



**Universidad  
Rey Juan Carlos**

**Grado en Ingeniería en Telemática**

Escuela Técnica Superior de Ingeniería Telecomunicación

Curso académico 2016/2017

**Trabajo Fin de Grado**

Servidor de mapas dinámico para la navegación en entornos  
domésticos

**Autor:** Jonathan Ginés Clavero **Tutor:** Francisco Martín Rico



*A*



# Agradecimientos



# Resumen



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Robótica móvil . . . . .	1
1.2. Mapeado . . . . .	1
1.3. RoboCup@Home . . . . .	1
1.4. Estructura de la memoria . . . . .	1
<b>2. Objetivos y Metodología</b>	<b>2</b>
2.1. Descripción del problema y requisitos . . . . .	2
2.2. Objetivo del proyecto . . . . .	2
2.3. Metodología de desarrollo . . . . .	3
2.4. Plan de trabajo . . . . .	5
<b>3. Entorno y herramientas</b>	<b>6</b>
3.1. Robot Kobuki . . . . .	6
3.2. ROS . . . . .	7
3.3. Software en ROS para el mapeado, localización y navegación . . . . .	10
3.3.1. Costmap_2D . . . . .	10
3.3.2. Map_server . . . . .	11
3.3.3. Map_saver . . . . .	11
3.3.4. AMCL . . . . .	11
3.3.5. Move_base . . . . .	14
3.3.6. gmapping . . . . .	14
<b>4. Montaje del robot</b>	<b>15</b>
4.1. Estructrura . . . . .	15
4.2. Diseño de conectividad . . . . .	15
4.3. Modelo de Gazebo . . . . .	15
<b>5. Mapas dinámicos</b>	<b>16</b>
5.1. Arquitectura del sistema . . . . .	16

5.2.	Servidor de mapas dinámico . . . . .	18
5.2.1.	Mapa estático . . . . .	18
5.2.2.	Mapa de corto plazo . . . . .	18
5.2.3.	Mapa de largo plazo . . . . .	21
5.3.	Construcción del mapa final . . . . .	23
<b>6.</b>	<b>Navegación semántica</b>	<b>28</b>
<b>7.</b>	<b>Experimentación</b>	<b>29</b>
7.1.	Mapeado en entorno doméstico . . . . .	29
7.2.	Navegación con obstáculos dinámicos . . . . .	30
7.3.	Experimentación en la Robocup . . . . .	30
<b>8.</b>	<b>Conclusiones y trabajos futuros</b>	<b>32</b>

# Índice de figuras

2.1. Modelo en espiral . . . . .	4
3.1. Robot kobuki. . . . .	6
3.2. Especificaciones robot kobuki. . . . .	7
3.3. Representación del modelo de comunicación de ROS. . . . .	9
3.4. A la izquierda el frame map y a la derecha los frames de odom y base_footprint . . . . .	10
3.5. Ejemplo visual de un costmap. . . . .	11
3.6. Inicialización del filtro de partículas . . . . .	12
3.7. El robot se va acercando a la posición ideal . . . . .	13
3.8. El robot se encuentra totalmente localizado . . . . .	13
5.1. Encuadre de nuestro sistema dentro del modelo de navegación de ROS	16
5.2. Esquema del sistema . . . . .	17
5.3. Mapa estático . . . . .	18
5.4. Esquematización de una lectura de láser con un objeto. . . . .	20
5.5. Escenario extendido . . . . .	25
5.6. Detalle de la localización . . . . .	25
5.7. Mapas del escenario extendido . . . . .	26
7.1. Visión del simulador, (a) y (b), y mapa a corto plazo (c). . . . .	29
7.2. Añadimos un objeto al escenario . . . . .	30
7.3. Eliminamos un objeto del escenario . . . . .	30
7.4. Añadimos un objeto al mapa de largo plazo . . . . .	31
7.5. Borramos un objeto del mapa de largo plazo . . . . .	31



# Capítulo 1

## Introducción

- 1.1. Robótica móvil**
- 1.2. Mapeado**
- 1.3. RoboCup@Home**
- 1.4. Estructura de la memoria**

# **Capítulo 2**

## **Objetivos y Metodología**

### **2.1. Descripción del problema y requisitos**

La navegación en entornos dinámicos, como puede ser una casa o una institución pública, supone un gran reto que superar ya que las personas o las mascotas pueden acercarse o cruzarse delante del robot o podríamos encontrarnos objetos del mobiliario que han sido movidos de su posición original y se encuentran en nuestro camino. En este escenario el robot no debería nunca ni chocar ni perderse en el entorno y debe llegar al destino impuesto por la mejor ruta disponible.

Por ejemplo, si nuestro robot está yendo desde el salón a la cocina de nuestra casa pero en su camino habitual y más óptimo se encuentra un mueble, el robot debe darse cuenta rápidamente, esquivarlo, proseguir con su camino y además recordarlo para que cuando volvamos al salón de vuelta podamos esquivarlo más fácilmente. Por otro lado en nuestra casa también habrá personas, estas personas se moverán casi constantemente por la estancia por lo que no será del todo correcto tenerlas en cuenta a la hora de planificar nuestra ruta para navegar de un sitio a otro de la casa pero si que será muy importante no chocar con ellas.

Para analizar el entorno del robot usaremos el sensor láser, este sensor destaca por su alta precisión y su corto tiempo de procesado.

### **2.2. Objetivo del proyecto**

Se quiere diseñar un algoritmo genere un mapa en tiempo real, el cual será usado por el nodo de navegación de ROS para navegar por un entorno doméstico, ya sea indicando una posición x,y en el mapa o indicándole una estancia a la que navegar.

Este mapa se construirá a partir de la mezcla de 3 mapas, mapa estático, mapa de largo plazo y mapa de corto plazo.

En primera instancia el algoritmo se validará haciendo uso de un simulador en el que se representa una casa con varios tipos de muebles, ya que resulta más fácil de depurar un algoritmo en un entorno virtual, que en un entorno real. Posteriormente el algoritmo se probará en distintas recreaciones de escenarios reales ,y se harán las modificaciones oportunas para adaptarlo al entorno real, y por ultimo se llevará a la competición.

Para simplificar la resolución del problema se ha dividido el proyecto en varios subobjetivos:

1. Se usarán las herramientas por defecto que nos ofrece ROS para creare el mapa de corto plazo en referencia a las mediciones tomadas por el laser. En un primer paso solo añadiremos los diferentes objetos que percibamos.
2. Se ampliará el algoritmo anterior para poder añadir y eliminar objetos que aparezcan o desaparezcan del entorno.
3. Se desarrollará un algoritmo para mezclar los mapas entre sí y así generar tanto el mapa de largo plazo como el mapa que usaremos para la navegación.
4. Se creará un servidor de mapas dinámicos que se inicializará con los mapas estático y de largo plazo y que generará el mapa final mezclando de loa mapas de largo plazo y de corto plazo.
5. Se usará el mapa final como parámetro del paquete de navegación de ROS, *move base*, y del paquete de localización de ROS *amcl*.
6. Se generará y usará un mapa semántico, en el que cada nivel de gris se asocie con una etiqueta para después ordenar al robot que navegue a dichas etiquetas.

## 2.3. Metodología de desarrollo

En el desarrollo del sistema descrito, el modelo de ciclo de vida utilizado ha sido el modelo en espiral basado en prototipos. Este modelo permite desarrollar el proyecto de forma incremental, aumentando la complejidad progresivamente y haciendo posible la generación de prototipos funcionales. Este planteamiento permite obtener productos parciales al final de cada ciclo que pueden ser evaluados, ya sea total o parcialmente. Esto facilita la adaptación a los cambios en los requisitos, circunstancia muy común en

los proyectos de investigación.

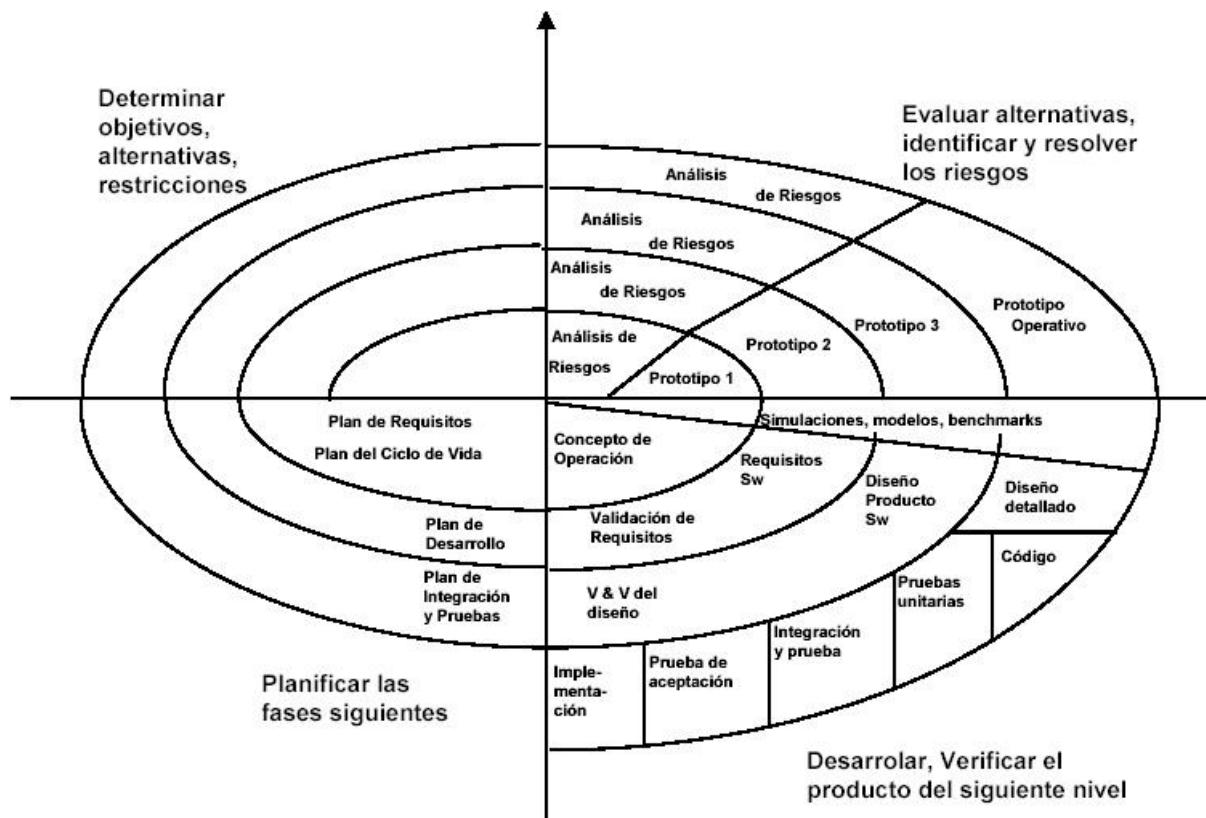


Figura 2.1: Modelo en espiral.

El ciclo de vida de este modelo está dividido en ciclos. Cada ciclo representa una fase del proyecto y está dividido, a su vez, en 4 partes. Cada una de las partes tiene un objetivo distinto:

- **Determinar objetivos.** Se establecen las necesidades que debe cumplir el sistema en cada iteración teniendo en cuenta los objetivos finales. Según avanzan las iteraciones aumenta el coste del ciclo y su complejidad.
- **Evaluar alternativas.** Se determinan diferentes formas de alcanzar los objetivos que se han establecido en la fase anterior. Se aborda el problema desde distintos puntos de vista, como, por ejemplo, el rendimiento del algoritmo en tiempo y espacio. Además, se consideran explícitamente los riesgos, intentando mitigarlos al máximo.
- **Desarrollar y verificar.** Se desarrolla el producto siguiendo la mejor alternativa para poder alcanzar los objetivos del ciclo. Una vez diseñado e implementado el producto, se realizan las pruebas necesarias para comprobar su funcionamiento.

- **Planificar.** Teniendo en cuenta los resultados de las pruebas realizadas, se planifica la siguiente iteración, se revisan los errores cometidos y se comienza un nuevo ciclo de la espiral.

## 2.4. Plan de trabajo

Para poder abordar el problema se han marcado una serie de subobjetivos ha completar. Dichos hitos son los siguientes:

1. Estudio y comprensión de la composición de un mapa y como construirlo. Nos apoyaremos en las herramientas ofrecidas por ROS y que resultaran básicas para este fin, dicha herramientas son *TF*<sup>1</sup> y *Costmap*<sup>2</sup>.
2. Primer subobjetivo. Una vez conocido como funcionan los *costmap* procederemos a crear un pequeño nodo en el que se cree un mapa con las observaciones instantáneas que percibimos con el láser.
3. Segundo subobjetivo. Extender el algoritmo anterior para poder añadir y eliminar objetos según entren o salgan de la escena.
4. Tercer subobjetivo. Modificar el paquete *map\_server* para que acepte varios mapas como entrada y estudiar la manera de mezclar los mapas entre sí.
5. Fase de pruebas. Se le pasará al paquete de navegación de ROS, *move\_base*, el mapa resultante y se harán pruebas de navegación en el simulador y en el robot real.
6. Cuarto subobjetivo. Crear el mapa semántico, especificando las etiquetas naturales que tendrá una casa, salón, cocina, habitación... y usarlo para poder ir a la estancia que le indiquemos.

---

<sup>1</sup><http://wiki.ros.org/tf>

<sup>2</sup>[http://wiki.ros.org/costmap\\_2D](http://wiki.ros.org/costmap_2D)

# Capítulo 3

## Entorno y herramientas

En este capítulo se presentaran y describirán las herramientas hardware y software que se han usado para el desarrollo del proyecto. En primer lugar se presentará el robot sobre el que hemos trabajado, el robot kobuki. En segundo lugar se presentará el *framework* sobre el que hemos construido el algoritmo, ROS , y por ultimo se describirán las herramientas de dicho framework que han resultado esenciales para la realización del proyecto.

### 3.1. Robot Kobuki

El robot kobuki o Turtlebot es un robot *lowcost* que cuenta con un software de código abierto. Su estructura principal es una base móvil de forma circular y una serie de baldas y barras metálicas para adaptar su configuración física a nuestras necesidades. Cuenta con una integración total con ROS, el framework que se usará para el proyecto, y además es el robot usado por la mayoría de alumnos de la universidad para cursar la asignatura de robótica.

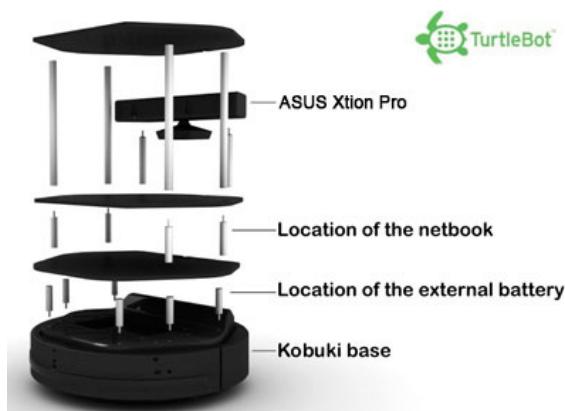


Figura 3.1: Robot kobuki.

Cuenta con una autonomía de unas 5 horas y es capaz de cargar 5 kg de peso. Además cuenta con multitud de puertos que nos facilitarán la alimentación de los nuevos sensores que queramos incluirle.

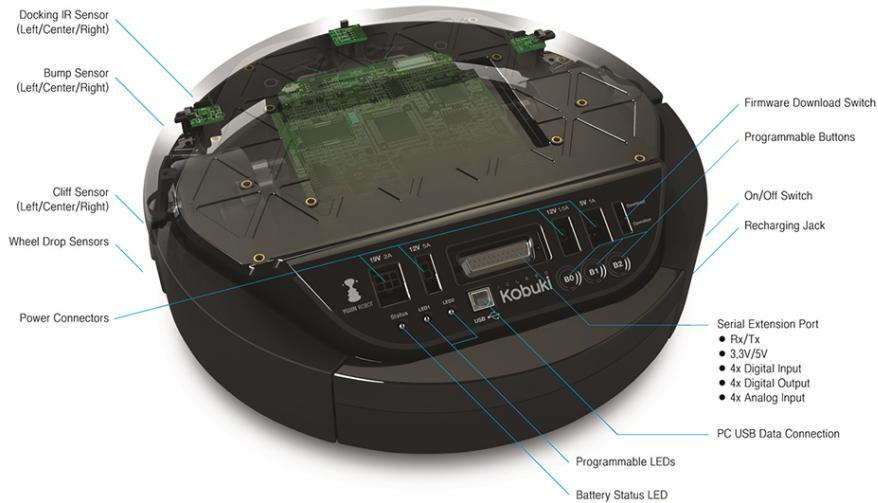


Figura 3.2: Especificaciones robot kobuki.

## 3.2. ROS

ROS, cuyas siglas significan *Robot Operating System*, es un framework muy flexible para la creación de software para robots. Cuenta con una amplia colección de herramientas y librerías que nos simplifican la generación de comportamientos, tanto simples como complejos, para una gran variedad de plataformas. Este framework se ha convertido en el estándar en el ámbito de la programación de software para robots por su gran versatilidad y su gran robustez y por proporcionarnos abstracción del hardware, control sobre los dispositivos de bajo nivel, paso de mensajes entre procesos y mantenimiento de paquetes gracias a sus repositorios.

La idea fundamental de ROS es la utilización de *topics* como modelo de comunicación entre procesos, que en ROS se les llama nodos. En estos *topics* los nodos publican los mensajes o la información que desean comunicar y otros nodos estarán suscritos a esos *topics* para leer dicha información. Dichos nodos están totalmente perfectamente distribuidos, lo que permite el procesamiento distribuido en múltiples núcleos, multiprocesamiento, GPUs y clústeres. Otro modelo de comunicación usado en ROS, aunque no es el principal, es el del uso de *actionlib*<sup>1</sup>. Esta herramienta puede ser descrita como la petición de la realización de una acción a un nodo y la espera de la finalización de esta.

ROS incluye paquetes que pueden cubrir diferentes áreas como:

- Percepción.
- Identificación de Objetos.
- Segmentación y reconocimiento.
- Reconocimiento facial.
- Reconocimiento de gestos.
- Seguimiento de objetos.
- Comprensión de movimiento.
- Estructura de movimientos (SFM).
- Visión estéreo: percepción de profundidad mediante el uso de dos cámaras.
- Visión 3D mediante el uso de cámaras RGBD.
- Movimientos.
- Robots móviles.
- Control.
- Planificación.
- Agarre de objetos.

ROS también cuenta con una integración perfecta con visualizadores de información, como puede ser *Rviz*, o con el simulador *Gazebo*. Esto resulta especialmente útil en un entorno de desarrollo robótico, ya que tanto los robots como los sensores que utilizamos habitualmente tienen un coste alto y cuanto más los cuidemos y más desarrollemos sobre un simulador más alargaremos su vida útil. Esto no significa que tengamos que desarrollar el 100 % de nuestros experimentos sobre el simulador, ya que todo algoritmo robótico llevado al robot real difiere bastante del algoritmo desarrollado para el simulador. En un entorno real nos encontraremos con muchos inconvenientes y problemas que el simulador nos da resueltos y tendremos que preparar nuestro algoritmo para que sea robusto frente a estos problemas.

---

<sup>1</sup><http://wiki.ros.org/actionlib>

En la figura 3.3 vemos un esquema de una aplicación y su modelo de comunicación. La aplicación en concreto es *move\_base*, de la que hablaremos un poco más adelante y que está representada con el cuadro negro central. Dentro de este cuadro tenemos los distintos nodos que componen la aplicación. En las flechas que entran a la aplicación tenemos los *topics* a los que está suscrito la aplicación y en las flechas de salida se representan los *topics* en los que la aplicación publica. En estas flechas también está representado el tipo de mensaje que se publica en estos *topics*. Estos mensajes son estándar de ROS, están creados y se pueden componer fácilmente en nuestros nodos, por lo que resulta la mejor manera de comunicar nuestros nodos. También podemos crear nuestros mensajes propios, pero esto rompe un poco el estándar.

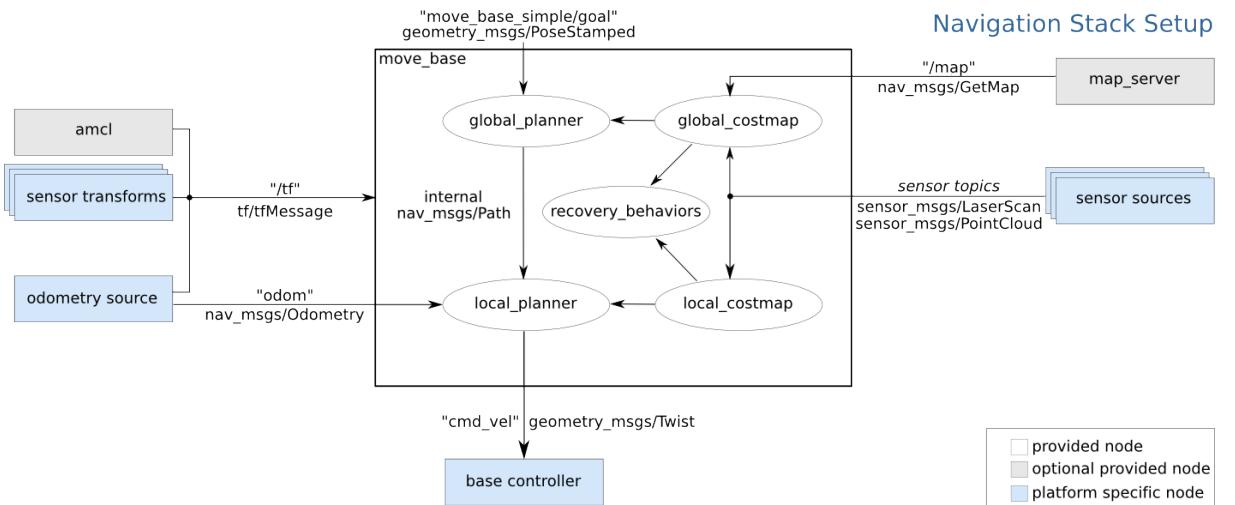


Figura 3.3: Representación del modelo de comunicación de ROS.

Ahora hablaremos un poco más en profundidad de la herramienta que está representada por el topic a la izquierda de la aplicación, *tf*

### **tf**

Cualquier robot está compuesto por multitud de piezas móviles, como puede ser la propia base del robot o la pinza de un brazo robótico. Cada una de estas piezas se pueden representar con un *frame*. Ademas existen también otros *frames* que pueden interesarnos, como puede ser el *frame* de world o el *frame* de map.

Usamos las *tf* para poder representar información relativa a uno de estos frames. Esto puede sernos de utilidad, por ejemplo, si queremos conocer la posición de un objeto que hemos cogido con nuestra pinza respecto a la base de nuestro robot, o cual es la posición relativa de un objeto que estamos percibiendo con el láser respecto a nosotros o respecto al mapa.

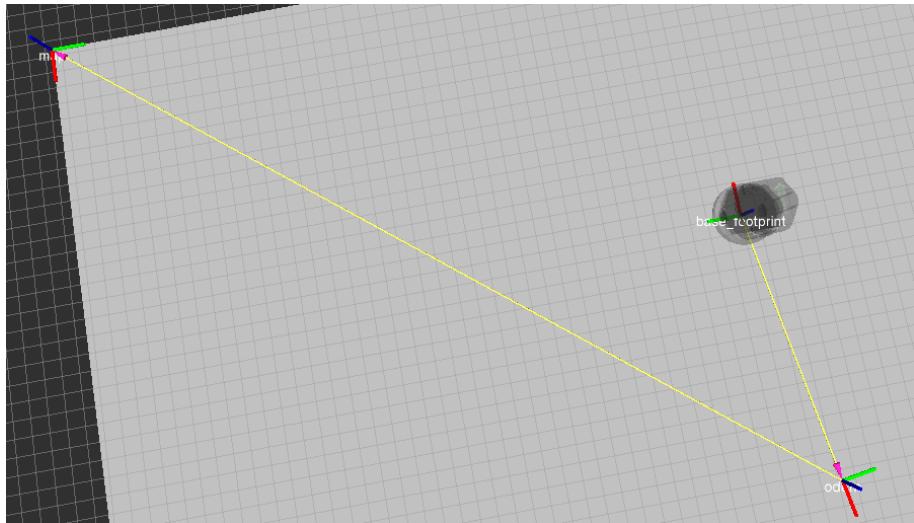


Figura 3.4: A la izquierda el frame *map* y a la derecha los frames de *odom* y *base\_footprint*

Cuando trabajamos con mapas es importante que todo lo que se representa en él sea respecto al frame *map*. De este modo nuestro mapa puede ser usado por otros nodos, como el nodo de navegación, o por otro robot situado en el escenario representado en el mapa.

### 3.3. Software en ROS para el mapeado, localización y navegación

En esta sección se describirá el funcionamiento de los diferentes paquetes que ROS nos proporciona y que resultan esenciales para la creación de mapas, la localización y la navegación por distintos entornos.

#### 3.3.1. Costmap\_2D

Un *costmap* es una estructura de datos ofrecida por ROS y compuesta por un grid de ocupación y los metadatos de este grid. Cada celda del grid toma valores entre 0 y 255, donde 0 corresponde a una celda vacía, los valores entre 1 y 254 representan la probabilidad de que una celda está ocupada y el valor 255 se reserva para el total desconocimiento sobre el estado de una celda. Cada valor se asocia con un nivel de gris, como se puede ver en la imagen.

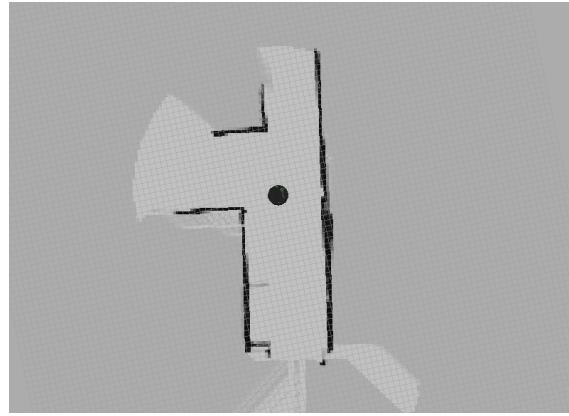


Figura 3.5: Ejemplo visual de un *costmap*.

Para poder representar la ocupación de un objeto en un *costmap* es necesario hacer uso de las transformadas entre frames que nos ofrece ROS.

### 3.3.2. Map\_server

El nodo *map\_server* es un nodo muy simple y muy útil. Se encarga de cargar un mapa en formato *.pgm* de un fichero y publicarlo en un topic para que nodos como *amcl* o *move\_base* se alimenten de él.

### 3.3.3. Map\_saver

El nodo *map\_saver* se encarga de guardar en un fichero *.pgm* el mapa que se está publicando en un topic.

### 3.3.4. AMCL

*AMCL*<sup>2</sup> es un paquete de localización que implementa un algoritmo de Monte Carlo, el cual usa un filtro de partículas para localizar al robot sobre un mapa que previamente le proporcionamos.

#### Filtro de partículas

El algoritmo del filtro de partículas se divide en 4 etapas: Inicialización, actualización, estimación y predicción.

1. Inicialización: En la etapa de inicialización se “lanzan” una serie de partículas cercanas a la posición inicial del robot. Estas partículas ademas de una posición

---

<sup>2</sup><http://wiki.ros.org/amcl>

en el espacio también tendrán una dirección. Podemos ver un ejemplo gráfico en la imagen 3.6. Vemos como se han generado muchas partículas alrededor del robot y que cada una tiene una dirección más o menos acertada con la dirección del robot.

2. Actualización: En este punto del algoritmo se compara la percepción del láser del robot en el punto en el que se encuentra con la percepción que tendría si tuviera la posición y la dirección de cada una de las partículas que generamos. Cuanto más acertada sea la suposición anterior, más valor se le da a esa partícula. Así nos encontraremos que las partículas que están más cercanas a la posición del robot cobran más valor y las que están más lejos y con una dirección totalmente errónea tienen menos valor.
3. Estimación: En esta fase nos quedamos con las partículas que más valor tenían para volver a lanzarlas en la siguiente fase del algoritmo.
4. Predicción: En esta última fase lanzamos las partículas de nuevo con el valor que tenían y su posición, añadiéndole un pequeño ruido. En este punto del algoritmo también se corrige la posición del robot a la posición de la partícula con más valor. Una vez completado el algoritmo se vuelve a la fase de Actualización y se repite hasta que el robot esté perfectamente localizado en el mapa.

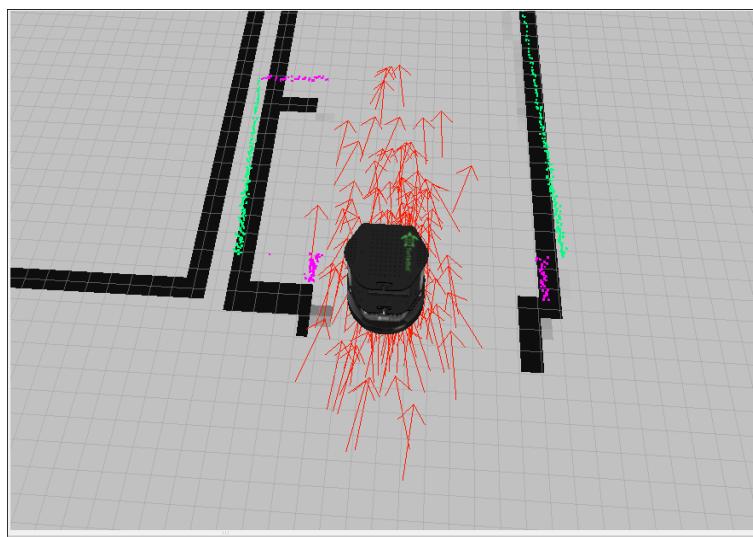


Figura 3.6: Inicialización del filtro de partículas

Observamos como la linea color de la parte derecha de la imagen 3.7, correspondiente a las muestras tomadas con el láser, está más cerca de la linea del mapa que en la imagen 3.6 que se aprecia que no está alineada. Esto es fruto de la corrección que se va

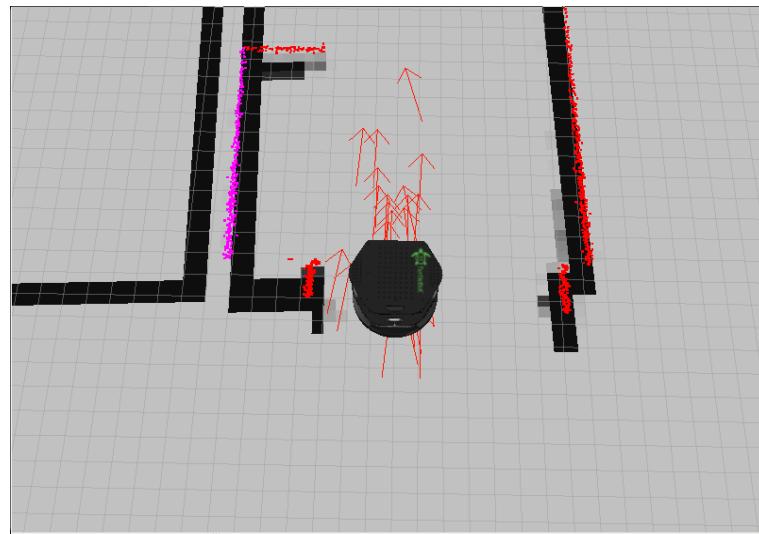


Figura 3.7: El robot se va acercando a la posición ideal

haciendo de la posición. También observamos que hay menos partículas, esto es fruto de la convergencia hacia la posición correcta.

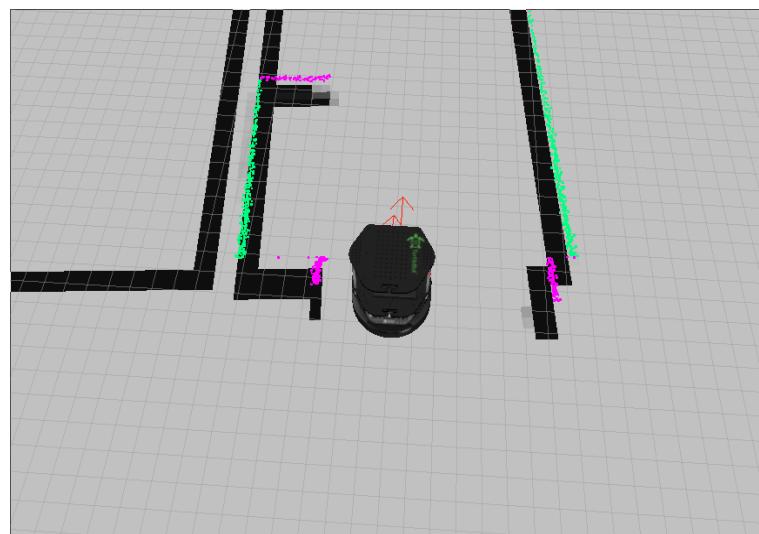


Figura 3.8: El robot se encuentra totalmente localizado

En la imagen 3.8 vemos como el número de partículas se ha reducido mucho, ya que se ha llegado a casi una estimación de la posición del robot muy cerca de la posición real. Vemos también que las líneas de color correspondientes a las muestras del láser están alineadas con el mapa.

(()) Habrá que ponerlo en otro sitio

Para el uso del paquete *amcl* en nuestro algoritmo fue necesaria la realización de una pequeña modificación. Esta modificación se refiere a que el paquete por defecto solo usa un mapa y lo obtiene al principio de la ejecución del algoritmo. Si le llegaba un

nuevo mapa reiniciaba por completo el algoritmo. Esto nos generaba un problema, ya que en el algoritmo propuesto se publica un mapa por cada iteración y el paquete por defecto se reiniciaba constantemente. El efecto que producía es que el robot siempre se encontraba en la posición inicial y aunque lo moviéramos siempre ocupaba la misma posición en el mapa. En nuestro *amcl* modificado se usa el mapa que se obtiene en cada iteración y sobre él se calcula la posición del robot, sin reiniciar en ningún momento el algoritmo.

(())()

### 3.3.5. Move\_base

El paquete principal para la navegación es el paquete *move\_base*<sup>3</sup>. Este paquete es el encargado de, proporcionándole un punto de meta y un mapa sobre el que navegar, calcular el plan necesario para llegar hasta dicho punto y ejecutar dicho plan para que el robot alcance su destino. *Move\_base* cuenta nodos que calculan y ejecutan la ruta, *global planner* y *local planner*. El *global planner* se encarga de planificar la ruta teniendo en cuenta el mapa que obtiene del servidor de mapas. Este nodo se asegurará de que la ruta no atraviesa ningún objeto y que la ruta calculada es la mejor para llegar al destino. Además calculará una nueva ruta si el robot se parara a causa de un objeto o si en medio del camino se da cuenta que la ruta no puede llevarse a cabo.

El *local planner* se encarga de ejecutar el plan calculado por el *global planner*, así como de esquivar objetos que no están en el mapa y de navegar de una forma segura, sin chocar con las paredes o los muebles del escenario.

### 3.3.6. gmapping

Este paquete nos proporciona un algoritmo de SLAM, Simultaneous Localization and Mapping, que nos construye un mapa del escenario en el que se encuentre el robot sin pasarle ningún mapa previo del lugar y además se localiza en ese mapa a la vez que lo construye. Este paquete hace uso del sensor láser para realizar la construcción del mapa.

---

<sup>3</sup>[http://wiki.ros.org/move\\_base?distro=indigo](http://wiki.ros.org/move_base?distro=indigo)

# **Capítulo 4**

## **Montaje del robot**

- 4.1. Estructrura**
- 4.2. Diseño de conectividad**
- 4.3. Modelo de Gazebo**

# Capítulo 5

## Mapas dinámicos

En este capítulo se expondrá el sistema creado para la composición de un mapa dinámico que pueda ser usado por paquetes de ROS como *move\_base* o *amcl* y que nos sirva para navegar por el ámbito doméstico de una forma más eficaz y segura de lo que lo haríamos usando un mapa estático o un sistema de *SLAM*.

### 5.1. Arquitectura del sistema

El sistema propuesto pretende sustituir al nodo *map\_server*, ya que este nodo solo publica un mapa estático y esto presenta una serie de problemas que veremos más adelante, por tanto nos centramos en esa parte del modelo de navegación de ROS.

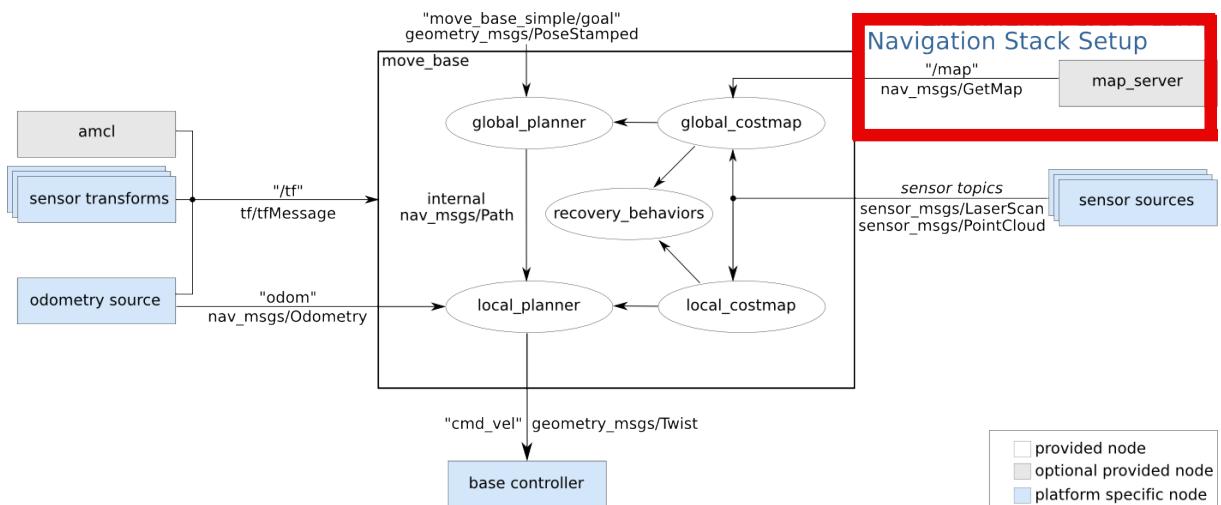


Figura 5.1: Encuadre de nuestro sistema dentro del modelo de navegación de ROS

El sistema de mapeado propuesto consta de 3 nodos principales:

1. *Map\_server*: Este nodo se encarga de cargar desde fichero el mapa estático y el mapa de largo plazo y activa un servicio para que cualquier otro nodo pueda pedir estos mapas.

2. Obs\_detector: La función principal de este nodo es la de detectar objetos, leyendo la información proporcionada por el láser, y componer con dicha información el mapa de corto plazo. Para ello solicita los Metadatos de los mapas cargados al nodo Map\_server, creando así un mapa de las mismas medidas y misma resolución que los mapas que maneja el map\_server. Este mapa es publicado para que sea utilizado por el nodo Map\_controller o sea visualizado por la herramienta Rviz.
3. Map\_controller: Nodo que se encarga de la composición del mapa final y de actualizar el mapa de largo plazo con la información obtenida del mapa de corto plazo. Estos mapas son también publicados en distintos topics para que sean utilizados por los nodos encargados de la navegación.

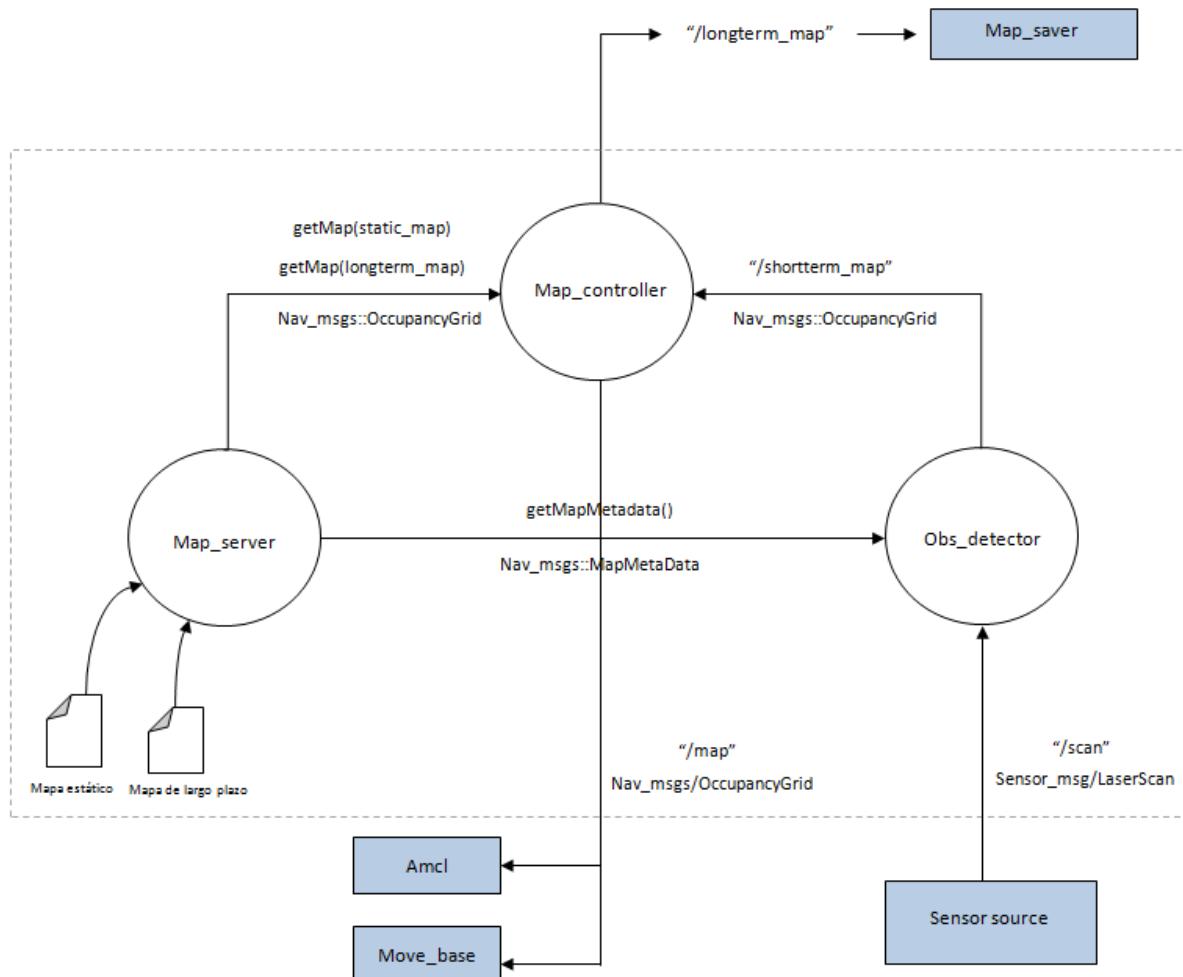


Figura 5.2: Esquema del sistema

## 5.2. Servidor de mapas dinámico

En esta sección se describirá el proceso de construcción de los distintos mapas que el servidor de mapas dinámico ofrece, mapa estático, mapa de largo plazo y mapa de corto plazo, así como el proceso de combinación entre ellos para dar lugar al mapa final que será usado por componentes de ROS vistos anteriormente.

### 5.2.1. Mapa estático

El mapa estático se caracteriza por incluir las partes inmutables del escenario, como son las paredes o las puertas. La mejor manera de construirlo es medir todo el escenario y crear el mapa usando una herramienta de diseño gráfico. En este caso se ha usado *Gimp*. Este mapa nos servirá como base para crear el mapa de largo plazo.

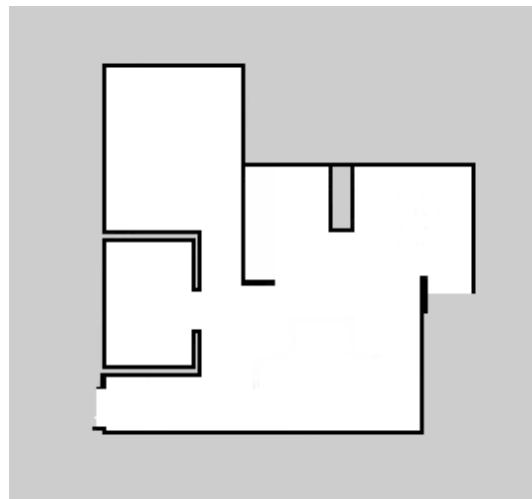


Figura 5.3: Mapa estático

### 5.2.2. Mapa de corto plazo

El mapa de corto plazo se caracteriza por ser un mapa en el que se representa los objetos que el robot va percibiendo. Este mapa se inicializa con el valor 255, lo que indica una incertidumbre total. En el instante en el que el algoritmo de construcción del mapa comienza a iterar comenzarán a corregirse estos valores iniciales, asignando el valor 0 a las celdas que corresponden con zonas libres e incrementando desde 0 hasta 254 el valor de las celdas que se perciben como ocupadas.

---

Código 5.1: Inicialización del cost\_map correspondiente al mapa de corto plazo

```
metadata = getMetadata();
tf::TransformListener tf(ros::Duration(10));
```

---

```

cells_size_x = metadata.width;
cells_size_y = metadata.height;
resolution = metadata.resolution;
origin_x = metadata.origin.position.x;
origin_y = metadata.origin.position.y;
default_value = 255;
scan_ready = false;
pos_ready = false;
cost_map.resizeMap(cells_size_x,cells_size_y, resolution, origin_x,
    origin_y);
cost_map.setDefaultValue(default_value);
cost_map.resetMap(0,0,cost_map.getSizeInCellsX(),
    cost_map.getSizeInCellsY());

```

---

El algoritmo propuesto destaca por la capacidad de, no solo añadir objetos al mapa, si no ademas eliminarlos si los objetos desaparecen del lugar que ocupaban. Esto con las herramientas de ROS no puede conseguirse, ya que si solo utilizamos el nodo map\_server para publicar nuestro mapa, tanto el algoritmo de navegación como el de localización cogerán ese mapa y calcularán sobre el nuestra localización y las rutas al destino, pero ese mapa no tendrá en cuenta cambios en el entorno como puede ser una persona paseando por la casa o un mueble cambiado de sitio, es un mapa estático, por lo que puede resultar totalmente erróneo. Por otro lado si usamos la herramienta gmapping para crear nuestro mapa y localizarnos simultáneamente en él, tenemos el mismo problema. En un entorno dinámico con personas moviéndose u objetos cambiando de sitio, se generará un mapa con muchísimo ruido en las zonas libres y con objetos que se mantienen en su lugar cuando ya no están, por lo que la localización empezará a fallar, ya que las muestras del láser no corresponderán con el mapa.

La solución a este problema se resuelve con la creación de un algoritmo que pueda borrar elementos del mapa. Para ello se compara cada muestra de datos con el mapa que estamos generando y si en dicha muestra existen celdas libres que en el mapa están ocupadas se decrementa el valor de dicha celda en el mapa. La figura 5.4 representa un modelo de una muestra del láser en el que podemos observar lo descrito anteriormente, celdas que marcaremos como ocupadas y celdas que marcaremos como libres. También observamos el límite del láser, que se sitúa en 2.5 metros, a las celdas de mas allá del límite no se les modificará su valor. La cuantía del decremento se puede modelar, consiguiendo así que el robot olvide más lentamente o más rápidamente los objetos que desaparecen del escenario.

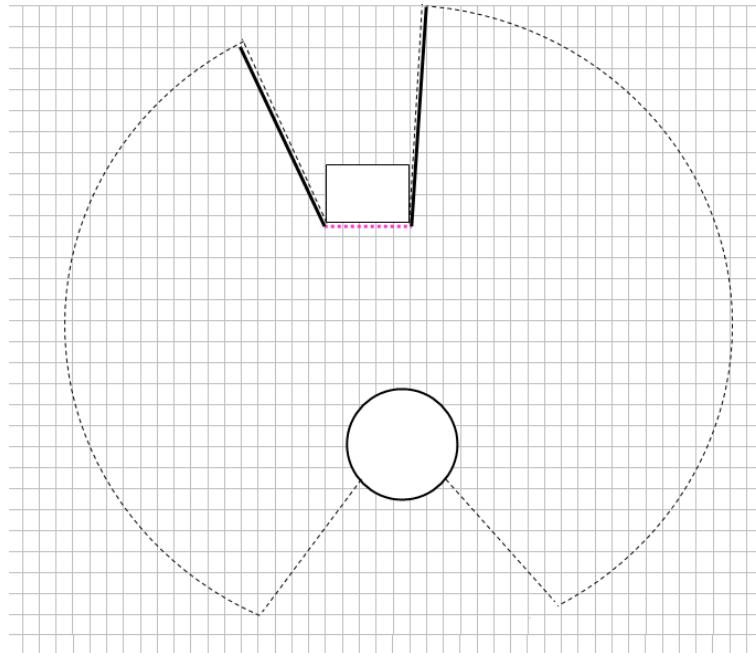


Figura 5.4: Esquematización de una lectura de láser con un objeto.

Código 5.2: Función que actualiza el mapa en cada iteración

---

```
void
ObstacleDetector::updateCostmap(){
    if(scan_ready && pos_ready){
        incrementCostProcedure();
        decrementCostProcedure();
    }
    pVectorList_.clear();
    pointList_.clear();
}
```

---

El proceso de añadir o eliminar un objeto puede ser más o menos rápido dependiendo de los valores de las variables que modelan estas operaciones. Para el caso de nuestro algoritmo contamos un fichero con extensión yaml en el que modificamos los valores de los parámetros.

Código 5.3: Fichero de configuración obstacle\_detector.yaml

---

```
cost_inc: 4
cost_dec: 1
min_lenght: 0.23
max_lenght: 2.5
```

---

Los valores asociados a cost\_inc y cost\_dec representan el incremento o decremento del valor de una celda por cada iteración del algoritmo. Los otros valores configurables

representan la longitud máxima y mínima que se tiene en cuenta para cada medida que el láser nos proporciona.

Al final de cada iteración del algoritmo el mapa de corto plazo es publicado en un topic para que pueda ser usado por el resto de nodos que conforman el servidor de mapas dinámico. Para llevar a cabo esta operación usamos una estructura de datos proporcionada por ROS, *Costmap2DPublisher*<sup>1</sup>. Este publicador publica nuestro mapa en el topic */shortterm\_map*.

Código 5.4: Step del nodo obstacle\_detector

---

```

void
ObstacleDetector::step(){
    updateCostmap();
    cost_map_publisher_.publishCostmap();
}

int
main(int argc, char** argv)
{
    ros::init(argc, argv, "obstacle_detector"); //Inicializa el nodo
    ObstacleDetector obs;
    ros::Rate loop_rate(5);

    while (ros::ok()){
        obs.step();
        ros::spinOnce();
        loop_rate.sleep();
    }
    return 0;
}

```

---

Existe un pequeño cambio en los valores del mapa que se publica. El tipo de mensaje que se usa para publicar un costmap es *nav\_msgs::OccupancyGrid*<sup>2</sup>. Este tipo de mensaje contiene, ademas de los metadatos asociados al mapa, un array de datos que representa al mapa. La principal diferencia con un costmap es que los valores en un OccupancyGrid van de 0 a 100 y -1 para las celdas desconocidas ,y no de 0 a 255 como en un costmap.

### 5.2.3. Mapa de largo plazo

El mapa de largo plazo se inicializa con los valores del mapa estático y se caracteriza por incluir los objetos que tienen un valor muy alto en el mapa de corto plazo, por tanto tenemos un mapa con objetos que han perdurado en el mapa a corto plazo durante

---

<sup>1</sup>[http://docs.ros.org/indigo/api/costmap\\_2d/html/classcostmap\\_2d\\_1\\_1Costmap2DPublisher.html](http://docs.ros.org/indigo/api/costmap_2d/html/classcostmap_2d_1_1Costmap2DPublisher.html)

<sup>2</sup>[http://docs.ros.org/indigo/api/nav\\_msgs/html/msg/OccupancyGrid.html](http://docs.ros.org/indigo/api/nav_msgs/html/msg/OccupancyGrid.html)

un largo periodo de tiempo y que podemos considerar que nos van a influir a la hora de planificar nuestra ruta por el escenario. Este mapa también es dinámico, por lo que también elimina los objetos del mapa si estos desaparecen o cambian su posición.

Código 5.5: Procedimiento para añadir un objeto al mapa de largo plazo

---

```
void
MapController::updateLongTermMap(nav_msgs::OccupancyGrid s_map){
    for(int i = 0;i<s_map.data.size();i++){
        if(s_map.data[i] > 95){
            longTerm_map.data[i] = s_map.data[i];
            .
            .
            .
        }
    }
}
```

---

En el código mostrado en 5.5 muestra el proceso de adición de un objeto al mapa de largo plazo. Para ello se recorren los valores del mapa de corto plazo y si alguno tiene un valor mayor a 95, valor que consideramos lo suficientemente alto como para que sea muy fiable que esa celda está ocupada, se incluye con ese valor al mapa.

Código 5.6: Procedimiento para eliminar un objeto al mapa de largo plazo

---

```
void
MapController::updateLongTermMap(nav_msgs::OccupancyGrid s_map){
    for(int i = 0;i<s_map.data.size();i++){
        .
        .
        .
    }else if(s_map.data[i] < 5 && s_map.data[i] >= 0 &&
             longTerm_map.data[i] >= longterm_cost_dec && static_map.data[i] <
             95){
        longTerm_map.data[i] = longTerm_map.data[i] - longterm_cost_dec;
    }
}
```

---

En el código mostrado en 5.6 se compara el mapa de largo plazo con el mapa de corto plazo. Si una celda en el mapa de largo plazo tiene un valor que indica que está ocupada y en el mapa de corto plazo esa misma celda tiene un valor que indica que está libre, siempre y cuando esa celda no pertenezca a una celda de una pared, se decrementa su valor en el mapa de largo plazo. La cuantía de este decremento también puede modelarse, longterm\_cost\_dec, regulando así la memoria que tenemos de los objetos que hemos añadido a este mapa.

Por último publicamos el mapa en el topic `/longTerm_map`.

Código 5.7: Publicación del mapa de largo plazo

---

```
longTermMap_pub = nh_.advertise<nav_msgs::OccupancyGrid>("/longTerm_map",
    5);

void
MapController::publishAll(){
    .
    .
    .
    longTerm_map.header.stamp = ros::Time::now();
    longTermMap_pub.publish(longTerm_map);
}
```

---

Adicionalmente este mapa se guarda cada cierto tiempo en un fichero, usando el nodo por defecto de ROS para este fin, `map_saver`<sup>3</sup>. Así por ejemplo podemos tener en cuenta una mesa dentro del salón de una casa para una futura ejecución del algoritmo. Esto nos permitiría planificar una ruta mejor para la navegación, ya que podríamos esquivar esta mesa con facilidad y llegar a nuestro destino lo más rápido posible. Si no tuviéramos esta mesa en cuenta el algoritmo podría planificar una ruta a través de las celdas ocupadas por la mesa. Seguramente el robot no llegaría a chocar, ya que el planificador de rutas usa un pequeño mapa local para evitar estos problemas, pero es seguro que tardaría más en alcanzar su destino ya que al encontrarse frente a la mesa el robot se pararía y tendría que recalcular una nueva ruta.

## 5.3. Construcción del mapa final

El mapa final será la composición de los mapas de largo plazo, que ya incluye el mapa estático, y de corto plazo. Este mapa será usado por el nodo de la navegación y por el nodo de la localización para navegar y localizar al robot en el escenario. La composición del mapa final o mapa efectivo será el resultado de la operación de máximo entre los mapas de corto y de largo plazo, como vemos en el código del apartado 5.8. De esta manera incluiremos en el mapa final toda la información relacionada con los objetos que llevan un tiempo en la escena y también incluiremos, aunque con un menor valor, personas o cosas que acaban de entrar en las inmediaciones del robot.

---

<sup>3</sup>[http://wiki.ros.org/action/fullsearch/map\\_server#map\\_saver](http://wiki.ros.org/action/fullsearch/map_server#map_saver)

Código 5.8: Composición del mapa final

---

```
void
MapController::buildEffectiveMap(nav_msgs::OccupancyGrid
    s_map,nav_msgs::OccupancyGrid l_map){
    for(int i = 0;i<s_map.data.size();i++){
        effective_map.data[i] = std::max(s_map.data[i],l_map.data[i]);
    }
}
```

---

Tras la creación del mapa este es publicado en el topic `/map`.

Código 5.9: Publicación del mapa final

---

```
effectiveMap_pub = nh_.advertise<nav_msgs::OccupancyGrid>("/map", 5);

void
MapController::publishAll(){
    effective_map.header.stamp = ros::Time::now();
    effectiveMap_pub.publish(effective_map);
    .
    .
    .
}
```

---

## Mapeado de zonas desconocidas

La creación de un mapa de corto plazo a partir de las muestras del láser y la posterior incorporación del mapa de largo plazo para generar un mapa conjunto cuenta con una gran versatilidad y eficacia, tanto que es posible mapear nuevas zonas o estancias del escenario. Generar un mapa así nos permite también localizarnos en este e incluso navegar por él.

No contamos con un sistema de SLAM, ya que partimos de unos mapas, pero poder incluir en un mapa objetos o nuevas estancias nos permite mantener nuestro robot doméstico por mucho más tiempo navegando por nuestra casa, lo que lo hace mucho más robusto y versátil. Esta ventaja podría, por ejemplo, aplicarse a un robot sirviente en una universidad. Al principio, como prueba de la tecnología, puede que nos interese darle una pequeña zona mapeada por la que moverse, pero si un día queremos que su rango de efectividad se amplíe solo tendremos que moverlo por la zona deseada y se añadirá al mapa perfectamente.

# Capítulo 6

## Navegación semántica

# Capítulo 7

## Experimentación

### 7.1. Mapeado en entorno doméstico

pruebas unitarias. CAMBIAR LAS IMAGENES DE DIRECTORIO

La figura 7.1 fue captada al iniciar el algoritmo. Observamos que la mayor parte del mapa de corto plazo se encuentra en una posición de desconocimiento y que se han ido incluyendo en este las zonas libres, las paredes y la estantería. Los puntos morados y verdes corresponden a la representación de las muestras tomadas por el láser.

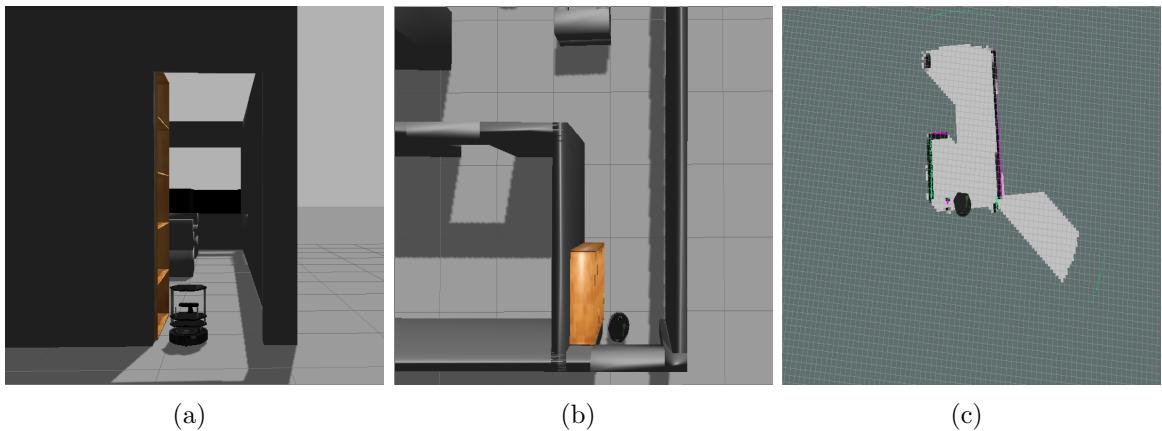


Figura 7.1: Visión del simulador, (a) y (b), y mapa a corto plazo (c).

Tras el inicio del algoritmo se añadió un objeto nuevo al escenario. Esto se representa en la figura 7.2. Vemos como el algoritmo añade el objeto al mapa y lo sitúa en una posición coherente respecto a la posición que ocupa el objeto en el escenario simulado.

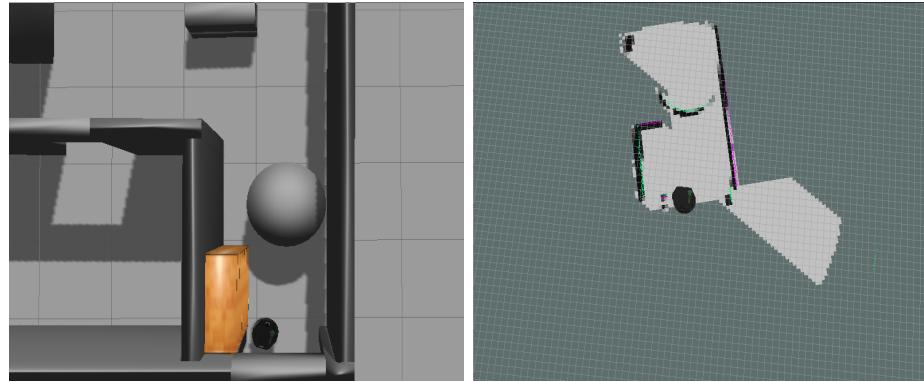


Figura 7.2: Añadimos un objeto al escenario

Una vez que el algoritmo ha incluido el objeto en el mapa procedemos a eliminarlo del escenario simulado. Esto se representa en la figura 7.3. Vemos como el algoritmo ha comenzado a borrar el objeto, por lo que el valor de las celdas que estaban ocupadas por el objeto ahora es mucho menor.

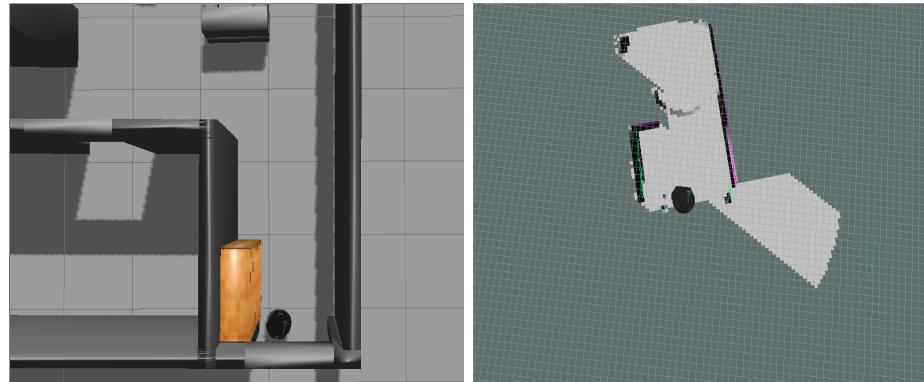


Figura 7.3: Eliminamos un objeto del escenario

## 7.2. Navegación con obstáculos dinámicos

## 7.3. Experimentación en la Robocup

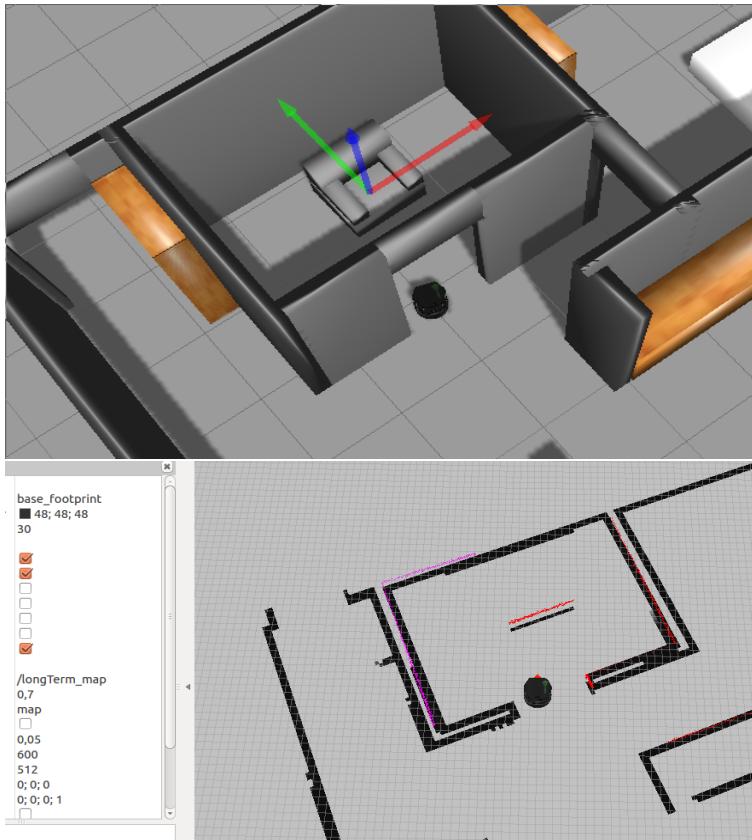


Figura 7.4: Añadimos un objeto al mapa de largo plazo

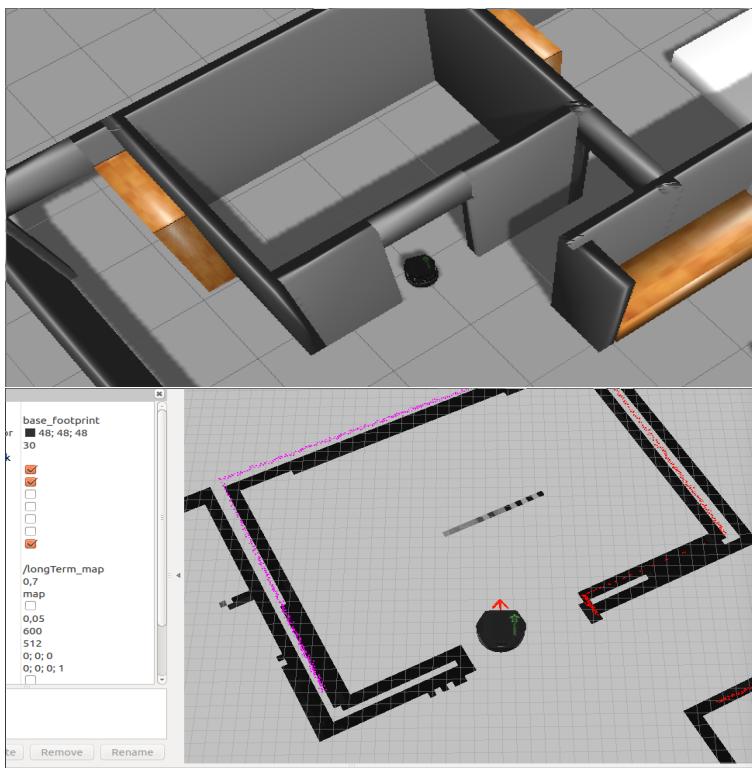


Figura 7.5: Borramos un objeto del mapa de largo plazo

## EXP ZONAS DESCONOCIDAS

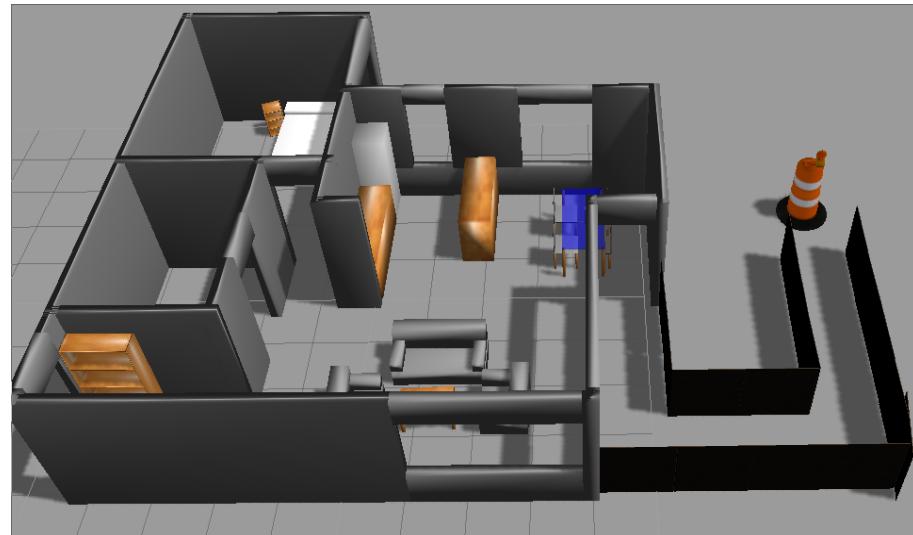


Figura 7.6: Escenario extendido

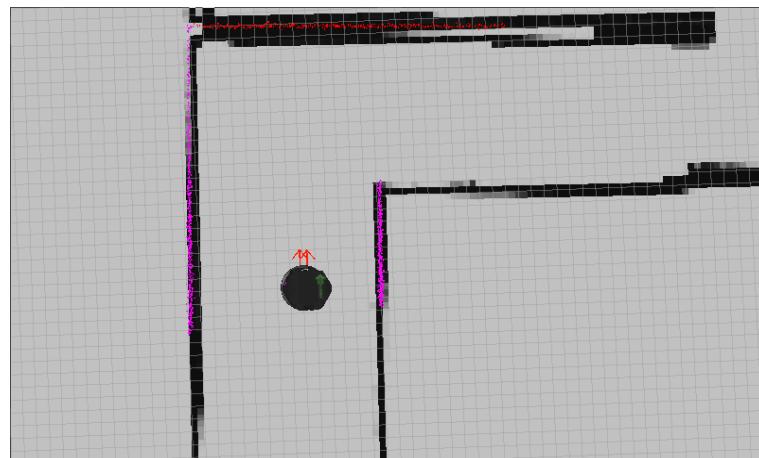
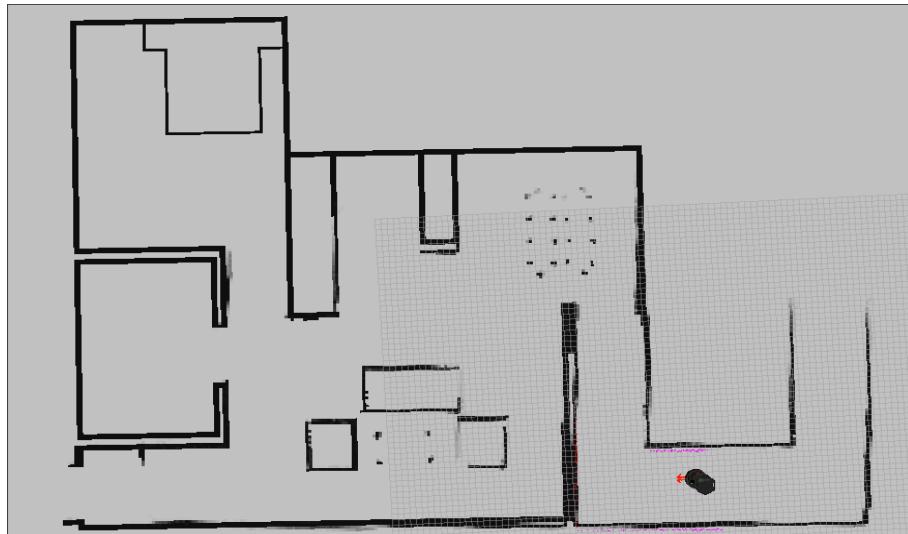


Figura 7.7: Detalle de la localización

En la figura 5.5 vemos como el escenario ha sido extendido, añadiéndole unos pasillos a la derecha de la casa. El mapa estático usado es el mismo que el mostrado en la figura 5.3. Vemos como después de recorrer la parte desconocida del mapa se ha añadido al mapa de largo plazo y también al mapa total, figura 5.7. Observamos en las flechas rojas bajo el robot que la incertidumbre en la posición del robot es mínima, figura 5.6.



(a) Mapa total



(b) Mapa de largo plazo



(c) Mapa de corto plazo

Figura 7.8: Mapas del escenario extendido

## **Capítulo 8**

### **Conclusiones y trabajos futuros**