UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

Navegación autónoma basada en un mapa topológico enriquecido semánticamente: Aplicación a un robot de servicio asistente en un entorno interior.

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA ELECTRÓNICA INDUSTRIAL Y AUTOMÁTICA

Autor: **Pablo López Ros**

Director: Dra. Nieves Pavón Pulido

Codirector: Dr. Juan Antonio López Riquelme

Cartagena, 10 de enero de 2021

AGRADECIMIENTOS

En primera instancia, agradecer a mi familia y amigos por haberme motivado a ampliar mis conocimientos y ponerlos a prueba en un proyecto de esta envergadura, especialmente en los tiempos que nos están tocando vivir a todos. Además, deseo agradecer a la Doctora Nieves por haberme dado a conocer este ámbito de desarrollo e investigación y haber sabido despertar mi interés por él en estos dos últimos años.

RESUMEN

En este trabajo se presenta el desarrollo de un paquete de software basado en ROS que proporciona la capacidad de generar y utilizar un mapa topológico enriquecido semánticamente para su uso en un vehículo autónomo inteligente terrestre en interiores. Para ello, se hará uso de técnicas de mapeado láser y se extraerá información del entorno mediante tecnologías de Deep Learning. Así mismo, se replicará un entorno real mediante herramientas CAD 3D para su posterior uso en las pruebas de concepto.

Contenido

Índice de Figuras

Índice de Tablas

1. Introducción

1.1. Objetivo

El objetivo principal de este trabajo es el de desarrollar un paquete de funcionalidad basado en ROS que permita la generación de mapas enriquecidos semánticamente en entornos interiores. De esta manera, se obtendrá un mapa con la morfología del entorno, en el cual las distintas estancias se encontrarán diferenciadas entre sí, se marcarán las zonas transitables y se mostrará información referente a cada estancia. Para la realización de este objetivo, es necesario el cumplimiento de unos requerimientos previos, que permitan tener la infraestructura necesaria para el desarrollo de dicho paquete.

1.2. Requerimientos

1.2.1. Creación de un entorno de trabajo

Para la creación de un paquete, es necesaria la puesta en marcha del entorno de ROS en una máquina local, asi como de otras herramientas para el desarrollo de código, como es el simulador Gazebo. A su vez, debe crearse una carpeta de trabajo donde desarrollar el paquete de funcionalidad e incluir otros paquetes de terceros que este necesite para funcionar. Dichos paquetes de terceros proporcionan funcionalidades extra como controlar los movimientos del robot, la generación de mapas métricos, la generación de rutas, etc. Estos paquetes no son siempre compatibles entre sí, por lo que se deberá hacer una correcta adaptación de estos.

1.2.2. Creación de un modelo virtual de prueba

A la hora de desarrollar un paquete para un robot, los tiempos requeridos para volcar el código en este y su posterior puesta en marcha son excesivamente largos. A su vez, el espacio requerido para poner a prueba algoritmos de generación de mapas es considerablemente grande. Por tanto, es necesario la creación de una planta de un edificio para incluirla en el simulador y, de esta manera, eliminar estas trabas en el desarrollo. Para este proyecto, se propone la creación de la planta baja de la Escuela Técnica Superior de Ingeniería Industrial.

1.2.3. Obtención de un mapa métrico

Para poder realizar un mapa enriquecido semánticamente, es necesario partir de un mapa métrico del entorno. Por tanto, es crítica la selección de un paquete de funcionalidad que brinde esta capacidad. A su vez, será necesaria su puesta a punto para funcionar eficientemente con el modelo del robot.

1.2.4. Extracción de información semántica

Para este objetivo, será necesario el desarrollo de un algoritmo capaz de interpretar el mapa métrico para extraer información de la composición del entorno y su estructuración. A su vez, se deberá crear un servicio en la nube con el que procesar imágenes generadas por el robot para obtener información adicional sobre el entorno con la que clasificar las estancias.

1.2.5. Creación de una relación métrico-semántica

Una vez interpretada la información del mapa métrico y obtenida información semántica del entorno, esta deberá ser apropiadamente estructurada y clasificada. La intención es obtener una red de elementos de la planta que guarden información sobre sí mismos y la relación que guardan entre ellos.

2. Estado del Arte

2.1. Robótica de servicios en la actualidad

En la actualidad, se pueden oír continuamente noticias sobre el avance en el campo de la robótica. Por suerte, en los últimos años se está invirtiendo mucho en el avance en la robótica de servicios. Los casos más sonados son los robots humanoides de empresas como Toshiba o Hanson Robotics (Hanson Robotics, s.f.), que han diseñado máquinas con un aspecto bastante realista, pero con funciones muy reducidas.

Sin embargo, estos no son los únicos robots existentes. Existe una gran variedad de robots dedicados al servicio a las personas que, si bien no tienen un aspecto tan "geminoide", son capaces de desempeñar un amplio catálogo de tareas que pueden resultar tediosas e incluso dificultosas para los seres humanos. Un buen ejemplo es Cloi Cartbot, un robot móvil diseñado por LG (KELLER, 2019), que te acompaña a hacer la compra. Cloi es la base sobre la que se han diseñado una serie de robots asistenciales, como son Cartbot, Porterbot, Cleanbot o Guidebot. Este último se trata de un robot de limpieza autónomo que se ha implementado satisfactoriamente en el Aeropuerto Internacional Incheon de Corea del Sur. Guidebot se mueve de forma autónoma por el aeropuerto proporcionando información de interés a los pasajeros, así como avisos. Probablemente, el ejemplo más interesante hasta la fecha es el caso de Porterbot. Porterbot se ha diseñado con la intención de ayudar en el *check-in* y *check-out* en hoteles y residencias. De forma autónoma, es capaz de transportar la maleta de los clientes y llevarlos hasta su habitación.

Figura 1. Robot Cloi Cleanbot.

Un buen ejemplo de entrega de paquetería es Scout, de Amazon. Se trata de un robot rodado autónomo con un volumen de carga medio. Su función es la entrega de un pedido en la casa del cliente, siendo capaz de evitar todos los obstáculos que se encuentre en el camino, personas, animales, etc. Actualmente, se Encuentra en fase de pruebas en el condado de Snohomish, en Washington (Scott, 2019).

Figura 2. Robot SCOUT de Amazon circulando en su etapa de pruebas por la población de Snohomish.

Por último, pero no por ello menos importante, se encuentra Thalon; un robot desarrollado por un grupo de 20 ingenieros de Bogotá (Vargas, 2019). Thalon es capaz de recibir una comanda de un cliente del hotel y llevársela hasta su propia habitación. Es capaz de llevar tanto platos principales, como comida y bebida refrigerada, aparte de hacer el servicio de lavandería. Para ello, Thalon cuenta con tecnología LIDAR para hacer un mapeado de todo el hotel en el que se encuentre. Gracias a este mapa, tiene la habilidad de moverse

de forma autónoma por el hotel y esquivar posibles obstáculos que se presenten con una serie de sensores ultrasónicos e infrarrojos. Además, el robot se encuentra servido con varias cámaras que le permiten llamar al ascensor para moverse cómodamente por el edificio y localizar la habitación de destino, donde llamará a dicha habitación o enviará una notificación por la App del hotel al comensal. Por otro lado, posee un software de reconocimiento facial, permitiendo suplir el trabajo de los recepcionistas de reconocer a los huéspedes y saludarles por su nombre cuando se los cruce.

Con este repaso nos damos cuenta de que hay un amplio desarrollo de la robótica movil en el sector servicios. Sin embargo, se aprecia que la inmensa mayoría de soluciones son de alto coste y se encuentran concentradas en sectores muy concretos. Por este motivo, este trabajo se va a centrar en el desarrollo de un prototipo de bajo coste y código abierto que, gracias a los avances en mapas semánticos, permita parecerse al comportamiento de un ser humano en tareas cotidianas, manteniendo un coste asequible para pequeñas empresas y/o organizaciones. A su vez, al ser de código abierto se permiten futuras modificaciones o extensiones de una forma más cómoda y específica para el usuario final.

2.2. Linux

Se denomina Linux, o GNU/Linux, al conjunto de sistemas operativos basados en UNIX (Linux ORG, s.f.). Se trata de sistemas que han sido desarrollados de forma modular y son de código abierto. Gracias a que se desarrollan bajo la Licencia Publica General, durante estas últimas décadas han experimentado una rápida expansión y aceptación por parte de la comunidad, especialmente en el ámbito del desarrollo informático.

Sin embargo, Linux es sólo el *kernel*, necesita de más elementos para su utilización por parte del usuario final. Dichos elementos son módulos de funcionalidades desarrollados por personas de todo el planeta y distribuidos de forma libre. Uno de los más importantes es el motor gráfico, que permite un entorno de utilización visual para el usuario.

Linux cuenta con una gran variedad de compiladores para la mayoría de los lenguajes de programación. A su vez, siempre pueden añadirse más desde el acceso a los mismos en repositorios. Gracias a esto, es comúnmente usado para el desarrollo de programas y aplicaciones, ya que brinda la posibilidad de desarrollarlos para cualquier plataforma. Para ello, utiliza un compilador cruzado, muy preciado a la hora de realizar aplicaciones multiplataforma.

Como es de entender, dichos módulos de funcionalidades ya han sido agrupados y compilados en paquetes listos para su uso. A Estos paquetes se les denomina distribuciones o *dristo*. La distro que se va a emplear a lo largo de este proyecto es Ubuntu. Ubuntu es una de las distros más conocidas, acaparando una cuota de mercado de más del 50% en su versión de escritorio y servidor. Además, ROS se desarrolla exclusivamente para esta distribución. La versión en la que está implementado este proyecto es concretamente Ubuntu 18.04.5 LTS (Ubuntu Canonical, s.f.), puesto que la versión más actualizada, la 20.04.1 LTS, aún no ha sufrido las portaciones de todos los paquetes de ROS. Esto, provoca que no se pueda garantizar la total compatibilidad de dichos paquetes, siendo así preferible usar la 18.

2.3. ROS

También conocido como Robot Operating System (ROS ORG, s.f.), es el principal marco de trabajo para el desarrollo de software del ámbito de la robótica móvil. Fue creado en 2007 bajo el nombre de Switchyard por el Laboratorio de Inteligencia Artificial de Standford con la intención de dar soporte al desarrollo del Stanford Intelligence Robot (STAIR). Con el tiempo se ha convertido en un entorno robusto y flexible, que permite la cooperación entre instituciones y grupos de desarrollo de todo el planeta para la creación de software para robótica.

La existencia de este entorno ha supuesto un punto clave para el rápido desarrollo de la robótica movil puesto que, como es de entender, diseñar un robot completamente funcional es una tarea realmente compleja; y casi imposible de realizarse por un único grupo de expertos. De esta manera, se permite la creación de grupos de especialización en ciertas tareas concretas, como el mapeo de interiores, y su posterior liberación de forma gratuita o bajo licencia de uso libre al resto de desarrolladores.

Como se puede observar, ROS sigue fielmente la filosofía con la que se creó LINUX, crear software libre y modular.

2.3.1. El entorno

ROS (ROS ORG, s.f.)cuenta con una gran cantidad de usuarios a lo largo de todo el planeta, con más de 1500 contribuyentes activos, una enorme comunidad de soporte en foros.

Este entorno de trabajo viene con una gran variedad de herramientas y funcionalidades de base. Por un lado, posee herramientas de visualización y *debug* para el proceso de desarrollo, como lo son *rviz* y *rqt*, que se verán más adelante. Dichas herramientas nos permiten comprobar el correcto funcionamiento de cada funcionalidad que se esté implementando, asi como el desempeño del robot en general. Por otro lado, nos brinda un total de 45 comandos, los cuales facilitan el lanzamiento, testeado, compilado y depuración (*debug*) de los diferentes nodosos y servicios que se estén desarrollando.

Además, ROS se encarga de la generación y gestión de toda la infraestructura de comunicaciones. Desde la publicación de mensajes, hasta la realización de llamadas asíncronas; pasando por el almacenamiento de los mensajes en un histórico revisable y reproducible.

Por otra parte, ros proporciona un standard para la creación de mensajes, contando de base con un gran número de tipos de mensaje; desde el envío de *strings* y *booleanos*, hasta información completa de mediciones de laser en un array de datos. Gracias a estas características, los desarrolladores pueden ir prácticamente directos al proceso de desarrollo de su software de funcionalidad.

2.3.2. Workspace

El espacio de trabajo o *Workspace*, contiene todo lo necesario para poder ejecutar uno o varios paquetes de ROS. Se trata de una carpeta creada mediante el comando de ros `catkin_make <ws_folder name>`. Dentro de esta carpeta se encuentran un total de otras 3 subcarpetas: *src*, *build* y *devel*.

Como se puede observar en la Figura 3, la carpeta *src* es donde se incluyen todos los paquetes de funcionalidades, ya sean paquetes previamente desarrollados por terceros o paquetes en desarrollo. Dichos paquetes pueden ser directamente descargados de repositorios de GitHub, bien sea en formato zip o mediante línea de comando, e incluidos en dicha carpeta para estar listos para funcionar.

La carpeta *build*, como su nombre indica, almacena todas las imágenes compiladas de los paquetes que se hayan incluido en *src* y posteriormente compilado. Una gran ventaja de ROS es que, al trabajar con las imágenes compiladas del código, se pueden realizar modificaciones al código fuente sin el riesgo de que afecten directamente a la ejecución. Puesto que estas no tendrán efecto hasta que sean compiladas, y si contienen errores, tampoco se generará dicha imagen.

La carpeta *devel* contiene todas las dependencias, entre los paquetes incluidos, y con otros paquetes y herramientas propias de ROS.

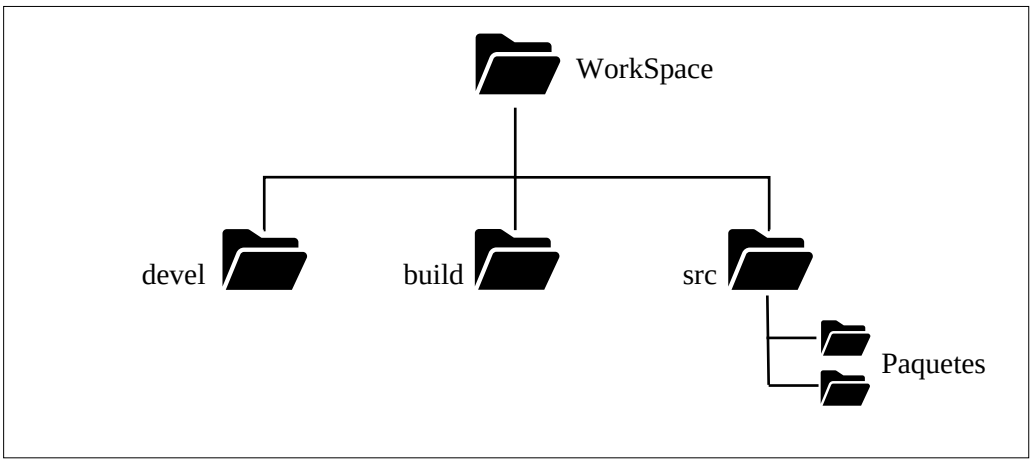

Figura 3. Estructura interna de un entorno de trabajo de ROS. Se encuentra dividido en 3 carpetas, siendo src donde se albergan todos los paquetes de funcionalidades.

2.3.3. Paquete

Como ya se ha mencionado anteriormente, ROS se basa en la modularidad, por lo que el software se desarrolla por módulos de funcionalidad, conocidos como paquetes. Cada paquete debe ser completamente funcional sin necesidad de elementos externos. Un paquete debe realizar una terea concreta y básica, como puede ser leer el input del teclado y transformarlo en comandos de velocidad para el robot. Los paquetes pueden comunicarse entre sí mediante los mensajes standard de ROS. De esta manera, los comandos de velocidad son enviados por el paquete que los genera y recibidos por otro encargado de aplicarlos. A su vez, un paquete puede contener varios paquetes en su interior.

2.3.3.1. SRC

En esta carpeta se debe incluir el código fuente. Pueden incluirse tantos archivos como se desee. Normalmente, cada archivo es un nodo de ROS, pero puede separarse en varios archivos en caso de que sea demasiado extenso e importarlos al que albergue el nodo.

2.3.3.2. Include

Cuando se tienen varios nodos, o archivos que son llamados desde otro nodo, se deben crear sus correspondientes ficheros de cabecera y colocarse en esta carpeta. Una vez hecho esto, simplemente se incluyen en el codigo donde se deseen llamar. Para que puedan localizarse correctamente, debe crearse una subcarpeta dentro de esta con el nombre del paquete donde están alojados dichos archivos.

2.3.3.3. Scripts

En ocasiones, es necesario crear ejecutables de Shell o scripts en otro lenguaje, como Python. Para tenerlos de forma organizada, el estándar recomienda la creación de una carpeta con el nombre de Scripts, e incluir en ella dichos ejecutables.

Estos scripts pueden ser ejecutados desde cualquier nodo que esté incluido en *src*. Brindando una gran oportunidad de crear códigos para emplear ciertas librerías que solo estén disponibles en un lenguaje en concreto y luego recibir la información necesaria mediante un mensaje estándar de ROS.

2.3.3.4. Config

Normalmente, cuando un paquete implementa una tarea o función que requiere de cierto reajuste de parámetros, o simplemente desea brindar la posibilidad de cambiar el nombre de los *topics* con los que se trabaja; se pueden crear archivos de configuración dentro de esta carpeta. Por ejemplo, si se trata de un paquete que sube imágenes a la nube, en este archivo podría incluirse una variable que permita modificar la frecuencia con la que estas imágenes son enviadas, o la resolución de las mismas. En el caso de un proceso de mapeado, normalmente se da la opción de modificar el nombre del *topic* de los datos del láser al que se debe subscribir.

2.3.3.5. Msg

Como se ha mencionado con anterioridad, ROS incluye de base una serie de mensajes estandarizados para permitir la comunicación entre nodos. Pero puede darse el caso de que se desee enviar una información que no termina de encajar en ningún tipo de los mensajes existentes; bien sea por que se deja gran parte de dicho mensaje vacío o porque directamente no alberga ninguna estructura de datos que coincida con la que se desea enviar.

En estos casos, ROS da la opción de crear estructuras de mensaje propias. La creación de estos mensajes se verá posteriormente. Para que puedan emplearse, la definición de estos tipos de mensaje debe incluirse en la carpeta *msg*.

2.3.3.6. MakeFile

Este es un archivo que se genera automáticamente al crear la carpeta del paquete. En él, se especifica toda la información necesaria para que el paquete pueda compilarse.

Si alguno de los nodos de este paquete emplea alguna estructura de mensaje, librería o tiene dependencia con otro paquete; debe especificarse aquí. En caso contrario, el codigo fuente no podrá compilarse. A su vez, si se desea poder lanzar algún nodo de ROS de este paquete, se debe especificar que se desea añadir un ejecutable para dicho nodo. Además, si se ha creado algún script, también debe incluirse en este archivo.

2.3.3.7. Package xml

Se trata mayoritariamente de un archivo informativo. En él se indica que tipo de licencia posee el paquete, quien es el autor, el correo, versión y fecha de revisión, etc. Por otro lado, también se indica si es necesaria alguna librería propia de ROS, tanto para la compilación como para la ejecución.

2.3.4. Nodo

ROS tiene un modo de funcionamiento basado en publicación-subscripción, donde cada paquete es un nodo, que tiene la posibilidad tanto de publicar como de subscribirse a mensajes. Estos mensajes se denominan *topics*, y se verán más adelante. Un nodo puede publicar un comando de velocidad, que será recibido por otro nodo que este subscrito a ese mensaje, como puede serlo el encargado de controlar los motores del robot.

Por supuesto, existe la posibilidad de tener más de un nodo por paquete, ya que estos se definen en el código fuente, en la carpeta *src*. Esto es especialmente útil cuando en el paquete se tienen dos o más funcionalidades muy diferenciadas una de otra, o simplemente se trata de un paquete que alberga varios paquetes en su interior.

Este sistema posee varias ventajas. La principal es que ayuda a crear un codigo mucho más simple y legible. A su vez, esto también ayuda a aumentar la modularidad. Por otro lado, aumenta la robustez del sistema en su conjunto puesto que, si se produce un error de ejecución en uno de los nodos, el resto no *crashearán*. Simplemente la funcionalidad que manejase ese nodo dejaría de ir, evitando asi reacciones impredecibles por parte del robot.

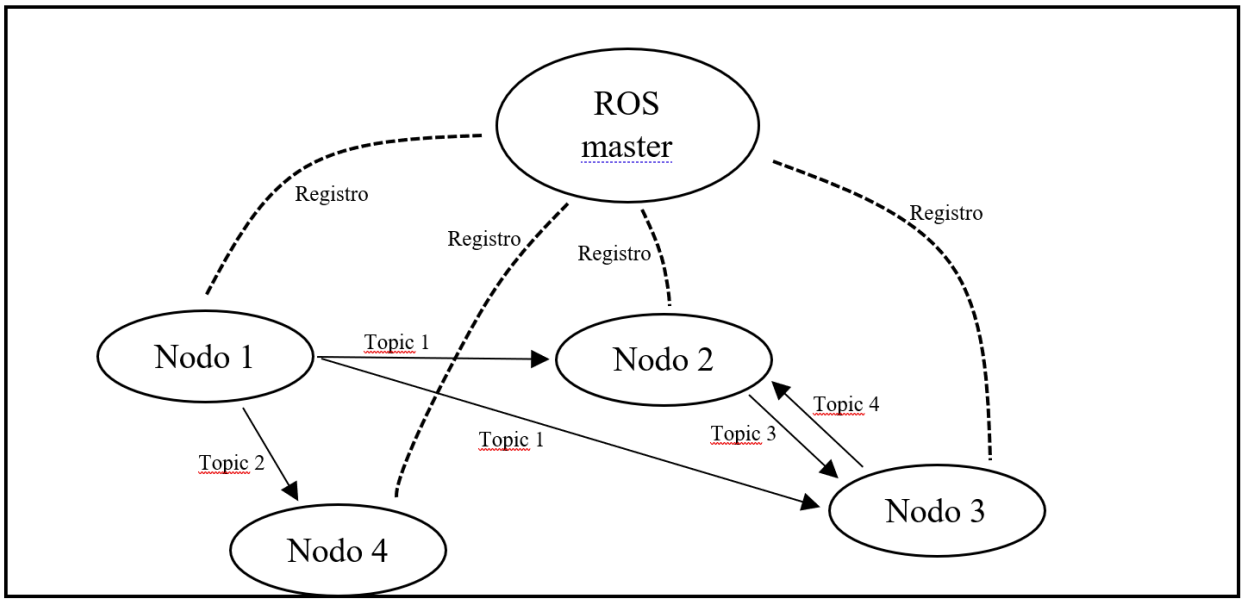

Figura 4. Esquema de comunicaciones entre varios nodos bajo una aplicación de ROS.

Una cosa a tener en cuenta es que, al estar distribuidos por nodos, es necesaria la existencia de un nodo que haga de gestor. Este nodo es denominado *master*, tal y como puede verse en la Figura 4. Por él, pasa todo el tráfico de mensajes. Es necesario que se lance el primero en tiempo de ejecución, pues el resto de los nodos intentarán comunicarse con él en primera instancia.

Para definir un nodo, debe incluirse lo siguiente en el codigo fuente:

```
  ros::init(argc, argv, "<nombre del nodo>");
  ros::NodeHandle n;

  ros::Rate looprate(4);

  while (ros::ok()){
    ros::spinOnce();
    looprate.sleep();
  }
```

Para poder ejecutar un nodo, debe incluirse un ejecutable en el *makefile*:

```
add_executable(<nombre del nodo> src/<nombre del archivo>.cpp)
target_link_libraries(
    <nombre del nodo>
    ${catkin_LIBRARIES}
)
```

Para lanzar un nodo, pueden emplearse cualquiera de los siguientes comandos de ROS:

```
$ rosrun <nobre del paquete> <nobre del nodo>
$ roslaunch <nombre del paquete> <nombre del nodo>
```

2.3.5. Topic

Como se ha explicado en el apartado anterior, los nodos se comunican entre si enviándose mensajes mediante el denominado proceso de publicación-subscripción. En este proceso, un nodo genera un mensaje de un tipo estándar y lo publica. Pero, para que un segundo nodo pueda subscribirse a un mensaje, se necesita una manera de poder identificarlo. Para ello nace el concepto de *topic*.

Un *topic* no es más que un identificador de mensajes. Se trata de una *pipeline* con un nombre concreto, a escoger por el desarrollador, por el cual se estarán publicando mensajes del mismo tipo y contexto. Por

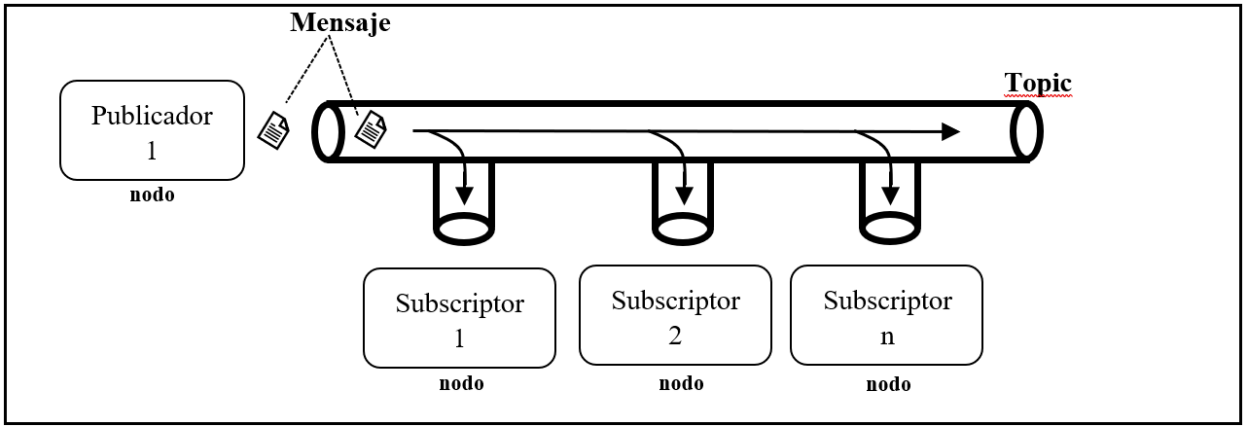

Figura 5. Envío de mensajes a través de una pipeline en una comunicación de publicación-subscripción.

ejemplo, para que el nodo que realiza el mapeado pueda recibir los datos que genera el nodo que controla el láser, se crea el topic */<nombre del robot>/laser/scan*. De esta manera el nodo del láser publica mensajes en este topic, que serán recibidos por el nodo de mapeado si este se ha subscrito a dicho *topic*.

Quien establece el nombre del *topic* es el nodo que publica los mensajes. El desarrollador debe asegurarse de que el nombre del *topic* al que el nodo subscriptor intenta conectarse es el correcto, de lo contrario nunca recibirá información. Se pueden tener un número indefinido de subscriptores. No es recomendable tener más de un publicador por *topic* puesto que puede dar lugar a errores en el receptor.

Un nodo puede tanto publicar como subscribirse a varios *topics*. A su vez, un nodo puede publicar mensajes tanto de forma síncrona como asíncrona, a modo de respuesta a eventos.

2.3.6. Servicio

El gran problema del método de comunicación de publicación-subscripción, es que se trata de un camino solo de ida. Un nodo puede enviar información a otro, pero no recibir una respuesta. Para esos casos existen los servicios.

Un servicio es un método de comunicación petición-respuesta, compuesto por un par de mensajes. En este caso, un nodo ofrece un servicio al que se puede llamar mediante un nombre concreto. El nodo cliente, envía un mensaje de petición a ese servicio, y queda a la espera de obtener respuesta por parte del proveedor del servicio. Una vez recibida, la ejecución continua.

2.3.7. Mensaje

Los nodos se comunican mediante mensajes publicados en *topics*. Un mensaje es una estructura simple de datos, la cual puede incluir varios campos. Los mensajes aceptan tanto los tipos de datos básicos (*booleanos*, *enteros*, *strings*, etc.), como *arrays* de los mismos. Además, también pueden enviarse estructuras de datos anidadas y crear *arrays* de las mismas.

Los mensajes se definen en archivos de texto planos, alojados en la carpeta *msg*. La estructura que siguen para su definición es <tipo de dato> <nombre>. Entre las estructuras de datos que puede contener un mensaje, se encuentra el HEADER. Se trata de una estructura capaz de almacenar metadatos útiles sobre el mensaje, como pueden ser la ID del mensaje o la hora de envío (*timestamp*).

2.3.8. Transformada

En ROS, un robot esta puede definirse por partes (las piezas físicas, los motores, sensores, LIDAR, etc.). Estos elementos son definidos por propiedades geométricas por lo que, para su correcta simulación, es necesario establecer la relación de posición entre ellas. Esto es especialmente importante para el caso de sensores y articulaciones motorizadas.

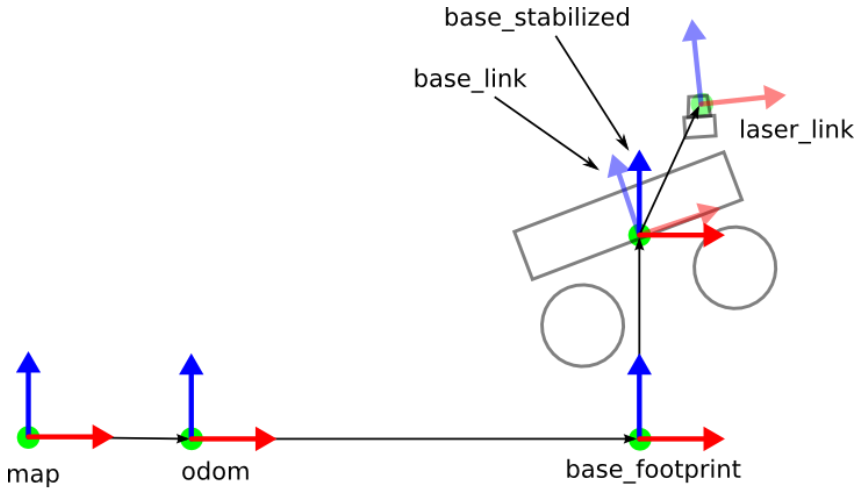

Figura 6. Ejemplo de distribución de los ejes de referencia presentes en una aplicación robótica. En este caso, map es el eje de referencia global y odom se encuentra fijo con una relación de posición determinada respecto a este. La base del robot sin información de elevación se denomina base_footprint y se trata de un eje de referencia móvil referenciado a odom.

Los paquetes de ROS trabajan siempre respecto a un punto de referencia común. La pose del robot se calcula respecto a este eje de referencia, que es fijo (ver Figura 6). Por tanto, resulta bastante intuitivo el hecho de que para obtener la información del láser y los sensores es necesario conocer la relación de posición de estos respecto a la base del robot y, por ende, su transformada.

Existe un paquete encargado de gestionar todas las transformadas del robot, tf. Sin embargo, en ocasiones es necesario especificar manualmente la transformada entre ciertos elementos. La existencia de estas transformadas facilita el desarrollo de nuevos paquetes y funcionalidades, puesto que no es necesario tener en cuenta los cálculos geométricos para todas y cada una de las medidas, ese trabajo se le deja a tf.

2.4. GitHub

2.4.1. La plataforma

GitHub es un servicio de almacenamiento en la nube (GitHub , s.f.). Se trata de la plataforma por excelencia a nivel mundial para desarrollar software, especialmente para sistemas de CI/CD (Continuous Integration and Continuous Delivery). Cuando se está desarrollando un software, siempre se desea tener una copia de seguridad para evitar la pérdida del código si algo le ocurre al hardware. En otros casos, cuando se están realizando cambios relevantes en el código o en sus funcionalidades, siempre es preferible tener alguna forma de poder mantener versiones anteriores para volver a ellas si es necesario, y siempre tener una versión del software funcional a mano.

Por su parte, GitHub nos brinda todas esas posibilidades, con la gran ventaja de que se trata de un servicio gratuito. En esta plataforma se puede crear un repositorio donde almacenar todo el código de manera segura y poder acceder a él cuando se desee. Además, cada vez que se sube un cambio a la plataforma, se crea una nueva rama con dicho cambio, manteniendo el codigo original intacto. De esta manera se pueden tener tantas versiones del codigo como se deseen, donde los cambios de una no afecten a la otra. Pero GitHub no se queda solo ahí. Esta plataforma ofrece también la posibilidad de revisar el código cada vez que se sube un cambio, realizarle test y hacer revisiones de este. Es una gran ventaja sobre

todo cuando hay varios desarrolladores trabajando en un mismo proyecto, puesto que puede hacerse comentarios sobre cambios, aprobar o rechazar los mismos, etc.

Una funcionalidad de gran utilidad para desarrollar software es enlazar GitHub con el editor de código con el que se vaya a trabajar, como se puede observar en la Figura 7. Uno de los editores más usados es Visual Studio Code, que posee una extensión para trabajar con GitHub, permitiendo ver la última fecha de edición de cada línea de código, asi como acceder a un histórico del mismo. Otra de las posibilidades que brinda es la de crear *commits* desde el propio editor y visualizar las ramas del repositorio.

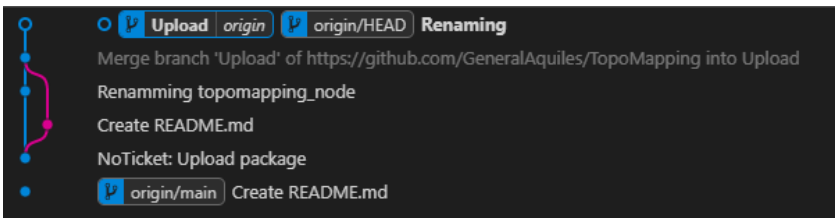

Figura 7. Visualización del historial de ramas y topics desde un plugin de git para Visual Studio Code. Puede observarse como, tras el commit "Upload package" se creó una rama nueva para incluir el archivo README. Esta rama se fusionó posteriormente con la rama principal, un commit por encima de donde se generó.

2.4.2. El repositorio

Desde GitHub, se puede obtener una vista general del repositorio. En ella se puede ver todos los archivos y carpetas de la rama por defecto, además de cambiar de rama. Puede revisarse el codigo fuente del paquete navegando entre las carpetas y archivos, tal y como se observa en la Figura 8. Panel de información general de un repositorio. Desde aquí, puede tenerse acceso a las carpetas del repositorio, obtener información sobre los lenguajes de programación empleados y seleccionar la rama que visualizar. En el botón verde puede obtenerse un enlace para clonar el repositorio, o descargarlo directamente en formato comprimido.. A su vez, estos pueden ser editados y realizar nuevos *commits* desde el navegador si se desea. Otra opción bastante útil es la de generar *pull request*, que *mergean* los cambios de una rama a otra. De esta manera se puede tener una rama principal a la que solo se le aplican modificaciones en el codigo cuando estas son completamente funcionales. Estas modificaciones provienen de ramas secundarias, donde se crea una rama en específico para cada funcionalidad nueva. El objetivo de trabajar de esta manera es evitar que errores que surjan durante el desarrollo de una funcionalidad afecten al resto.

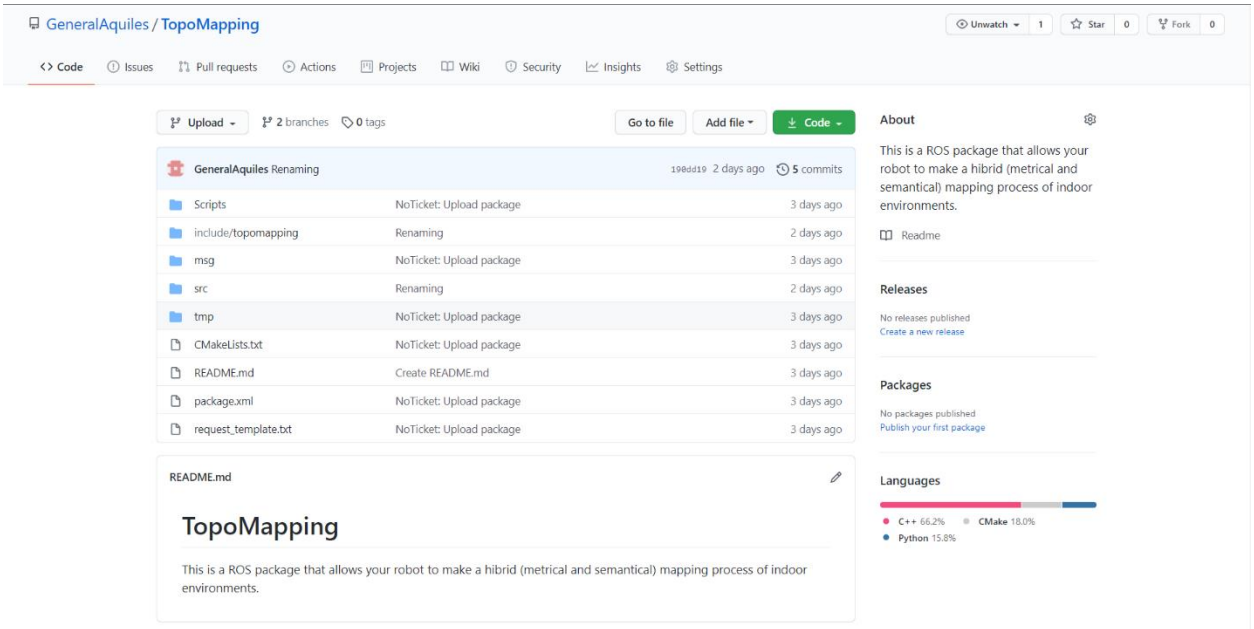

Figura 8. Panel de información general de un repositorio. Desde aquí, puede tenerse acceso a las carpetas del repositorio, obtener información sobre los lenguajes de programación empleados y seleccionar la rama que visualizar. En el botón verde puede obtenerse un enlace para clonar el repositorio, o descargarlo directamente en formato comprimido.

2.4.3. Usar GitHub

Lo más habitual cuando se está desarrollando un paquete software, es subir el repositorio a plataformas de integración continua en la nube. Especialmente en el caso de ROS, como para otros muchos ecosistemas, la comunidad se ha decantado por el uso de GitHub. En gran parte esto es debido a la gran compatibilidad que tiene con todas las plataformas y programas de terceros. Una de sus mayores utilidades, es la posibilidad de volver a versiones antiguas de un mismo código sin necesidad de realizar copias de seguridad de forma manual. Para crear un paquete de ROS con GitHub se deben seguir los siguientes pasos.

2.4.3.1. Crear el repositorio

Tras haberse creado una cuenta en la plataforma, hay que dirigirse a la página principal; donde en la zona superior izquierda de la pantalla aparecerán todos los repositorios existentes a los que se tienen acceso. Si no se ha trabajado previamente con GitHub, esta lista aparecerá vacía. En esa zona se clica sobre "new" para crear un repositorio nuevo.

Para crear un nuevo repositorio, debemos especificar una serie campos (ver Figura 9). Es recomendable que el nombre del repositorio coincida con el nombre del paquete de ROS que se vaya a desarrollar, puesto que cuando el resto de usuarios lo busquen, aparecerá el nombre del repositorio, no del paquete.

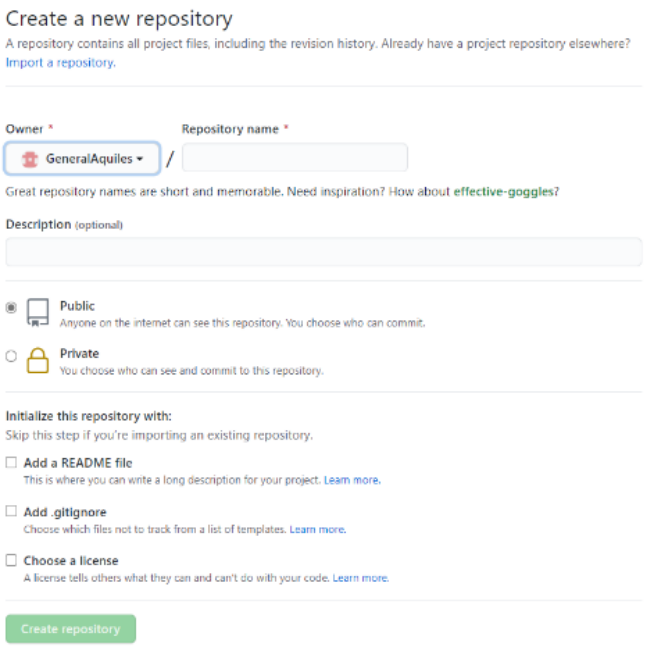

Figura 9. Opciones de configuración de un nuevo repositorio. Puede incluirse una descripción que se mostrará en la página principal de este. Un repositorio puede definirse público o privado, la modificación de este parámetro implica la pérdida del número de visitas al repositorio.

Para continuar con la filosofía de ROS, es recomendable declarar el repositorio como público, así cualquier persona puede verlo. Sin embargo, no es necesario ya que puede modificarse la visibilidad del repositorio en cualquier momento. Por último, es una práctica común incluir en el repositorio un archivo *readme* donde se puede dar algo de información e instrucciones de uso del paquete. Además, existe la posibilidad de añadir una licencia de uso para indicar a los futuros usuarios que permisos tienen en lo referente al uso que se le permite dar a dicho paquete (ver Figura 9. Opciones de configuración de un nuevo repositorio. Puede incluirse una descripción que se mostrará en la página principal de este. Un repositorio puede definirse público o privado, la modificación de este parámetro implica la pérdida del número de visitas al repositorio.).

2.4.3.2. Descargar el repositorio

Ya tenemos creado el repositorio, ahora se procede a descargar el repositorio en nuestra maquina local. Para ello, seleccionamos el botón verde que indica *code*, visible en la Figura 8. Panel de información general de un repositorio. Desde aquí, puede tenerse acceso a las carpetas del repositorio, obtener información sobre los lenguajes de programación empleados y seleccionar la rama que visualizar. En el botón verde puede obtenerse un enlace para clonar el repositorio, o descargarlo directamente en formato comprimido., y copiamos en el portapapeles el enlace del repositorio. A continuación, en nuestra maquina local nos movemos a la carpeta *src* del *workspace* desde una ventana de comandos. Una vez en el sitio deseado, clonamos el repositorio con el enlace adquirido.

```
Git clone https://github.com/GeneralAquiles/TopoMapping.git
```

A continuación, ya se puede crear contenido dentro de la carpeta que se habrá generado en *src* con el nombre del repositorio.

2.4.3.3. Subir cambios al repositorio

Para subir a GitHub cualquier cambio que se realice en nuestra copia local del repositorio, es necesario seguir una serie de pasos.

Lo primero de todo es asegurarse de que se ha configurado *git* correctamente para realizar cambios. Para ello, se indicará en nombre de usuario de *git* y correo, además de habilitar la salida con color para una mejor visualización.

```
$ git config --global user.name "tu nombre de usuario"
$ git config --global user.email "tu email"
$ git config --global color.ui true
$ git config -global core.editor emacs
```

Si es la primera vez que se va a realizar un *commit* a este repositorio, es necesario establecer una clave *ssh*. Si aún no se posee ninguna, se genera automáticamente con el comando que se muestra a continuación y se guarda con el nombre que se desee. Una vez hecho esto, se accede al archivo que se ha generado y se copia la clave.

```
$ ssh-keygen -t rsa -C "tu email"
```

A continuación, en GitHub debemos acceder a la configuración de nuestro perfil, al apartado *SSH and GPG keys*. Una vez allí, le damos a añadir nueva clave SSH, copiamos la clave y le damos un nombre. Una vez hecho esto, ya se pueden subir cambios al repositorio. Para asegurarse de que se ha realizado el proceso correctamente, puede escribirse el siguiente comando, donde es posible que pregunte por una confirmación. Al aceptar la confirmación, si todo ha ido bien, debe salir el siguiente mensaje:

```
Hi <tu usuario>! You've succesfully authenticated, but GitHub does not provide shell access.
```

A partir de este momento, los pasos para subir cambios serán siempre los mismos. En primera instancia se deben añadir los cambios que se desean subir, a este proceso se le llama *stage*. Una vez hecho esto, debe crearse un *commit* con un nombre que sea representativo de los cambios que se estén realizando. Por último, se suben los cambios a la rama deseada.

```
$ git add ./<carpeta>/<cualquier fichero>
$ git status   #Sirve para comprobar que se han añadido todos los archivos deseados
$ git commit -m "Nombre del commit"
$ git log -n   #Permite ver todos los commits realizados hasta la fecha
$ git push origin <nombre de la rama>
```

Con estas acciones ya estarán los cambios subidos a la rama que se haya indicado y visibles para todo el que tenga acceso.

2.5. Gazebo

Gazebo (Gazebo Sim, s.f.) es una herramienta imprescindible para el desarrollo de un paquete de ROS. Es una aplicación donde se puede simular el comportamiento del robot sin necesidad de tenerlo físicamente, acelerando el proceso de prueba-error. El motor gráfico de gazebo incluye físicas muy completas, como lo son los cálculos de gravedad, inercia e iluminación. Gracias a estas características puede incluirse un modelo virtual del robot que se esté desarrollando bastante completo, con todas las articulaciones, sensores, masa, texturas, etc. A su vez, Gazebo permite la inclusión de objetos 3D o incluso crearlos en tiempo de ejecución. Esto da grandes posibilidades al testeo del comportamiento de un robot, ya que se puede crear un entorno de pruebas fiel a la realidad, con edificios, mobiliario e iluminación.

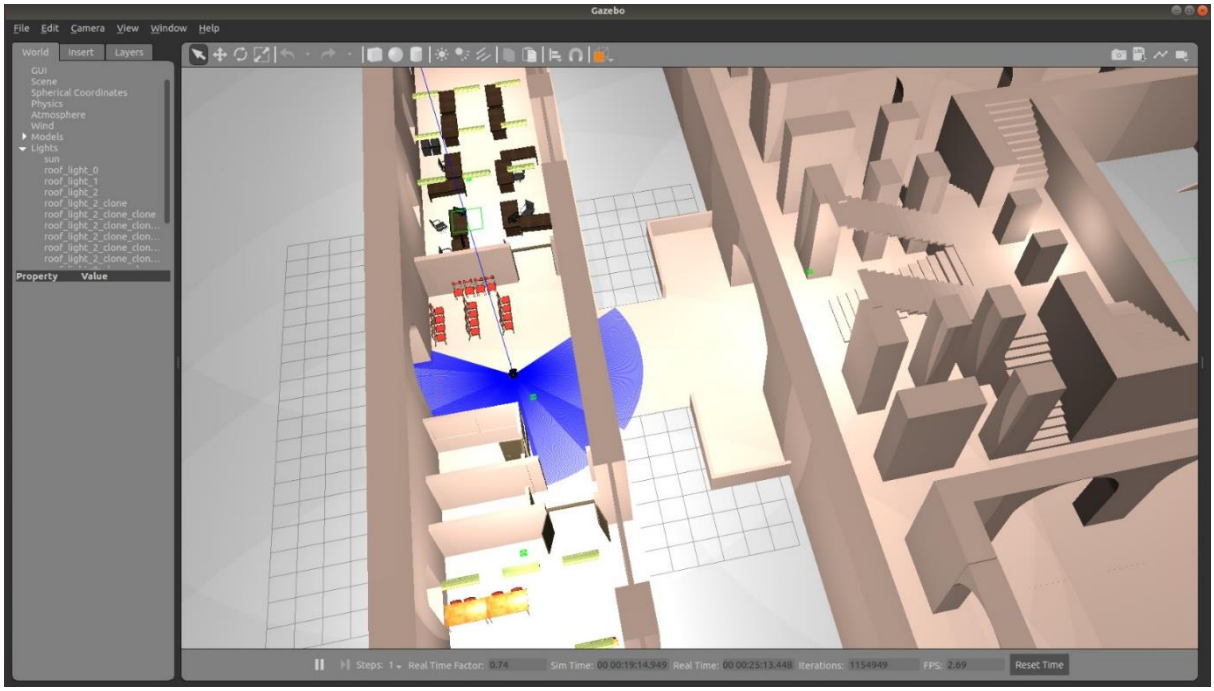

Figura 10. Captura de la ventana principal de gazebo con un mundo cargado. En este programa se pueden renderizar varios objetos 3D simultáneamente y simular el comportamiento del robot con respecto a estos.

2.5.1. World

Por defecto, Gazebo genera un mundo vacío cuando se inicia. Tal y como se ha mencionado anteriormente, dentro de la aplicación se pueden insertar figuras geométricas e iluminación, modificar su tamaño, color y posición. Esto es útil para hacer cambios en tiempo de ejecución. Sin embargo, es más eficiente tener un archivo predefinido con todos los modelos que se desean incluir en el simulador. De esta manera no se tienen que meter manualmente cada vez que se arranque la aplicación. A este archivo se le denomina *world file*.

Un *world file* está compuesto por una serie de objetos en formato *json*, que definen la inclusión de distintos modelos 3D. En este documento se puede especificar la posición y rotación del objeto. Cuando se desea incluir un modelo más de una vez, debe especificarse un nombre distinto para cada pieza, a modo de identificación. A su vez, en el modelo pueden incluirse varios tipos de iluminación. Esto es especialmente útil en edificios de gran tamaño con varias estancias, ya que se generarán sombras que empeoran el desempeño de la cámara RGB.

```
<?xml version="1.0" ?>
<sdf version="1.4">
       <world name="default">
       <include>
              <uri>model://(nombre del modelo)</uri>
              <name>model_N</name>
              <pose>x y z θx θy θz</pose>
       </include>

       …

        <light type="point" name="light_N">
            <pose> x y z θx θy θz</pose>
            <diffuse>100 100 100 255</diffuse>
            <specular>25 25 25 255</specular>
            <attenuation>
                <range>20</range>
               <linear>0.1</linear>
               <constant>0.5</constant>
               <quadratic>0.001</quadratic>
           </attenuation>
           <cast_shadows>true</cast_shadows>
        </light>

       …

  </world>
</sdf>
```

De esta manera, cada vez que se lance Gazebo con este fichero, se cargarán todos los modelos incluidos en él en la posición especificada, tal y como puede verse en la Figura 10.

```
Roslaunch gazebo gazebo_world
```

2.5.2. Modelo

Previamente se ha mencionado el concepto de modelo. En Gazebo se denomina modelo a un objeto 3D, que esté definido dentro de las librerías de Gazebo, concretamente en la carpeta *models*. Dentro de esta carpeta, debe crearse un nuevo subdirectorio para cada modelo que se desee crear. El nombre que tenga el directorio debe coincidir con el nombre deseado para el modelo. Dentro de esta carpeta se incluyen una serie de elementos que son los que definen el objeto (ver Figura 11).

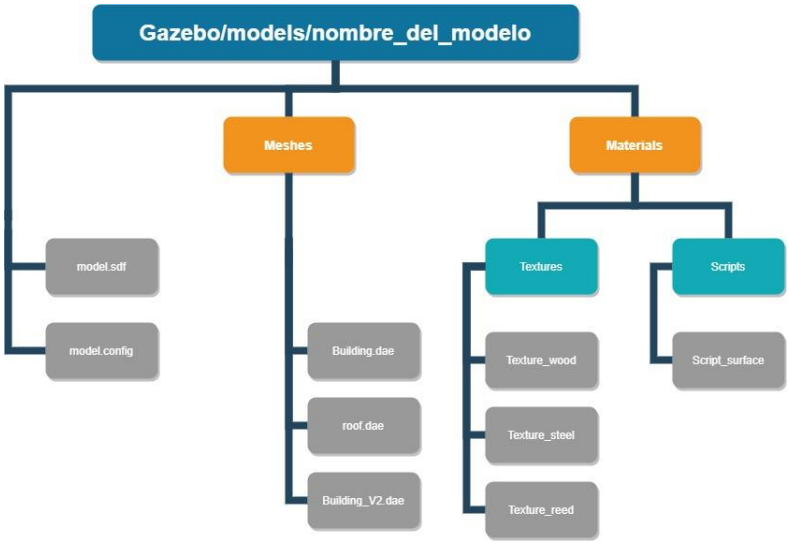

Figura 11. Distribución de archivos dentro de la definición de un modelo 3D para gazebo.

2.5.2.1. Meshes

La carpeta *meshes* es donde se debe alojar el objeto 3D en sí. Puede utilizarse tanto un archivo descargado de repositorios públicos como uno de diseño propio. El único formato que acepta Gazebo para un elemento 3D es el formato *Collada* (.dae).

2.5.2.2. Textures

Existe otra carpeta dentro del directorio de un modelo, esta es la carpeta `textures`. Como su nombre indica, en ella se encuentran las texturas del objeto 3D en cuestión. En este caso, Gazebo acepta prácticamente cualquier formato de imagen, el más usado para las librerías de modelo es *PGN*. Cuando se cargue el modelo 3D, el archivo collada lleva una instanciación al nombre de la textura para poder cargarla, por lo que hay que asegurarse de que estos coincidan.

2.5.2.3. Model.sdf

Se trata de un archivo de configuración del modelo. En él se especifica el nombre del modelo, qué archivo Collada escoger para el mallado del objeto y cuál escoger para la malla de colisión, ya que en ciertos casos puede ser interesante que no sea el mismo. En este fichero también se da la opción de modificar la posición del objeto, hay que tener en cuenta que esta posición es respecto al origen del archivo, no del mundo en el que se incluye.

2.5.2.4. Model.config

Una vez más, este archivo tiene la intención de aportar una breve explicación del modelo. En él se añade información como el nombre del autor, correo de contacto y licencia de uso.

2.5.3. Crear un modelo 3D

Para crear un modelo 3D uno de los programas más ampliamente usados en la industria es SolidWorks (Dassault Systems, s.f.). Se trata de un software de alto grado de profesionalidad que permite hacer cualquier diseño, tanto desde plano a 3D, como en 3D directamente. Además, este software permite hacer estudios de esfuerzo para el modelo, así como emplear una amplia gama de texturas.

Este programa usa un formato propio para sus archivos, denominado *SLDPRT* (SolidWorks Part) o *SLDASM* (SolidWorks Assembly). El gran hándicap para esta aplicación es que SolidWorks no es compatible con el formato *Collada*. *Collada* es un formato de objeto 3D público, por lo que cualquier software de CAD gratuito puede tener compatibilidad con él. Desgraciadamente, al ser un formato simple en cuanto a detalle del objeto, no es ampliamente usado en la actualidad, por lo que es más difícil de encontrar un programa que permita exportar a dicho formato. Finalmente, un software libre que si tiene opción de exportación a *Collada* es FreeCAD (FreeCAD, s.f.).

Por tanto, para la obtención de un objeto 3D en Collada se deben seguir los siguientes pasos.

- Diseñar el objeto en SolidWorks.
- Exportar el objeto en formato *STL*.
- Importar el *STL* en FreeCAD.
- Exportar el documento como *Collada*.

2.6. RVIZ

Se trata de una herramienta gráfica que permite visualizar los diferentes mensajes de los *topics* que estén siendo generados por los nodos en ejecución (RosWiki, s.f.). Este programa resulta una herramienta muy potente a la hora de comprobar el correcto funcionamiento del robot.

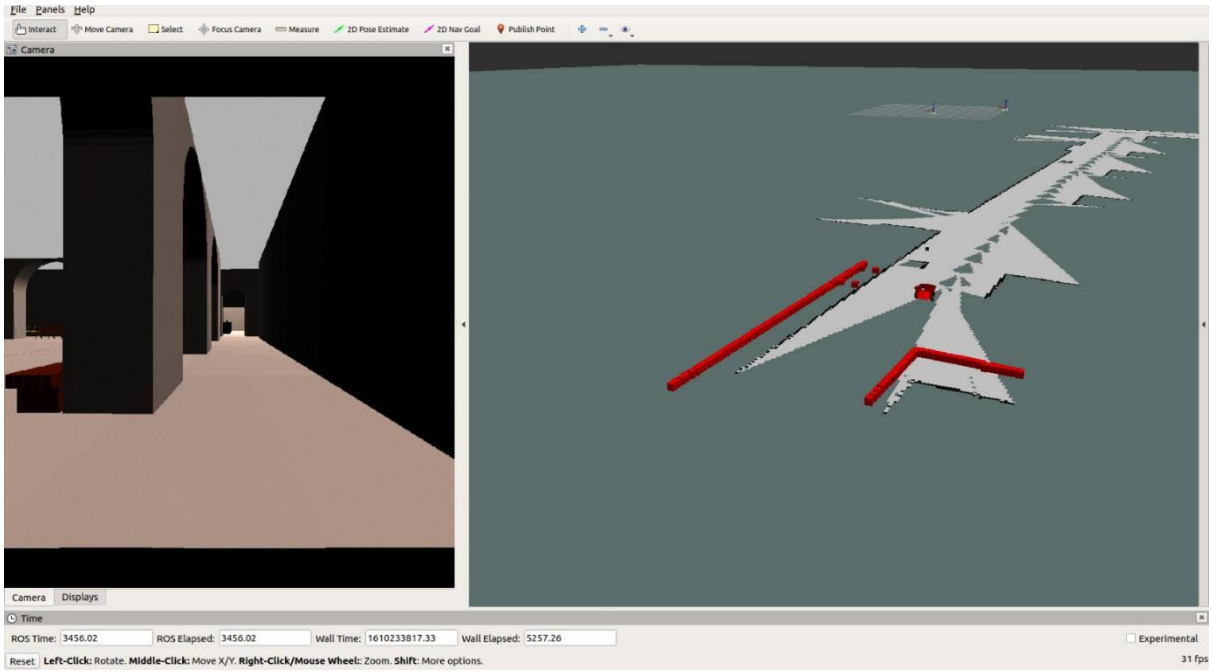

Figura 12. Captura de la herramienta de visualización RVIZ. En esta imagen se muestra la cámara del robot (izquierda) y la posición de este sobre el mapa que se está generando (derecha). Los puntos rojos son las distintas medidas obtenidas por el LIDAR.

Cuando se desea utilizar el programa, es necesario establecer un plano o eje de referencia desde el que visualizar dichos *topics*. Como ya se ha comentado previamente, ROS basa el comportamiento del robot en ejes de coordenadas móviles; por lo que, si se desea visualizar información, es necesario indicar respecto a qué eje se quiere recibir esta. De esa manera el propio sistema de Ros se encarga de realizar los cálculos pertinentes, sin necesidad de que el desarrollador tenga que preocuparse por ese aspecto de la programación.

Una vez establecido el plano de referencia, solo hay que añadir los *topics* que se deseen mostrar. En estos *topics* se incluye cualquiera que contenga información gráfica, puede ser desde el modelo del robot, hasta la trayectoria que va a realizar; pasando por las medidas del láser, el mapa que se está generando, o incluso la covarianza de la posición del robot en dicho plano.

De hecho, puede incluso servir como pantalla de monitoreo sobre el estado de los robots cuando estos ya estén en etapa de uso, ya que puede cargarse el plano de la estancia en la que se encuentre e incluir tantos modelos de robot como unidades haya en funcionamiento, puesto que cada una publica su propio modelo. De esta manera puede tenerse un control en todo momento de donde está cada unidad y hacia dónde se dirige.

2.6.1. Plano de referencia

Un modelo de robot se encuentra divido por partes y articulaciones, para que estas puedan ser movidas y modificadas si están motorizadas. A su vez, un robot incluye sensores, los cuales extraen información de su entorno, pero siempre respecto a ellos mismos. Por tanto, para trabajar con un robot como un todo, es necesario establecer un punto global para todo el robot. Este punto general se denomina eje de referencia móvil del robot, que recibe el nombre estándar de *base_link*. Esto es debido a que, por convenio, este eje de referencia se establece en la base del robot, normalmente en el centro del eje de giro del mismo. Como ya se ha explicado en apartados anteriores, cada pieza y sensor del robot deberá tener una *transformada* respecto a este eje de referencia. Si se cumplen estas condiciones, al establecer la base del robot como eje de referencia en RVIZ podrá verse toda la información respecto al robot. De esta manera, las medidas del

láser se ven a la distancia correcta. Al haber establecido *base_link* como eje de referencia, si se está trabajando con un proceso de mapeado, se podrá visualizar el mapa generado sin problema. Sin embargo, cuando el robot se desplace, lo que se moverá en el visualizador no será el modelo, si no el mapa; dando asi la sensación de estar montado sobre el robot.

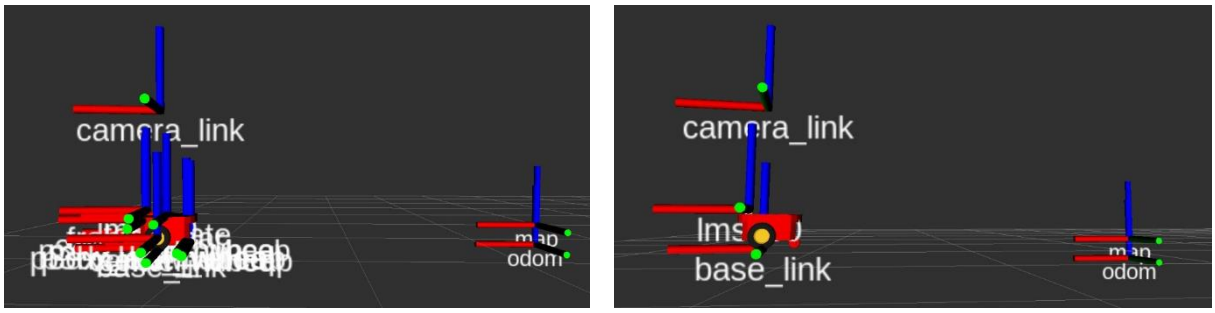

Figura 13. Representación de los ejes de coordenadas de todos los frames disponibles (izquierda) y discretización de los frames relevantes para el desarrollo (derecha)

Cuando se desea visualizar el mapa estático, y que sea el robot el que se mueva, se debe cambiar el plano de referencia al del mapa; el cuál normalmente es *map*. En esta situación, sucederá todo lo contrario. El robot será el que desplace sobre el mapa generado, obteniendo así una visualización más realista. Como ya se ha mencionado, los distintos sensores poseen relaciones de posición respecto a la base del robot, por lo que sus medidas se irán desplazando en concordancia con este.

2.6.2. Incluir elementos

Para modificar el plano de referencia, hay que dirigirse al desplegable de la zona izquierda del visualizador. En él, habrá un primer apartado denominado *global options*. Dentro de este apartado, el primer elemento es *fixed frame*, con un desplegable a su vez. Ese es el plano de referencia que se está usando. En este segundo desplegable, se mostrarán todos los nombres de los distintos planos o ejes de referencia que existan durante la ejecución, tanto fijos como móviles. Basta con seleccionar el deseado.

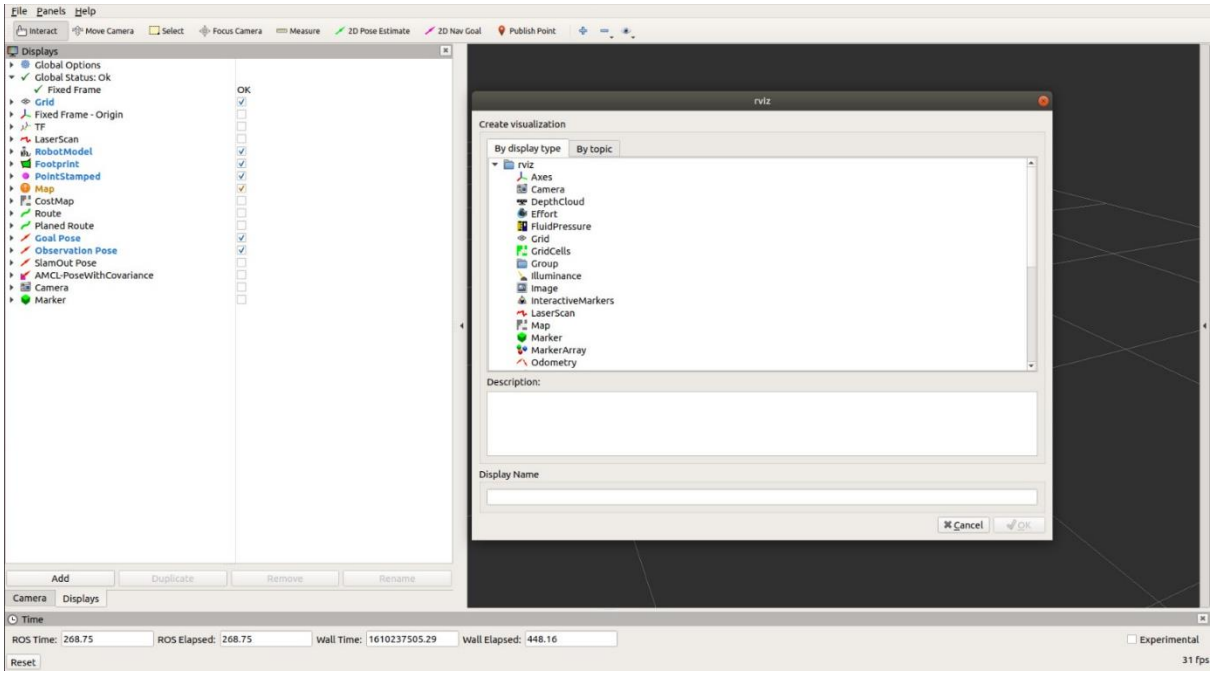

Figura 14. Panel de visualización de los topics activos y ventana de selección de nuevos topics.

A la hora de incluir un *topic*, en la parte inferior del desplegable usado anteriormente aparecerá un botón denominado *add* (ver Figura 14). Al hacer clic, se abrirá una ventana donde se podrá elegir que *topic* mostrar, teniendo la opción de ordenarlos por tipo de mensaje (de pose, de láser, de mapa, etc.), o por nombre. Se selecciona el deseado y aparecerá en el listado del desplegable inicial. Si el *topic* ha sido bien configurado respecto al plano de referencia elegido, la información debe mostrarse automáticamente.

Por último, si se ha incluido una cámara en el modelo del robot, esta puede visualizarse si se desea. Basta con añadirla como un *topic* más. Una vez se haya añadido, aparecerá una segunda pestaña en el desplegable derecho, donde se estará mostrando la imagen recibida de la cámara.

2.7. Mapeado métrico

Un mapa o plano métrico, es una representación gráfica exacta a escala de la realidad. En el ámbito de la robótica móvil, un mapa métrico representa la geometría del entorno, ya sea exterior o de interior, en el que el robot se desplaza. Para el proceso de mapeado se emplean diversos sensores, donde destacan por su uso habitual un sensor láser tipo LIDAR y los sensores de *odometría* de los que este dispuestos el robot. En él, se emplean las medidas de *odometría* para calcular la variación de posición de la unidad robótica y de esta manera obtener la relación de posición respecto a la cual reconstruir el mapa a partir de las medidas de láser, que representarán obstáculos del entorno.

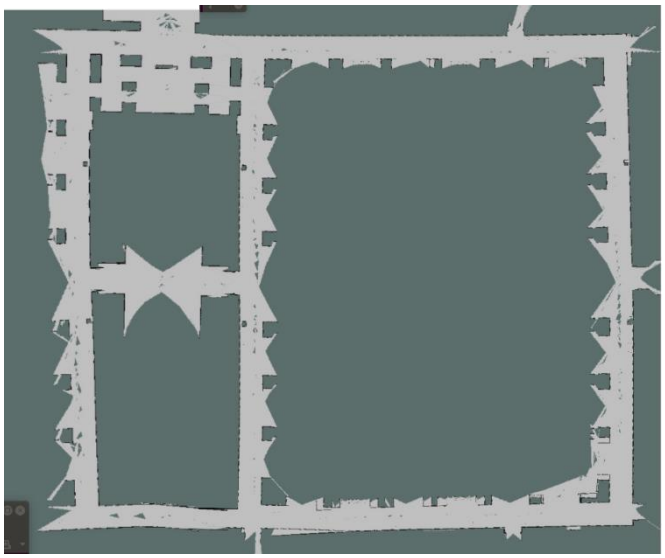

Figura 15. Mapa métrico del patio este, entrada a graderío y pasillo del patio oeste generado mediante procesos de SLAM. En estos mapas se tiene una posición exacta de cada punto, pero se desconoce completamente su significado y relevancia.

2.8. Mapeado semántico

Un mapa o plano semántico, es una relación gráfica de conceptos o elementos. Se trata de un método de organización muy útil para la clasificación de entidades. En el ámbito robótico, estos planos pueden relacionar elementos como son el número de estancias y las interconexiones entre ellas, o que tipos de objetos hay en su interior. Se trata de mapas muy ricos en información para el usuario, con el añadido de su fácil lectura. Sin embargo, esta metodología sufre de la abstracción total de la geografía del entorno, por lo que para tareas de orientación no es el más idóneo.

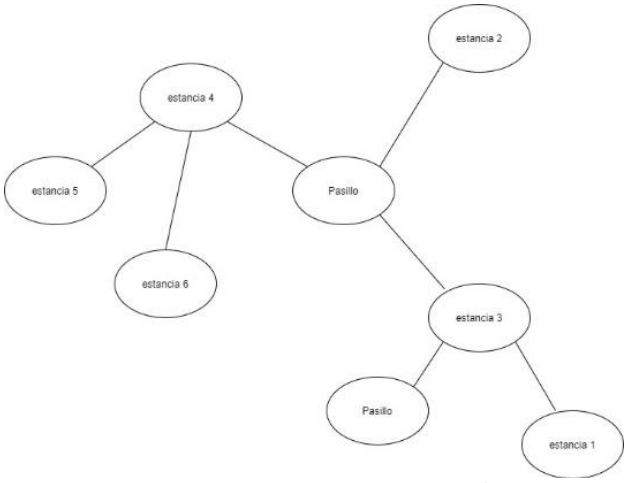

Figura 16. Ejemplo representativo de un mapa semántico de varias estancias. En este mapa se guarda una relación de conexión entre los elementos, pero no se tiene información sobre la posición de estos.

2.9. Mapeado híbrido

El proceso de mapeado híbrido, de ahora en adelante mapeado topológico, resulta de la fusión de sendos métodos (ver Figura 17). En estos mapas, se reduce el nivel de precisión métrica a favor de extraer de ella la información clave que se necesita para una correcta localización y estructuración del entorno. A su vez, añaden información semántica que ayuda a interpretar de una manera más lógica el mapa. De esta manera se alcanza un nivel de abstracción medio del entorno, en el que se pueden establecer relaciones lógicas entre estancias, sin llegar a perder la morfología del terreno.

Estos mapas, al estar estructurados semánticamente, permiten la inclusión de información extra obtenida por otros medios, como puede ser una cámara con visión artificial. Esta información puede ayudar a obtener más información del entorno, como de qué tipo de estancia se puede tratar, el aforo, etc.

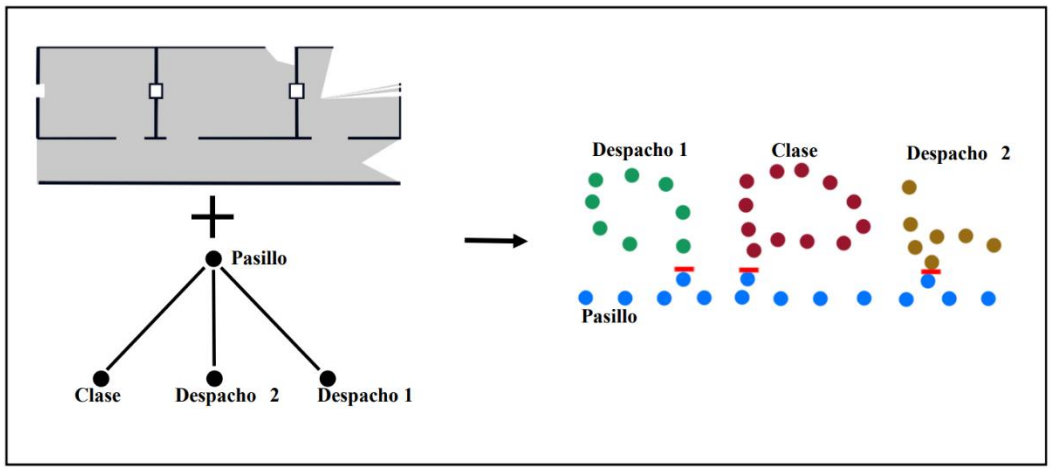

Figura 17. En un mapa topológico, se fusiona un mapa métrico (superior) con un mapa semántico (inferior), extrayendo información relevante de ambos y relacionándola. De esta manera se obtiene un mapa mucho más rico en información que los otros dos por separado. A simple vista puede extraerse una relación de conexión entre las estancias, así como su tamaño, forma y nombre.

2.10. Google Cloud Services

La empresa de electrónica y servicios informáticos Google ofrece desde hace un tiempo en su plataforma Google Cloud una serie de más de 100 servicios de almacenamiento, tratamiento y procesado de datos en la nube (Google, s.f.). Se tratan de servicios de pago, en los que se cobra por lotes de servicio, ya sean *requests* a una unidad de procesamiento de imagen o una cantidad determinada de espacio de almacenamiento en la nube. Sin embargo, en la mayoría de estos servicios se ofrece un periodo de prueba que es más que suficiente para acelerar la etapa de desarrollo de prototipos que requieren esta clase de productos.

Al utilizar servicios en la nube, se puede reducir la carga de procesamiento en las unidades robóticas móviles. De esta manera se puede dedicar el grueso de la capacidad de procesamiento a tareas más críticas como el control del desplazamiento del robot. A su vez permiten abaratar costes puesto que, en caso de trabajar con visión artificial, ya no es necesario que el hardware incorpore una unidad de procesamiento grafico dedicada. De esta manera se pueden proporcionar unidades más asequibles al usuario final.

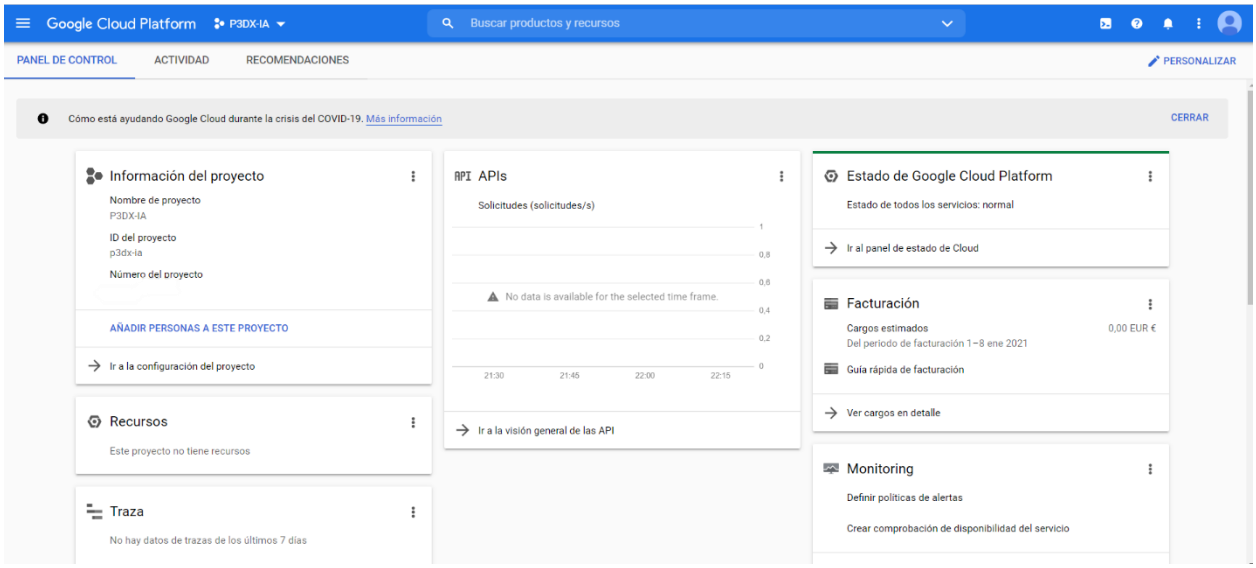

Figura 18. Panel de control de los servicios de Google Cloud. En esta ventana puede obtenerse información rápida del estado del proyecto, la facturación y el consumo que las distintas APIs han experimentado en un grafo temporal.

Algunos de los productos más usados proporcionados por Google son Cloud Storage, Cloud SQL, Speech-to-Text, Text-to-Speech, Cloud Translation y Vision AI. En este proyecto se ha hecho uso de esta última, Vision AI.

2.10.1. Vision AI

Se trata de un servicio en la nube de Google (Google Cloud Services, s.f.) que consta de una plataforma a la que se pueden subir imágenes. Estas imágenes pueden subirse de forma gráfica, mediante el uso de navegador, o realizando una petición *http* desde un script, empleando las librerías que proporciona la compañía. Una vez cargadas, la aplicación emplea una gran serie de modelos de aprendizaje automático, entrenados previamente, para detectar varios tipos de elementos en la imagen. Los elementos que esta API es capaz de detectar son:

- Detección de rostro. Incluyendo detección de la expresión facial e inclinación de la cara en tres dimensiones.
- Detección de objetos. Incluyendo nombre del objeto y posición de los vértices del rectángulo envolvente.
- Detección de texto. Incluyendo texto, tamaño y posición.
- Deteccion de logo. Extrae logos de empresas.
- Detección de localización geográfica.
- Detección de contenido inapropiado. Dando un grado de probabilidad para cada nivel de contenido.
- Generación de etiquetas temáticas. Extrae el contexto de la imagen, la categoría de los objetos que se pueden reconocer en ella, el estilo de la imagen, incluso conceptos más abstractos como el hecho de enseñar.
- Extracción de secciones relevantes. Recorta la sección de la imagen de donde se ha podido extraer más información.
- Propiedades de la imagen. Como es el caso de la gama cromática de la imagen y el color predominante.
- Detección web. Detecta etiquetas relevantes de la imagen y muestra búsquedas relacionadas en la web.

Para establecer una comunicación con esta API desde un nodo de ROS, es necesario realizar una petición *http* a la *url* del servicio, en la que se envíe un archivo en formato *JSON*. En este archivo se debe especificar qué tipo de detecciones se desean realizar de las mencionadas anteriormente. Esto es debido a que cada tipo de detección se interpreta como una petición independiente. Una vez procesada la imagen, la Api devolverá otro archivo en formato *JSON* con los resultados detectados, y sus probabilidades asociadas, agrupados por tipo.

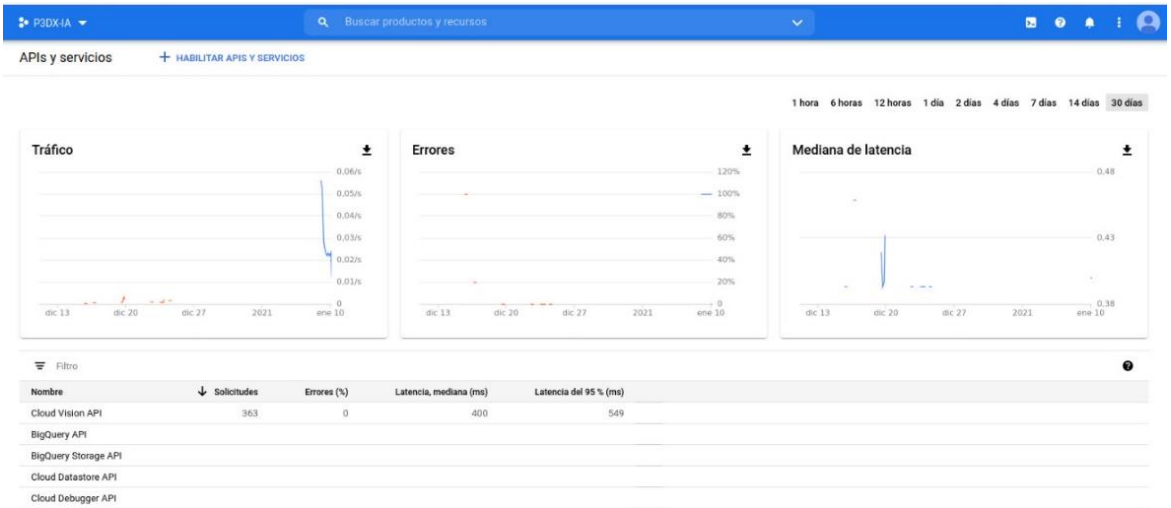

Figura 19. Panel de información del tráfico de peticiones sobre las distintas APIs que se tienen a disposición. En el grafo de tráfico, puede verse claramente cuando se ha realizado una prueba de su uso al crecer instantáneamente este.

Como los demás servicios de Google, este es un servicio de pago. Sin embargo, se ofrecen las primeras 1000 peticiones de forma gratuita, cada mes. Las peticiones no son acumulables, por lo que se pueden realizar 1000 peticiones de cada tipo. Por tanto, se puede hacer un total de 1000 detecciones de imágenes al mes de forma gratuita, para cada tipo de detección. Una vez pasado este límite, hay una tabla de precios para cada tipo de detección. Los precios asignados a cada elemento pueden verse en la Tabla 1.

Tabla 1. Precios de API Vision AI desglosados por servicio y cantidad de peticiones

Función	**Precio por cada 1000 unidades**		
	Primeras 1000 unidades al mes	**De 1001 a 5.000.000 de unidades al mes**	**De 5.000.001 a 20.000.000 de unidades al mes**
Detección de etiquetas	Gratis	1,50 USD	1,00 USD
Detección de texto	Gratis	1,50 USD	0,60 USD
Detección de texto en documentos	Gratis	1,50 USD	0,60 USD
Detección de Búsqueda Segura (contenido explícito)	Gratis	Gratis con la detección de etiquetas, o bien 1,50 USD	Gratis con la detección de etiquetas, o bien 0,60 USD
Detección facial	Gratis	1,50 USD	0,60 USD
Detección facial: reconocimiento de famosos	Gratis	1,50 USD	0,60 USD
Detección de puntos de referencia	Gratis	1,50 USD	0,60 USD
Detección de logotipos	Gratis	1,50 USD	0,60 USD
Propiedades de la imagen	Gratis	1,50 USD	0,60 USD
Pistas de recorte	Gratis	Gratis con propiedades de la imagen, o bien 1,50 USD	Gratis con propiedades de la imagen, o bien 0,60 USD
Detección web	Gratis	3,50 USD	-
Localización de objetos	Gratis	2,25 USD	1,50 USD

2.11. SolidWorks

Se trata del software de CAD por excelencia para el ámbito de diseño en ingeniería (Dassault Systems, s.f.). Permite la realización de piezas tanto desde planos 2D, que posteriormente generan de forma automática el objeto 3D, como la construcción de este directamente en 3D. A su vez, da posibilidad a generar planos de cotas completamente configurables para que coincidan con el diseño deseado. Por otro lado, influye gran cantidad de simulaciones para mejorar el desarrollo del producto sin necesidad de fabricarlo, como son ensayos de esfuerzo, simulaciones de temperatura y simulaciones aerodinámicas, entre otras.

En otro ámbito, SolidWorks incorpora una serie de paquetes extra que le permiten la integración con otras etapas del desarrollo del producto, como es el caso de SolidWorks Electrical. Este plugin permite la importación de componentes electrónicos diseñados en otras plataformas comerciales con las que tiene establecido *partnership*, como es el caso de Altium Designer. Además, permite el trazado del cableado en 3D, seleccionando el número de conductores, la sección, radio de curvatura, etc. Otro plugin interesante es el de renderizado, permitiendo aplicar texturas a la pieza, cara por cara, además de la inclusión de escenarios e iluminación. De esta manera permite obtener una imagen bastante fiel a la realidad de cómo se verá el objeto una vez finalizado.

Cabe mencionar que esta aplicación también permite la realización de ensamblajes a partir de piezas previamente diseñadas, los que aporta gran eficacia a la hora de modificar diseños complejos que incorporan gran cantidad de elementos.

Por último, cabe destacar que SolidWorks tiene la capacidad de importar y exportar archivos en la mayoría de los formatos 3D más usados en la industria; como son *STEP*, *STL*, *IGS*, *3DXML*, *OBJ*.

3. Materiales y métodos

3.1. Modelo del Robot – P3DX

El robot sobre el que se ha basado el desarrollo de este proyecto es el Pioneer 3DX (Jelinek, s.f.), un modelo de Omron de propósito genérico. Este modelo en concreto está constituido por dos ruedas motorizadas y una tercera rueda conducida. A su vez, posee motorización diferencial, es decir, cada motor se controla por separado. Existen varias opciones de configuración, la escogida para este proyecto incluye un array de sensores de ultrasonidos frontales, un LIDAR en el frontal del robot, (Hokuyo Sensors, s.f.) y una cámara RGB colocada a un metro de altura.

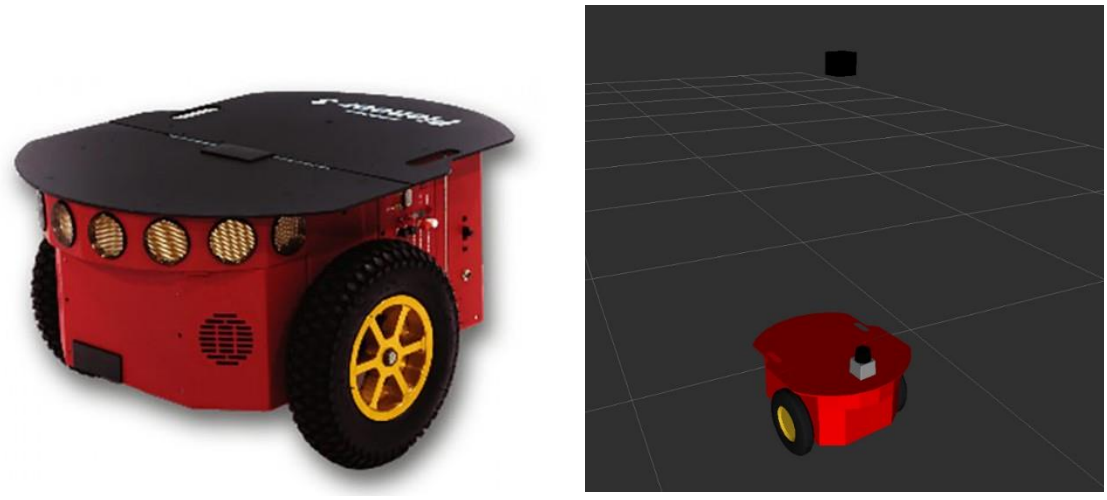

Figura 20. Modelo robótico Pioneer 3DX (izquierda) y su representación simplificada (derecha) en el visualizador RVIZ. En esta representación se incluye un LIDAR (frontal del robot) y una cámara RGB (situada a un metro de altura).

En los repositorios de GitHub de ROS se ofrece un paquete con los sólidos del modelo y la configuración necesaria para su implementación tanto en Gazebo como RVIZ. Dicho paquete, se encuentra dividido en 3 carpetas, que se explican a continuación.

3.1.1. p3dx_control

Este apartado contiene la configuración necesaria para el control de la tracción del robot. Contiene dos subcarpetas:

- Control: Contiene el archivo *pioneer3dx.yaml*, donde se especifican los distintos parámetros para los controladores, como la frecuencia de publicación, posición y unión del robot a la que se corresponden.
- Launch: Contiene el archivo *control.launch*. Este archivo permite carga la configuración especificada en *pioneer3dx.yaml*, para después lanzar el paquete *controller_manager* con dichos argumentos. Por último, lanza el paquete *robot_state_publisher*, que se encarga de generar las transformadas de las distintas uniones del robot.

3.1.2. p3dx_description

Aquí es donde se encuentran alojados los modelos 3D de las distintas partes del robot y la definición de las uniones y relaciones de posición entre estas. Se divide a su vez en 3 subcarpetas:

- launch: En él, se aloja el archivo *rviz.launch*, el cual lanza una instancia de *RVIZ* donde se cargará el modelo del robot.
- meshes: Esta carpeta engloba todos los archivos 3D de las distintas partes del modelo.
- urdf: Se trata de la carpeta de definición del modelo en sí. En ella nos encontramos una serie de archivos donde se indica como reconstruir el robot. Los archivos son:
    - *pioneer3dx.xacro* es un archivo en formato macro basado en xml donde se especifica las distintas partes del modelo del robot. Estas partes se organizan por *links*. Esto es debido a que dichos links luego son empleados por el *robot_state_publisher* para generar las transformadas de cada parte. En cada link, se define un nombre, una posición, masa del objeto, archivo 3D correspondiente a la figura visual y archivo 3D correspondiente a figura de colisión.
    - *pioneer3dx_wheel.xacro* es otro archivo de especificación de links, en este caso específico para la tracción. Esto es debido a que la definición es bastante extensa y puede separarse del resto de la definición.
    - *materials.xacro* es un apartado para la creación de texturas que posteriormente pueden ser usadas en las piezas del robot. Aquí puede definirse el tono, reflectividad, etc. Además, se le asocia un nombre a cada configuración de color para que posteriormente sea más sencillo asignarlas a las piezas.
    - En *pioneer3dx.gazebo* se indica la configuración de cada parte del robot. En este documento es donde se pueden aplicar las texturas definidas previamente en *materials.xacro* a cada sólido del robot. Por otro lado, aquí se configuran los parámetros para los controladores de los distintos sensores y actuadores del modelo. En estos parámetros se incluyen el nombre del *topic* sobre el que publicar los mensajes, el *frame* al que están referenciados, la frecuencia de publicación, asi como parámetros de ruido y precisión para obtener unas medidas con mayor o menor error y aleatoriedad y, de esta manera, ajustarse en mayor medida a la realidad.

El paquete de base no incluye una cámara, por lo que es necesario añadirla a la descripción de este. Para ello, se añaden en *pioneer3dx.xacro* las siguientes líneas de código:

```
<link name="camera_link">

         <collision>
                  <origin xyz="0 0 0" rpy="0 0 0"/>
                  <geometry>
                     <box size="0.05 0.05 0.05"/>
                  </geometry>
         </collision>
         <visual>
                  <origin xyz="0 0 0" rpy="0 0 0"/>
                  <geometry>
                     <box size="0.05 0.05 0.05"/>
                  </geometry>
                  <material name="red"/>
         </visual>
         <inertial>
                  <mass value="1e-5" />
                  <origin xyz="0 0 0" rpy="0 0 0"/>
                  <inertia ixx="1e-6" ixy="0" ixz="0" iyy="1e-6" iyz="0" izz="1e-6" />
         </inertial>
</link>

<joint name="camera_joint" type="fixed">
         <axis xyz="0 1 0" />
         <origin xyz="0 0 1" rpy="0 0 0"/>
         <parent link="base_link"/>
         <child link="camera_link"/>
</joint>
```

Por otro lado, en *pioneer3dx.gazebo* se añade el siguiente trozo:

```
<gazebo reference="camera_link">
         <visual>
              <material>
                     <ambient>0.5 0.5 0.5 1.0</ambient>
                     <diffuse>0.0 0.0 0.0 1.0</diffuse>
                     <specular>0.0 0.0 0.0 128</specular>
                     <emissive>0.0 0.0 0.0 1.0</emissive>
              </material>
         </visual>

         <sensor type="camera" name="camera1">
           <update_rate>30.0</update_rate>
           <camera name="head">
              <horizontal_fov>1.3962634</horizontal_fov>
              <image>
                     <width>800</width>
                     <height>800</height>
                     <format>R8G8B8</format>
              </image>
              <clip>
                     <near>0.02</near>
                      <far>300</far>
              </clip>
              <noise>
                <type>gaussian</type>
                <!-- Noise is sampled independently per pixel on each frame.
                     That pixel's noise value is added to each of its color
                     channels, which at that point lie in the range [0,1]. -->
                <mean>0.0</mean>
                <stddev>0.007</stddev>
              </noise>
           </camera>

           <plugin name="camera_controller" filename="libgazebo_ros_camera.so">
              <alwaysOn>true</alwaysOn>
              <updateRate>0.0</updateRate>
              <cameraName>rrbot/camera1</cameraName>
              <imageTopicName>image_raw</imageTopicName>
              <cameraInfoTopicName>camera_info</cameraInfoTopicName>
              <frameName>camera_link</frameName>
              <hackBaseline>0.07</hackBaseline>
              <distortionK1>0.0</distortionK1>
              <distortionK2>0.0</distortionK2>
              <distortionK3>0.0</distortionK3>
              <distortionT1>0.0</distortionT1>
              <distortionT2>0.0</distortionT2>
           </plugin>
         </sensor>
</gazebo>
```

3.1.3. p3dx_gazebo

Esta carpeta es la más relevante del paquete. En ella están incluidos los dos archivos necesarios para que el modelo pueda usarse óptimamente:

- *gazebo.launch*: Es el archivo que se ejecuta al lanzar la aplicación de gazebo. En él, se especifica qué mundo generar, qué descripción del robot utilizar y en qué coordenadas generar el robot. Además, resulta interesante el hecho de que desde aquí también se pueden lanzar otros nodos que sean necesarios siempre que se genere un robot, como lo son el *joint_state_publisher* y el *robot_state_publisher*, para que se generen las transformadas.
- *p3dx*.world: Se trata de un archivo en formato *xml* donde se incluyen todos los modelos de gazebo que queremos que aparezcan en el simulador. Pueden incluirse tantos como sean necesarios, e incluso repetir un mismo modelo varias veces. Para esto último, es necesario especificar un identificador distinto para cada modelo repetido. De esta manera, se puede generar un mundo con un modelo de la planta de un edificio e incluir en el mobiliario e iluminación. En este proyecto, se ha empleado la planta baja de la Escuela Técnica Superior de Ingeniería Industrial (ETSII); incluyéndose mobiliario de ejemplo para ciertas estancias, como mesas, sillas, pizarras, despachos, ordenadores, puertas, luces de techo, etc.

  La estructura para incluir un modelo en este documento es la siguiente:

```
<include>
        <uri>model://(nombre del modelo)</uri>
        <name>(identificador único para cada elemento)</name>
        <pose>x y z θx θy θz</pose> (distancia en metros y rotación en radianes)
</include>
```

3.2. Teleop

Este paquete es utilizado en cualquier proyecto de ROS durante la etapa de desarrollo, pues permite el control manual del robot en cualquier momento (Hendrix, s.f.). Da la opción de enviar comandos de velocidad y dirección al modelo, tanto en el simulador como al robot real. Para ello, solo hay que indicarle al paquete sobre qué *topic* se deben publicar los mensajes para que sean escuchados por el robot, en este caso el *topic* es */p3dx/cmd_vel*.

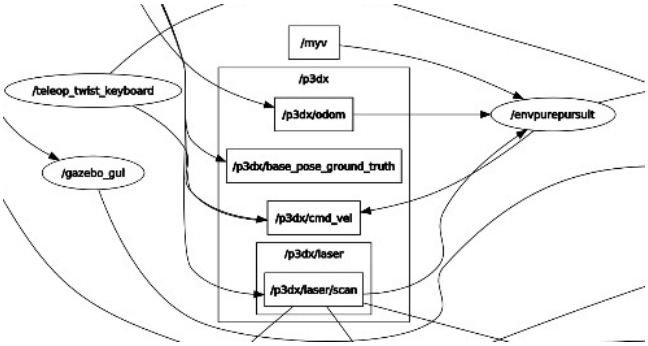

Figura 21. El nodo teleop (teleop_twist_keyboard) recibe comandos del teclado que se traducen en mensajes de velocidad. Estos mensajes están siendo enviados a cmd_vel, que también puede recibir comandos de velocidad de otros nodos, tal y como se puede observar.

3.3. Gmapping

En lo referente a la generación de un mapa en un entorno desconocido, existen varias soluciones. Una de las más conocidas es la aportada por el paquete *Gmapping* (Gerkey, s.f.). Este paquete proporciona, un algoritmo de SLAM basado en las medidas de un láser 2D y la odometría del robot. Como salida, genera un mapa de ocupación en dos dimensiones, a partir de dichas medidas y de la pose estimada por los sensores propioceptivos del robot.

Este paquete fue originalmente desarrollado para la generación de mapas de entornos de interior en ambientes controlados, como edificios públicos, campus, etc. Surgió como una solución a la necesidad de la reducción del número de posibles poses que muestrear y analizar, ya que los métodos previos de SLAM eran computacionalmente costosos. Con el algoritmo de este paquete se consigue reducir en un orden de magnitud el número de partículas necesarias. Se recomienda su uso con láseres de larga distancia, pero esto no implica que no pueda ser adaptados para modelos de corto alcance.

3.3.1. Hardware necesario

Basta con un modelo de robot y un láser tipo LIDAR montado horizontalmente.

3.3.2. Transformadas necesarias

Para que este paquete funcione adecuadamente son necesarias las transformaciones de las medidas del láser a la base del robot (*base_link*) y de la base del robot a el *frame* de la odometría (*odom*). Por su parte, este paquete proporciona la transformada de *odom* al origen del mapa (*map*).

3.3.3. Lanzamiento

Este paquete posee un único nodo que realiza todo el proceso. Para lanzar este nodo debe hacerse un *remapping* de las dos transformadas requeridas por el paquete. Para ello, se debe emplear la siguiente línea de comandos:

```
rosrun gmapping slam_gmapping scan:=/p3dx/laser/scan _odom_frame:=odom
```

3.3.4. Principio de funcionamiento

El paquete *Gmapping* se basa en el algoritmo de OpenSLAM. La gran eficacia de este algoritmo es debido a que reduce el número de posibles resultados que muestrear mediante el empleo del Rao-Blackwellized Particle Filter (RBPF) y un muestreo reiterativo adaptativo. En este algoritmo, tal y como menciona (Dieter Fox), no se tiene en cuenta solo la medida del LIDAR o la de la odometría para estimar un modelo. En oposición, se basa en centrar el rango de posibilidades generados por los datos de odometría en la moda de la distribución probabilística de los datos de láser, y suponer una distribución gaussiana. De esta manera se fusionan ambas medidas para adaptar la cantidad de posibles soluciones a la situación. Por último, el algoritmo Rao-Blackwellized realiza un segundo proceso de muestreo para eliminar aquellas soluciones que poseen un menor peso. Sin embargo, en ciertas ocasiones esto puede derivar en la eliminación de la solución correcta, por ello se emplea un método adaptativo para decidir si la proposición actual ya es lo suficientemente acertada como para no tener que repetir el muestreo.

Por otro lado, este paquete incluye el desarrollo de un algoritmo de lazo cerrado, que permite la corrección del mapa cuando se vuelve a un punto previamente visitado desde otra dirección. Esto es debido a que el algoritmo RBPF almacena todas las posiciones previas con todas las muestras tenidas en cuenta. De esta manera, si se produce un solapamiento del mapa (correspondiente con un lazo cerrado en la trayectoria) en el que no coinciden las muestras estimadas, se realiza una reestimación de todas las poses previas, asignándole unos pesos distintos a cada solución.

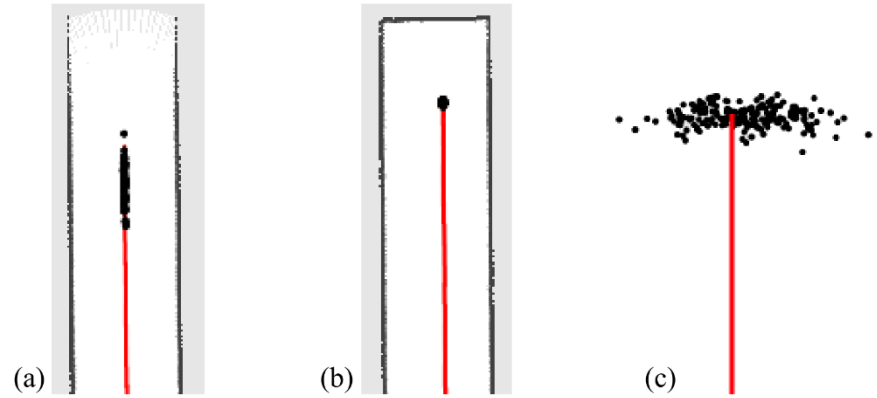

Figura 22. Ejemplo aclarativo de la utilidad de RBPF. En la figura (a) se limitan las potenciales poses del robot al tener en cuenta las medidas del láser, que tras varias detecciones de las paredes del pasillo a la misma distancia reduce la incertidumbre de la pose a la longitudinal de la dirección. En la figura (b) se muestra cómo, conforme se va acercando a un objeto frontal, esta incertidumbre longitudinal se va reduciendo hasta ser prácticamente puntual. En el caso (c) se representan todas las posibles poses del robot si no se obtienen medidas de láser con las que delimitar la región de incertidumbre.

3.4. Hector mapping

Este paquete se trata de otra solución de mapeado (Meyer, s.f.). La gran diferencia con respecto al paquete anterior es que realiza todo el proceso sin tener en cuenta los datos de odometría. A su vez, tampoco posee la capacidad de utilizar un lazo cerrado. Sin embargo, esta especialmente diseñado para sensores más modernos como los de Hokuyo (empleado en este proyecto), por lo que suple esta falta con una mayor tasa de lectura.

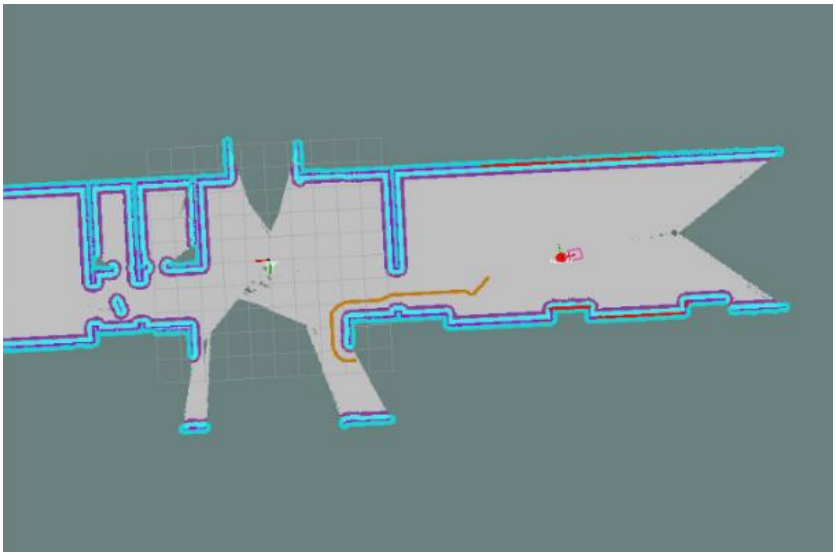

Figura 23. Generación de un mapa con el paquete de SLAM de Hector Mapping. Se puede apreciar como la tendencia es a mantener trazados rectos y estructuras poligonales.

Hector mapping fue inicialmente desarrollado como una solución para la competición mundial *RoboRescue*, donde los robots deben encontrar objetivos en un mapa muy accidentado. Por este motivo se decide prescindir de la odometría, ya que el relieve tan cambiante empeora la calidad de las medidas recibidas. Por otro lado, aporta la implementación de otras muchas funcionalidades, como la posibilidad de crear un mapa de ocupación en 3D, mediante la creación de un mapa de profundidad de partículas con una cámara RGB. Debido a que en esta competición los robots deben moverse de forma autónoma, el equipo de *Hector* ha desarrollado un paquete específico para la planeación de trayectorias. Para asegurar la compatibilidad, *Hector mapping* contiene varios paquetes de transmisión de datos.

3.4.1. Hardware necesario

Basta con un modelo de robot y un láser tipo LIDAR montado horizontalmente.

3.4.2. Transformadas necesarias

Al prescindir de los datos de odometría no se necesita el *frame odom*, por lo que solo se requiere la transformada del sensor a la base del robot. A su vez, el paquete proporcionará la transformada de la base del robot al mapa.

3.4.3. Lanzamiento

Para lanzar la funcionalidad básica, el mapeado 2D, deben configurarse una serie de parámetros. El archivo donde se pueden indicar estos parámetros es *hector_slam/hector_mapping/launch/mapping_default.launch*. Estos parámetros son, el *frame* sobre el que publicar el mapa, el *frame* del robot, el de la odometría si existe, y el del láser. Además, si se desea, se puede modificar el tamaño máximo del mapa generado desde este documento.

```
  <arg name="tf_map_scanmatch_transform_frame_name" default="scanmatcher_frame"/>
  <arg name="base_frame" default="base_link"/>
  <arg name="odom_frame" default="odom"/>
  <arg name="pub_map_odom_transform" default="true"/>
  <arg name="scan_subscriber_queue_size" default="5"/>
  <arg name="scan_topic" default="p3dx/laser/scan"/>
  <arg name="map_size" default="4096"/>
```

Una vez configurados dichos parámetros, podemos lanzar los nodos requeridos para la realización de SLAM de forma satisfactoria. Para ello, existe un archivo de lanzamiento, *hector_slam/hector_slam_launch/launch/myp3dx.launch*. En él, se introducen dichos nodos y se importa la configuración definida previamente:

```
<?xml version="1.0"?>

<launch>
<include file="$(find hector_mapping)/launch/mapping_default.launch"/>

  <include file="$(find hector_geotiff)/launch/geotiff_mapper.launch">
    <arg name="geotiff_map_file_path"
default="/home/pablo/catkin_ws/src/hector_slam/hector_geotiff/maps"/>
    <arg name="trajectory_source_frame_name" value="scanmatcher_frame"/>
    <arg name="map_file_path" value="$(arg geotiff_map_file_path)"/>
    <arg name="map_file_base_name" type="string" value="ETSII_MAP_" />
    <arg name="geotiff_save_period" type="double" value="30" />
  </include>
  <node pkg="hector_map_server" type="hector_map_server" name="hector_map_server" output="screen"
/>
  <node pkg="hector_trajectory_server" type="hector_trajectory_server"
name="hector_trajectory_server" output="screen" />
  <node pkg="topic_tools" type="throttle" name="map_throttle" args="messages map 0.1
throttled_map" />

</launch>
```

3.4.4. Principio de funcionamiento

Para este paquete, se carece de documentación oficial que permita conocer la propuesta exacta de SLAM que se sigue para realizar el algoritmo se este paquete. Sin embargo, puede estimarse que el modelo seguido es un filtro de partículas basado en la distribución probabilística de las medidas obtenidas del láser. Este modelo, al no tener en cuenta los datos de odometría ni incorporar un algoritmo de lazo cerrado, resulta mucho más simple computacionalmente. Sin embargo, se necesita de un sensor con un alto grado de precisión. De lo contrario, a la desviación típica de las muestras producirá un alto grado de incertidumbre que se traduce en deformaciones y solapamientos en el mapa.

3.5. Navigation Stack

Este paquete está centrado en la generación de trayectorias de forma autónoma para un robot, dado un punto de destino. Para ello, emplea los datos recibidos de odometría y sensores exteroceptivos que le permitan recopilar información métrica de su entorno. Con estos datos, se puede estimar la posición del robot dentro de un mapa previamente generado, mediante el algoritmo de Adaptative Monte Carlo Localization (amcl), y modificar la ruta planeada para evitar así los obstáculos que vayan apareciendo a su paso.

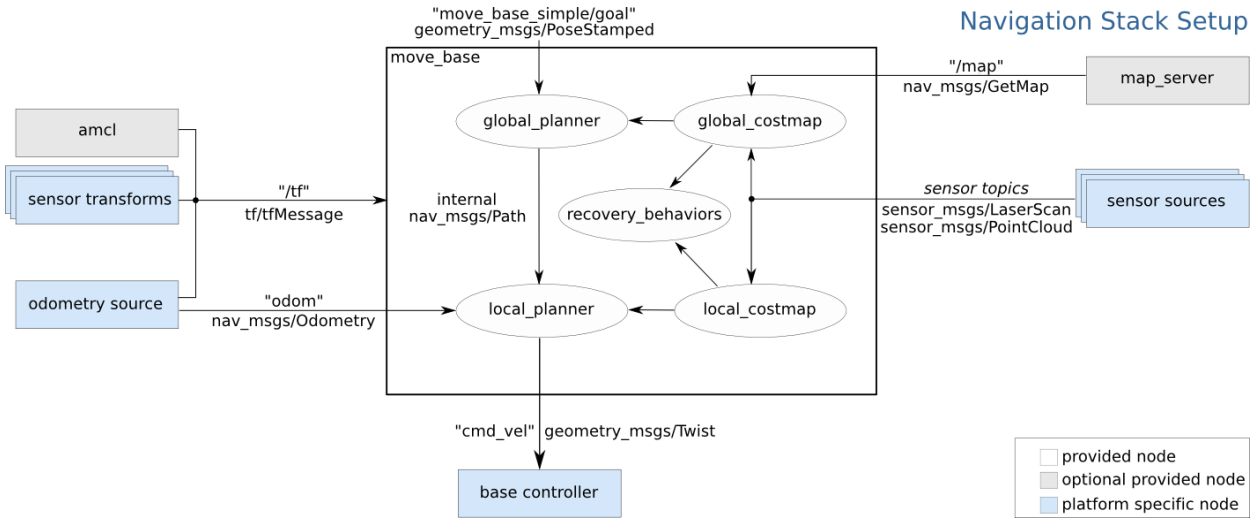

Figura 24. Esquema del flujo de información en el paquete Navigation. Se recibe información exterior del mapa, los sensores, la odometría y las transformadas. Internamente, se genera un mapa de ocupación global y otro local, que aportan la información necesaria para generar una ruta completa que puede ser modificable de forma local.

3.5.1. Hardware necesario

Como característica principal, es necesaria la presencia de un sensor láser tipo LIDAR orientado paralelo respecto al suelo, para que pueda construirse un mapa local del entorno con el que planear las rutas y detectar obstáculos.

Otra característica importante, es el hecho de que este paquete se desarrolló para ser implementado en robots que, o bien tengan un control diferencial (como es el caso del P3DX), o bien sean holomónicos. Esto implica que los radios de curvatura de las rutas planeadas pueden no ser lo suficientemente grandes para robots basados en otra estructura de movimiento.

Por último, este paquete fue originalmente desarrollado para implementarse en un robot con un chasis cuadrado, lo que implica que puede no ser eficaz en robots que sean muy alargados o no tengan una morfología simétrica. Un ejemplo de los posibles problemas que pueden surgir es al pasar por puertas, ya que, si el robot es demasiado rectangular, puede resultar que entre con cierto ángulo y acabe rozando los marcos.

3.5.2. Transformadas necesarias

Al hacer uso de la odometría y del mapa, es necesario que además del *frame base_link* estén tanto *odom*, como *map*. Se da por supuesto que, llegado al punto de implementar este paquete, las transformadas de los sensores propioceptivos y exteroceptivos del robot están ya correctamente definidas y activas.

3.5.3. Configuración

Este paquete se ha desarrollado con la intencionalidad de ser lo más genérico posible, dentro de las limitaciones físicas de cada robot. Por ello, previo a su puesta en marcha, es necesario realizar una serie de especificaciones, tanto de su funcionamiento como de las características del robot en el que se vaya a implementar. Para ello, se van a definir una serie de documentos de configuración. De esta manera se facilita su lectura y posterior accesibilidad en caso de ser necesario.

3.5.3.1. costmap_common_params.yaml

Como ya se ha mencionado previamente, Navigation Stack ofrece la posibilidad del empleo de mapas para la planificación de trayectorias de larga distancia más efectivas. Para ello, se sirve de un total de dos mapas; un mapa global, que es el que se le proporciona al paquete, y otro local, generado por este paquete para la reconstrucción de mapas a corta distancia. Estos dos mapas poseen una parte de configuración común, por lo que se especifica en un único documento.

```
obstacle_range: 2.5
raytrace_range: 3.0
#footprint: [[x0, y0], [x1, y1], ... [xn, yn]]
robot_radius: 0.30
inflation_radius: 0.15

observation_sources: laser_scan_sensor

laser_scan_sensor: {sensor_frame: lms100, data_type: LaserScan, topic: p3dx/laser/scan, marking:
true, clearing: true}
```

En esta figura, se pueden apreciar los distintos campos configurables para los mapas. Los dos primeros, *obstacle_range* y *raytrace_range*, permiten ajustar la distancia máxima a la cual se actualizará el mapa con la presencia de un obstáculo y se intentará tener una vía libre obstáculos respectivamente.

Por otro lado, se presenta el footprint que, como es de entender, delimita el espacio ocupado por el robot para poder calcular por dónde hay espacio suficiente para pasar. Footprint permite generar una figura poligonal mediante la definición de sus vértices. En el caso concreto de este proyecto, el robot P3DX posee una huella circular, por lo que este parámetro no es válido. En su defecto, se define el siguiente, *robot_radius*, que generará una figura circular del radio especificado.

A continuación, se define el radio de inflación, *inflation_radius*. Esta medida delimita la zona de incertidumbre alrededor de los obstáculos. De esta manera, se elimina el posible error de medida de los instrumentos, asegurando así que todo lo que quede fuera de esta zona es terreno libre.

Por último, queda la enumeración de las distintas medidas que se deben tener en cuenta. Dichas medidas son definidas posteriormente de manera independiente, siendo necesario especificar el *frame* al que pertenecen, el tipo de mensaje y el *topic* sobre el que se publican. Las dos últimas variables son empleadas para indicar si los mensajes de este sensor pueden ser usadas o no para añadir y/o eliminar obstáculos del mapa.

3.5.3.2. Global_costmap_params.yaml

Este documento es empleado para definir las características del mapa global de la planta.

```
global_costmap:
  global_frame: /map
  robot_base_frame: base_link
  update_frequency: 5.0
  static_map: true
```

Como puede verse en la figura anterior, es necesario indicar el *frame* sobre el que está referenciado el mapa, y el *frame* de la base del robot. Otras opciones interesantes de configurar son la frecuencia a la que se actualiza el mapa y si se va a proporcionar un mapa previamente generado o no.

3.5.3.3. Local_costmap_params.yaml

En el caso de la generación del mapa local, se deben indicar una serie de parámetros adicionales, ya que este no es proporcionado al paquete.

```
local_costmap:
  global_frame: odom
  robot_base_frame: base_link
  update_frequency: 5.0
  publish_frequency: 2.0
  static_map: false
  rolling_window: true
  width: 6.0
  height: 6.0
  resolution: 0.05
```

En este caso, el *global_frame* debe asignarse a *odom* debido a que el mapa se crea centrado en el robot. Por tanto, para poder relacionarlo con secuencias previas, Navigation se basa en las medidas de odometría; quedándose referenciado así al origen del *frame odom*.

Al no ser un mapa proporcionado, se establece el parámetro *static_map* en falso, y se especifica una frecuencia de publicación de este. Por otro lado, el parámetro *rolling_window* nos permite indicar si deseamos que el mapa permanezca centrado en el robot conforme se vaya moviendo.

Finalmente, se encuentran los parámetros para especificar la dimensión y resolución que debe tener este mapa. Al tratarse de un mapa local, se recomienda que, si se utiliza centrado en el robot, el tamaño de este no supere la distancia máxima de detección del sensor; puesto que no se generará más información y se aumentará el tiempo de procesado.

3.5.3.4. Base_local_planner_params.yaml

Como se ha mencionado con anterioridad, este paquete está desarrollado de forma genérica, por lo que también es necesario especificar ciertas características del robot en el que se vaya a implementar. A continuación, se indican las características más relevantes, pero es posible modificar una gran variedad de parámetros.

```
TrajectoryPlannerROS:
  max_vel_x: 0.40
  min_vel_x: 0.1
  max_vel_theta: 0.40
  min_in_place_vel_theta: 0.1

  acc_lim_theta: 3.2
  acc_lim_x: 2
  acc_lim_y: 2

  holonomic_robot: false #P3DX is a diferential drive robot
```

Debido a la morfología y al propio peso, cada vehículo motorizado posee unos límites de velocidad y aceleración a partir de los cuales el robot puede desestabilizarse. Por ello, es necesario indicar dichos límites al planificador, ya que este paquete genera directamente los comandos de velocidad que deben enviarse al robot. Normalmente, el desarrollador de un modelo de robot especifica cual es el rango de velocidades en las que es seguro operar el modelo. Sin embargo, en ocasiones en velocidades altas la estabilidad de la plataforma es reducida, generando mucho ruido en las medidas. Por ello, no está de más limitar aún más los valores máximos, mediante ensayo y error, hasta un punto en el que el error en las medidas sea aceptable. Como se puede observar en la figura, se puede especificar la velocidad lineal y la rotación en el eje vertical.

Por último, es necesario especificar si el robot sigue una estructura holomónica o no. En caso negativo, se supone un control diferencial.

3.5.4. Lanzamiento

A la hora de poner en marcha este paquete, es necesario generar un archivo de lanzamiento para poder incluir todos los documentos de configuración crear previamente. Este archivo, recibe el nombre de *move_base.launch*.

```
<launch>

    <master auto="start"/>
 <!-- Run the map server -->
    <node name="map_server" pkg="map_server" type="map_server"
args="/home/pablo/catkin_ws/maps/A_THISMAP_ETSII_MAP_071220_00:35:11.pgm 0.05"/>

 <!--- Run AMCL -->
    <include file="$(find amcl)/examples/amcl_diff.launch" />

    <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen">
        <rosparam file="$(find robot_config)/costmap_common_params.yaml" command="load"
ns="global_costmap" />
        <rosparam file="$(find robot_config)/costmap_common_params.yaml" command="load"
ns="local_costmap" />
        <rosparam file="$(find robot_config)/local_costmap_params.yaml" command="load" />
        <rosparam file="$(find robot_config)/global_costmap_params.yaml" command="load" />
        <rosparam file="$(find robot_config)/base_local_planner_params.yaml" command="load" />
    </node>

</launch>
```

Como se va a proporcionar un mapa previamente generado, es necesario lanzar el servicio que *Gmapping* posee para publicar mapas. Este servicio lee el contenido del documento que se le indique e intenta publicarlo. A continuación, se incluye el documento predefinido en el paquete para el nodo de *amcl*. Este nodo es el encargado de realizar el proceso de localización dentro del mapa y, de esta manera, dar un punto de partida al planificador de rutas.

Finalmente, se lanza el nodo *move_base* con los archivos de configuración que han sido creados previamente. Este nodo quedará a la espera de recibir mensajes de destino. Cuando se reciba uno, comenzará el proceso de calcular la ruta más eficiente al destino, y se irá actualizando conforme vayan apareciendo obstáculos que no constaban en el mapa.

3.5.5. Principio de funcionamiento

El flujo de proceso que sigue este paquete es fácilmente diferenciable en dos etapas, la etapa de localización, y la de planificación.

3.5.5.1. Localización

Primero de todo, es necesario la existencia de un mapa sobre el que poder hacer el análisis. Por lo tanto, se parte de la base de haber generado un mapa y publicado mediante el servicio específico de *Gmapping*. Una vez cargado, hace una lectura de la odometría para realizar una primera estimación de la pose actual del robot en el mapa. Esta pose será orientativa y muy probablemente no coincida a la perfección con la posición real, puesto que se basa únicamente en el valor de la pose respecto al origen de *odom*.

A partir de ese momento, entra en juego el algoritmo de Adaptative Monte Carlo Localization (AMCL) que usa un filtro de partículas para comparar las medidas del láser con secciones de un mapa conocido. Al estar inicialmente mal localizado el robot, el filtro necesitará que el robot se desplace unos pocos metros para poder reducir la incertidumbre de la pose del robot. Este filtro se basa en los obstáculos detectados por el láser, por lo que cuantos más obstáculos estén presentes, más fácil será para este algoritmo localizar la posición correcta. A su vez, cuanto más este alejada la estimación inicial de la real, más costoso será llegar a la solución correcta. Es por esto, que, si el robot se encuentra demasiado alejado del origen del mapa, *amcl* sea incapaz de encontrar por si sólo la pose correcta, debido a la gran variabilidad existente de base. En estos casos, puede indicarse una posición orientativa del robot sobre el mapa mediante herramientas como *rviz*. Con esta nueva pose estimada el nodo ya será capaz por si solo de terminar de pulir la posición exacta.

Una vez estimada la pose, esta será publicada en el *topic amcl_pose*, que posteriormente será recibido por el nodo de planificación de ruta.

3.5.5.2. Planificación

Para la planificación, se necesita tener un punto de destino, por lo que el nodo permanece a la espera de recibir un mensaje tipo *goal*. Una vez recibido, se procede al cálculo de la trayectoria desde la posición estimada por *amcl*. Para ello, se planifica la ruta más corta posible con los datos obtenidos del mapa global, generando hitos intermedios no visibles para el usuario. En el cálculo de la ruta se han tenido en cuenta la zona de exclusión y el *footprint* del robot definidos en la configuración, por lo que se evitará cualquier espacio seguro que no tenga el tamaño suficiente para que el robot pase libremente. A continuación, entra en juego el mapa local, que, tras hacer una lectura de su entorno, el nodo genera un mapa de ocupación de la zona circundante incluyendo las zonas de incertidumbre. El planificador buscará el camino más despejado en este entorno local para llegar a su destino. Si se presentan obstáculos no esperados en la ruta planificada, se hará un replanteamiento de la ruta a corto plazo para sobrepasar dicho obstáculo, extendiendo el camino a seguir lo menos posible.

Se debe tener en cuenta, que, al haber enviado un punto de destino, el robot va a intentar llegar a la posición exacta especificada. Por lo que, si llega al destino con una pequeña desviación, el robot comenzará a realizar trayectorias circulares en un afán de aproximarse mejor al punto. Es por esto por lo que es necesaria la definición de una buena horquilla, que permita cierto error en el destino sin afectar en demasía al resultado final.

3.6. Hector Exploration Planner

Se trata del segundo paquete desarrollado por el equipo de Hector (Stefan Kohlbrecher, s.f.). Estos paquetes fueron originalmente pensados para mejorar la automatización de tareas de Búsqueda y Rescate Urbanos (USAR). Exploration Planner en concreto, proporciona la capacidad de generar rutas a destinos en entornos desconocidos, asi como la generación automática de dichos destinos. De esta manera se consiguen explorar zonas desconocidas en busca de objetivos.

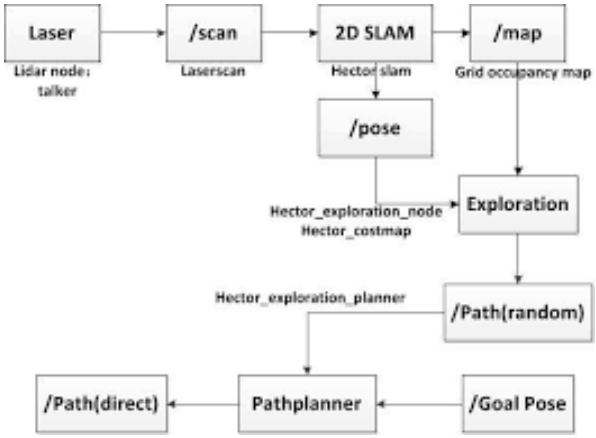

Figura 25. Flujo conceptual del proceso de generación de rutas a partir de la información del láser únicamente. Se realiza un proceso de localización mediante la búsqueda de coincidencias del patrón generado por láser. A partir de esa coincidencia, se genera una pose estimada, que junto con el mapa se emplea para la generación de las rutas.

3.6.1. Transformadas necesarias

Tal Y como ha ocurrido hasta ahora con todos los paquetes que trabajan con mapas, se necesita la presencia de la transformada de la base del robot al *frame* del mapa. A su vez, se necesitará la transformada de todos los sensores que se vayan a emplear.

3.6.2. Configuración

Por desgracia, este paquete presenta graves problemas de compatibilidad en la versión de ROS en la que se está desarrollando este proyecto. Para solucionarlo, han de instalarse varios paquetes extra:

- costmap_2d
- hector_slam
- geographic_info
- unique_identifier
- ceres_catkin
- catkin_simple
- geometry2
- navigation_msgs

Todos estos paquetes extra son necesarios debido a que *Hector Exploration* reutiliza muchas de sus funcionalidades. El paquete más problemático con diferencia es *ceres*. Pero, al ser únicamente utilizado para la localización GPS, puede prescindirse de él si se suprime también dicha funcionalidad. Una vez el paquete compile satisfactoriamente, se da paso a la edición de los archivos de configuración. La funcionalidad de cada uno de los paquetes restantes se explicará más adelante.

3.6.2.1. hector_exploration_node/config/costmap.yaml

Este primer documento establece las especificaciones para la interpretación del mapa que se haya generado previamente, el mapa global.

```
global_costmap:

  map_type: costmap
  track_unknown_space: true
  unknown_cost_value: 255
  obstacle_range: 2.5
  raytrace_range: 3.0
  footprint: [[0.25, 0.21],
              [-0.25, 0.21],
              [-0.25, -0.21],
              [0.25, -0.21]]
  inflation_radius: 0.32
  transform_tolerance: 1.0
  inscribed_radius: 0.3
  circumscribed_radius: 0.32

  #layer definitions
  static:
    map_topic: map
    subscribe_to_updates: true
  global_frame: map
  robot_base_frame: base_link
  update_frequency: 0.5
  publish_frequency: 0.1
  static_map: true
  rolling_window: false
```

Como se puede observar, las especificaciones son bastante similares al paquete *Navigation* visto anteriormente, esto es debido a que utiliza alguna de sus funcionalidades, por lo que se mantiene el formato. De primeras, se encuentra la identificación de qué tipo de mapa está tratando, en este caso se indica *costmap* al ser un mapa de ocupación. La siguiente variable permite decidir si el planificador debe tratar de intentar explorar zonas no mapeadas o no. A continuación, vuelven a aparecer los parámetros *obstacle_range* y *raytrace_range*, que quedan establecidos al mismo valor. En cuanto al footprint del robot, en esta ocasión no se da la posibilidad de definir una forma circular, por lo que se debe definir la siguiente forma poligonal más ajustada, un cuadrado de lado igual al diámetro del robot. De seguido, se encuentra la definición del área de exclusión mediante el parámetro *inflation_radius*. Cabe destacar la presencia de un parámetro para indicar el tiempo máximo permisible sin recibir mensajes de la transformada del mapa.

Por último, se especifica el *topic* sobre el que se está publicando el mapa, el *frame* de este y el del robot. También se indica que se trata de un mapa previamente generado, estableciendo *static_map* a cierto, y que no se trata de un mapa centrado en la base del robot, por lo que *rolling_window* es falsa.

3.6.3. Lanzamiento

3.6.3.1. My_expoloration_planner.launch

Una vez especificado el mapa, solo queda lanzar los distintos nodos necesarios para realizar una exploración autónoma basada únicamente en los datos del LIDAR. Para ello, se lanzarán un total de 4 nodos: *hector_costmap*, *hector_driving_aid_markers*, *hector_path_follower* y *hector_exploration_node*. Los parámetros de cada uno se especifican en este mismo *launch file*.

```
<?xml version="1.0"?>

<launch>
       <node pkg="hector_costmap" type="hector_costmap" name="hector_costmap" output="screen"
respawn="false">
           <param name="cost_map_topic" value=" map" />
           <param name="map_frame_id" value="map" />
           <param name="local_transform_frame_id" value="base_link" />
           <param name="initial_free_cells_radius" value="0.3" />
           <param name="update_radius" value="4.0"/>
           <param name="costmap_pub_freq" value="4.0" />
           <param name="sys_msg_topic" value="syscommand" />

           <param name="use_grid_map" value="true" />
           <param name="grid_map_topic" value="scanmatcher_map" />

           <param name="use_elevation_map" value="false" />
           <param name="elevation_map_topic" value="elevation_map_local" />

           <param name="use_cloud_map" value="false" />
           <param name="cloud_topic" value="openni/depth/points" />
       </node>

       <node pkg="hector_driving_aid_markers" type="hector_driving_aid_markers_node"
name="hector_driving_aid_markers" output="screen">
           <param name="left_side_y_outer" value="0.205"/>
           <param name="left_side_y_inner" value="0.11"/>
           <param name="right_side_y_outer" value="-0.205"/>
           <param name="right_side_y_inner" value="-0.11"/>
       </node>

       <!-- Not necessary to set, set by scanmatcher -->
           <!--param name="map_resolution" value="0.05" /-->
           <!--param name="max_grid_size_x" value="1024" /-->
           <!--param name="max_grid_size_y" value="1024" /-->
       <!--</node>-->

       <node pkg="hector_exploration_node" type="exploration_planner_node"
name="hector_exploration_node" output="screen">

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

       </node>

       <node pkg="hector_path_follower" type="path_follower" name="hector_path_follower_node"
output="screen">
           <param name="holonomic" value="false"/>
           <param name="max_vel_lin" value="0.4"/>
           <param name="min_vel_lin" value="0.1"/>
           <param name="max_vel_th" value="2.0"/>
           <param name="min_vel_th" value="0.1"/>
           <param name="robot base frame" value="base link"/>
           <param name="global_frame" value="map"/>
           <param name="cmd_vel" value="/p3dx/cmd_vel"/>
       </node>
</launch>
```

En este documento, se puede observar que cargan todos los nodos indicando el nombre del *frame* de la base del robot y el *frame* global. Más específicamente, para *costmap* se indica qué tipo de mapa se desea emplear, en este caso solo se desea un mapa tipo *grid*, dejando el de elevación y de nube de puntos negados. En el caso del *driving_aid_marker*, tan solo es necesario indicar la posición y tamaño de las ruedas, referenciadas al centro del robot. Para el *exploration_planner*, es necesario importar el archivo de configuración del mapa que se ha definido previamente. Por último, a *path_folllower* se le debe indicar los límites de velocidad y el modo de conducción del robot, en este caso diferencial. Es de vital importancia especificar que el *topic* de velocidad es */p3dx/cmd_vel*, de lo contrario el robot no se desplazará, pero tampoco saltará ningún aviso.

3.6.4. Principio de funcionamiento

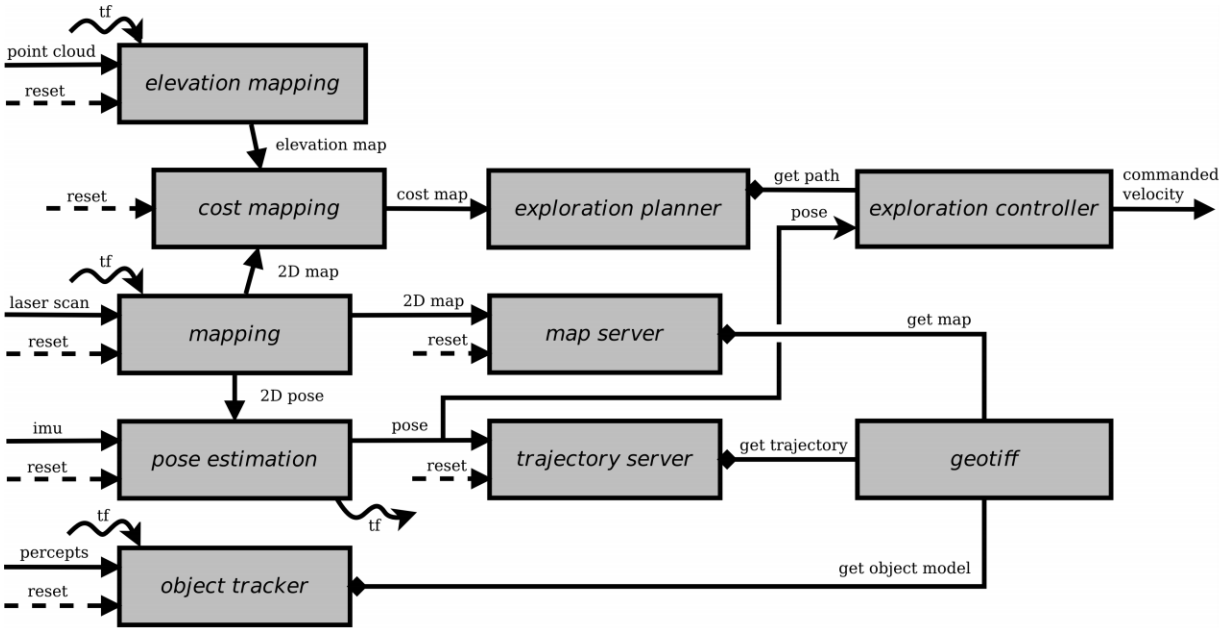

Figura 26. Representación del flujo de información entra los distintos procesos llevados a cabo por el paquete de Hector Exploration. Se genera un mapa, o bien se publica uno previo mediante map_server, que es empleado para una estimación de la pose y la generación del mapa de ocupación. Una vez generada la ruta por exploration_planner se envía a exploration_controller para que genere los comandos de velocidad pertinentes en función de la pose estimada.

Como se puede observar en la Figura 26, se parte de un mapa de ocupación, este mapa puede ser bidimensional o tridimensional. Para el desarrollo de este proyecto se ha decidido optar por una generación bidimensional. Como el mapa ya ha sido previamente generado, se obtiene mediante el nodo *map_server*, que emplea el servicio *get_map* de *geotiff* para recuperar el mapa de los archivos. Una vez cargado el mapa, se envía a *exploration_planner*. Este nodo, se encarga de la generación de una ruta segura para llegar al destino deseado. Por último, *exploration_controller* analiza la ruta a seguir para generar los comandos de velocidad adecuados para seguir esta.

3.7. Otros Paquetes necesarios

3.7.1. unique_identifier

Este paquete (O'Quin, UUID, s.f.), se encarga de definir un mensaje para un Identificador Único Universal (UUID). Actualmente es únicamente empleado para resolver dependencias en paquetes con problemas de compatibilidad.

3.7.2. tf_transformacion

Es un pequeño paquete adicional, de desarrollo propio. Es empleado para alojar en el todas las transformadas de elementos que no vengan definidas por defecto. Un ejemplo de ello son las transformada del *frame lms100* a *base_link*, y la transformada de *odom* a *map*.

3.7.3. image_pipeline

Se trata de un paquete que contiene las funcionalidades necesarias para convertir una imagen en bruto proporcionada por el *driver* de una cámara en una imagen procesada que pueda ser utilizada por otros paquetes (Rabaud, s.f.). Sus funcionalidades principales son:

- Calibrado de objetivo
- Procesado de imagen monocular
- Procesado de imagen estéreo
- Procesado de imagen de profundidad
- Visualización de imágenes

3.7.4. geometry2

Es otro meta paquete (Foote, s.f.), que permite la utilización de la segunda generación de la librería de transformadas. Esta librería es más comúnmente conocida como *tf2*.

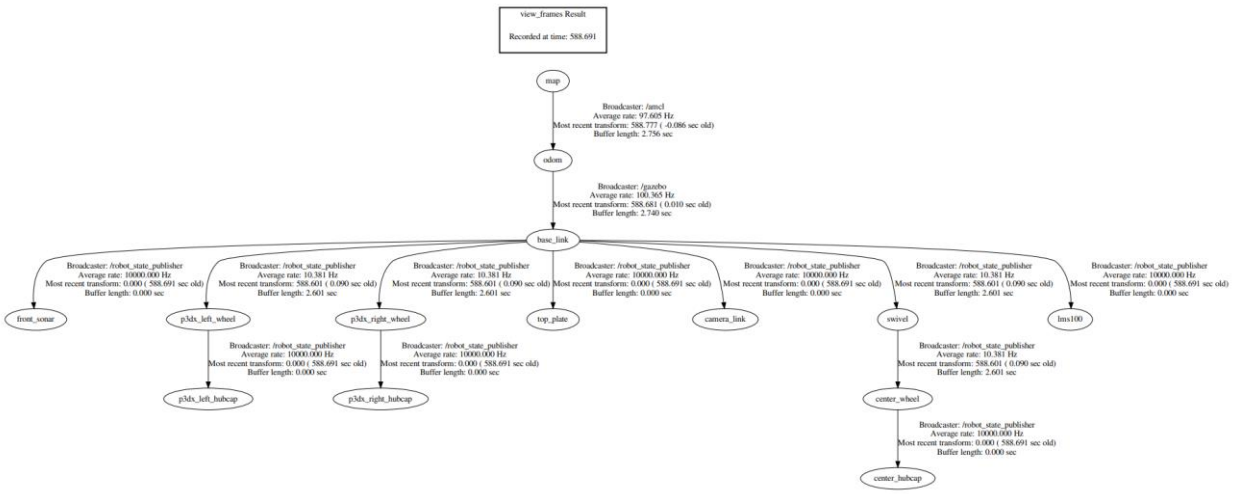

Figura 27. Representación gráfica de las relaciones de transformación entre los distintos frames presentes en la aplicación para este trabajo. Para la generación de la transformación, todos los nodos usan de base el paquete geometry2.

3.7.5. geographic_info

Se trata de otro paquete empleado para la portabilidad de *stacks* desarrollados con *rosbuild* que tenían dependencias con *geographic_msgs* y *geodesy* (O'Quin, geographic_info, s.f.). Aunque con *catkin* ya no es necesario, hay paquetes que aún siguen manteniendo dichas dependencias, por lo que *geografic_info* se encarga de brindarles esa capacidad. Principalmente proporciona métodos de conversión para mapas y coordenadas.

3.7.6. catkin_simple

La intención de este paquete es la de simplificar los archivos *CMakeLists* para hacerlos más legibles, encargándose este del procesado de dicho archivo en una capa intermedia para que sea comprensible por el entorno.

3.7.7. navigation_msgs

Únicamente contiene la definición de los mensajes empleados posteriormente por paquetes como *navigation_stack* y *hector_exploration_planner* (Hidalgo, s.f.).

3.8. Desarrollo del modelo virtual de la ETSII

Cuando se está trabajando en el desarrollo de una nueva funcionalidad para un robot, es necesario hacer pruebas de ensayo para comprobar el correcto funcionamiento del sistema. Estos ensayos normalmente se realizan con cada pequeño cambio en el código que pueda ser susceptible de modificar el comportamiento general del robot. Por este motivo, realizar las dichas pruebas en un modelo robótico real no es viable, ya que el proceso de volcado de todo el código sobre el robot y posterior puesta en marcha es lento. Esto supone una gran pérdida de tiempo. Además, cuando se intentan poner a prueba funcionalidades de mapeado y localización, se necesita un campo de pruebas considerablemente grande y que permanezca inalterado entre prueba y prueba. Estos problemas motivan al desarrollo de un modelo propio compatible con gazebo para eliminar gastos de tiempo y espacio.

Para el testeo de el algoritmo desarrollado en este proyecto, se ha procedido al desarrollo en paralelo de la planta baja de la Escuela Técnica Superior de Ingeniería Industrial (ETSII) de la Universidad Politécnica de Cartagena (UPCT)

3.8.1. Diseño de la planta

Todo el desarrollo de este modelo se ha realizado en la herramienta de CAD 3D *SolidWorks* (Dassault Systems, s.f.). Debido a que los planos oficiales de este edificio no son de dominio público, este diseño se ha basado en planos orientativos proporcionados por la propia institución (Figura 28).

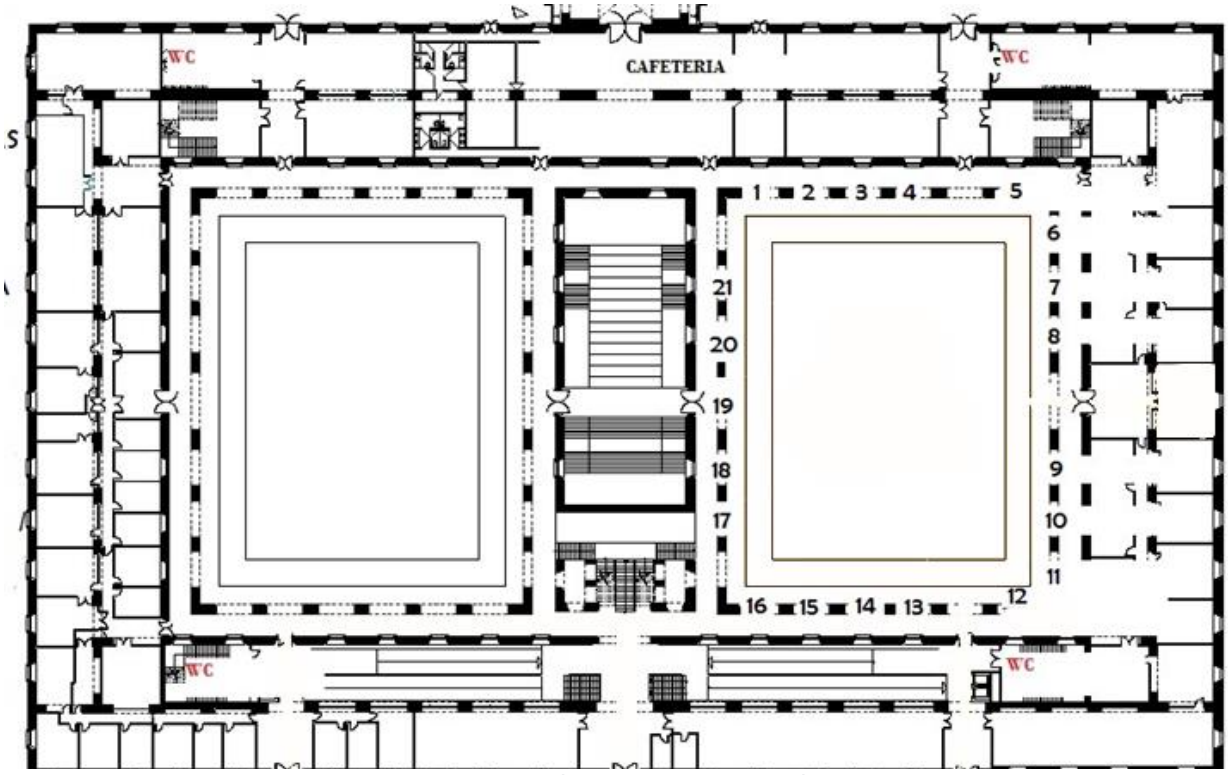

Figura 28. Plano orientativo de la planta baja de la Escuela Técnica Superior de Ingeniería Industrial. Este plano se ha obtenido mediante la edición del existente para foro de empleo de Cartagena 2017 (AENAE, s.f.).

Como este plano carece de cota alguna, para una obtención aproximada del tamaño real de la edificación se realizó una medida de distancia sobre un mapa satelital mediante la herramienta (Google Earth, s.f.). De

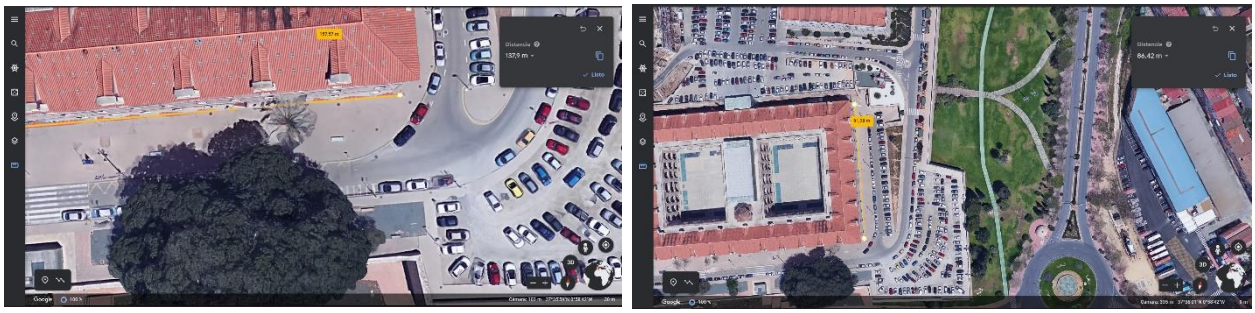

Figura 29. Proceso de toma de medidas de la ETSII dentro de la herramienta online (Google Earth, s.f.)

esta manera, puede insertarse el plano en un croquis de la pieza y posteriormente aplicarle un reescalado a la medida real del edifico. Las medidas obtenidas son 137.9 × 86.42 metros.

A la hora de realizar las operaciones de modelado 3D, se tomó la decisión de dividir los distintos croquis por elementos de construcción (pilares centrales, arcos, tabiques, muros de contención, escaleras, etc.). A su vez, estos elementos se agrupan en carpetas por zona del edificio (ala norte, este, sur y oeste). El motivo de esta planificación es el de facilitar la posterior modificación de este archivo a terceras personas, bien sea para aumentar el nivel de detalle o corregir defectos.

Para los requerimientos de este proyecto, se estimó necesaria la presencia de los siguientes elementos estructurales del edificio:

- Muros exteriores del edificio
- Muros interiores del patio
- Dos anillos interiores de columnas estructurales
- Columnas y arcos de sendos patios
- Tabiques de aulas, estancias y despachos
- Accesos a pasillos y graderío central
- Parte inferior de escaleras, tanto la central como las periféricas
- Ascensores
- Cristalería de las estancias del ala sur
- Cristalería de sendos patios
- Ventanales del ala sur

Debido a la ausencia de cotas, todas las separaciones de las distintas estancias están basadas en la disposición del plano orientativo. Estas separaciones han tenido que ser normalizadas para guardar la consistencia en el reparto de espacios, en especial para los despachos. En lo referente al tamaño y espesor de los elementos estructurales, se han normalizado para coincidir en el rango de medidas habituales en cada tipo de construcción. Siendo así un espesor de 800mm para los muros de carga externos e internos, 150mm para tabiques de obra y 100mm para paredes de pladur. En cuanto al espaciado de las entradas a las distintas estancias, se han establecido un total de 3 medidas. Para puertas individuales, un ancho de hoja de 900mm. Para puertas dobles, 1800mm. Por último, para puertas principales una distancia de 2500mm.

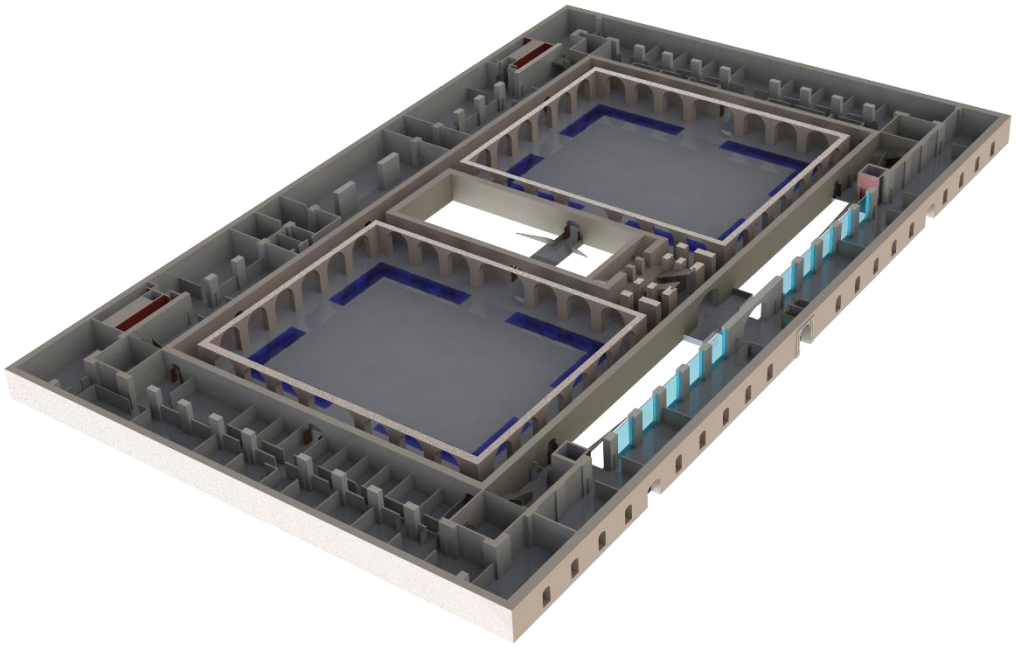

figura 30. Vista cenital del modelo 3D de la planta baja de la ETSII UPCT.

Una vez terminado el desarrollo, el diseño se exporta en formato *STL*. Esto es debido a que, a continuación, es importado al software libre *FreeCAD*, donde finalmente se genera el archivo *COLLADA* que será utilizado en Gazebo.

3.8.2. Objetos de enriquecimiento

El modelo de la planta vacío es más que suficiente para testear el comportamiento del robot en el terreno y la correcta generación del mapa métrico. Sin embargo, no es suficiente para la generación de un mapa semántico, ya que no puede obtener información distinguible entre una estancia y otra. Para solventar resta tarea, se han diseñado una serie de objetos de mobiliario. Otros, en cambio, han sido obtenidos de repositorios de modelos de Gazebo.

Todos los objetos diseñados serán coleccionados en un repositorio de GitHub para que sea accesible a la comunidad.

3.8.2.1. Puertas

Uno de los principales elementos a tener en cuenta en la decoración de interiores son las puertas. Debido a la morfología específica de la ETSII, ha sido necesario diseñar 3 modelos de puerta. Una puerta blanca individual con pomo para las entradas a las aulas, laboratorios y despachos; una puerta de madera doble sin pomo para las entradas de los pasillos; y una puerta de madera doble de 2.5 metros para la entrada central de las alas. No se ha desarrollado una puerta para la entrada principal del edificio por que el robot nunca va a cruzar dicha entrada en la simulación.

figura 31. Modelos de las puertas de la ETSII. (a) Puerta de 90cm de longitud de hoja. (b) Puerta de 180cm de longitud de doble hoja. (c) Puerta de 180cm de longitud de doble hoja. (d) Puerta de 250cm de longitud de doble hoja.

Para no sobrecargar el archivo de definición del mundo de Gazebo, se ha realizado un ensamblaje con todas las puertas en su posición exacta en el edificio. De esta manera, se exporta un único documento para todas las puertas.

3.8.2.2. Bancos

Para añadir más contenido a los patios del edificio, se diseñó el banco de madera existente. Una vez más, se hizo un ensamblaje con todos los bancos del patio y se exportó como un único documento.

figura 32. Modelos 3D réplica de los existentes en el patio de la ETSII. Estos modelos han sido dispuestos en matriz en el simulador, debajo de los arcos y delante de los pilares respectivamente.

3.8.2.3. Sillas y asientos

Ya existe cierta variedad de sillas desarrolladas para Gazebo, por lo que se escogió un grupo de estas para insertarla en el edificio. De esta manera, se pueden distinguir sillas de clase, de cocina, de despacho y sillas de espera para las estancias comunes.

Figura 33. Sillas empleadas en las diferentes estancias del entorno simulado.

Al estar ya definidas como un modelo de gazebo, tan solo es necesario incluirlas en la carpeta de modelos.

3.8.2.4. Pizarra

No existe actualmente ningún modelo de pizarra de pared para gazebo, por lo que se ha diseñado una doble que encaje en las aulas donde se imparte clases actualmente.

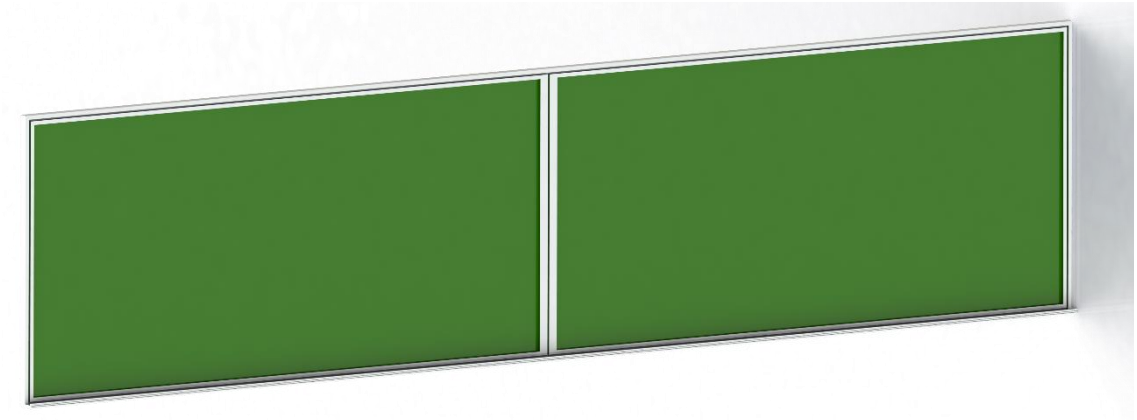

Figura 34. Modelo 3D de dos pizarras de pared.

3.8.2.5. Mesas

Otro elemento bastante común en los mundos de gazebo son las mesas. Por lo que se escogieron un total de 3 modelos, para despachos, aulas y la cantina.

Figura 35. Renderizado de los cuatro modelos de mesa empleados en la simulación de este trabajo. Están pensados para despachos, aulas, cantina y oficina.

3.8.2.6. Elementos de ofimática

Dentro de los elementos diferenciales entre aulas y despachos están los distintos tipos de elementos electrónicos de ofimática. Para las clases se incluyeron proyectores, tanto digitales como analógicos. Para los despachos y estancias de gestión académica se han incluido varios monitores, ordenadores y teclados.

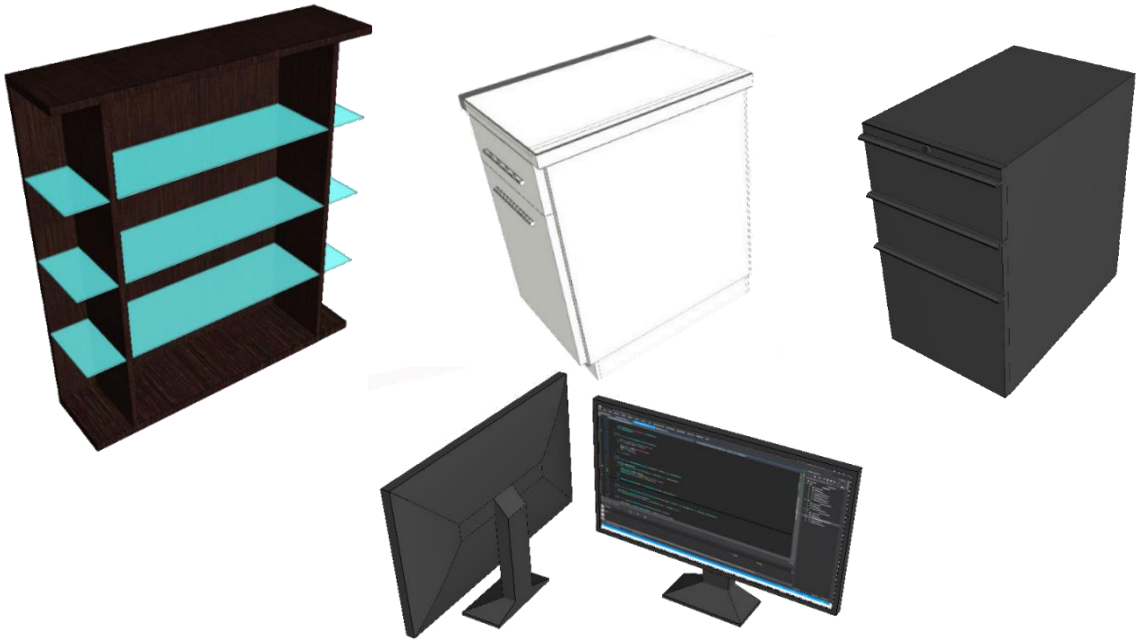

Figura 36. En el entorno de pruebas se han incluido varios elementos de ofimática para enriquecer las estancias de tipo oficina y despacho. Los objetos que aquí se representan son algunos de ellos.

3.8.2.7. Elementos de cocina

Para cargar de contenido la cantina, se han diseñado la barra principal en forma de “L” y la zona de menús. Esto permite a su vez depositar otros elementos sobre ellos. En este caso se ha optado por incluir modelos ya existentes de latas, vasos platos y una cafetera. A su vez, se han colocado más vasos y platos sobre las mesas, para que estas no estén completamente vacías.

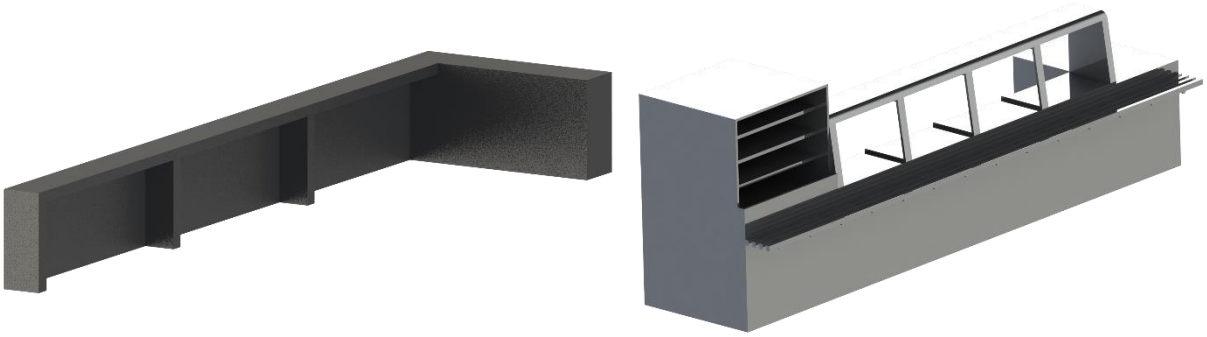

Figura 38. Mobiliario específicamente diseñado para la cantina de la ETSII. Estos objetos son visibles desde la entrada de la cantina y dominan la estancia, por tanto, es imprescindible su presencia para un buen ajuste a la realidad.

Figura 37. Modelos de código abierto desarrollados por terceros para su uso en Gazebo. Han sido empleados para añadir variedad a la cantina.

3.8.2.8. Iluminación

En aquellas estancias de mayor tamaño, resulta interesante insertar elementos de iluminación ya que, normalmente, ocupan gran parte del espacio superior de estas. Se obtuvo un diseño de una lámpara incandescente de techo para ser fieles a la realidad.

3.8.3. Resultado final

Una vez diseñados todos los elementos y exportados a COLLADA, se importaron como modelos a la librería de modelos de gazebo. A continuación, se puede ver el resultado obtenido, en la Figura 39, tras llenar de mobiliario algunas estancias de ejemplo para poder poner a prueba el robot.

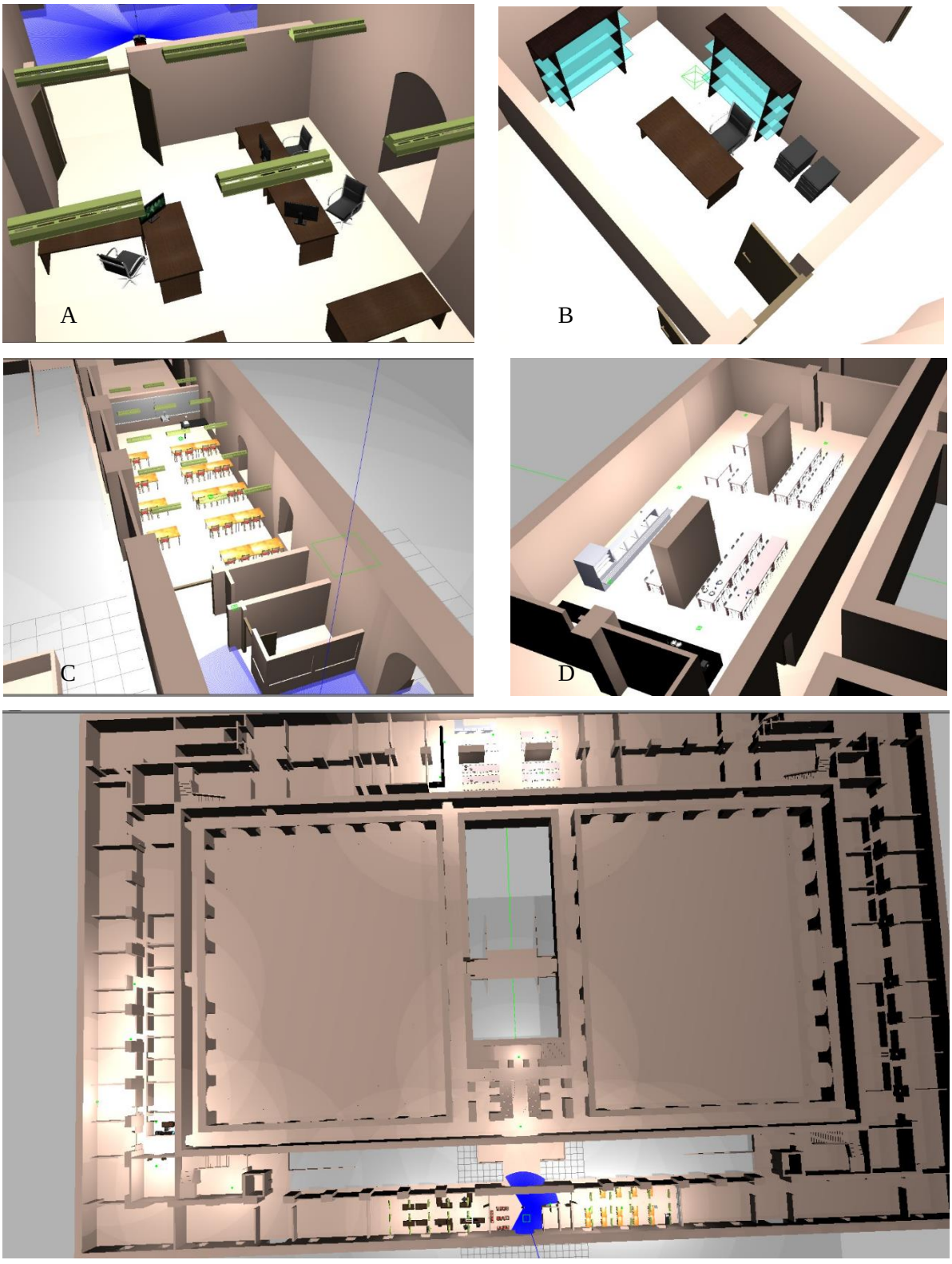

Figura 39. Para este trabajo se han definido 4 estancias tipo estas son la oficina (a), un despacho (b), una clase (c) y la cantina de la escuela (d).

3.9. Mapeado Hibrido

En el proceso de la creación de un mapa híbrido, se distinguen tres fases bien diferenciadas. Por un lado, se encuentra la extracción de información métrica del entorno. Por otro lado, se obtiene información semántica mediante imágenes, que permite la clasificación de estancias por tipos. Por último, se realiza el proceso de integración de dicha información semántica y la información métrica interpretada en un mapa topológico.

Para realizar una correcta integración de todos los elementos, es necesario establecer previamente una estructura semántica de datos, que se verá a continuación.

3.9.1. Estructuración del mapa

Para la correcta interpretación de los distintos elementos presentes en un mapa semántico, se van a definir una serie de estructuras semánticas. Estos elementos, permiten la interpretación y segmentación del mapa métrico en estructuras independientes de mayor grado de abstracción, la relación de estos elementos puede observarse en la Figura 40. De esta manera, se obtiene una interpretación del entorno más útil para el usuario, asi como se facilita la adición de nuevos elementos de forma modular. Al tener elementos definidos, puede añadirse información a cada uno por separado y establecer relaciones entre dichos elementos, generando una estructura rica semánticamente.

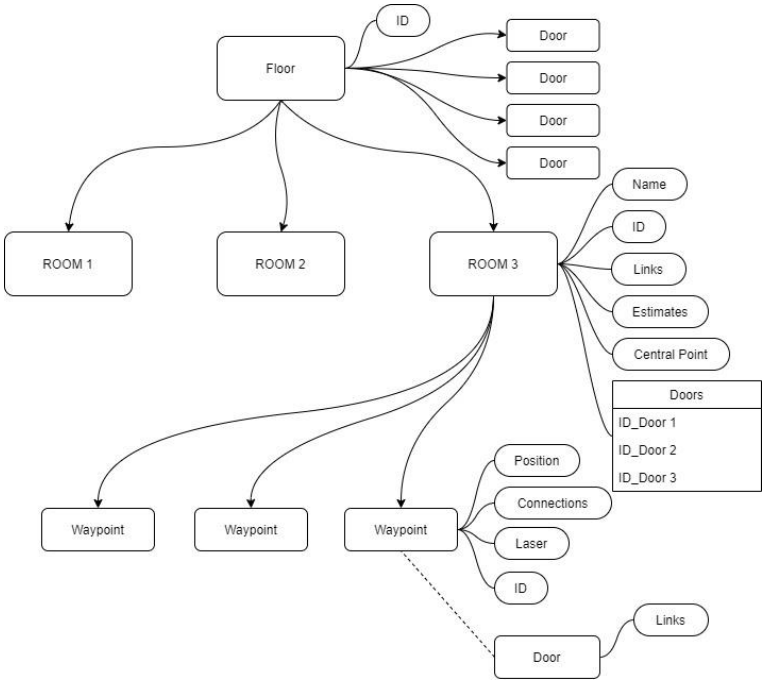

Figura 40. Definición de la estructura de datos. El tipo planta contiene a todos los tipo estancia y puerta. Estancia a su vez, contiene los waypoint.

3.9.1.1. Planta

Una planta es un **elemento constituido por un conjunto de estancias, con una cantidad definida de puertas**. Su función principal es la de constituir el mapa semántico de una planta de un edificio. Ya que contiene toda la información necesaria anidada por grupos. Sus características son:

- ID único: actualmente no es requerido que una planta tenga una ID puesto que solo se tiene acceso a una. Sin embargo, se ha incluido para que en futuras investigaciones pueda trabajarse con varias plantas.
- Estancias: se trata de un array de elementos estancia. De esta manera se anida toda la información de la planta agrupada por estancias, además de dar la capacidad de saber cuántas estancias hay por planta.
- Puertas: guarda una copia de los números de identificación de cada una de las puertas existentes en dicha planta.

3.9.1.2. Estancia

Este elemento ha sido creado para albergar toda la información semántica referente a cada habitación. Una estancia puede definirse en este contexto como un **conjunto de puntos aislados, con una temática similar, que guarda una relación con otros grupos a través de conexiones mediante puertas**. Las características principales de una estancia son:

- ID único: cada elemento posee un número único identificativo. De esta manera, puede realizarse un rastreo de este de forma segura.
- Nombre de la estancia: se genera automáticamente como el tipo de estancia del que se trata
- Links: alberga los IDs del resto de estancias con las que tiene una conexión física. Se entiende por conexión física una puerta en común.
- Puertas: contiene los IDs de las puertas que dan a esta estancia. Son empleados para poder hacer una reasignación o eliminación de las mismas cuando hay una corrección en el mapeado.
- Waypoints: es un array elementos tipo Waypoint. Esto permite definir la topología de cada estancia, y acceder así a las características de cada waypoint.
- Estimaciones recibidas: es un array que contiene todas las estimaciones asociadas a esta estancia. Es utilizado para calcular la estimación con mayor probabilidad y así asignar un nombre a la estancia.

3.9.1.3. Waypoint

Se entiende por waypoint un punto geográfico del mapa. Estos puntos suponen el enlace entre el nivel métrico y el nivel topológico. Dichos puntos se generan equidistantes unos de otros, para constituir una malla. Por tanto, **un conjunto de waypoints definen una estancia**. Las características de un waypoint son:

- ID único: una vez más, para poder realizar un seguimiento de este.
- Posición: elemento de tipo *geometry_msgs::Point* que guarda las coordenadas del punto respecto al origen del mapa.
- Conexiones: array de IDs de los waypoints con los que se encuentra conectado.
- Laser_data: elemento *sensor_msgs::LaserScan* para almacenar los datos del láser obtenidos en ese punto. Actualmente no son utilizados, pero se ha mantenido su definición para en trabajos futuros generar estructuras poligonales más realistas.

3.9.1.4. Puerta

Una puerta es un **tipo concreto de waypoint, por encontrarse en el punto de conexión entre dos estancias**. Al ser un tipo de waypoint, hereda todas las características de este. Solo se añade unacaracterística adicional:

- Link: vector que contiene las dos IDs de las estancias que interconecta.

3.9.2. Segmentación de estancias

El elemento base para una correcta construcción de un mapa topológico es la obtención de las distintas estancias que componen una planta. Para definir la morfología de una estancia, puede extraerse dicha información mediante la recopilación de las distintas poses del robot durante su estancia en la habitación. Para no sobrecargar de datos el sistema, empeorando así las tareas de búsqueda posteriores, se ha optado por recopilar dicha información únicamente cada cierta distancia. Una distancia óptima es entre 1m y 1.2m; de manera que se recopilan pocos puntos, pero se sigue manteniendo la estructura de la estancia.

El problema clave es cómo detectar dónde acaba una estancia, es decir, dónde se está cambiando de lugar. Lógicamente, una habitación finaliza en una puerta, que da a otra habitación. Por lo tanto, es necesario desarrollar un algoritmo capaz de detectar cuándo se está cruzando por una puerta y, de esa manera, poder indicar que ha habido un cambio de estancia y guardar la presencia de dicha puerta.

3.9.2.1. Detección de cruce por puerta

Para resolver la tarea de detectar cuando se está cambiando de estancia, se ha procedido a extraer información del láser. De esta manera, se busca, mediante el análisis de las lecturas del LIDAR, encontrar un patrón que coincida con la forma de un marco de puerta. La forma característica de un marco es una línea recta a cada lado del robot, momentáneamente interrumpida por un salto en la distancia de las medidas. Esta reducción del espacio entre la pared y el robot se corresponde con el marco de la puerta.

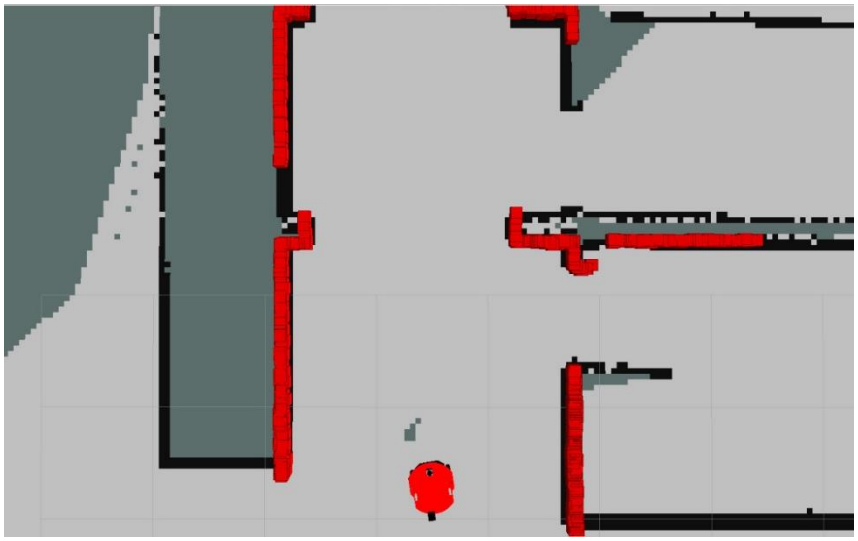

Figura 41. Las medidas recibidas del láser componen un patrón con la forma de los marcos de las puertas. Esta forma puede ser empleada para la detección de las mismas.

El gran hándicap de este método de detección es que existen una gran variedad de casos en los que se puede cumplir esta condición. Por ello, es necesario definir una serie de condiciones extra para eliminar dichas posibilidades.

La primera, es definir la longitud que debe tener la detección para que se considere un posible marco de puerta. Como en esta planta de entrenamiento hay 3 tamaños de puertas distintos, se establecen 3 posibles distancias correctas. La longitud es la suma de la distancia existente entre cada lado y el centro del robot. Para acotar el abanico de posibilidades en las que esto puede ocurrir, se establece que el robot solo puede atacar la puerta perpendicularmente. Resulta intuitivo añadir esta restricción puesto que, tanto los algoritmos de generación de trayectorias como un operador humano, intentarán hacer que el robot cruce perpendicularmente la puerta para evitar posibles colisiones. Como un ángulo de 90º exacto no se va a poder conseguir en la mayoría de las situaciones, hay que definir una horquilla. Tras varios ensayos, se estimó que un rango θ de ±3º es suficiente para detectar cualquier cruce.

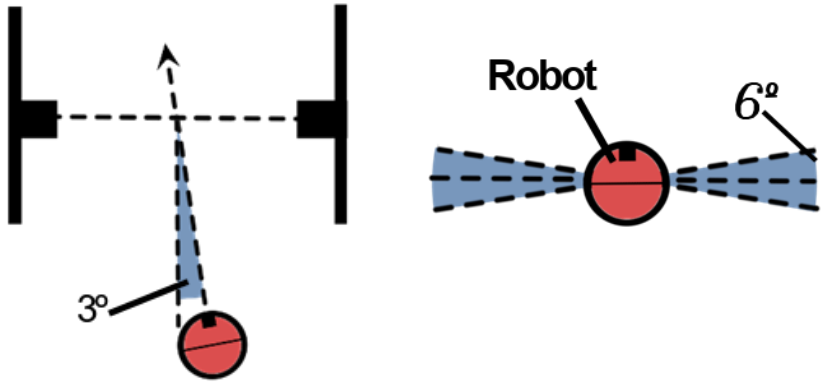

Figura 42. El ángulo de entrada del robot puede ser de ±3º respecto a la normal de la puerta. Esto implica que hay que tener en cuenta todas las medidas en una ventana de 6º para calcular la distancia de la puerta.

Para poder obtener una medida de longitud independientemente del ángulo de entrada, se calcula la media aritmética de todas las distancias dentro de este rango de ángulos. Al calcular el valor medio, la distancia resultante no será la medida exacta de la puerta. Es por esto, que también debe asignarse una variabilidad en la medida. Tras varias pruebas, se asignó la variabilidad en ±0.1m.

$$l = \frac{1}{N} \times \sum_{n=0}^{N} d_n \; ; \; N = número\ de\ muestras\ en\ \theta$$

$$d = l_{izq} + l_{der}$$

De esta manera ya se puede detectar satisfactoriamente marcos de puertas de distintas longitudes. Sin embargo, también serán interpretados como puertas cualquier otro objeto o grupo de objetos que mantengan una distancia similar entre sí. Para solucionar esta cuestión, hay que introducir otro concepto, el de separación. Al instalar una puerta, el marco de esta hace que haya una diferencia de distancia entre la pared y la puerta en sí. Por tanto, para considerar que una superficie, que tiene la longitud correcta, es una puerta, debe existir una distancia mínima entre esta y el resto de la pared. Por ende, es necesario calcular también la distancia existente entre cada pared y el robot. Para ello, se reutiliza la misma fórmula previa para calcular la distancia l, pero en este caso el ángulo empleado θ es de ±30º.

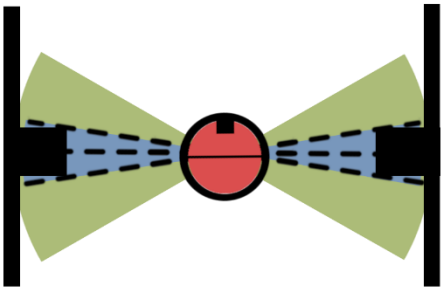

Figura 43. Para calcular el salto de distancias, es necesario crear otra ventana de medida más grande, a parte de la ya existente. Se ha establecido el valor de esta a 60º, siendo 10 veces mayor que la encargada de medir la distancia de la puerta.

3.9.2.2. Flujo de segmentación

Ya se ha visto el patrón a buscar para poder afirmar que se trata de una potencial puerta. Sin embargo, no basta solo con tomar medidas, ya que existen varios casos en los que un objeto puede tener las distancias correctas, pero no ser una puerta. Por ello, es necesario establecer una serie de criterios lógicos.

Puede darse el caso de que la suma de la distancia a cada lado del robot sea la longitud de una puerta, pero estas distancias pueden ser L (longitud de la puerta) y 0. Si se estuviese cruzando por una puerta esta situación no es posible, puesto que el hecho de medir 0 implica que el láser no ha rebotado en ningún objeto. Esto quiere decir que lo q se está midiendo es un objeto a un lateral del robot que justamente está a la distancia exacta de una puerta. Para solucionarlo, se aplica la condición de que ambos lados de la puerta tengan una distancia mínima mayor de 0.

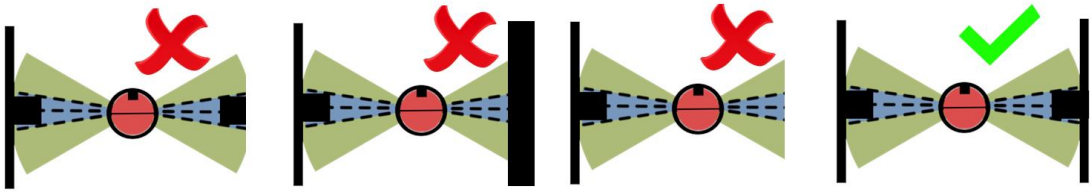

Figura 44. Representación gráfica de las condiciones que se deben cumplir para que se pueda considerar una puerta. Tal y como se puede ver en la última imagen, debe haber dos paredes a una distancia mínima del marco de la puerta a cada lado.

Este mismo efecto también se aplica a las distancias respecto a la pared, por lo que será necesario comprobarlo también. A su vez, se debe comprobar que la distancia mínima entre el marco y la pared se cumple a cada lado, puesto que de lo contrario pueden detectarse objetos erróneamente, como patas de mesa.

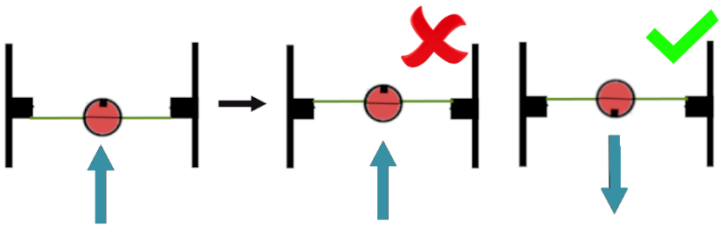

Figura 45. Cuando se detecta una puerta cruzando en una dirección, no se vuelve a transmitir la detección hasta que la vuelva a cruzar en sentido contrario, o cruce otra puerta distinta.

Una vez eliminados los casos de posibles falsos positivos, es momento de adentrarse en evitar la repetitividad. Cuando el robot pasa por una puerta, la velocidad de las lecturas hace que una misma puerta

pueda ser detectada sucesivas veces mientras es cruzada. Para este problema, se han de tener en cuenta dos características del robot, la posición y la dirección. Resulta intuitivo comprender que hay una distancia mínima entre una puerta y la siguiente, lo suficientemente grande como para poder asegurar que hay una zona alrededor de cada puerta que, si se da un positivo, se trata de esa puerta. Por tanto, si se detecta una nueva puerta y acto seguido se sigue detectando un positivo, se puede asumir con seguridad que se trata de la misma, y por tanto no debe de notificarse. Sin embargo, esto por sí solo invalida el algoritmo entero, ya que, si impedimos que se vuelva a detectar la última puerta, cuando se desee salir de la estancia no se notificará que se ha cruzado la puerta. Para solucionar esto, entra en juego la dirección del robot. Supóngase el ejemplo de que el robot va a cruzar una puerta en dirección norte. Una vez entre a la estancia, si desea salir por la misma puerta, deberá hacerlo en dirección sur. Por tanto, basta con impedir la detección de una misma puerta únicamente si se ha detectado en la misma dirección que la vez anterior, estando dentro del área de confianza de la puerta.

3.9.3. Extracción de información semántica

Para la obtención de información sobre el entorno, se ha utilizado una cámara RGB, que va tomando capturas desde el robot conforme se va desplazando. Para el tratamiento de la imagen, esta es enviada a la nube donde se extrae el contexto de la imagen. Este contexto puede ser aplicado posteriormente a cada estancia.

3.9.3.1. Obtención de imágenes

Para la generación de imágenes, cada vez que el nodo encargado de crear el mapa topológico genera un nuevo *waypoint*, se envía una petición al servicio *image_save* para que se guarde el fotograma actual en una carpeta temporal específicamente creada para ello. Acto seguido, se publica un mensaje booleano sobre el topic *topomapping/upload_photo* para indicar al nodo encargado de comunicarse con la nube. De esta manera, el nodo busca la última imagen guardada para subirla.

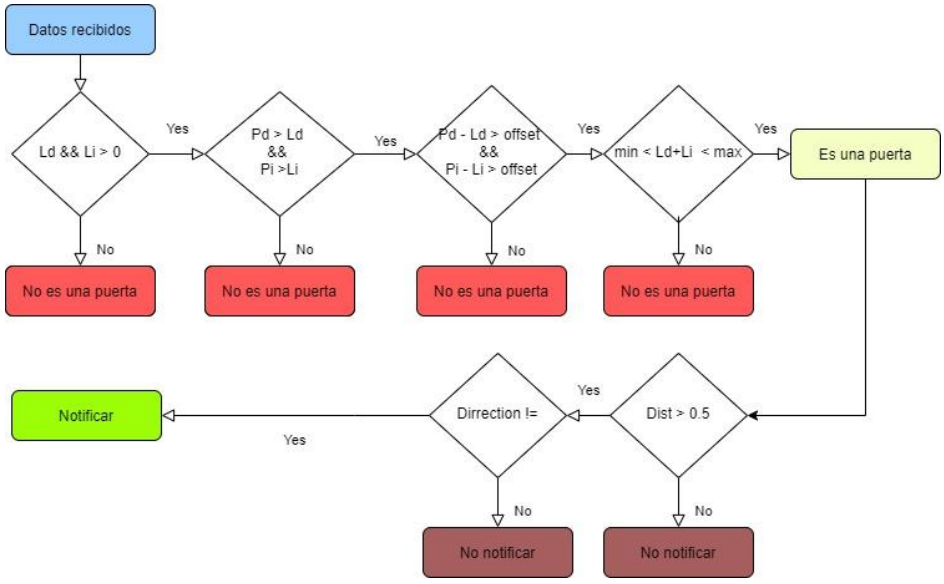

Figura 46. Flujo de comprobaciones para la notificación de una segmentación.

3.9.3.2. Petición de análisis

El servicio de Google utilizar en este proyecto es *Google Vision AI API*. Este servicio en la nube tiene su propia librería de Python para comunicarse con él. El método de comunicación es una petición http en la que se envía un archivo en formato JSON, donde se incluye la imagen codificada en base 64 y todos los tipos de detección que deseamos que se realicen.

Para realizar esta petición con la librería, basta con indicar la ruta absoluta a la imagen, además de indicar el tipo de análisis, como se indica a continuación.

```
class Vision_AI:

    @staticmethod
    def Make request(path):
        # Imports the Google Cloud client library


        # Instantiates a client
        client = vision.ImageAnnotatorClient()

        # The name of the image file to annotate
        file_name = os.path.abspath(path)

        # Loads the image into memory
        with io.open(file_name, 'rb') as image_file:
            content = image file.read()

        image = vision.Image(content=content)

        # Performs label detection on the image file
        response = client.label_detection(image=image)
        labels = response.label_annotations

        return labels


folder = "/home/pablo/catkin_ws/images/"
files_path = os.path.join(folder, '*.jpg')
files = sorted( glob.iglob(files_path), key=os.path.getctime, reverse=True)
this_file = files[0]

cloud = Vision_AI()
result = cloud.Make_request(this_file)

with open('/home/pablo/catkin_ws/src/topomapping/tmp/Vision_AI_output.txt', 'w') as outfile:
    outfile.write(str(result))
```

Como se observa en el método, nos devuelve la respuesta de la detección en un array de elementos tipo *dict*. Este array pasa a ser escrito en un documento de texto temporal, de manera que pueda ser leído por el nodo de ros encargado de la clasificación de los resultados.

3.9.3.3. Clasificación de resultados

Como respuesta a una petición de detección, Vision AI devuelve un listado de todos los resultados encontrados con una probabilidad asociada a cada uno. Estos resultados son guardados en un documento de texto para que el nodo publicador pueda recopilarlos y clasificarlos. Debido a que son tags estandarizadas y genéricas, no se adaptan correctamente a los resultados que se esperan. Para poder realizar una correcta clasificación, es necesario crear los tipos de estancias entre los que se desee que se clasifiquen. En esta definición, se especifica un nombre para ese tipo de estancia y un listado de los tags generados por Vision AI que están relacionados con esa estancia.

Figura 47. Respuesta de ejemplo al análisis de una imagen. Se recibe un grupo de tags procedentes del análisis. Este grupo es filtrado por las listas de tags relevantes, que han sido definidas con prioridad. A partir de los tags recibidos, se calcula un índice de confianza para cada tipo de estancia.

Todos los tags generados no son de igual relevancia. Por ejemplo, si en una clase se detecta el tag "colorido" y el tag "pizarra", está claro que "pizarra" es un tag clave para la estancia clase, mientras que la posibilidad de que aparezca en otras estancias es nula. Por esto, se han creado dos clases de tags diferenciables, las normales y las exclusivas. Cuando se recibe un tag normal, se añade al listado de tags detectados del tipo de estancia correspondiente y se calcula la probabilidad media total. Sin embargo, si el tag recibido es exclusivo de una estancia, se añade a dicha estancia pero, acto seguido, no se siguen analizando más tags. Esta solución resulta un método para evitar que otros tipos de estancias obtengan una mayor probabilidad. Así, la presencia de un tag exclusivo asegura la selección de la estancia correspondiente, pero sin perder la probabilidad asociada de la estancia.

3.9.3.4. Publicación de resultados

Una vez calculadas las probabilidades medias asociadas a cada estación, son enviadas en un mensaje de formato propio, creado en específico para este paquete. En este mensaje se incluye la fecha y un vector de elementos tipo nombre-probabilidad.

```
#Definition of probability_element
string name
float64  prob

# Definition of probability_element_array message
Header header
time stamp
probability_element[] items
```

Este mensaje es posteriormente recibido por el nodo generador del mapa semántico para asociarlo a la estancia correspondiente.

3.9.4. Creación del mapa híbrido

Finalmente, vamos a ver como se realiza el proceso principal de este proyecto, que es la unificación de toda la información recibida en único mapa que contenta una representación más enriquecida del entorno.

3.9.4.1. Flujo de recepción de datos métricos

Anteriormente se ha visto como se realiza la segmentación de estancias mediante la detección de puertas. Este proceso es llevado a cabo por un nodo independiente que va analizando todos los mensajes generados por el láser. Tras analizar dichas medidas, este nodo publica un mensaje booleano especificando de si se trata de una puerta o no.

Por otro lado, en el nodo principal está subscrito tanto a este *topic* como al *topic* donde se publica la pose del robot. Esta pose es utilizada para mantener siempre el valor de la posición del robot. De esta manera, cuando se reciba un mensaje confirmando si se ha detectado una puerta o no, el nodo ya posee la información de la posición del robot.

Al recibir el mensaje, se comprueba si es una puerta o no. En caso de serlo, entonces se revisarán las puertas ya existentes en la planta para comprobar si ya existe. En caso de no existir, se crea una nueva, asignándola a las estancias que corresponda. En caso de existir, se procede a cambiar de estancia a la que esté asociada a la puerta correspondiente.

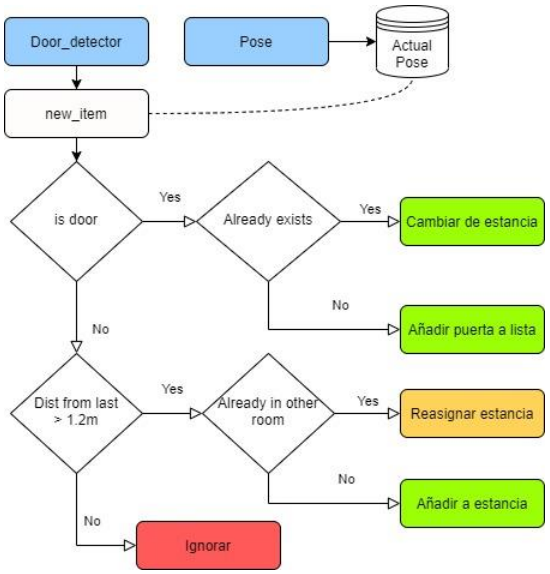

Figura 48. Flujograma de recepción de datos métricos. En esta figura aparece incluido el lazo cerrado que permite reasignar habitaciones si fuese necesario.

Cuando el mensaje recibido no se corresponde con una puerta, entonces se pasa a la inclusión de waypoints. La condición principal para generar un waypoint es que se encuentra a una distancia mínima de 1.2 m de todos los waypoints ya existentes en la estancia actual. Cumpliendo estas condiciones, se añade el waypoint a la estancia correspondiente.

3.9.4.2. Clasificación de estancias

Tal y como se ha comentado con anterioridad, para la obtención de información extra del entorno, se realiza un análisis de imágenes en la nube que devuelve una serie de tags con probabilidades asociadas. Estos tags son posteriormente analizados para calcular qué estancia se ha detectado y con qué probabilidad. Para ello, es necesaria la definición de un listado de estancias y asignar a cada una los tags con las que se encuentren relacionados. De esta manera, el nodo encargado de la detección puede calcular la probabilidad de que sea una estancia a partir de los tags recibidos.

Cuando la respuesta es recibida por el nodo gestor, este las almacena en la estancia correspondiente. De esta manera, se realiza un cálculo para averiguar cuál es la moda de las respuestas recibidas, y se calcula la probabilidad media de esta a partir de todas sus respuestas.

3.9.4.3. Corrección con lazo cerrado

Hay ocasiones, en las que existen estancias a las que se puede acceder desde varios puntos. En otros casos, varias estancias dan a otra común (un pasillo) y, a su vez, estas están interconectadas entre sí. Esto es un problema potencial a la hora de generación de mapas, ya que se puede generar dos veces un mismo pasillo y no tener todas las conexiones en cada copia. Para solventar este problema, se ha implementado un algoritmo de lazo cerrado que permita fusionar las estancias cuando ocurren estos casos.

La forma de implementarlo se trata de, tras comprobar que un waypoint cumple las condiciones para ser guardado, se comprueba si esa posición ya existe en otras estancias. Si se cumple que ya existe en otra estancia, se inicia un proceso de migración. En este proceso se copian todos los waypoints de la estancia actual a la estancia donde se ha realizado la detección. Tras esto, se modifica las puertas de la estancia actual, cambiando su id por la de la estancia destino. Acto seguido, se copian dichas puertas a la estancia destino. Por último, se elimina por completo la estancia actual y se cambia de estancia a la de destino.

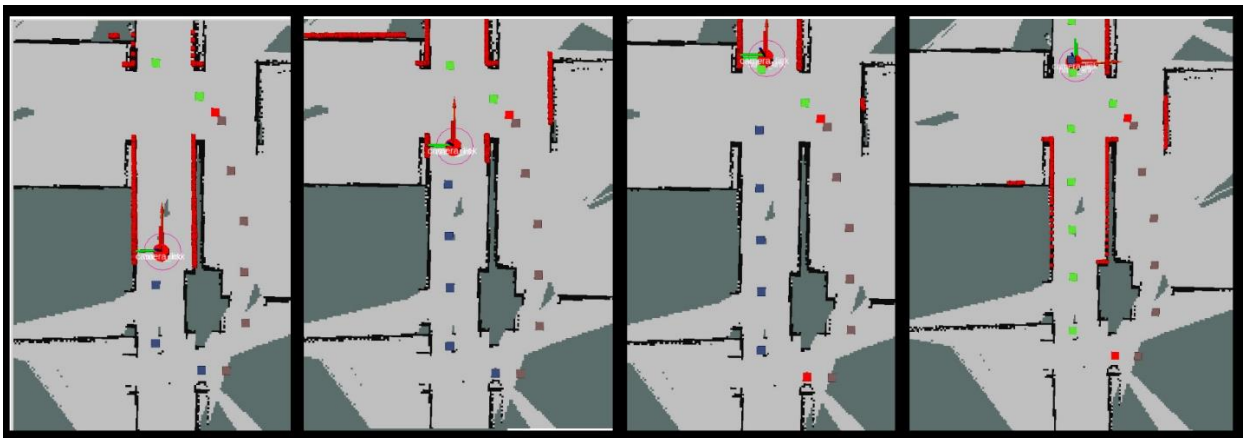

Figura 49.En esta secuencia el robot vuelve al pasillo central desde una estancia diferente de por la que entró. Va creando una estancia nueva hasta que topa con los waypoints del pasillo original (verdes). Como no ha cruzado una puerta de por medio, da por hecho que se trata de la misma habitación, reasignando todos sus puntos (azules) a esta.

3.9.4.4. Asignación de información semántica

El nodo generador del mapa se subscribe al *topic* donde se están publicando los resultados de las detecciones. Cuando un mensaje es recibido, se asignan los resultados a la estancia actual. Acto seguido, se lanza una función para recalcular las probabilidades medias de la estancia. De esta manera lo que se consigue es hacer que, a mayor cantidad de imágenes de una estancia, más posibilidades se tiene de aumentar la probabilidad media de la solución correcta. Tras hacer el cálculo de los valores medios, se realiza una normalización de estos valores.

Finalmente, se asigna al nombre de la habitación el nombre de la estancia más probable con su probabilidad normalizada.

3.9.4.5. Generación del mapa

Tras la generación de cada nuevo waypoint o puerta, se llama a una función encargada de realizar la representación de todo el mapa topológico obtenido hasta el momento. Para realizar esta tarea se va a publicar un total de 3 mensajes.

En el primer mensaje se envían todos los waypoints. Para ello, se va iterando entre las distintas estancias de la planta. A cada estancia se le asigna un color generado de manera aleatoria y almacenado para evitar que se repita. El tipo de mensaje es un *visualization_msgs::Marker::SPHERE_LIST* , donde los waypoints quedan agrupados en listas por estancias. Estas esferas han sido definidas para que se generen a medio metro de altura. De esta manera se facilita su visibilidad al no solaparlas con el mapa ni el robot.

El segundo mensaje alberga las puertas de la planta. El tipo de mensaje escogido en este caso es *visualization_msgs::Marker::POINT*, para que sean distinguibles en forma respecto a los *waypoints*. A su

vez, se les ha asignado el color rojo, para que todos tengan un color uniforme y puedan identificarse rápidamente al visualizar el mapa.

El último mensaje es el que contiene los nombres de las estancias con sus probabilidades. Para ello, se ha escogido el mensaje *visualization_msgs::Marker::TEXT_VIEW_FACING*. Este tipo tiene la característica especial de mantener el texto siempre orientado al usuario, de manera que siempre serán legibles todos los nombres. En este caso, se generan los nombres a una altura de 1m, para separarlos de los waypoints. En lo referente a donde colocar el texto, se ha creado una variable asociada a cada estancia, donde se guarda el centro de masa de la estancia. Este centro de masa es obtenido calculando el valor medio de la posición de todos sus waypoints.

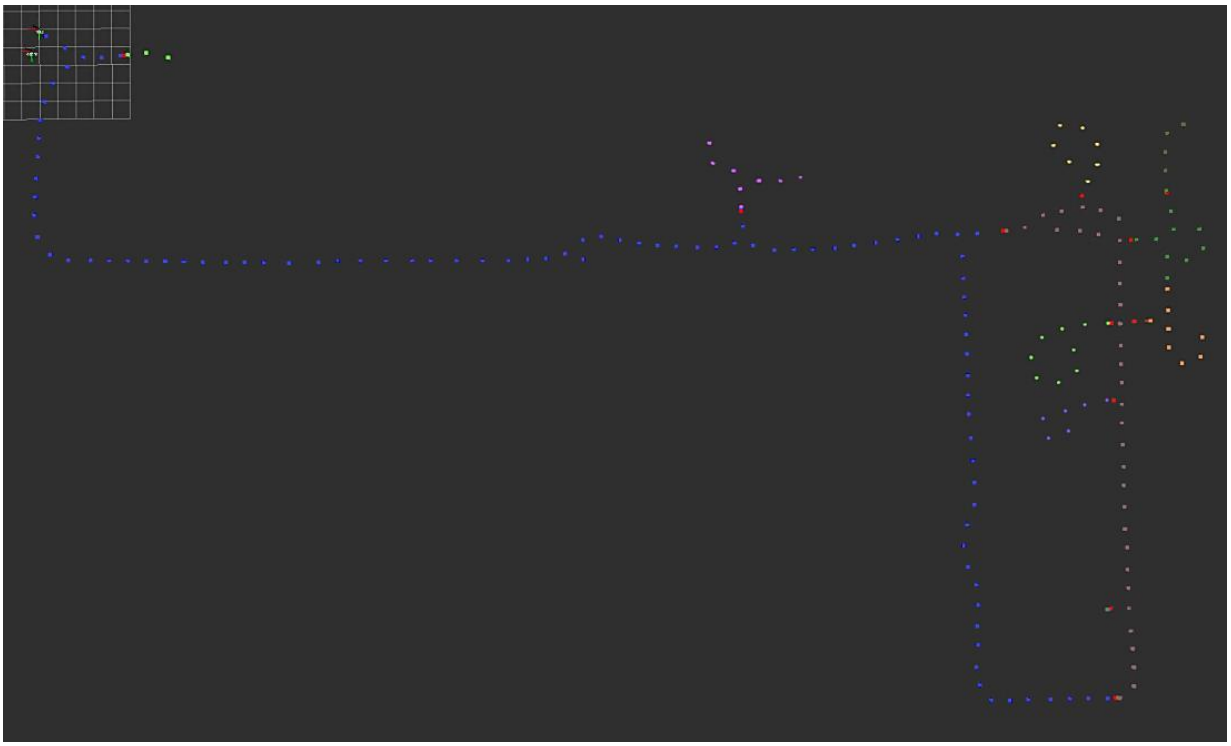

Figura 50. Mapa topológico generado siguiendo la estrategia propuesta en este trabajo. Puede observarse la segmentación de estancias por colores y las puertas resaltadas en rojo. Se ha desactivado la visualización de los nombres de las estancias para un mayor detalle.

4. Análisis de resultados

4.1. Selección entre Gmapping y Hector Slam

Durante el desarrollo de este proyecto, se ha analizado y puesto en marcha 2 paquetes de software centrados en el proceso de SLAM. Estos dos paquetes son capaces tanto de localizarse dentro de un entorno desconocido como de generar un mapa de este. Sin embargo, estos paquetes se han desarrollado desde dos puntos diferentes para llegar a una misma meta.

Por un lado, Gmapping aporta una solución al problema de la realización de SLAM focalizada a su uso para la generación de mapas de gran extensión, en entornos con cierto grado de control. Para la realización de estos mapas, se basa en los datos recibidos de un LIDAR y de la odometría del robot. Este paquete está pensado para usarlo con unidades de larga distancia, de manera que se puedan obtener varias marcas de referencia visibles durante varios metros de recorrido. Sin embargo, el modelo usado para este proyecto tiene un alcance máximo de 6 metros, lo cual no lo hace idílico para trabajar con Gmapping. A pesar de ello, no acaba resultando en un problema ya que, aunque en las estancias más grandes tiende a generarse cierta curvatura en las paredes, este paquete posee un robusto algoritmo de lazo cerrado, que le permite corregir estas desviaciones en situaciones críticas.

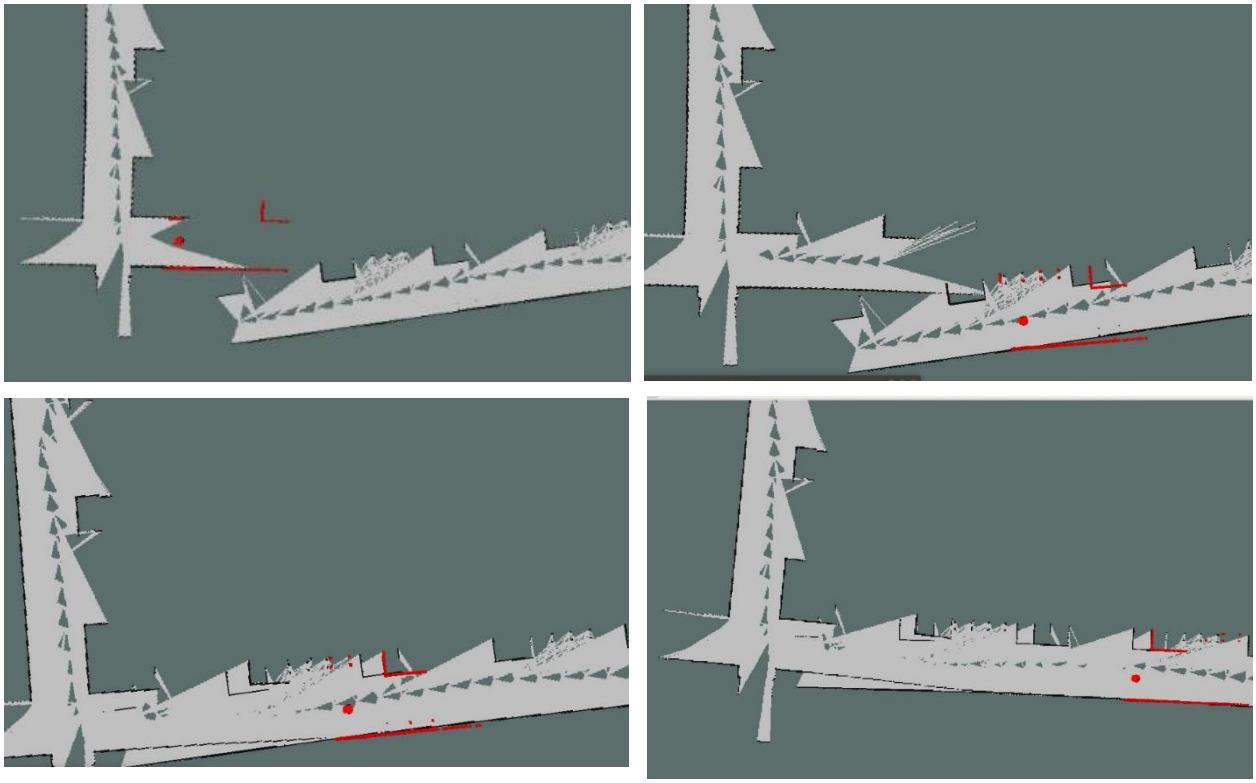

Figura 51. Secuencia de generación del mapa en la que entra en juego el lazo cerrado. Al dar la vuelta a todo el patio existe una discordancia en las distancias de este. Por tanto, al pasar por una zona que ya había recorrido antes, intenta localizarse en el punto más cercano y corrige todo el mapa.

Un buen ejemplo de esta solución es en la generación de los patios de la ETSII. El patio hace las veces de pasillo para interconectar varios departamentos. Las dimensiones del patio son de varias decenas de metros, por lo que claramente se van a generar curvaturas en el mapa. Por suerte, gracias al lazo cerrado, cuando el robot completa una vuelta al patio, y llega a una zon a por la que había pasado previamente, realiza una corrección de todo el mapa generado del patio para hacerlo coincidir. De esta manera se reducen considerablemente las deformidades, aunque sin llegar a desaparecer. A pesar de dichas deformidades, como se ha conseguido hacer coincidir el pasillo, estas no afectan al funcionamiento adecuado del robot.

La única desventaja de este paquete es que requiere un mayor consumo de capacidad de procesamiento. Sin embargo, se encuentra dentro de las capacidades de este robot ya que mediante el empleo del RBPF se consigue optimizar el análisis y centrarlo en las soluciones potencialmente acertadas.

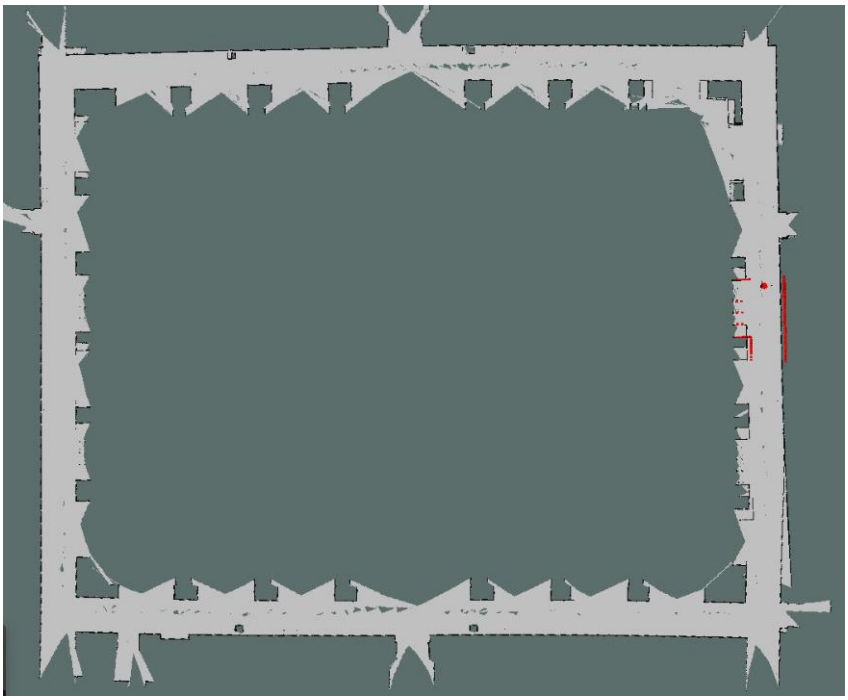

Figura 52. Mapa del patio oeste tras haber realizado varias vueltas alrededor de él, en ambos sentidos. Puede observarse como la discordancia original ha sido solventada. A su vez, se han corregido varias deformidades en la longitud de los pasillos.

Por otro lado, Hector intenta aportar una solución de bajos requerimientos para que sea adaptable a cualquier unidad. Claramente, esta cualidad no se encuentra exenta de contras. El primero contra es la no posesión de un lazo cerrado que permita la corrección del mapa. Otra característica es que prescinde del uso de los datos de odometría. Esto puede resultar una ventaja en entornos altamente accidentados, para los cuales fue pensado el uso de este paquete, ya que los desniveles o posibles desplazamientos repentinos del robot no afectan directamente a la localización del robot. Esto es debido a que, en un desplazamiento imprevisto, tanto lateral como en altura, puede provocar una indicación errónea del desplazamiento real realizado por la odometría. En contrapartida, el algoritmo necesita encontrar muchos más puntos de referencia en las lecturas del láser para mantener una localización consistente.

Como sea mencionado, en entornos accidentados esto no supone un gran problema ya que tienden a existir una gran cantidad de objetos a corta distancia del robot, permitiendo así un cálculo sencillo del desplazamiento. Para la aplicación para la que se está desarrollando este proyecto, su uso en grandes interiores, esta característica no es nada favorable. De hecho, resulta en una completa pérdida de la distancia recorrida en estancias grandes donde apenas existen objetos de referencia, como es el caso de los pasillos.

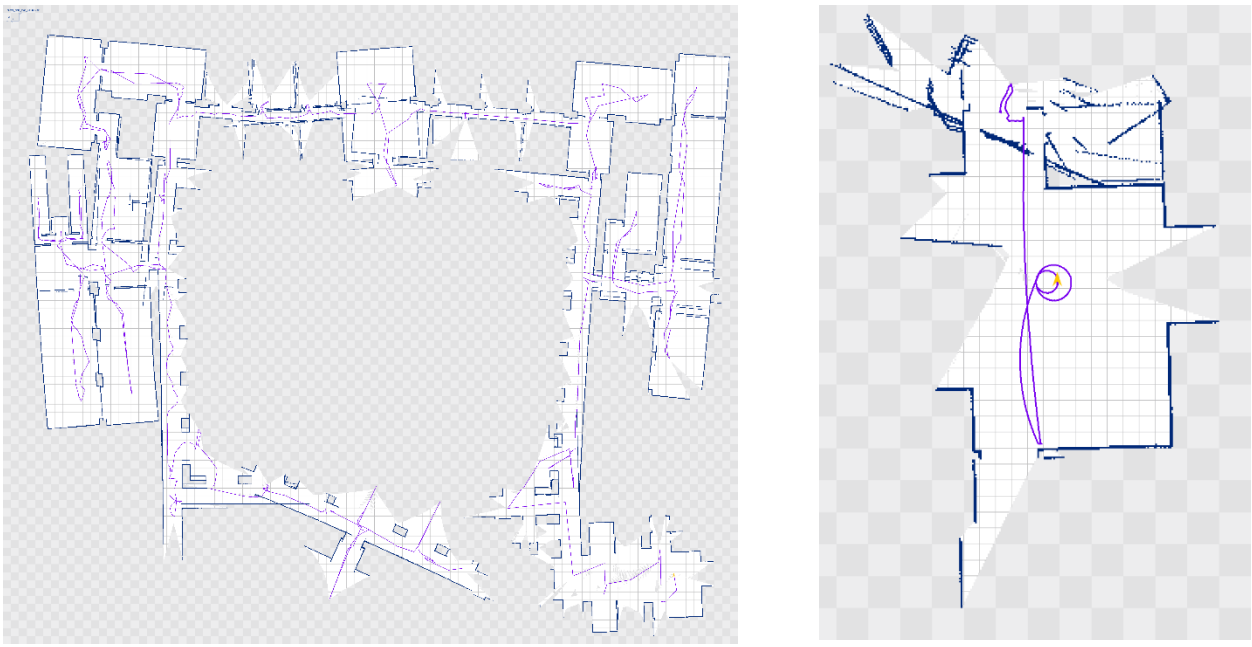

Figura 53. Mapa generado en una de las varias pruebas de funcionamiento realizadas a Hector Mapping (izquierda). En la imagen ampliada (derecha), se puede observar cómo Hector Exploration Controller traza trayectorias circulares, lejos de seguir la ruta planeada y termina chocando con objetos del entorno.

Esto es debido a que la varianza de las posibles poses es excesivamente grande por la falta de geometría de referencia con la que contrastar.

Como se ha podido ver, el hecho de que Hector Slam prescinda de las lecturas de odometría hace que no sea viable su aplicación en este proyecto. Cada vez que haya un fallo de localización se van a generar los objetos en una parte distinta del mapa a la que estaban originalmente, provocando que se produzcan copias de la misma estructura desplazada tanto lateral como angularmente. Esto genera una gran imprecisión en el resultado final. A su vez, la ausencia de un algoritmo de lazo cerrado hace que las deformaciones generadas en estancias grandes resulten en un mapa solapado y, por lo tanto, inservible para posteriores localizaciones.

Por tanto, aunque Gmapping es un paquete más pesado, ofrece una solución más adecuada para campo de aplicación en el que se está trabajando. En su mayoría, gracias a la robustez que le proporciona la posesión de un lazo cerrado. Para ello, se han realizado varias tandas de pruebas de recorrido en el modelo de la ETSII del entorno simulado para ambos paquetes.

4.2. Algoritmo de segmentación

Tras las pruebas de detección de puertas realizadas en la planta de la ETSII, se ha podido comprobar que, si bien la detección de patrones por sí sola no es suficiente, al añadirle las condiciones lógicas se consiguen eliminar todos los falsos positivos existentes. Claramente, al haber reducido tanto el ángulo permisible en el que se puede detectar una puerta, existen casos en los que se produce un falso negativo. Estos falsos negativos se producen la mayoría de las ocasiones en las puertas dobles. Esto es debido a que, al ser más ancha la zona de cruce, se posible para el robot atravesarlas con un ángulo más pronunciado.

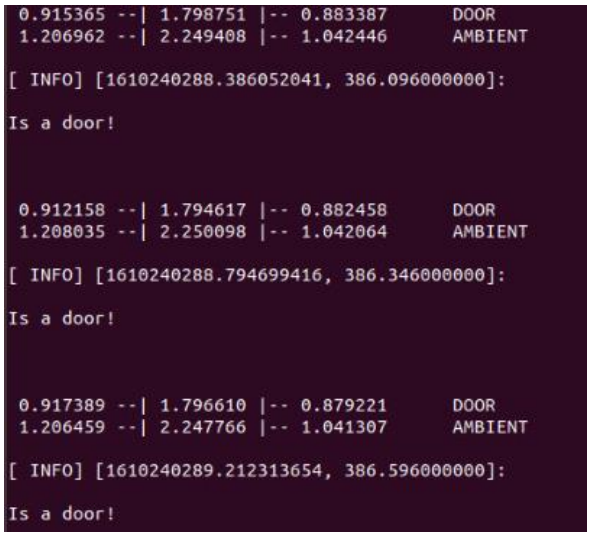

Figura 54. Información mostrada por el nodo detector de puertas al usuario. Puede observarse la distancia medida en cada dirección.

Sin embargo, es más sencillo para el usuario reconocer dónde es posible que haya una estancia que no se ha detectado, que tener que discernir cuál de las estancias detectadas es un error de detección. Es por este motivo que se ha dado preferencia a la eliminación de falsos positivos, tarea resuelta satisfactoriamente.

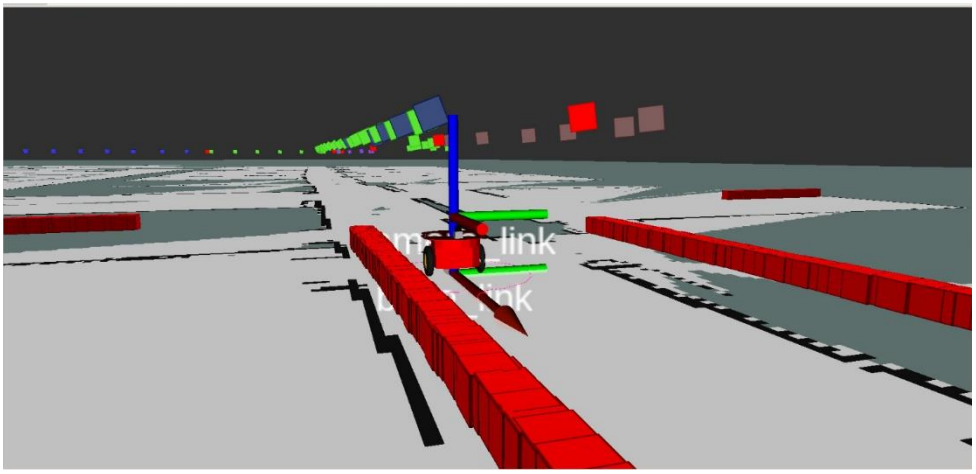

Figura 55. Visualización del proceso de segmentación topológica desde RVIZ.

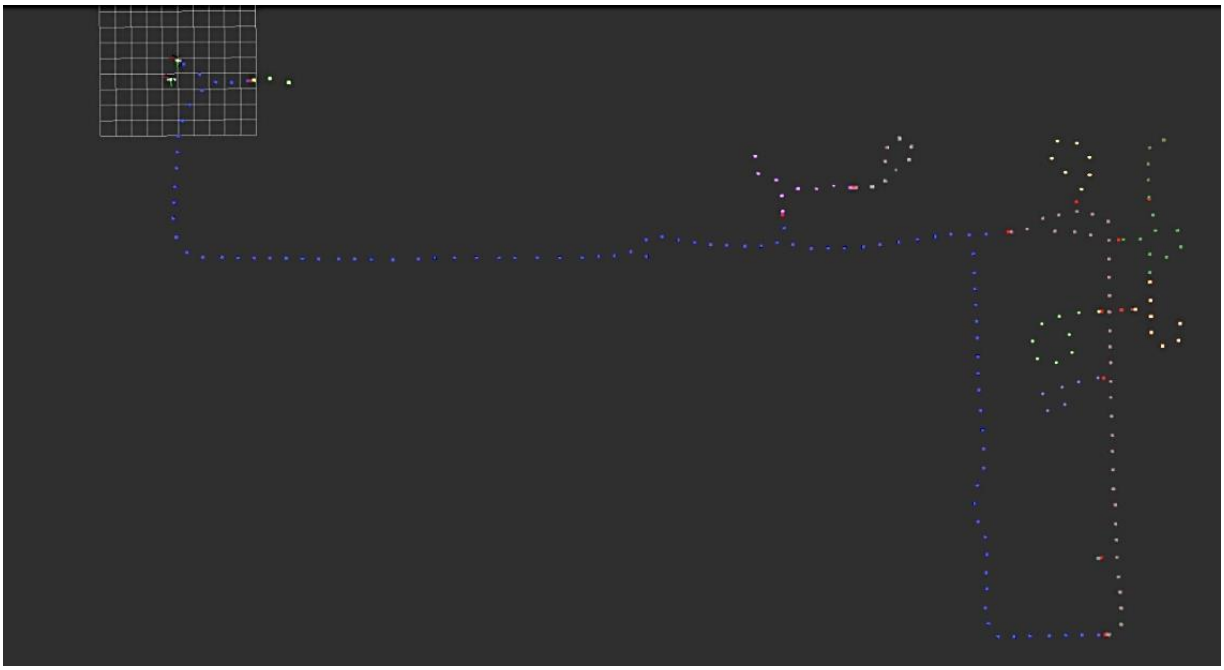

Figura 56. Mapa de segmentación topológica generado en una prueba.

4.3. Extracción de información semántica

En lo referente a la generación de imágenes y posterior creación de peticiones en la nube, el paquete desarrollado ha resultado resolver dicha tarea eficientemente. Para ello, se ha realizado una colección de capturas de estancias modelo de cada tipo en el entorno de simulación. Tras analizar cada una de ellas, se han proporcionado las respuestas al nodo clasificador, el cual ha devuelto la estancia correcta en todos los casos. A su vez, se ha realizado el mismo procedimiento con un set de imágenes reales del edificio para asegurar que el modelo es eficiente en un entorno real.

En cuanto a la calidad de la clasificación de las estancias es necesario hacer un apunte. Con la creación de un listado de tags para cada tipo de estancia, se ha conseguido que siempre obtenga un porcentaje de probabilidad mayor la estancia correcta. Sin embargo, se ha encontrado una clara diferencia entre imágenes obtenidas en el simulador e imágenes de entornos reales.

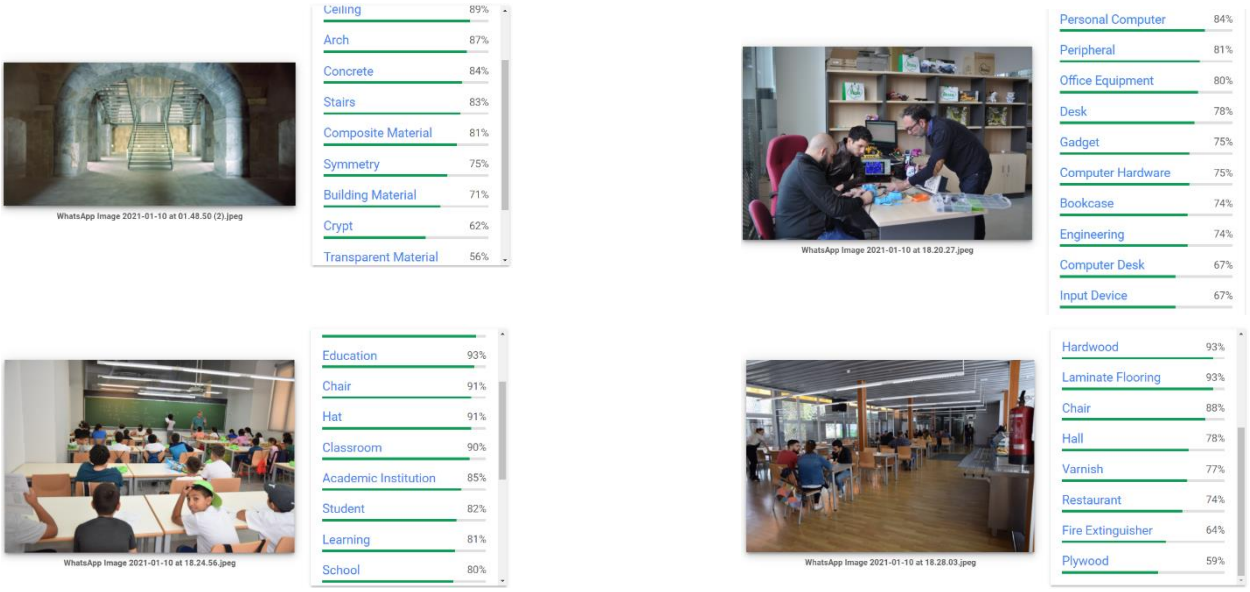

Figura 57. Pruebas de extracción de información con imágenes reales de ejemplo. Se puede observar como en todos los casos puede obtenerse al menos una respuesta fuertemente ligada al tipo de estancia.

Cuando se realiza un análisis de una imagen creada en el simulador, la probabilidad de que sea la estancia correcta es baja, aunque sigue siendo la mayor respecto al resto de estancias posibles. Mientras que una imagen real para la misma estancia muestra resultados mucho más favorables. Esto es debido a que cuando se envía una imagen simulada, los tags devueltos por la red neuronal de esta api son en su mayoría referentes a que se trata de un software (simulador, videojuego, programa de CAD, diseño, etc.) como contexto de la imagen. Mientras tanto, si la imagen es real, el contexto que devuelve es en la mayoría de los casos el acertado. Esta gran diferencia muestra claramente que esta red no se encuentra específicamente entrenada para el reconocimiento de estancias, sino que es una red neuronal de propósito general, dentro de la detección de contexto en la imagen.

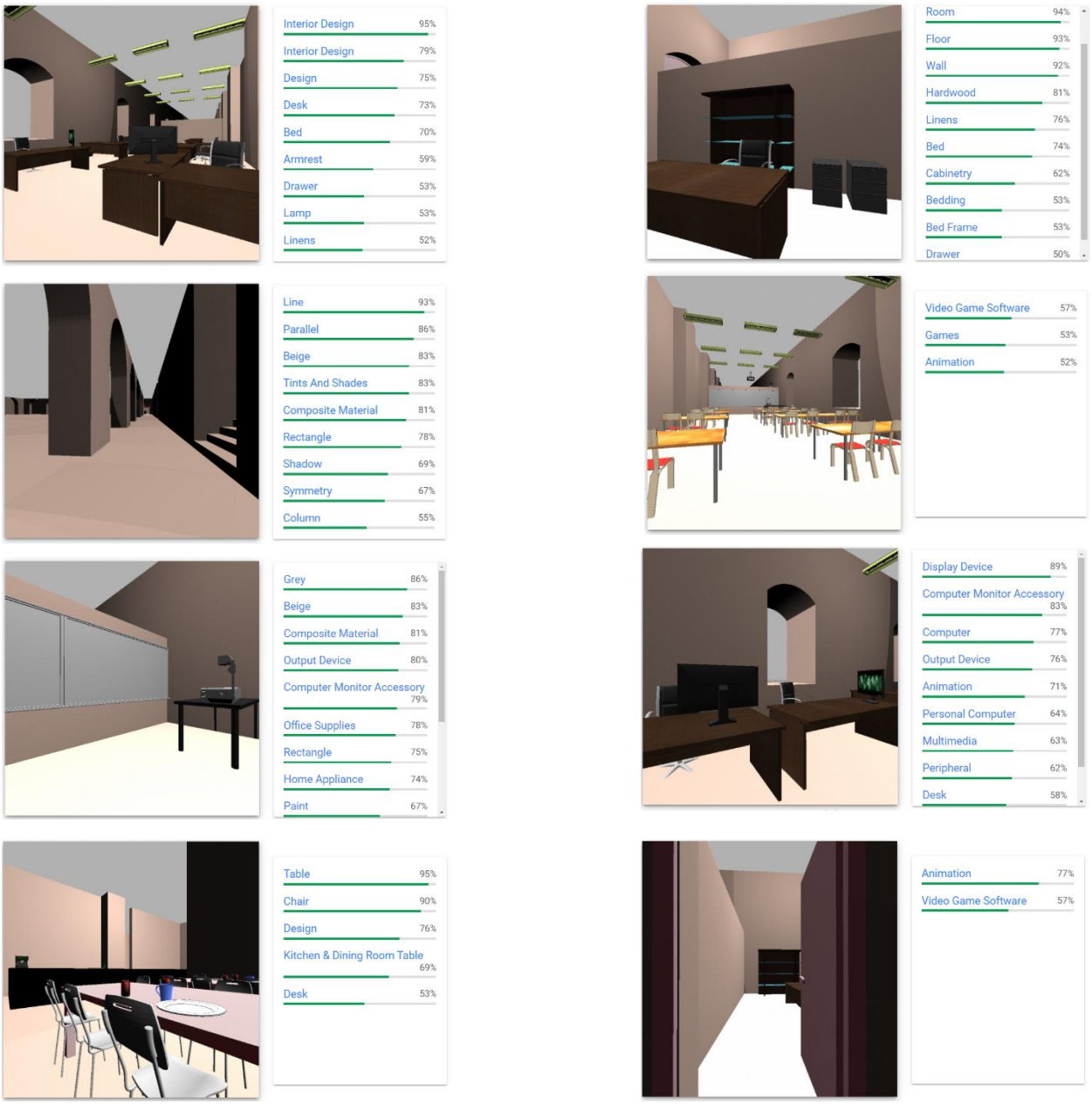

Figura 58. Resultados del análisis semántico de las imágenes capturadas por el robot durante una simulación. Se puede observar cómo no en todas las capturas es posible obtenerse información relevante, debido a una tendencia del modelo de IA a deducir que el contexto es un videojuego o simulación.

A pesar de ello, no es un problema para el desarrollo de este proyecto, puesto que estos paquetes son desarrollados para su uso en entornos reales. Los entornos simulados son únicamente empleados en fase de prueba. Aun así, esta API sigue siendo capaz de detectar correctamente las estancias en entornos simulados, únicamente con un menor grado de confianza.

4.4. Generación del mapa topológico

La creación de un mapa topológico mediante la generación de puntos equidistantes ha permitido mantener la morfología básica de las estancias, pero librando al usuario de una interpretación de bajo nivel (identificación de elementos estructurales, zonas transitables, etc.). De esta manera, se recibe directamente el espacio transitable existente y las zonas de interconexión entre estancias. Por otro lado, la clasificación de las estancias por tipo permite un fácil reconocimiento del entorno para el usuario.

Tras la generación de varios mapas en el entorno simulado, se puede apreciar como la inclusión de condiciones lógicas en la detección de puertas ha ayudado sustancialmente a la detección de estancias. A su vez, el hecho de la estructura en forma de malla de los waypoints ha hecho posible la detección de un lazo cerrado en una estancia abstrayéndose de la geometría del entorno; sin el uso de las medidas del láser para buscar coincidencias previas. Esto es debido a que el espacio entre los waypoints permite la definición de una región de inclusión en la que, si el robot se encuentra dentro, se puede asumir que se encuentra en ese waypoint.

Se considera que puede llegar a darse el caso en el que, estando en una habitación, el robot se encuentre dentro de la zona de inclusión de un waypoint de la estancia contigua. Sin embargo, no se ha podido conseguir este efecto en las diversas pruebas realizadas, por lo que este método se considera seguro. En caso de que llegase a suceder, puede solucionarse reduciendo el radio del área de inclusión. Por defecto, este radio se encuentra definido a menos de la mitad de la distancia entre dos waypoints.

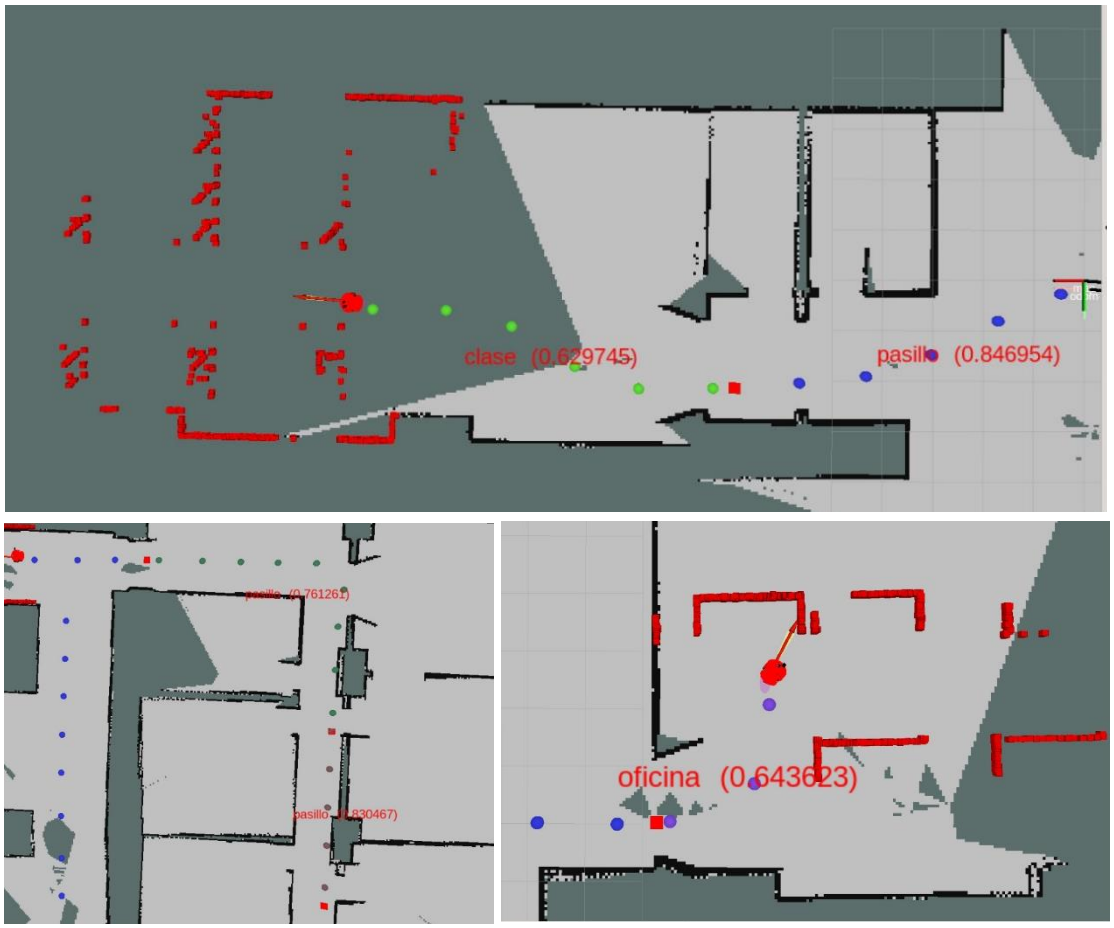

Figura 59. Resultados de creación de un mapa topológico enriquecido semánticamente. En este mapa puede verse la morfología de las estancias, separadas por colores junto al tipo de estancia del que se trata (se aporta un grado de confianza respecto a la estimación).

5. Conclusiones y vías de avance futuras

5.1. Conclusiones

Como se comentó al comienzo de este proyecto, actualmente existen varias soluciones para la generación de mapas de entornos interiores. Sin embargo, estos se encuentran fuertemente enfocados a la automatización de los desplazamientos de las unidades robóticas móviles en entornos muy controlados y definidos. En cambio, aún no existe un método de generación de mapas autónomo definido que permita su uso directo por parte del usuario final y, de esta manera, mantener una comunicación más fluida con el robot. Tras la realización de los diferentes hitos necesarios en este proyecto, como la obtención de un método de realizar SLAM, la creación de un entorno virtual de la ETSII a escala 1:1 y la comunicación con servicios de Google en la nube; se ha conseguido implementar satisfactoriamente un nodo de ROS que sea capaz de fusionar el mapeado métrico con la estructuración semántica de un entorno. A su vez, se ha integrado todas las funcionalidades en un paquete de este ecosistema, siguiendo los estándares de la comunidad en cuanto a nomenclatura y estructuración.

De esta manera, se ha conseguido el desarrollo de un paquete que proporcione la creación de mapas más ricos en información y fácilmente interpretables por el usuario final. Manteniendo un consumo de recursos reducido, mediante el procesado de imágenes con servicios en la nube, se permite la implantación de esta funcionalidad en todo tipo de robots, incluyendo aquellos de bajo coste que poseen un hardware más simple. Por último, se ha puesto a disposición de la comunidad de desarrolladores de ROS mediante la creación de un repositorio público, específico para este paquete, en la herramienta de integración continua GitHub.

Se espera que los avances conseguidos en este trabajo sirvan para el desarrollo de aplicaciones más autónomas y versátiles en el ámbito de la robótica de servicios, donde los robots necesitan alejarse de los métodos de guiado tan restrictivos empleados hasta en el momento en el sector industrial. En los últimos años puede apreciarse claramente la tendencia al empleo de herramientas de inteligencia artificial en el sector autónomo, por lo que es de obligada necesidad la creación de paquetes de funcionalidades orientadas a la obtención de información semántica que permita y motive la integración de estos sistemas inteligentes. De esta manera, se conseguirá una mayor cantidad de información del entorno, provocando que los robots sean cada vez más capaces.

5.2. Vías de avance futuras

5.2.1. Mejoras en el modelo de la ETSII UPCT

5.2.1.1. Creación de texturas específicas para la edificación

Si se desea mejorar la eficiencia de los métodos actuales para la extracción de información semántica en un entorno simulado en Gazebo, es necesario la creación de unas texturas para el edificio. Actualmente, el método empleado para exportar el archivo a formato COLLADA hace que se pierdan completamente las texturas aplicadas al edificio. Es por esto por lo que se observa un color uniforme en todo el edificio en las imágenes del simulador. Por tanto, es necesario encontrar otro software capaz de exportar un paquete de texturas para el modelo o, en su defecto, la creación de estas y su posterior inclusión manual.

5.2.1.2. Enriquecimiento del entorno

Otra vía para aumentar el realismo de las imágenes obtenidas en Gazebo es la inclusión de más variedad de elementos en las distintas estancias de la planta. Algunos de estos elementos pueden ser mochilas, carteles, vegetación, estanterías, libros, cubiertos, tablones de información, etc. Hay que tener en cuenta, que cuantos más modelos se incluyan en el mundo, más recursos requerirá el simulador para poder generarlo. Actualmente ya es un archivo considerablemente pesado de cargar, por lo que debe plantearse la posibilidad de seccionar la planta en varias partes si se desea avanzar por esta vía.

5.2.1.3. Creación de más plantas de la ETSII

Este proyecto se ha centrado en el desarrollo de la planta baja, ya que esta incluye una gran variedad de tipos de estancia. Sin embargo, resulta interesante la creación del resto de plantas, ya que de esta manera puede extenderse las funcionalidades del paquete a la de crear varios mapas por planta. Otra utilidad es la es emplear este modelo 3D de la Escuela para hacer un entorno virtual para conocerla a distancia.

5.2.2. Mejoras en el algoritmo de segmentación

Como se ha visto, el algoritmo desarrollado para segmentar las estancias mediante la detección de puertas ha sido capaz de detectar todas las puertas presentes en el entorno. Sin embargo, la condiciones respecto al ángulo de incidencia del robot al entrar en la puerta son bastante restrictivas para puertas de gran tamaño. Por tanto, sería una buena opción modificar el algoritmo para ser capaz de detectar el patrón de una puerta independientemente del ángulo de incidencia.

Esta solución es relativamente intuitiva de desarrollar puesto que podría buscarse primero el patrón de un lado de la puerta y a continuación comparar que el otro se encuentra en la dirección opuesta. Sin embargo, puede que este modelo introduzca nuevas posibilidades de generar falsos positivos en las detecciones. Ya que podrían detectarse columnas, esquinas u otro tipo de mobiliario con una estructura similar. Por eso, puede ser que fuese necesario desarrollar un método de reconocimiento de puertas mediante visión artificial, para sólo confirmar la existencia de una puerta cuando esta ha sido recientemente capturada por imagen.

5.2.3. Desarrollo de una red neuronal específica

Como se ha podido comprobar, los servicios en la nube de Google suponen una buena solución al problema de la extracción de información semántica. Se puede extraer información suficiente de una imagen para poder clasificar una estancia. Sin embargo, el desarrollo y entrenamiento de una red neuronal específica para este cometido siempre será más eficiente que una de propósito general. Dicho esto, cabe remarcar que no es viable albergar el procesamiento de imágenes en el robot debido a la cantidad de recursos que consume dicha tarea. Por ello, sería también necesario la creación de un host en la nube que aloje la IA y al que poder realizarle las peticiones.

5.2.4. Desarrollo de un planificador de rutas topológicas

Una opción muy interesante de vía de desarrollo es la creación de un planificador de rutas. Al poseer información semántica estructurada por estancias, se le podría mandar al robot una petición de destino textual. Por ejemplo, ir al despacho, o ir al despacho del doctor anónimo. Al ya tener esta información presente, el algoritmo puede ser capaz de generar directamente la ruta de su posición actual a la estancia destino. Esto resulta en una comunicación mucho más rica y fluida entre el usuario y el robot.

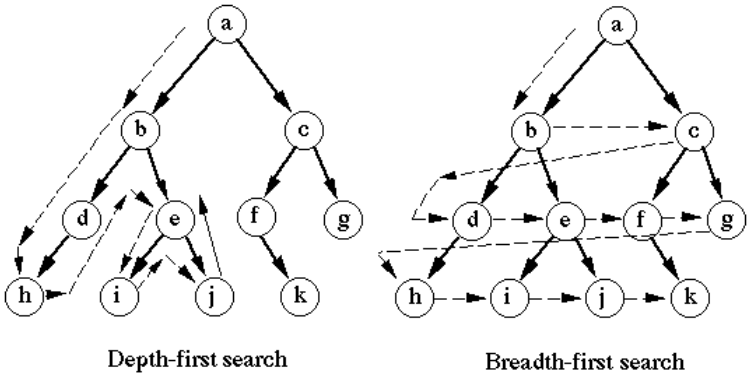

Figura 60. Comparativa entre el método de búsqueda por herencia (Depth-first search) y búsqueda en anchura (Breadth-first search) (Wilson, 2001). En este ejemplo se ve claramente que la forma más eficiente de encontrar la ruta más corta al nodo deseado es mediante la comprobación por niveles (breadth first).

Otra de las grandes ventajas es el hecho de que, al tener toda la planta segmentada en estancias y cada estancia segmentada en waypoints, puede agilizarse aún más el proceso de cálculo de rutas. Gracias a esta estructura, la interpretación de una planta es una red de puntos interconectados entre sí. Por tanto, para encontrar la ruta más corta del punto A al punto B, solo es necesario desarrollar un algoritmo de vaya analizando las conexiones por capas. El teorema que define este método de cálculo se denomina Breath First Search (BFS) o Búsqueda en Anchura en castellano.

El algoritmo de BFS va analizando las conexiones entre puntos o nodos por capas. Es decir, partiendo de un punto A, realiza una primera búsqueda de coincidencia con el punto B en sus nodos vecinos o, lo que es lo mismo, en los nodos con los que guarda una relación de conexión directa. Una vez analizados todos, realiza una segunda búsqueda en los nodos con los que cada uno de estos nodos vecinos tienen conexión. Esta búsqueda anidada por capas de conexión se repite indefinidamente hasta encontrar el nodo destino B. De esta manera, se asegura que la primera vez que se detecte el nodo de destino, la ruta al mismo es la más corta posible. En el contexto del mapeado topológico, estos nodos pueden ser tanto estancias de una planta, como los waypoints de una estancia.

El flujo para generar la ruta podría ser el siguiente. Primero de todo, se emplea el algoritmo BFS para calcular por qué estancias se debe pasar para llegar al destino, al saber por qué estancias cruzar, se sabe a qué puertas se debe llegar. En segundo lugar, cada vez que se entre a una estancia, volver a aplicar este algoritmo para calcular la ruta más corta por la red de waypoints de la estancia para llegar a la puerta por la que se debe abandonar esta. Con esto, se reduce drásticamente la consumición de cómputo, ya que no es necesario calcular la ruta completa en un mapa métrico, donde se debe ser más preciso. Por último, para saber moverse entre waypoints, se pueden convertir estos waypoints en rutas de destino. Así, el paquete navigation sólo tendrá que calcular rutas de alrededor de 1 metro, haciendo incluso posible prescindir del mapa global y trabajar sólo con un mapa de ocupación local.

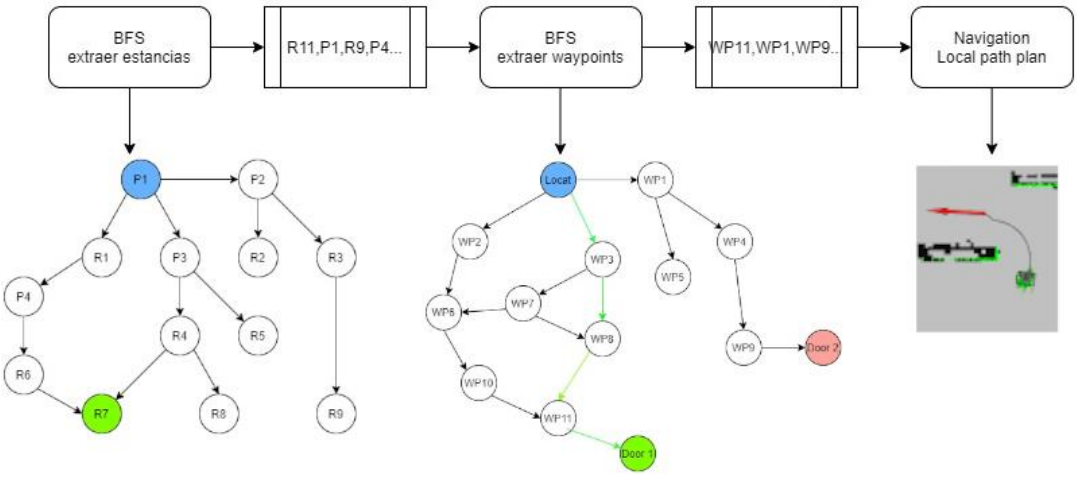

Figura 61. Propuesta de flujo de planificación de rutas basadas en información topológica del entorno. Primero se extrae el array de estancias más corto posible al destino, posteriormente se calculan los waypoints por los que pasar y, finalmente, se pasan los goals al planificador de rutas de Navigation.

Tal y como esta estructurados los distintos elementos semánticos de este proyecto, no sería necesario realizar ninguna adaptación, tan solo habría que crear un nodo específico para este proceso, con las funciones necesarias.

6. Referencias

AENAE. (s.f.). *Foro de Empleo de Cartagena 2017*. Obtenido de https://www.enae.es/resumen_foros_empleo_2017/cartagena/img/distribucion_stands_foro_empleo_cartagena.jpg

Dassault Systems. (s.f.). Obtenido de https://www.solidworks.com/

Dieter Fox, W. B. (s.f.). *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots.*

Foote, T. (s.f.). *geometry2*. Obtenido de geometry2: http://wiki.ros.org/geometry2

FreeCAD. (s.f.). Obtenido de https://www.freecadweb.org/?lang=es_ES

Gazebo Sim. (s.f.). Obtenido de Gazebo: http://gazebosim.org/

Gerkey, B. (s.f.). *Gmapping*. Obtenido de Gmapping: http://wiki.ros.org/slam_gmapping

GitHub . (s.f.). Obtenido de https://github.com/

Google Cloud Services. (s.f.). *Api Vision*. Obtenido de https://cloud.google.com/vision#industry-leading-accuracy-for-image-understanding

Google Earth. (s.f.). Obtenido de Google Earth: https://earth.google.com/web/

Google. (s.f.). *Google Cloud*. Obtenido de Google Cloud: https://cloud.google.com/

Hanson Robotics. (s.f.). *hansonrobotics*. Obtenido de hansonrobotics: https://www.hansonrobotics.com/sophia/

Hendrix, A. (s.f.). *ROS::teleop_twist_keyboard*. Obtenido de ROS::teleop_twist_keyboard: http://wiki.ros.org/teleop_twist_keyboard

Hidalgo, M. (s.f.). *nav_msgs*. Obtenido de nav_msgs: http://wiki.ros.org/nav_msgs

Hokuyo Sensors. (s.f.). Obtenido de https://www.hokuyo-aut.jp/

Jelinek, L. (s.f.). Obtenido de http://wiki.ros.org/p3dx_urdf_model

KELLER, V. (17 de 01 de 2019). *LG*. Obtenido de https://www.lg.com/es/lg-magazine/sabias-que/robots-cloi-ces-2019

Linux ORG. (s.f.). Obtenido de https://www.linux.org/

Meyer, J. (s.f.). *Hector SLAM*. Obtenido de Hector SLAM: http://wiki.ros.org/hector_slam

O'Quin, J. (s.f.). *geographic_info*. Obtenido de geographic_info: http://wiki.ros.org/geographic_info

O'Quin, J. (s.f.). *UUID*. Obtenido de http://wiki.ros.org/unique_identifier

Rabaud, V. (s.f.). *image_pipeline*. Obtenido de http://wiki.ros.org/image_pipeline

ROS ORG. (s.f.). Obtenido de ROS: https://www.ros.org/

ROS ORG. (s.f.). Obtenido de ros melodic: http://wiki.ros.org/melodic

RosWiki. (s.f.). Obtenido de http://wiki.ros.org/rviz

Scott, S. (23 de January de 2019). *Amazon*. Obtenido de https://www.aboutamazon.com/news/transportation/meet-scout

Stefan Kohlbrecher, T. G. (s.f.). *Hector Navigation*. Obtenido de http://wiki.ros.org/hector_navigation

Ubuntu Canonical. (s.f.). Obtenido de https://releases.ubuntu.com/18.04/

Vargas, M. (11 de Mayo de 2019). Obtenido de https://expansion.mx/tecnologia/2019/05/11/el-es-thalon-y-es-el-primer-robot-colombiano

Wilson, B. (Julio de 2001). *UNSW Sydney*. Obtenido de http://www.cse.unsw.edu.au/~billw/Justsearch.html