, en una nueva terminal se escribe el comando:

roslaunch open_manipulator_controller open_manipulator_controller.launch use_platform:=false

Este comando consigue ejecutar el controlador para permitir el movimiento del brazo. Al final de la frase se le da valor *false* al parámetro use_platform ya que, como se ha dicho en este capítulo 4., queremos que los datos se transmitan a Gazebo y no al brazo real.

Tras tener ejecutado el controlador, ejecutamos el launch encargado de llamar al nodo implicado en la telemanipulación en una nueva terminal escribiendo el comando:

roslaunch open_manipulator_teleop open_manipulator_teleop_keyboard.launch

Gracias a este *launch*, se puede manejar el brazo de forma parecida a la que se mueve la base de forma manipulada (sección 3.2.). Se puede controlar el brazo con las diferentes teclas de un teclado (Fig. 45):

- w: incremento de 0.01 m de la posición del brazo en el eje x.
- s: decremento de 0.01 m de la posición del brazo en el eje x.
- a: incremento de 0.01 m de la posición del brazo en el eje y.
- d: decremento de 0.01 m de la posición del brazo en el eje y.
- z: incremento de 0.01 m de la posición del brazo en el eje z.
- x: decremento de 0.01 m de la posición del brazo en el eje z.
- y: incremento de 0.5 rad del ángulo de la articulación 1.
- h: decremento de 0.5 rad del ángulo de la articulación 1.
- u: incremento de 0.5 rad del ángulo de la articulación 2.
- j: decremento de 0.5 rad del ángulo de la articulación 2.

- i: incremento de 0.5 rad del ángulo de la articulación 3.
- k: decremento de 0.5 rad del ángulo de la articulación 3.
- o: incremento de 0.5 rad del ángulo de la articulación 4.
- l: decremento de 0.5 rad del ángulo de la articulación 4.
- g: abrir la pinza.
- f: cerrar la pinza.
- 1: posición inicial.
- 2: posición 'home'.
- q: salir de la telemanipulación.

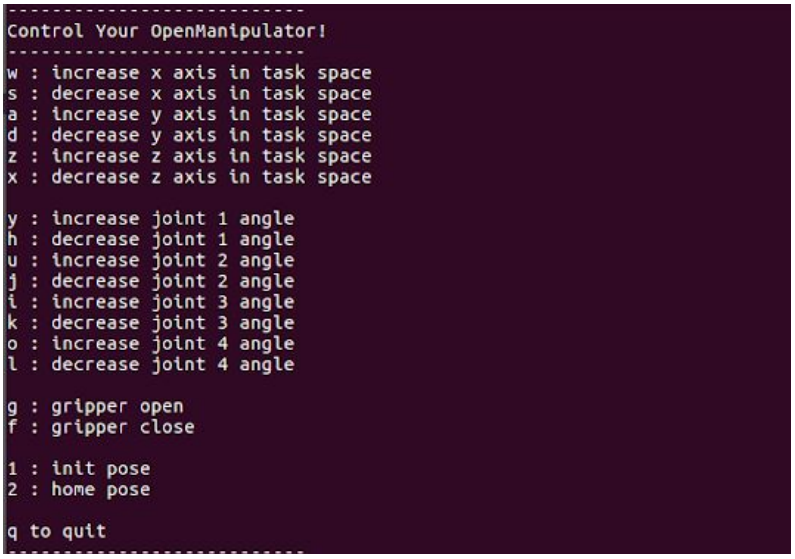

Fig. 45: Terminal con el mensaje de los controles al ejecutar la telemanipulación.

Si seleccionamos la tecla 1, el robot se pondrá automáticamente en su posición inicial (Fig. 46).

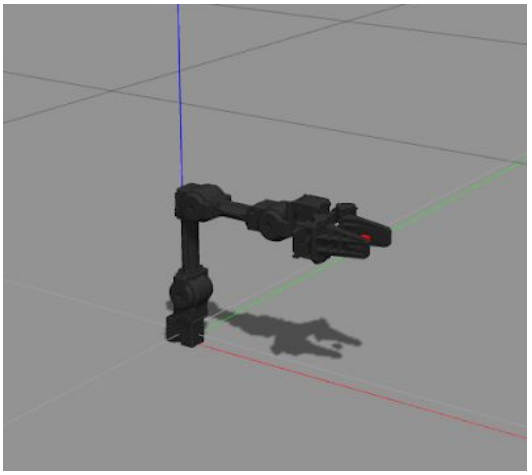

Fig. 46: Posición inicial del brazo robot de OM.

Si seleccionamos la tecla 2, el robot se pondrá automáticamente en su posición *home* (Fig. 47).

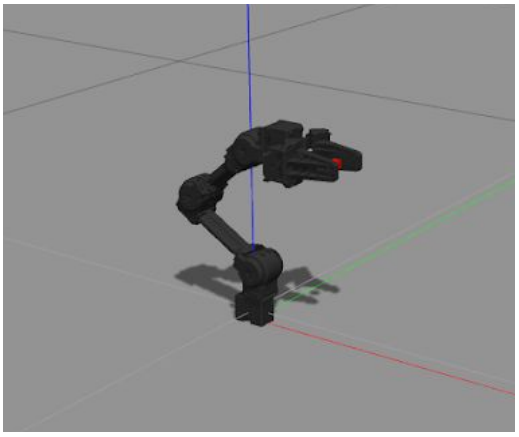

Fig. 47: Posición home del brazo robot OM.

Por otro lado, si se selecciona cualquier otra tecla del control, el brazo va a reaccionar al incremento o decremento de los ángulos de los brazos o de las distancias en *x*, *y* y *z* hasta que llegue a su límite. Además de poder abrir y cerrar la pinza con la g y la f respectivamente. Por ejemplo, en la siguiente imagen (Fig. 48) se muestra como se ha bajado el brazo robot por el eje z, solamente modifica la coordenada z en su posición manteniendo el valor de x e y, tomando como referencia la herramienta de OM (efector final o pinza).

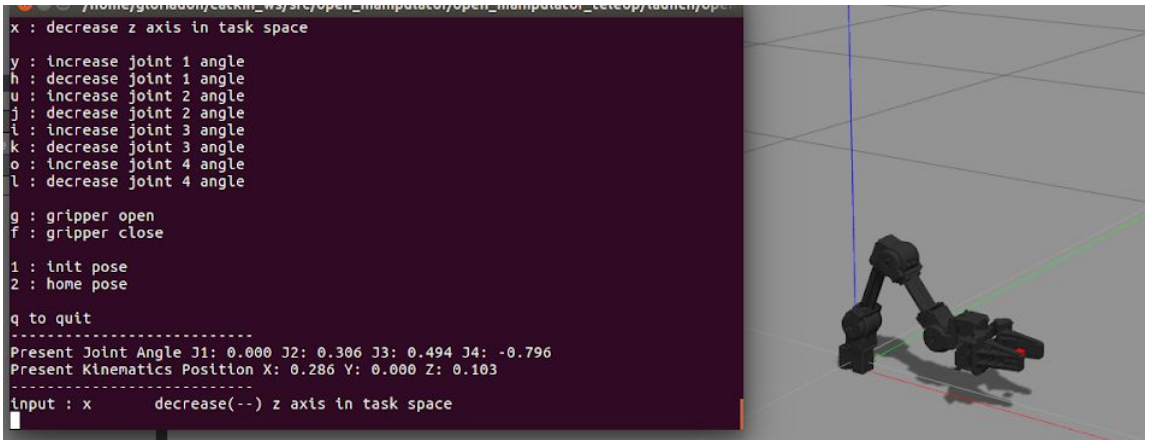

Fig. 48: Ejemplo de movimiento telemanipulado del brazo robot. Movimiento en el eje z.

Con el movimiento telemanipulado, los nodos y *topics* activos se muestran en el siguiente Rosgraph (Fig. 49).

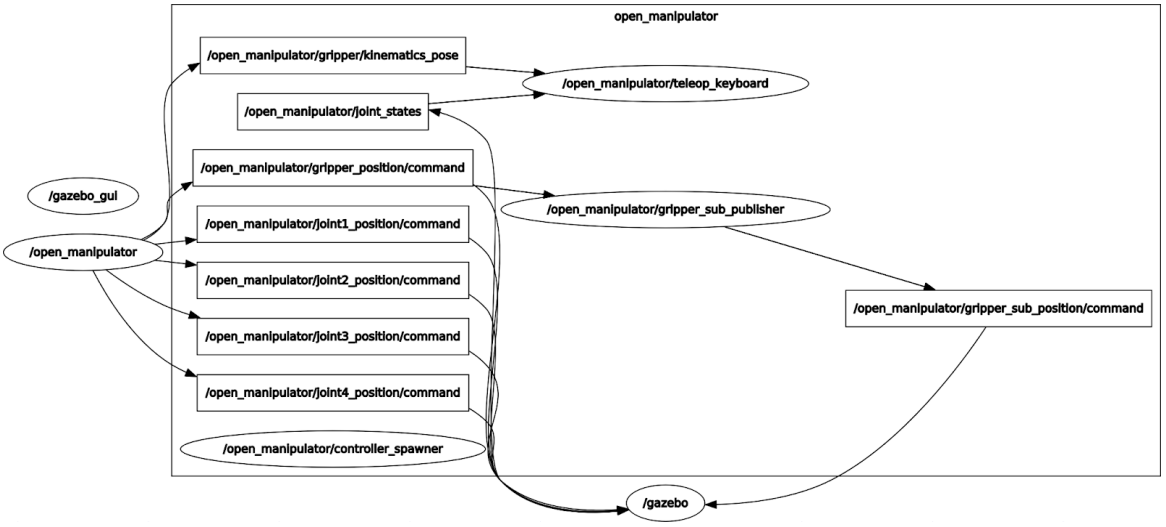

Fig. 49: Rosgraph del movimiento telemanipulado del brazo robot OM.

El nodo desconocido es el teleop_keyboard, es el encargado de llevar a cabo la aplicación de movimiento manipulado. Los demás nodos y todos los *topics* que aparecen en este *rosgraph* ya se han visto durante esta memoria.

El movimiento del brazo, como se ha visto durante este capítulo (4.), se puede llevar a cabo de múltiples formas. Depende de lo que se quiera, será mejor aplicar el movimiento de una forma u otra. En este proyecto, interesan los topics y nodos que están funcionando en cada una de ellas para, posteriormente, utilizarlos a favor del objetivo final para crear una aplicación.

5. OpenManipulator con TurtleBot 3

OpenManipulator es compatible con TurtleBot 3, por lo que se van a fusionar ambos para poder tener un brazo manipulador móvil, esto hace más extenso el campo de actividades que puede llevar a cabo el robot.

Para cada una de las aplicaciones, se va a tener el master en proceso, como se ha comentado constantemente en el capítulo 3. de esta memoria y cada proceso o *launch* necesario para el desarrollo de cada aplicación o funcionalidad, se ejecutará en una nueva terminal.

5.1. Simulación en Gazebo

Al igual que en la sección 3.1., primero se tiene que simular en Gazebo el modelo del robot en un entorno, en este caso, los archivos que ejecutan Gazebo con el modelo robot posicionado en un entorno son:

- empty_world.launch (Fig. 50): posiciona el robot de OM con TB3 en un entorno vacío en simulación en Gazebo.

- rooms.launch (Fig. 51): posiciona el robot de OM con TB3 en un entorno con varias habitaciones en simulación en Gazebo.

Antes de ejecutar los *launch* que ejecutan Gazebo, se debe de seleccionar el modelo de la base móvil por el comando de ROS export, en una nueva terminal se escribe la línea de comando:

```
export TURTLEBOT3_MODEL=waffle
```

El modelo burger de la base no está disponible con el brazo de OpenManipulator en estos paquetes de ROS, por lo que se puede elegir solamente entre los modelos waffle y waffle PI.

Se ejecuta Gazebo en la misma terminal en la que se ha elegido el modelo de la base móvil:

- Para el entorno vacío se escribe la línea de comando:

roslaunch open_manipulator_with_tb3_gazebo empty_world.launch

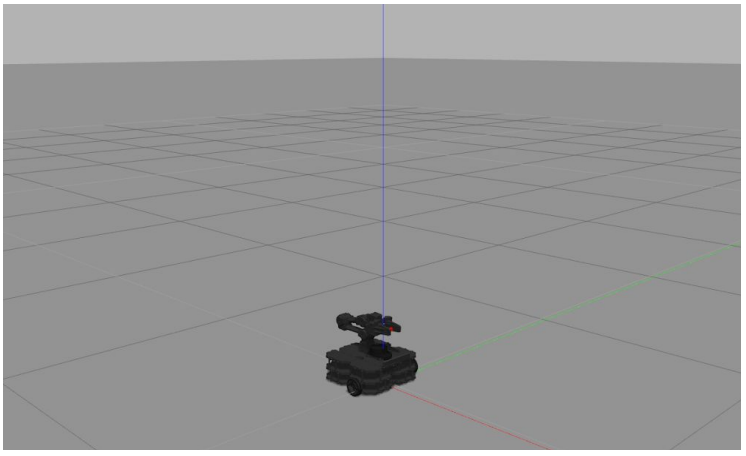

Fig. 50: Modelo Waffle en el entorno vacío.

- Para el entorno de diferentes habitáculos se escribe la línea de comando:

roslaunch open_manipulator_with_tb3_gazebo rooms.launch

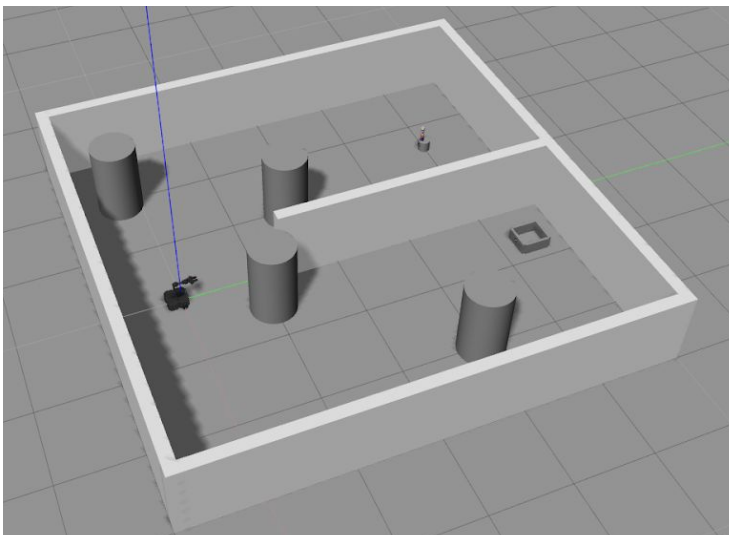

Fig. 51: Modelo Waffle en el entorno rooms.

Estos *launch*, además de ejecutar Gazebo, el modelo robot (URDF) y el entorno (world), ejecutan otros nodos y algunos archivos de configuración general. Ejecutan el archivo gazebo_controller.yaml, donde se configuran las ganancias del control PID (parámetros proporcional, integral, y derivativo) de las articulaciones del brazo. Se ejecuta el nodo robot_state_publisher y el ejecutable joint_controller.launch. Este último ejecuta el archivo de configuración joint_controller.yaml y los nodos controller_spawner y gripper_sub_publisher. Todos ellos para la configuración del brazo de OpenManipulator.

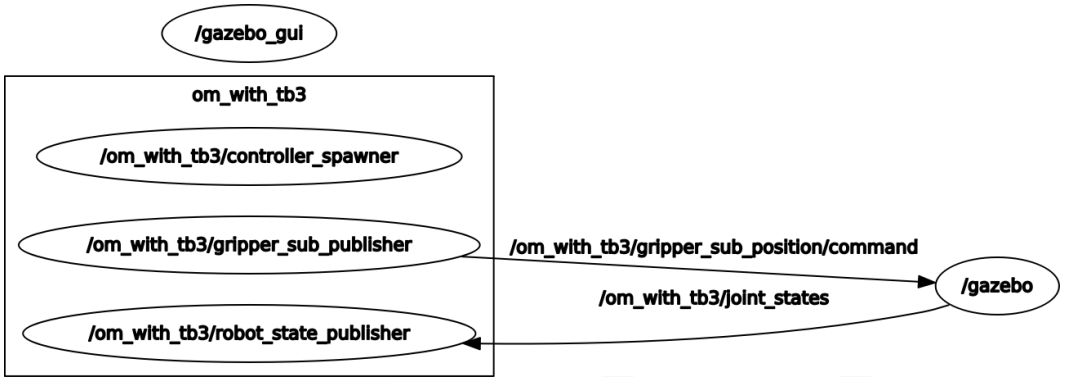

Fig. 52: Rosgraph de nodos y topics de empty_world.launch y rooms.launch.

En la imagen (Fig. 52) se muestran los nodos y *topics* activos cuando se ejecutan estos *launch*. Se puede observar que están activos todos aquellos nodos que ejecutan los propios *launch* (gazebo, controller_spawner, gripper_sub_publisher y robot_state_publisher) y se comunican a través de los *topics* joint_states y gripper_sub_position/command. Los nodos y los *topics*, exceptuando gazebo, están referenciados al robot, en este caso, om_with_tb3.

Todos estos nodos y *topics* ya se han visto en el capítulo anterior, capítulo 4., en el que se ha explicado qué es y cómo funciona el brazo robot de OpenManipulator.

5.2. Movimiento

El movimiento del conjunto robot, formado por el brazo manipulador sobre la base móvil, se lleva a cabo moviendo la base por un lado y el brazo por otro lado. Ambos movimientos se llevan a cabo de igual manera que en la sección 3.2. para la base móvil de TB3 y que en el capítulo 4. para el brazo de OM. Sumando algunas restricciones o pequeños cambios, el cambio más notable a tener en cuenta es el cambio del nombre del robot.

Primero, se va a repasar el movimiento de la base. Con *roscore* en proceso y Gazebo ejecutado, es decir, el modelo robot simulado en un entorno para tener los datos en tiempo real, se va a ejecutar la funcionalidad de telemanipulación en una nueva terminal. En la terminal se escriben los comandos:

```
export TURTLEBOT3_MODEL=waffle
```

```
ROS_NAMESPACE=om_with_tb3 roslaunch turtlebot3_teleop
turtlebot3_teleop_key.launch
```

Para aplicar el movimiento a la base del robot om_with_tb3 se debe de añadir el comando ROS_NAMESPACE para elegir los *topics* adecuados a los que suscribirse o en los que publicar información. Tras elegir el nombre del robot al que se va a manipular, se ejecuta el *launch* indicado para llevar a cabo la funcionalidad, como siempre, indicando el directorio en el que se encuentra y el nombre del archivo.

En el caso de este robot, no se puede llevar a cabo la simulación que ofrece ROS, ya que el nodo que la lleva a cabo publica en el topic cmd_vel y, ahora, el *topic* que lleva la información de las velocidades de este robot es om_with_tb3/cmd_vel. Por lo que habría que hacer modificaciones en el código para que publicara correctamente en los *topics* deseados.

En segundo lugar, se va a repasar el movimiento del brazo. Al igual que con la base, se tiene que tener *roscore* en proceso y Gazebo ejecutado. Hay varias formas en las que llevar a cabo el movimiento del brazo robot, todas vistas en el capítulo 4.. En este apartado, solo se van a repasar dos: se va a mover el brazo a través de comandos de ROS y a través de MoveIt! (con el MPP en RViz). Las formas de llevar a cabo el movimiento del brazo no difieren demasiado de la forma en la que se lleva a cabo en el capítulo 4., tan solo cambia alguna variable, como el nombre del robot.

El movimiento del brazo a través de comandos ROS consiste en publicar en los *topics* implicados en registrar los valores de las diferentes articulaciones del brazo robot y de la pinza. Los *topics* implicados son:

- om_with_tb3/joint1_position/command
- om_with_tb3/joint2_position/command
- om_with_tb3/joint3_position/command
- om_with_tb3/joint4_position/command
- om_with_tb3/gripper_position/command

La línea de comando sería igual a la utilizada en el capítulo 4. cambiando el nombre del *topic* (teniendo en cuenta que va asociado con el nombre del robot) y el valor, que cada vez es variable según la posición en radianes deseada para cada articulación:

```
rostopic pub /om_with_tb3/joint2_position/command std_msgs/Float64 "data:
-0.21" --once
```

Si se tiene la base móvil en movimiento por telemanipulación y, a su vez, se mueve el brazo a través de la terminal con un comandos ROS, los nodos y *topics* activos se ven en el *rosgraph* siguiente (Fig. 53).

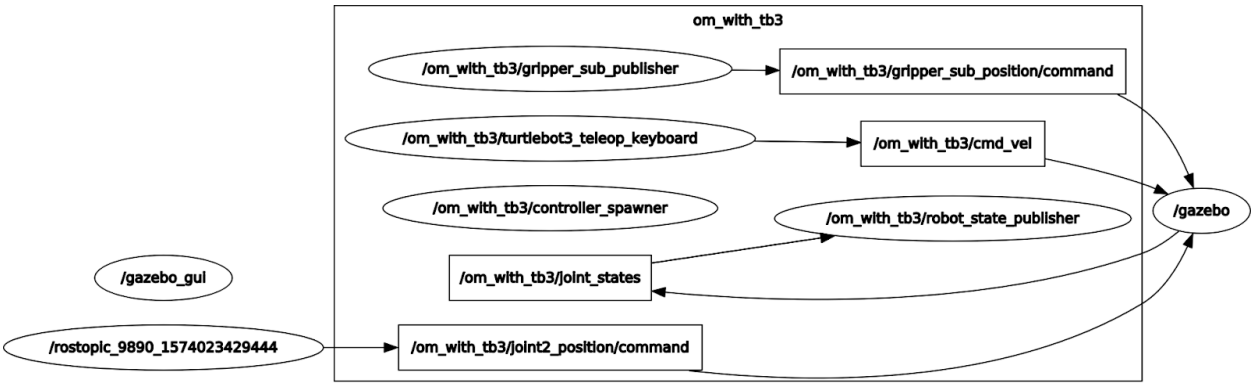

Fig. 53: Rosgraph de nodos y topics activos en el movimiento de la base y el brazo robot.

En la imagen (Fig. 53) aparece el nodo turtlebot3_teleop_keyboard, que se ocupa de manejar la base. También aparece el nodo n_rostopic_9890_1574023429444, encargado de informar del movimiento de la articulación 2 del brazo. Esta información la ha mandado a Gazebo a través del topic om_with_tb3/joint2_position/command. Además de estos dos nodos principales para la realización de ambos movimientos, aparecen los mismos nodos que los de el *rosgraph* del movimiento por comandos ROS del brazo (Fig. 39) junto con los del *rosgraph* del movimiento de la base telemanipuladamente (Fig. 15). La única diferencia es el nombre del robot, en este caso, om_with_tb3 en vez de turtlebot3 para el caso de la base y om_with_tb3 en vez de

open_manipulator para el caso del brazo. Se añaden el nodo robot_state_publisher y el *topic* om_with_tb3/joint_states, ambos ya vistos también en el capitulo 4..

Se va a dar un valor diferente en cada topic para mover cada articulación del brazo. El resultado de cada uno viene en las imágenes que siguen. La figura 54 muestra el cambio de la primera articulación de valor 0 a valor 1.

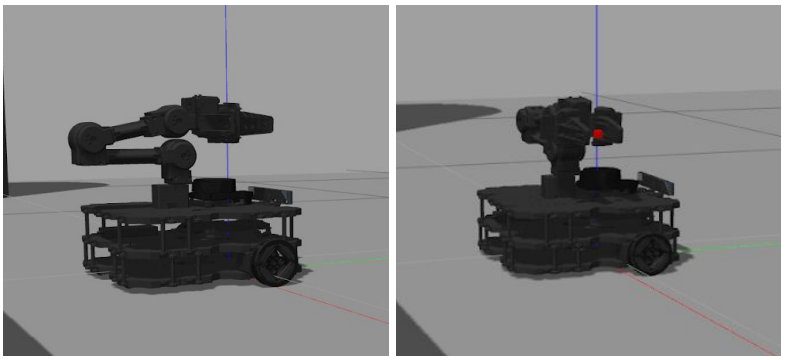

Fig. 54: Articulación 1: Posición inicial (izq.) y posición después del movimiento (dcha.)

La figura 55 muestra el movimiento de la articulación dos del brazo del robot.

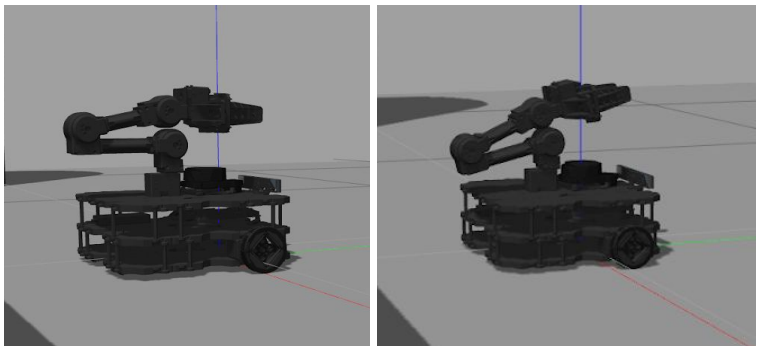

Fig. 55: Articulación 2: Posición inicial (izq.) y posición después del movimiento (dcha.)

La figura 56 muestra el movimiento de la articulación 3. Se le ha dado un valor negativo en la imagen central y un valor positivo en la imagen de la derecha. Ya que el valor cero para esta articulación es cuando se posiciona perpendicular a la parte del brazo que une la articulación dos y la tres.

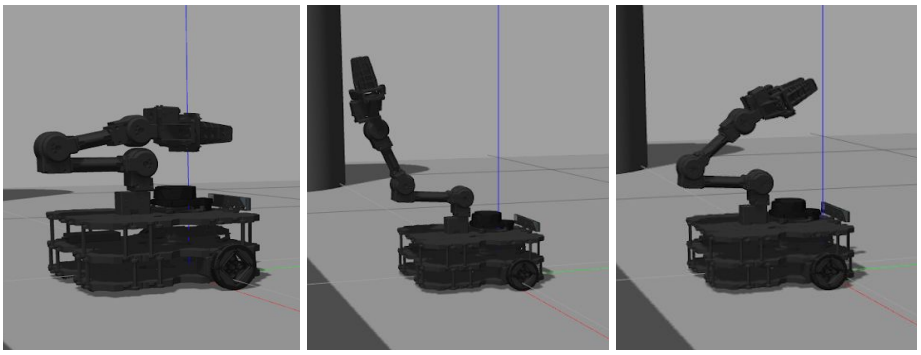

Fig. 56: Movimiento de la articulación 3: Posición inicial (izq.) y posiciones después del movimiento.

La figura 57 muestra el cambio en la cuarta articulación, la que maneja el ángulo de la pinza. Se le ha asignado un valor negativo a la articulación, el valor cero de la cuarta articulación sigue la dirección del brazo que une la tercera y la cuarta articulación.

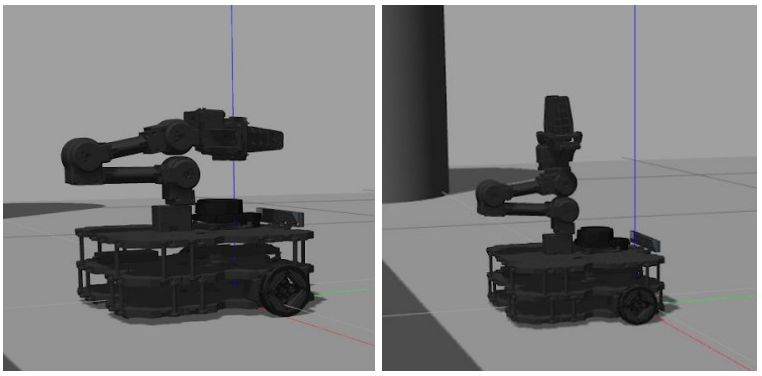

Fig. 57: Articulación 4: Posición inicial (izq.) y posición después del movimiento (dcha.)

Por último, se muestra la puesta a cero del robot. Si todas las articulaciones tienen un valor cero, el brazo tiene la posición de la imagen (Fig. 58).

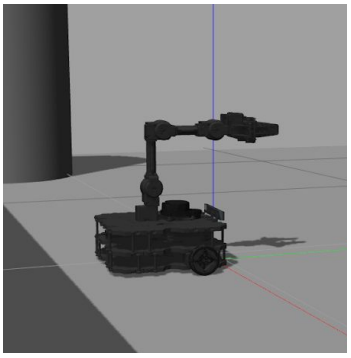

Fig. 58: Posición del brazo con las articulaciones a cero.

Por otro lado, se va a ver el movimiento del brazo a través de MoveIt!. Se va a repasar la forma de ejecutar MoveIt! y se van a ver diferentes planificaciones de trayectorias. Trayectorias de una posición inicial a otra posición objetivo.

Con *roscore* en proceso y Gazebo ejecutado, en una nueva terminal se va a ejecutar MoveIt!. En la terminal se elige primero el modelo de la base, se escribe en la terminal la línea de comando:

```
export TURTLEBOT3_MODEL=waffle
```

En la misma terminal en la que se ha exportado el nombre del modelo de la base móvil se ejecuta el *launch* indicado para llevar a cabo la funcionalidad, como siempre, indicando el directorio en el que se encuentra y el nombre del archivo:

```
roslaunch open_manipulator_with_tb3_tools manipulation.launch
use_platform:=false
```

Al parámetro use_platform se le asigna el valor *false* para que no mande los mensajes al brazo real, sino que los mande a Gazebo.

Tras la aplicación en proceso, podemos ver los nodos y *topics* activos en la misma (Fig. 59).

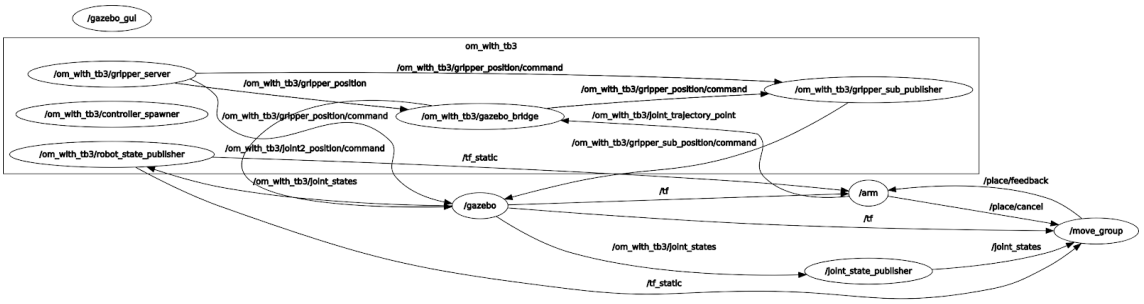

Fig. 59: Rosgraph de los nodos y topics al ejecutar Gazebo y MoveIt! en RViz.

Como se ve, es un rosgraph mucho más complejo que el del brazo de OpenManipulator (Fig. 38). Aparecen los nodos gripper_server, gazebo_bridge y arm. También, aparecen los *topics* place/cancel y place/feedback. El nodo gripper_server se ocupa de la pinza. El nodo gazebo_bridge es el nodo intermediario entre MoveIt y Gazebo, se ocupa de transformar los datos recogidos de algunos *topics* para publicarlos en otros topics a los que MoveIt tiene acceso o utiliza. El nodo arm y los *topics* place/cancel y place/feedback son propios del brazo de OpenManipulator con MoveIt!

En este apartado, se van a ver varios ejemplos de trayectorias seguidas por el brazo del robot, calculadas a partir de MPP de MoveIt!.

Con MPP se puede seleccionar una posición objetivo aleatoria (Fig. 60). La cual, queda marcada en color naranja en RViz. Además, en gris aparece la posición anterior o inicial. Una vez se tienen marcadas las posiciones inicial y final, se puede planificar una trayectoria desde la posición inicial hasta la posición cargada final (Fig. 61).

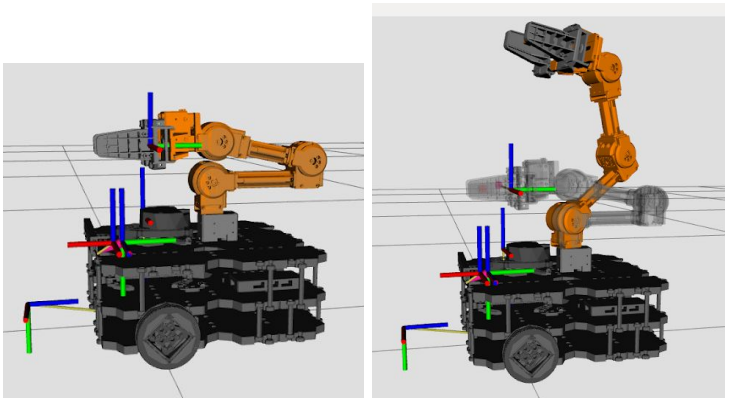

Fig. 60: Posición inicial (izq.) y posición final (dcha) en RViz.

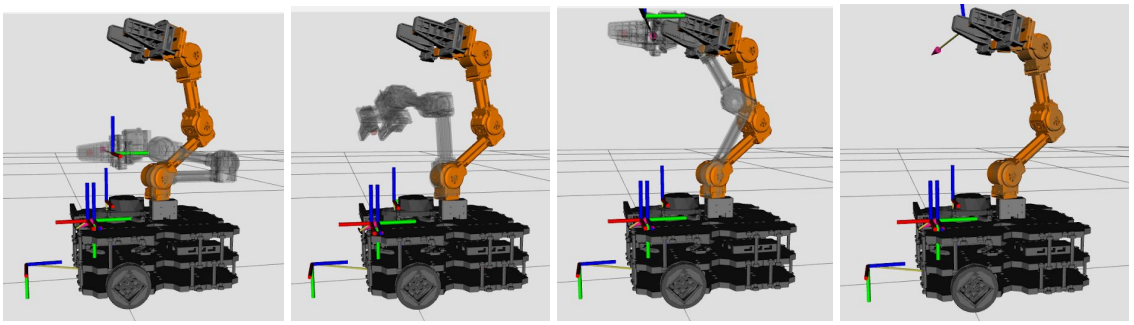

Fig. 61: Plan de ejecución de una trayectoria en RViz

Una vez se ha planificado la trayectoria con éxito en RViz con Motion Planning Plugin se puede ejecutar. En el caso de este proyecto, en Gazebo se mueve el brazo siguiendo la trayectoria planificada. Si no fuera por simulación, el brazo real seguiría la trayectoria planificada.

Un ejemplo de trayectorias ejecutadas en Gazebo son las imágenes siguientes (Fig. 62, Fig. 63 y Fig. 64). Estas tres trayectorias parten de una posición inicial y, tras planificar y ejecutar todas, el brazo vuelve a una posición parecida a la inicial. Es decir, la posición final de la última trayectoria es la posición inicial de la primera.

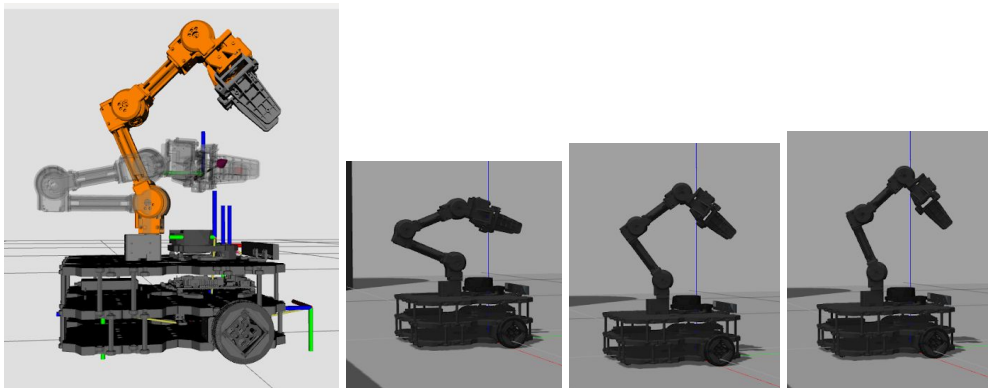

Fig. 62: Posición inicial y final (izq.) y ejecución en Gazebo de la primera trayectoria planificada.

En esta imagen (Fig. 62) se observa como el brazo parte de una posición recogida y termina en la posición indicada en RViz como posición final.

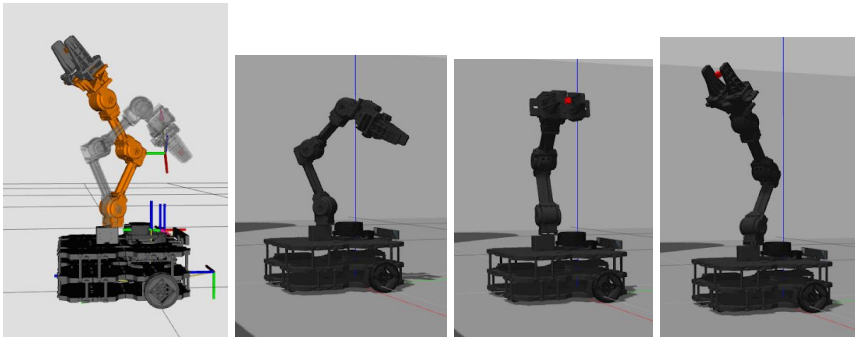

Fig. 63: Posición inicial y final (izq.) y ejecución en Gazebo de la segunda trayectoria planificada.

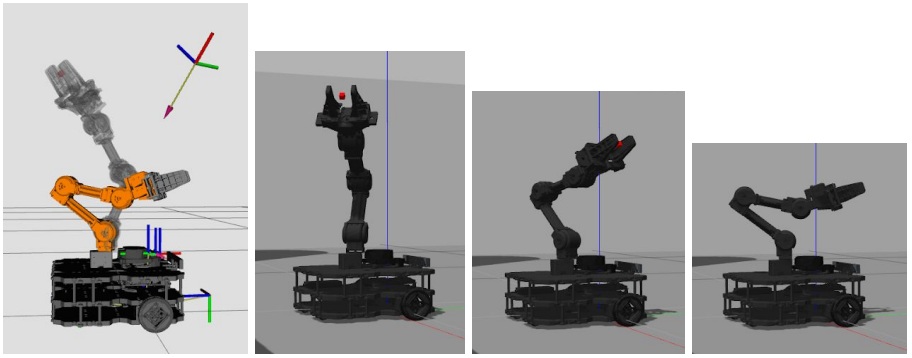

Fig. 64: Posición inicial y final (izq.) y ejecución en Gazebo de la tercera trayectoria planificada.

En estas últimas imágenes (Fig. 63 y Fig. 64) se ve la trayectoria que sigue en Gazebo el robot (las tres imágenes de la derecha) comenzando por la posición inicial (en gris en RViz) hasta la posición final indicada (en naranja en RViz).

Una vez visto el movimiento de la base y el brazo juntos, se van a repasar las funcionalidades de SLAM y Navegación con este robot constituido por una base móvil y un brazo robótico.

5.3. SLAM

Se va a hacer SLAM con el brazo montado en la base móvil de TB3. Ya se ha explicado en qué consiste la funcionalidad de SLAM en la sección 3.3. SLAM.

Se va a abrir una terminal para el proceso de *master* y otra terminal con Gazebo simulando el robot en el entorno. Para esta funcionalidad se va a ejecutar el archivo rooms.launch para abrir Gazebo y posicionar el robot en el entorno de las habitaciones, el mismo entorno ejecutado en la sección 3.1..

En otra terminal, se ejecuta el *launch* que realiza el mapeo. En la terminal se escriben las líneas de comando:

export TURTLEBOT3_MODEL=waffle

roslaunch open_manipulator_with_tb3_tools slam.launch use_platform:=false

De nuevo, se elige la base del robot en la misma terminal que se va a ejecutar el *launch* pertinente. También se le da valor *false* al parámetro use_platform, porque queremos hacerlo por simulación. Tal y como se ha hecho en el capítulo 4..

Se abre la herramienta de RViz mostrando la parte del mapa que ve o recibe por los sensores láser el robot (Fig. 65).

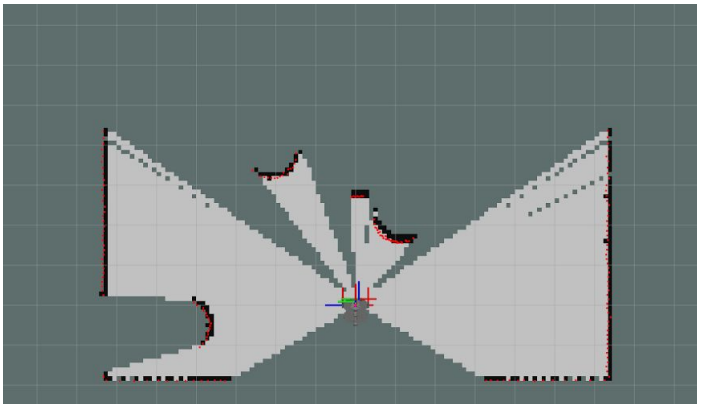

Fig. 65: RViz con el launch de SLAM ejecutado.

Para esta tarea, no es necesario el movimiento del brazo, por lo que el robot se moverá por el entorno telemanipuladamente como se ha mostrado en el apartado 3.2., en una nueva terminal se escriben las líneas de comando:

export TURTLEBOT3_MODEL=waffle

ROS_NAMESPACE=om_with_tb3 roslaunch turtlebot3_teleop
turtlebot3_teleop_key.launch

Escribiendo y ejecutando cada una de estas líneas en la terminal, se selecciona el modelo de la base del robot y se aplica la telemanipulación al robot. El cual, está simulado en Gazebo y se llama, en este caso, om_with_tb3.

Una vez se ha recorrido el espacio completamente, se guarda el mapa con map_saver gracias al paquete de map_server. En una nueva terminal se escribe la línea de comando:

```
ROS_NAMESPACE=om_with_tb3 rosrun map_server map_saver -f ~/NombreMapa
```

En la parte final de la línea de comando se indica la ruta y el nombre del mapa, los cuales, podemos modificar en la propia línea para llamarlo como se quiera y guardarlo en la carpeta deseada.

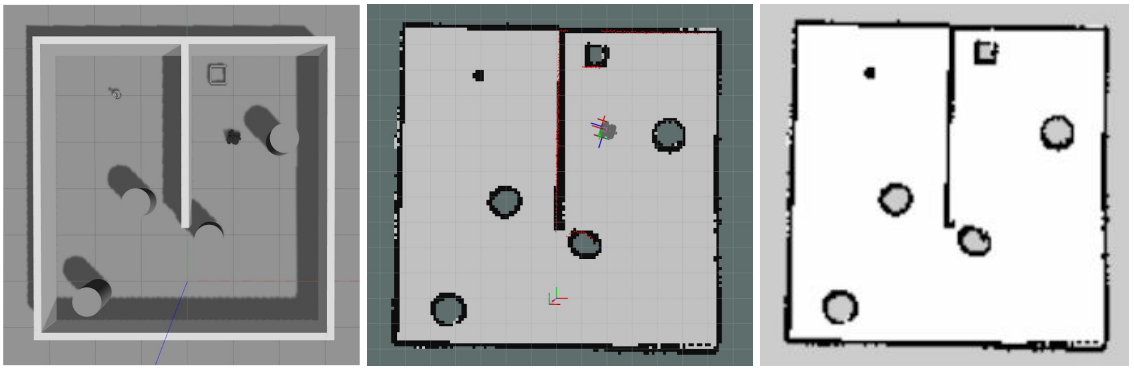

Fig. 66: Comparación del entorno en Gazebo (izq.), mapa en RViz (centro) y la imagen del mapa (dcha.).

En las imágenes anteriores (Fig. 66) se ve el entorno simulado en Gazebo (imagen izquierda), el entorno recorrido por el robot en RViz haciendo mapeo (imagen central) y el resultado del mapa tras guardarlo en formato png (imagen derecha).

Por otro lado, los nodos y *topics* activos se ven representados en el siguiente gráfico *rosgraph* (Fig. 67).

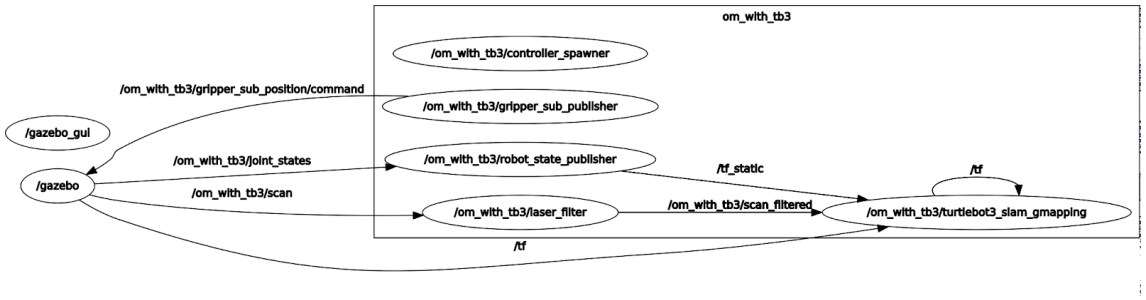

Fig. 67: Rosgraph de los nodos y topics activos durante la navegación.

ROS nos proporciona un diagrama con las comunicaciones activas entre nodos. En este caso, dentro del grupo om_with_tb3, el robot utilizado, existen varias de estas comunicaciones.

Se observa que el nodo om_with_tb3/controller_spawner no se comunica con nada, es porque durante el mapeo solo se utiliza activamente la base móvil, ya que el brazo no se mueve.

Todos los nodos y *topics* activos en esta funcionalidad ya se han visto en los capítulos 3 y 4.

5.4. Navegación

En la sección 3.4. ya se ha visto el concepto de la funcionalidad de navegación. Esta funcionalidad se lleva a cabo con el conjunto de brazo y base móvil de la misma forma que con la base móvil de TB3.

Tras tener en proceso *roscore* y Gazebo, en una nueva terminal se va a ejecutar el *launch* que permite realizar esta funcionalidad. Primero, en la nueva terminal se elige

el modelo de la base y, luego, se ejecuta el *launch*. Esto se pone en marcha a través de las líneas de comando que siguen:

```
export TURTLEBOT3_MODEL=waffle
```

```
roslaunch open_manipulator_with_tb3_tools navigation.launch use_platform:=false
map_file:=$HOME/NombreMapa.yaml
```

El parámetro use_platform sigue siendo falso, ya que se realiza por simulación. Y el parámetro map_file se puede utilizar para elegir el directorio en el que se encuentra el mapa por el que navegar y su configuración yaml, una de las configuraciones que carga el *launch* de navegación.

Una vez ejecutada la funcionalidad, de la misma forma que en el apartado 3.4, se indica a través de la herramienta de RViz la posición objetivo, entonces, el robot calcula la trayectoria y va hasta allí sin chocar con nada.

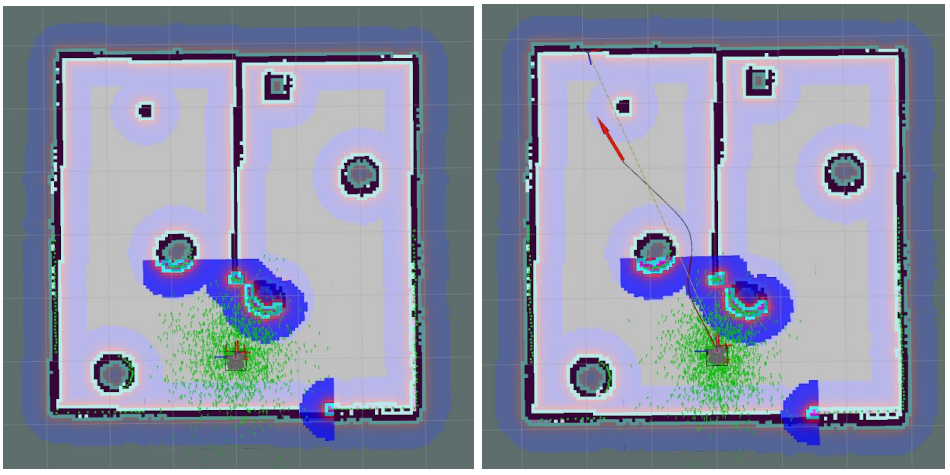

Fig. 68: RViz nada más ejecutarlo (izq.) y RViz con la posición objetivo marcada (dcha.).

En la imagen anterior (Fig. 68), se puede observar la trayectoria calculada hasta la posición final indicada con la flecha roja (imagen de la derecha).

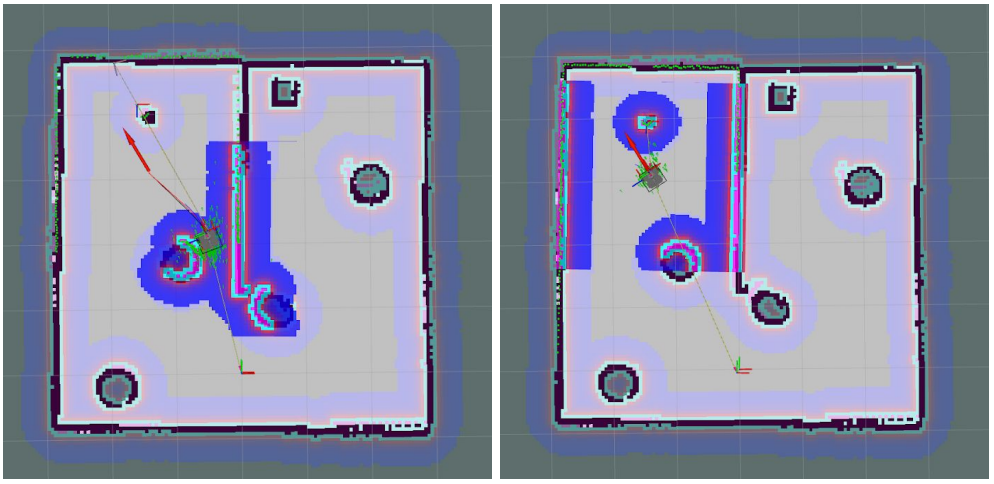

Fig. 69: Posiciones intermedias del robot siguiendo la trayectoria hasta la posición objetivo.

En la figura 69 se puede observar el recorrido del robot por la trayectoria y los bordes de las paredes y los objetos que tiene en cuenta (zonas azules).

Además, el *rosgraph* de esta funcionalidad con este robot (Fig. 70) difiere algo del *rosgraph* de la navegación con sólo la base de TB3(Fig. 22). Siendo el primero más complejo, ya que no solo lleva una base, sino que es una base y un brazo al mismo tiempo.

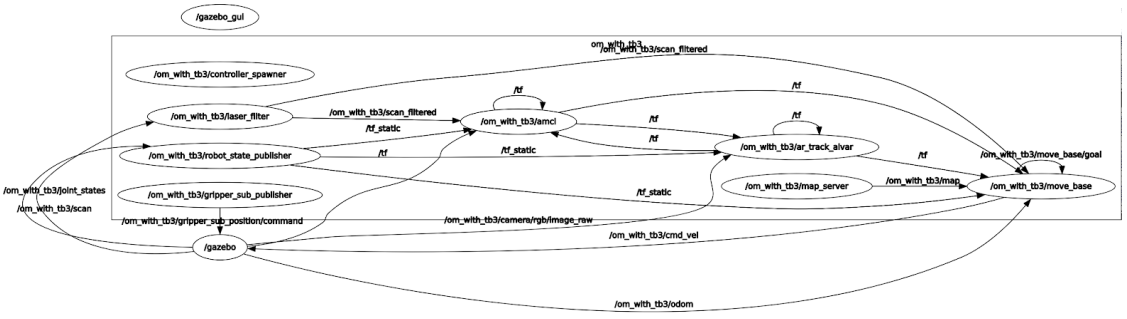

Fig. 70: Rosgraph de nodos y topics activos en la navegación de OM con TB3.

Como se ha visto, se pueden realizar todas las funcionalidades con el robot con la base y el brazo de una forma muy parecida a la realización de las mismas funcionalidades solo con la base o solo con el brazo.

6. Diseño e implementación de las aplicaciones robóticas

En este apartado se va a hablar de las funcionalidades durante este trabajo. En particular, se presentan diversas funcionalidades para completar el tutorial de ROS de TB3 y OM y una aplicación para realizar la tarea objetivo combinando aspectos multirobot y de manipulación.

6.1. Extensión de funcionalidades para TurtleBot 3

En este subapartado se van a explicar algunas de las aplicaciones que considero relevantes para completar el tutorial, para que haya un desarrollo completo que recoja todas las funcionalidades y características vistas a lo largo de esta memoria.

6.1.1. Telemanipulación en entorno multirobot

En el caso de la telemanipulación, es muy tedioso tener varias terminales abiertas y tener que ejecutar el nodo teleop_key en una terminal diferente para cada robot en el espacio multirobot. Por lo que se ha creado un nuevo nodo que lleva el control de los tres robots en el entorno multirobot que nos ofrece ROS del que se ha hablado en la sección 3.5..

Este nodo permite manejar los tres robots desde el teclado. Su función consiste en publicar con diferentes teclas las velocidades lineal y angular de cada robot en los *topics* propicios para hacer llegar esa información a Gazebo. Este nodo implementado, también tiene la posibilidad de mover los tres robots simultáneamente. Para utilizar esta funcionalidad es necesario tener en proceso el *master* y el robot posicionado en un entorno simulados en Gazebo de la misma forma que se ha realizado en la sección 3.5.. En una nueva terminal se escribe el comando:

rosrun turtlebot3_teleop teleoperacion.py

Una vez en proceso el nodo creado para esta aplicación, teleoperación.py, se puede comenzar a manipular las tres bases robot con los diferentes controles (Fig. 71).

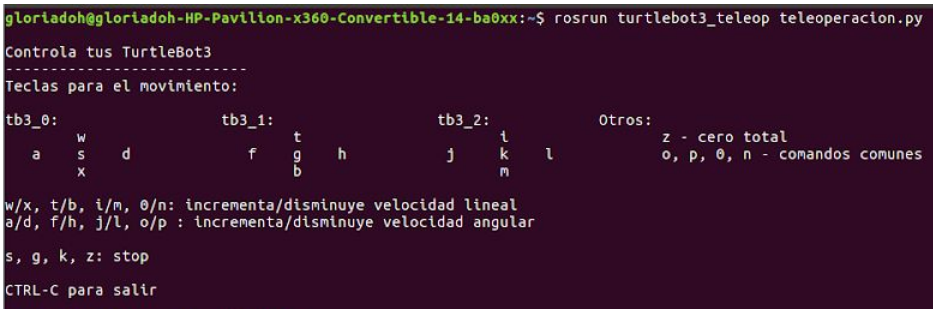

Fig. 71: Terminal con los controles para la telemanipulación multirobot.

Como se ve en la imagen (Fig. 71), para manejar la base con el nombre tb3_0 se usarán las teclas: a, w, s, d y x. Para la base llamada tb3_1 se reservan las teclas: t, f, g, h y b. Mientras que, para la última base (tb3_2), se utilizan las teclas: i, j, k, l y m. Además, las teclas z, o, p, 0 y n mueven los tres robots simultáneamente cambiando los valores de las velocidades lineal y angular al mismo tiempo. En el mensaje de la terminal (Fig. 71) se explica la función de cada tecla: las teclas dedicadas a la puesta de las velocidades a cero (stop) y al incremento y el decremento de las mismas.

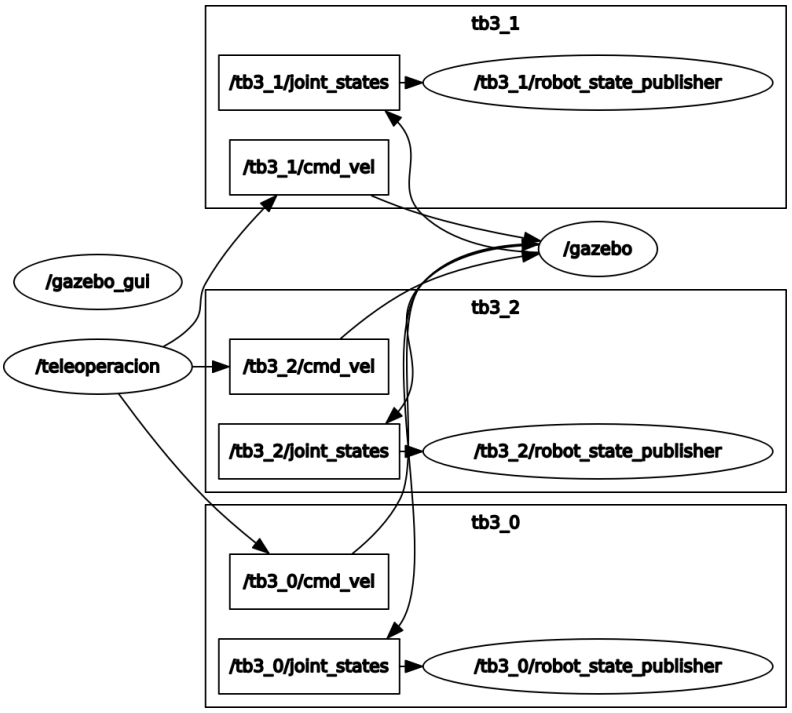

Fig. 72: Rosgraph de nodos y topics activos en el movimiento multirobot con telemanipulación.

En el *rosgraph* (Fig. 72) se puede observar en el lado izquierdo de la figura que, esta vez, solo hay un nodo para el movimiento de los robots llamado teleoperación, a diferencia que en el rosgraph multirobot en el que aparece un nodo de telemanipulación para cada grupo o robot (Fig. 32). Además, es muy parecido al rosgraph del movimiento telemanipulado de una sola base (Fig. 15). Simplemente se añade el conjunto del nodo y topic que se encarga de controlar el estado del brazo y de sus articulaciones. Pero la telemanipulación sigue el mismo proceso, coge datos de teclado y se los envía a través de los *topics* cmd_vel, asignados a cada base móvil, al robot simulado en Gazebo. Esta funcionalidad hace más cómoda la tarea de manejar los diferentes robots en un espacio multirobot y puede ser una herramienta más útil y práctica para que el usuario pueda interactuar más fácilmente con el entorno multirobot.

6.1.2. Movimiento autónomo en entorno multirobot

El movimiento autónomo disponible en los paquetes de ROS está implementado para un solo robot, pero no está preparado para el entorno multirobot. Sería mucho más cómodo poder recorrer un mapa, de la forma que se ha visto en la sección 3.3., con tres robots moviéndose libremente por el entorno sin tener que ir controlando los tres por separado telemanipuladamente como ocurre en la sección 3.5..

Como se ha visto en la sección 3.2., la funcionalidad de movimiento autónomo por parte del robot, se lleva a cabo gracias al nodo turtlebot3_drive. En este nodo, se encuentra la publicación en el topic cmd_vel, el cual, es válido para cuando hay un solo robot porque no se necesita referenciar a ningún robot, ya que hay una sola base y es inequívoco. El mencionado, es el nombre por defecto con el que se llama al *topic* que maneja los datos de velocidad lineal y angular cuando solo existe una sola base en funcionamiento. Sin embargo, en caso de tener varias bases, el *topic* encargado de los datos de velocidad, está referenciado a cada robot, por lo que el nombre del *topic* pasa a nombreRobot/cmd_vel. Siendo "nombreRobot", el nombre correspondiente a cada uno.

Ocurre algo similar con los *topics* del que es suscriptor el nodo turtlebot3_drive, scan y odom. Por lo que en el caso de multirobot en vez de ser cmd_vel, scan y odom los tres *topics* en proceso para cada robot serían:

- tb3_0/cmd_vel: *topic* con los datos de las velocidades para el primer robot.
- tb3_0/scan: *topic* con los datos del escáner para el primer robot.
- tb3_0/odom: *topic* con los datos de posición de la base para el primer robot.
- tb3_1/cmd_vel: *topic* con los datos de las velocidades para el segundo robot.
- tb3_1/scan: *topic* con los datos del escáner para el segundo robot.
- tb3_1/odom: *topic* con los datos de posición de la base para el segundo robot.
- tb3_2/cmd_vel: *topic* con los datos de las velocidades para el tercer robot.
- tb3_2/scan: *topic* con los datos del escáner para el tercer robot.

- tb3_2/odom: *topic* con los datos de posición de la base para el tercer robot.

Para realizar esta tarea, se pueden llevar a cabo dos soluciones: la primera consta en crear tres nodos que sean suscriptores y editores de cada uno de los *topics* adecuados para cada robot, así se ejecutará un nodo para cada robot en el mismo *launch*. La segunda consta de pasar por parámetro los nombres de los *topics* en el *launch* y, en el mismo launch, añadir tres veces el mismo nodo, uno por robot, parametrizado de forma diferente.

Si se sigue la primera opción, en el *launch* que ejecuta el nodo turtlebot3_drive, turtlebot3_simulation.launch, se va a ejecutar el nodo indicado para cada robot. En ese nodo indicado, el nombre de los *topics* de los que el nodo es suscriptor y editor se va a tener una copia modificada del nodo turtlebot3_drive ya existente para que coincida con el nombre de los *topics* comentados para cada robot. Por lo que se va a crear un nodo por cada robot con la configuración de nombre correcta. Lo más propicio va ser que los nombres de esos nodos sean tb3_0_drive, tb3_1_drive y tb3_2_drive.

Con los pasos comentados, en el *launch* que ejecuta la funcionalidad de movimiento autónomo, se ejecutan los tres nodos. Se pueden elegir los tres ficheros cargando el nodo de la misma forma que en el *launch* actual (Fig. 73), añadiendo otros dos argumentos nombre o, directamente, indicar el nombre del nodo en “node name”.

```
<arg name="name" default="turtlebot3"/>
<param name="cmd_vel_topic_name" value="$(arg name)/cmd_vel"/>
<node name="$(arg name)_drive" pkg="turtlebot3_gazebo" type="turtlebot3_drive" required="true" output="screen"/>
```

Fig. 73: Parte de código del archivo turtlebot3_simulation.launch.

Como se ve en la figura 73, en este mismo *launch*, se está pasando como parámetro el nombre del *topic* encargado de los datos de las velocidades lineal y angular. De la misma forma, se le pueden pasar por parámetro los nombres de todos los *topics* implicados. Se ejecuta tres veces el mismo nodo cambiando los parámetros, es

decir, introduciendo los diferentes nombres de los topics en cada ejecución correspondientes con cada robot.

Añadiendo estas modificaciones, se conseguiría cumplir el objetivo propuesto de mover los tres robots del entorno multirobot visto en la sección 3.5. autónomamente de la misma forma que se mueve la TB3 en la sección 3.2..

6.2. OpenManipulator con TurtleBot 3 multirobot

En el caso de tener la base de Turtlebot 3 junto con el brazo de Open Manipulator, en la documentación disponible no se considera un entorno multirobot. Solo existe el entorno multirobot de varias TB3 pero no de la unión de TB3 y OM. En este apartado se proponen dos entornos multirobot, uno constituido por tres robots con base y brazo y el otro entorno multirobot constituido por un robot con base y brazo y otro siendo solo la base de TB3. El mapa de ambos entornos va a coincidir, siendo que Gazebo tiene unas amplias posibilidades para crear diferentes entornos sin dificultad.

6.2.1. Simulación en Gazebo y movimiento

Se han construido dos archivos en los que se ejecuta un entorno multirobot. Estos archivos de tipo *launch* son multirobot_brazo_base.launch y multirobot_tres_brazos_rooms.launch. Ambos tienen en común que no hace falta elegir el modelo del robot, ya que se ha añadido por defecto, la base móvil será la base waffle, aunque siempre se puede configurar en el *launch* (Anexo 1 y Anexo 2) para que el modelo robot sea diferente, en la línea (Fig. 74):

```
<arg name="model" default="waffle"/>
```

Fig. 74: Línea de código dentro de los launch del entorno multirobot.

También tienen en común el entorno en le que se cargan los modelos robot, pero al igual que con el modelo de la base móvil, se puede cargar cualquier fichero .world en el *launch* (Anexo 1 y Anexo 2) en la línea (Fig. 75):

```
<arg name="world_name" value="$(find open_manipulator_with_tb3_gazebo)/worlds/turtlebot3_rooms.world"/>
```

Fig. 75: Línea de código dentro de los launch del entorno multirobot.

Tras tener en proceso *roscore* se ejecuta el *launch* deseado en una nueva terminal:

- Para el escenario con una base y una base con brazo se escribe la línea de comando:

```
roslaunch open_manipulator_with_tb3_gazebo multirobot_brazo_base.launch
```

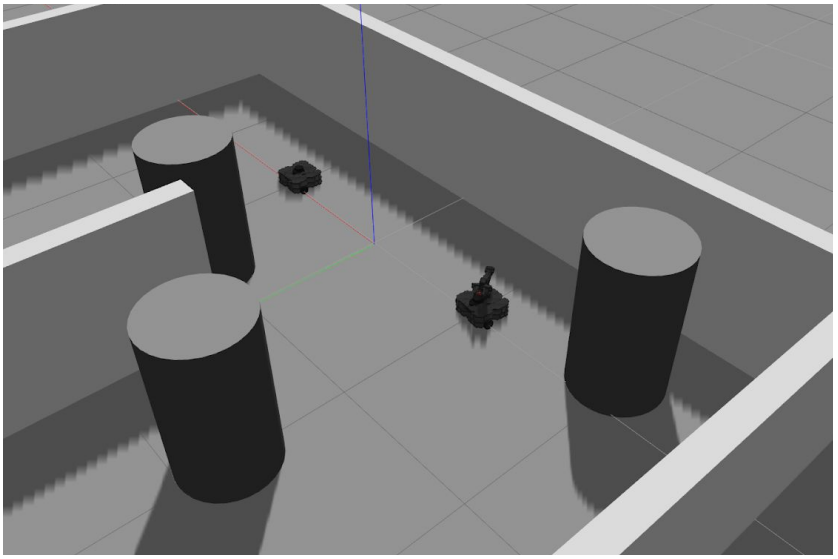

Fig. 76: Entorno multirobot con una TB3 y el conjunto de TB3 con OM.

En la imagen anterior (Fig. 76) se observa el entorno simulado en gazebo con los dos robots en la posición indicada en el *launch* correspondiente (Anexo 1). Además, los nodos y topics activos en este *launch* (Fig. 77) se agrupan en dos grupos, uno por robot con un nodo en común que es gazebo.

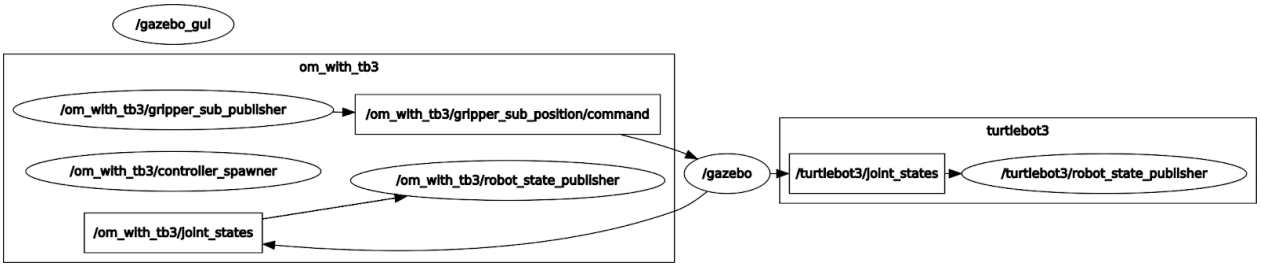

Fig. 77: Rosgraph de nodos y topics activos en el entorno multirobot de base y base con brazo.

- Para el escenario con tres bases con brazo se escribe la línea de comando:

roslaunch open_manipulator_with_tb3_gazebo multirobot_tres_brazos_rooms.launch

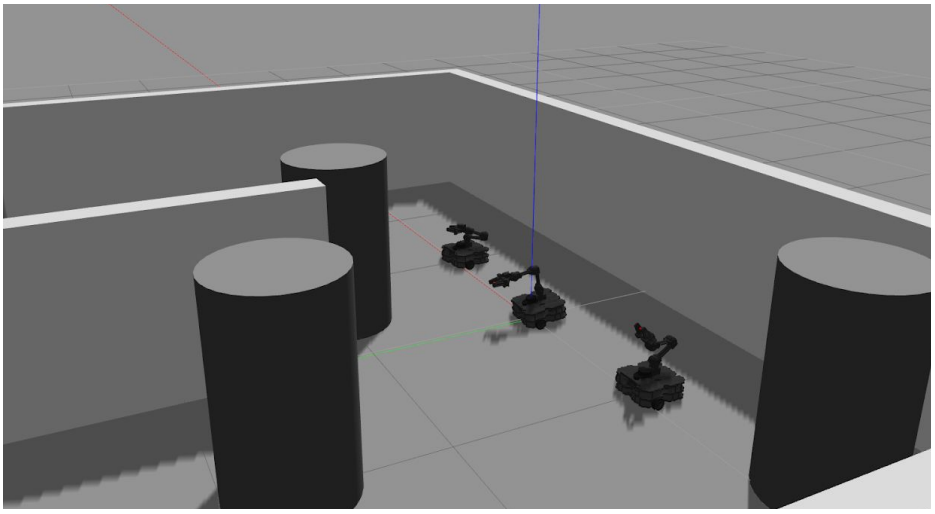

Fig. 78: Entorno multirobot con tres TB3 con OM

En la imagen anterior (Fig. 78) se observa el entorno simulado en gazebo con los tres robots en la posición indicada en el *launch* correspondiente (Anexo 2) y con los ángulos de los brazos indicados también en el *launch,* en las tres líneas de parametrización del robot (Fig. 79):

```
<node pkg="gazebo_ros" type="spawn_model" name="urdf_spawner" respawn="false" output="screen"
    args="-urdf -model $(arg robot_tres_name) -x 1.0 -y 0.0 -Y 1.5708 -J joint1 -1.0 -J joint2 -1.5707 -J joint3 1.37 -J joint4 0.2258 -param robot_description"/>
</group>
```

Fig. 79: Línea de código en la que se indica la posición del brazo y de la base en el entorno.

Además, los nodos y topics activos en este *launch*, se dividen en tres grupos idénticos. Simplemente cambia el nombre del robot. Se observa que el nodo gazebo es común para todos, al igual que en el caso anterior (Fig. 77). Este *rosgraph* (Fig. 80) es muy parecido al *rosgraph* multirobot de la sección 2.2.5. (Fig. 32), además de que en este caso no se muestran los nodos y *topics* que no se dirigen al modelo robot, se añaden algunos nodos asociados al brazo de OpenManipulator.

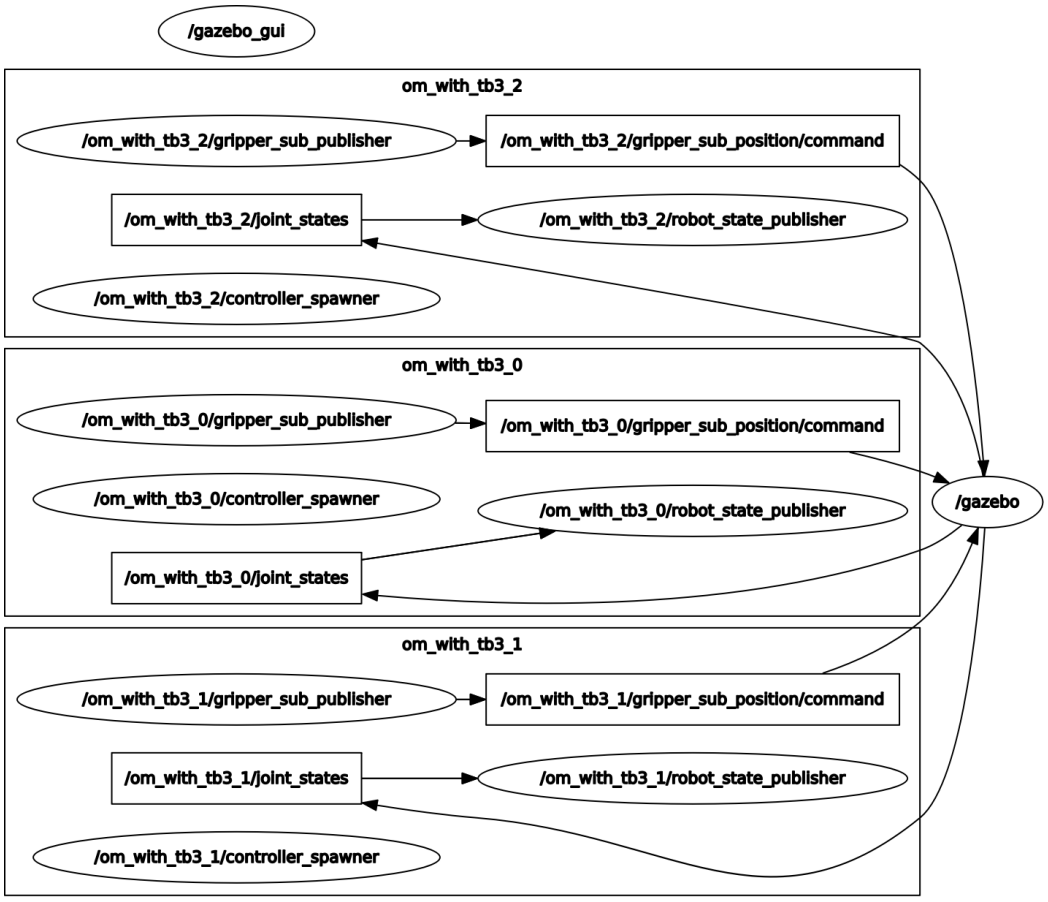

Fig. 80: Rosgraph de nodos y topics activos en el entorno multirobot de tres bases con brazo.

Para llevar a cabo la posición de varios robots en un mismo entorno, se ha tenido en cuenta la configuración de cada uno de ellos, así como añadir configuraciones para diferentes robots (Anexo 3 y Anexo 4) y fijarse en el espacio de nombres.

Además, el movimiento de todos estos robots se lleva a cabo de la misma manera que en la sección 5.2., se pueden manejar todos ellos moviendo la base y, por otro lado, se pueden mover los brazos de todas las formas descritas en la capítulo 4..

6.2.2. Recogida de objetos y comunicación entre robots

Para llevar a cabo una aplicación de recogida de objetos y comunicación entre robots se usaría el entorno multirobot con una base de TB3 y una base con un brazo. La función de la base móvil sería buscar los objetos por un entorno conocido o desconocido y, la función de la base con brazo, sería estar catalogando la basura hasta que la base encontrara un objeto. Entonces, se lo comunicaría a este último e iría a buscarlo por navegación. En esta tarea se podría hacer reutilizando código, ya que la base móvil se moverá automáticamente por el entorno como se explica en la sección 3.2., modificando la máquina de estados.

El robot explora baldosa a baldosa el entorno sin chocarse con ningún objeto y, en cuanto se encuentre un objeto, manda la ubicación del mismo al robot con brazo. Para una prueba inicial, sin reconocimiento de objetos, se pueden guardar unas variables con los valores de unas posiciones fijas en el entorno donde están los objetos, por lo que cuando se encuentra un objeto de frente (cuando la distancia calculada por el sensor láser es menos a un valor determinado, por ejemplo, 0.5 metros o media baldosa), se calcula la distancia a él y se compara la ubicación del objeto encontrado con las ubicaciones en las que se puede encontrar el objeto reciclable. En el caso de que coincidan, se le manda la ubicación al robot con el brazo, en el caso de que no, se esquiva y se sigue buscando. Esa ubicación se envía a través del *topic* move_base_simple/goal, en este caso om_with_tb3/move_base_simple/goal, ya que es el *topic* del que coge los datos el launch de navegación para llevar a cabo la misma hasta esa posición objetivo.

Por parte del robot de telemanipulación, en cuanto reciba una posición a la que ir, se utiliza el código de navegación desarrollado en la sección 5.4.. Se le puede mandar por un *topic* nuevo, un mensaje String, para que cuando le llegue al robot con brazo "objeto encontrado", aplique la funcionalidad de navegación cogiendo la posición que le ha proporcionado la base. Utilizando el planteamiento presentado, también puede haber

diferentes posibilidades como en el entorno de los robots con brazo, coger un objeto entre los tres e ir simultáneamente navegando por un entorno para transportarlo.

7. Conclusiones y trabajo futuro

En este apartado se presentan las conclusiones del proyecto realizado. También se va a hacer una reflexión personal del presente proyecto, así como lo que me ha aportado la realización del presente trabajo.

7.1. Conclusiones

Se puede afirmar que se ha cumplido el objetivo principal de aprender a utilizar el entorno de trabajo ROS en el sistema operativo Linux junto con el simulador gráfico de Gazebo y otras herramientas. Los conocimientos adquiridos han permitido seguir los varios tutoriales seguidos, haber entendido cada funcionalidad aportada por ROS y, sobre todo, haber conseguido aplicar parte del conocimiento adquirido sobre Linux, ROS y las plataformas de TB3 y OM en diversas aplicaciones desarrolladas durante este trabajo de fin de grado.

Por lo tanto, el objetivo final de desarrollar algunas aplicaciones en el entorno de programación, utilizando todo lo aprendido durante el desarrollo de este proyecto, también se ha cumplido satisfactoriamente.

7.2. Trabajo futuro

Este proyecto queda abierto a muchas posibilidades para desarrollar futuras aplicaciones o tareas. Principalmente, quedan las ideas del reconocimiento de objetos mediante visión, para que reconozca botellas u otros objetos que entren en el campo de "basura" a través de sistemas de visión ya existentes. Con bases de datos, mallas, etc.

Además, para completar la funcionalidad de reciclaje, en vista al futuro, se le puede incluir un sensor capacitivo en la pinza del brazo de OpenManipulator para que el propio robot sea capaz de reconocer el tipo objeto que es (papel, plástico, vidrio u otro material) para tirarlo en el cubo de basura indicado. También, se pueden transportar objetos grandes gracias a la comunicación posible entre varios robots, por lo que hace factible el agarrar un solo objeto de gran tamaño por varios robots manipuladores para transportar a una posición objetivo el objeto recogido por todos. En resumen, una gran cantidad de posibilidades se abren para resolver problemas con estos pequeños robots.

7.3. Valoración personal

Personalmente, el proyecto llevado a cabo me ha aportado conocimientos sobre un entorno de trabajo principal en robótica, del cual no sabía demasiado. También, he adquirido conocimientos sobre el sistema operativo de Linux y ha abierto las puertas de aprender a trabajar con diferentes sistemas operativos y la variedad que existe. No solo estar encerrado en Windows.

Además, me ha reforzado la forma esquemática de pensar y de programar, por el tipo de comunicación que se lleva a cabo a través de ROS. He llevado a cabo este proyecto con gran autonomía, lo que ha hecho que refuerce la forma de buscar eficientemente en internet y saber buscar solución a los problemas que surgen.

Por otro lado, me ha dado el último empujón de satisfacción como ingeniera, el volcar años de conocimientos adquiridos para aprender, sobre todo, a pensar. Y, por fin, pensar en ser graduada en Ingeniería Electrónica y Automática.

8. Bibliografía

[1] SÁNCHEZ, Cristina. Shakey, el primer robot inteligente de la historia y el abuelo del coche autónomo, 2017. *elDiario.es.* 11 de abril. [Consulta: 18 noviembre 2019]. Disponible en: https://www.eldiario.es/.

[2] MURILLO, Ana Cristina y MONTANO, Luis. *Introducción Robots Autónomos*. Zaragoza: Departamento de Informática e Ingeniería de Sistema, s.a. Universidad de Zaragoza.

[3] *ROBOTIS e-manual, Overview*. ROBOTIS. [Consulta: 18 noviembre 2019]. Disponible en: http://emanual.robotis.com/.

[4] *open_manipulator*. ROS Wiki. 30 abril 2019, 07:50. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/open_manipulator.

[5] *gazebo_ros_pkgs*. ROS Wiki. 14 junio 2018, 17:58. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/gazebo_ros_pkgs.

[6] *"MoveIt!", un estándar para la manipulación móvil.* Robotnik. 08 agosto 2014. [Consulta: 18 noviembre 2019]. Disponible en: https://www.robotnik.es/.

[7] *rqt*. ROS Wiki. 30 agosto 2016, 18:41. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/rqt.

[8] RAGEL DE LA TORRE, Ricardo. 17 septiembre 2012. PFC, Modelado, control y simulación de un quadrotor equipado con un brazo manipulador robótico. Universidad de Sevilla. Recuperado de: http://bibing.us.es/proyectos/

[9] *¿Cómo abrir un archivo LAUNCH?.* file.extension. [Consulta: 18 noviembre 2019]. Disponible en: https://www.file-extension.info/es/format/launch

[10] *YAML*. Wikipedia, la enciclopedia libre. 11 agosto 2019, 12:53. [Consulta: 18 noviembre 2019]. Disponible en: https://es.wikipedia.org/wiki/YAML

[11] *xacro*. ROS Wiki. 25 octubre 2019, 15:40. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/xacro

[12] *urdf*. ROS Wiki. 11 enero 2019, 01:15. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/urdf

[13] *Terminal (informática)*. Wikipedia, la enciclopedia libre. 13 septiembre 2019, 02:18. [Consulta: 18 noviembre 2019]. Disponible en: https://es.wikipedia.org/wiki/Terminal_(inform%C3%A1tica)#Terminales_de_texto

[14] *ROS-Servicios*. Geek Gasteiz. 17 marzo 2018 [Consulta: 18 noviembre 2019]. Disponible en: https://geekgasteiz.wordpress.com/

[15] *Messages*. ROS Wiki. 26 agosto 2016, 07:21. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/Messages

[16] YoonSeok Pyo, HanCheol Cho, RyuWoon Jung, TaeHoon Lim (Dec 22, 2017): *ROS Robot Programming.* Seoul, República de Corea: ROBOTIS Co.,Ltd.

[17] *rosgraph*. ROS Wiki. 06 febrero 2015, 04:40. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/rosgraph

[18] *ROS/EnvironmentVariables*. ROS Wiki. 13 septiembre 2019, 13:17. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/ROS/EnvironmentVariables

[19] *Slam*. Wikipedia, la enciclopedia libre. 17 diciembre 2017, 18:09. [Consulta: 18 noviembre 2019]. Disponible en: https://es.wikipedia.org/wiki/Slam

[20] *robot_state_publisher.* ROS Wiki. 04 diciembre 2016, 00:19. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/robot_state_publisher

[21] *joint_state_publisher*. ROS Wiki. 15 octubre 2015, 22:20. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/joint_state_publisher

[22] *tf*. ROS Wiki. 02 octubre 2017, 13:40. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/tf

[23] *Questions*. ROSAnswers: Open Source Q&A Forum. [Consulta: 8 noviembre 2019]. Disponible en: https://answers.ros.org/

[24] *map_server*. ROS Wiki. 07 febrero 2019, 18:31. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/map_server

[25] *amcl*. ROS Wiki. 27 junio 2019, 03:41. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/amcl

[26] *move_base*. ROS Wiki. 27 septiembre 2018, 21:33. [Consulta: 18 noviembre 2019]. Disponible en: http://wiki.ros.org/move_base

[27] *ROBOTIS e-manual*, *Controller Package.* ROBOTIS. [Consulta: 18 noviembre 2019]. Disponible en: http://emanual.robotis.com/

[28] *ROBOTIS e-manual, Simulation.* ROBOTIS. [Consulta: 18 noviembre 2019]. Disponible en: http://emanual.robotis.com

[29] *robots*. GOOGLE. [Fecha de consulta: 18 noviembre 2019]. Disponible en: https://www.google.com

[30]. *ROBOTIS e-manual, Manipulation.* ROBOTIS. [Fecha de consulta: 18 noviembre 2019]. Disponible en: http://emanual.robotis.com/

[31] CASSEL, David. Remembering Shakey, the First Intelligent Robot, 2017. *thenewstack.io*. 5 de Marzo. [Consulta: 18 noviembre 2019]. Disponible en: https://thenewstack.io/remembering-shakey-first-intelligent-robot/

[32] *What is a TurtleBot?*. TurtleBot. [Consulta: 18 noviembre 2019]. Disponible en: https://www.turtlebot.com/

9. Anexos

En los siguientes anexos se muestran algunos de los principales ficheros realizados en la programación de las distintas tareas desarrolladas durante la realización de este proyecto.

A.1. Archivo multirobot_brazo_base.launch.

```
<launch>
  <arg name="model" default="waffle"/>
  <arg name="use_robot_brazo_name" default="om_with_tb3"/>
  <arg name="use_robot_base_name" default="turtlebot3"/>

  <!-- These are the arguments you can pass this launch file, for example paused:=true
-->
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

<rosparam file="$(find
open_manipulator_with_tb3_gazebo)/config/gazebo_controller.yaml" command="load"
/>

  <!-- We resume the logic in empty_world.launch, changing only the name of the world
to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find
open_manipulator_with_tb3_gazebo)/worlds/multiWorld.world"/>
        <arg name="debug" value="$(arg debug)" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="paused" value="$(arg paused)"/>
        <arg name="use_sim_time" value="$(arg use_sim_time)"/>
        <arg name="headless" value="$(arg headless)"/>
  </include>
```

```
  <!-- Load the URDF into the ROS Parameter Server -->
  <group ns = "$(arg use_robot_brazo_name)">
        <param name="robot_description"
        command="$(find xacro)/xacro --inorder '$(find
open_manipulator_with_tb3_description)/urdf/open_manipulator_with_tb3_$(arg
model).urdf.xacro'"/>

        <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen">
        <param name="publish_frequency" type="double" value="100.0" />
        <param name="tf_prefix" value="$(arg use_robot_brazo_name)" />
        </node>

        <node pkg="gazebo_ros" type="spawn_model" name="urdf_spawner"
respawn="false" output="screen"
        args="-urdf -model $(arg use_robot_brazo_name) -x -1.0 -y 0.0 -z 0.0 -Y 1.5708
-J joint1 1.0 -J joint2 -1.5707 -J joint3 1.37 -J joint4 0.2258 -param robot_description"/>
  </group>
  <group ns = "$(arg use_robot_base_name)">
        <param name="robot_description" command="$(find xacro)/xacro --inorder
$(find turtlebot3_description)/urdf/turtlebot3_$(arg model).urdf.xacro" />

        <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen">
        <param name="publish_frequency" type="double" value="50.0" />
        <param name="tf_prefix" value="$(arg use_robot_base_name)" />
        </node>

        <node name="spawn_urdf" pkg="gazebo_ros" type="spawn_model"
args="-urdf -model $(arg use_robot_base_name) -x 1.0 -y 0.0 -z 0.0 -Y 1.5708 -param
robot_description" />
  </group>

  <!-- ros_control robotis manipulator launch file -->
  <include file="$(find
open_manipulator_with_tb3_gazebo)/launch/joint_controller.launch">
        <arg name="use_robot_name" value="$(arg use_robot_brazo_name)"/>
  </include>

</launch>
```

A.2. Archivo multirobot_tres_brazos_rooms.launch

```
<launch>
  <arg name="model" default="waffle"/>
  <arg name="robot_uno_name" default="om_with_tb3_0"/>
  <arg name="robot_dos_name" default="om_with_tb3_1"/>
  <arg name="robot_tres_name" default="om_with_tb3_2"/>

  <!-- These are the arguments you can pass this launch file, for example paused:=true
-->
  <arg name="paused" default="false"/>
  <arg name="use_sim_time" default="true"/>
  <arg name="gui" default="true"/>
  <arg name="headless" default="false"/>
  <arg name="debug" default="false"/>

  <!-- rosparam file="$(find
open_manipulator_with_tb3_gazebo)/config/gazebo_controller.yaml" command="load"
/ -->

  <!-- We resume the logic in empty_world.launch, changing only the name of the world
to be launched -->
  <include file="$(find gazebo_ros)/launch/empty_world.launch">
        <arg name="world_name" value="$(find
open_manipulator_with_tb3_gazebo)/worlds/turtlebot3_rooms.world"/>
        <arg name="debug" value="$(arg debug)" />
        <arg name="gui" value="$(arg gui)" />
        <arg name="paused" value="$(arg paused)"/>
        <arg name="use_sim_time" value="$(arg use_sim_time)"/>
        <arg name="headless" value="$(arg headless)"/>
```

```
  </include>

  <!-- Load the URDF into the ROS Parameter Server -->
  <group ns = "$(arg robot_uno_name)">
        <param name="robot_description"
        command="$(find xacro)/xacro --inorder '$(find
open_manipulator_with_tb3_description)/urdf/open_manipulator_with_tb3_$(arg
model).urdf.xacro'"/>

        <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen">
        <param name="publish_frequency" type="double" value="100.0" />
        <param name="tf_prefix" value="$(arg robot_uno_name)" />
        </node>

        <node pkg="gazebo_ros" type="spawn_model" name="urdf_spawner"
respawn="false" output="screen"
        args="-urdf -model $(arg robot_uno_name) -x 0.0 -y 0.0 -Y 1.5708 -J joint1 1.0
-J joint2 0.0000 -J joint3 0.0 -J joint4 0.0 -param robot_description"/>
  </group>

  <group ns = "$(arg robot_dos_name)">
        <param name="robot_description"
        command="$(find xacro)/xacro --inorder '$(find
open_manipulator_with_tb3_description)/urdf/open_manipulator_with_tb3_$(arg
model).urdf.xacro'"/>

        <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen">
        <param name="publish_frequency" type="double" value="100.0" />
        <param name="tf_prefix" value="$(arg robot_dos_name)" />
```

```
        </node>

        <node pkg="gazebo_ros" type="spawn_model" name="urdf_spawner"
respawn="false" output="screen"
        args="-urdf -model $(arg robot_dos_name) -x -1.0 -y 0.0 -Y 1.5708 -J joint1 0.0
-J joint2 -1.0 -J joint3 1.0 -J joint4 -1.0 -param robot_description"/>
 </group>

 <group ns = "$(arg robot_tres_name)">
        <param name="robot_description"
        command="$(find xacro)/xacro --inorder '$(find
open_manipulator_with_tb3_description)/urdf/open_manipulator_with_tb3_$(arg
model).urdf.xacro'"/>

        <node pkg="robot_state_publisher" type="robot_state_publisher"
name="robot_state_publisher" output="screen">
        <param name="publish_frequency" type="double" value="100.0" />
        <param name="tf_prefix" value="$(arg robot_tres_name)" />
        </node>

        <node pkg="gazebo_ros" type="spawn_model" name="urdf_spawner"
respawn="false" output="screen"
        args="-urdf -model $(arg robot_tres_name) -x 1.0 -y 0.0 -Y 1.5708 -J joint1 -1.0
-J joint2 -1.5707 -J joint3 1.37 -J joint4 0.2258 -param robot_description"/>
 </group>

 <!-- ros_control robotis manipulator launch file -->
 <include file="$(find
open_manipulator_with_tb3_gazebo)/launch/multirobot_joint_controller.launch">
        <arg name="robot_uno_name" value="$(arg robot_uno_name)"/>
        <arg name="robot_dos_name" value="$(arg robot_dos_name)"/>
```

```
        <arg name="robot_tres_name" value="$(arg robot_tres_name)"/>
  </include>
</launch>
```

A.3. Archivo multirobot_joint_controller.launch

```
<launch>
  <arg name="robot_uno_name" default="om_with_tb3_0"/>
  <arg name="robot_dos_name" default="om_with_tb3_1"/>
  <arg name="robot_tres_name" default="om_with_tb3_2"/>

  <!-- Load joint controller configurations from YAML file to parameter server -->
  <rosparam file="$(find
open_manipulator_with_tb3_gazebo)/config/multirobot_joint_controller.yaml"
command="load"/>

  <!-- load the controllers -->
  <node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false"
  output="screen" ns="$(arg robot_uno_name)" args="joint_state_controller
                                        joint1_position
                                        joint2_position
                                        joint3_position
                                        joint4_position
                                        gripper_position
                                        gripper_sub_position"/>

  <node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false"
  output="screen" ns="$(arg robot_dos_name)" args="joint_state_controller
                                        joint1_position
                                        joint2_position
                                        joint3_position
                                        joint4_position
```

```
                                    gripper_position
                                    gripper_sub_position"/>

  <node name="controller_spawner" pkg="controller_manager" type="spawner"
respawn="false"
  output="screen" ns="$(arg robot_tres_name)" args="joint_state_controller
                                    joint1_position
                                    joint2_position
                                    joint3_position
                                    joint4_position
                                    gripper_position
                                    gripper_sub_position"/>

  <!-- Run gripper sub position publisher -->
  <node name="gripper_sub_publisher" pkg="open_manipulator_gazebo"
type="gripper_sub_publisher" output="screen" ns="$(arg robot_uno_name)"/>
  <node name="gripper_sub_publisher" pkg="open_manipulator_gazebo"
type="gripper_sub_publisher" output="screen" ns="$(arg robot_dos_name)"/>
  <node name="gripper_sub_publisher" pkg="open_manipulator_gazebo"
type="gripper_sub_publisher" output="screen" ns="$(arg robot_tres_name)"/>

</launch>
```

A.4. Archivo de configuración multirobot_joint_controller.yaml

```
om_with_tb3_0:
  # Publish all joint states ----------------------------------
  joint_state_controller:
        type: joint_state_controller/JointStateController
        publish_rate: 1000

  # Position Controllers ---------------------------------------
  joint1_position:
        type: position_controllers/JointPositionController
        joint: joint1

  joint2_position:
        type: position_controllers/JointPositionController
        joint: joint2

  joint3_position:
        type: position_controllers/JointPositionController
        joint: joint3

  joint4_position:
        type: position_controllers/JointPositionController
        joint: joint4

  gripper_position:
        type: position_controllers/JointPositionController
        joint: gripper

  gripper_sub_position:
```

```
        type: position_controllers/JointPositionController
        joint: gripper_sub

om_with_tb3_1:
  # Publish all joint states ----------------------------------
  joint_state_controller:
        type: joint_state_controller/JointStateController
        publish_rate: 1000

  # Position Controllers ---------------------------------------
  joint1_position:
        type: position_controllers/JointPositionController
        joint: joint1

  joint2_position:
        type: position_controllers/JointPositionController
        joint: joint2

  joint3_position:
        type: position_controllers/JointPositionController
        joint: joint3

  joint4_position:
        type: position_controllers/JointPositionController
        joint: joint4

  gripper_position:
        type: position_controllers/JointPositionController
        joint: gripper

  gripper_sub_position:
```

```
        type: position_controllers/JointPositionController
        joint: gripper_sub

om_with_tb3_2:
  # Publish all joint states ----------------------------------
  joint_state_controller:
        type: joint_state_controller/JointStateController
        publish_rate: 1000

  # Position Controllers ---------------------------------------
  joint1_position:
        type: position_controllers/JointPositionController
        joint: joint1

  joint2_position:
        type: position_controllers/JointPositionController
        joint: joint2

  joint3_position:
        type: position_controllers/JointPositionController
        joint: joint3

  joint4_position:
        type: position_controllers/JointPositionController
        joint: joint4

  gripper_position:
        type: position_controllers/JointPositionController
        joint: gripper

  gripper_sub_position:
```

type: position_controllers/JointPositionController

joint: gripper_sub