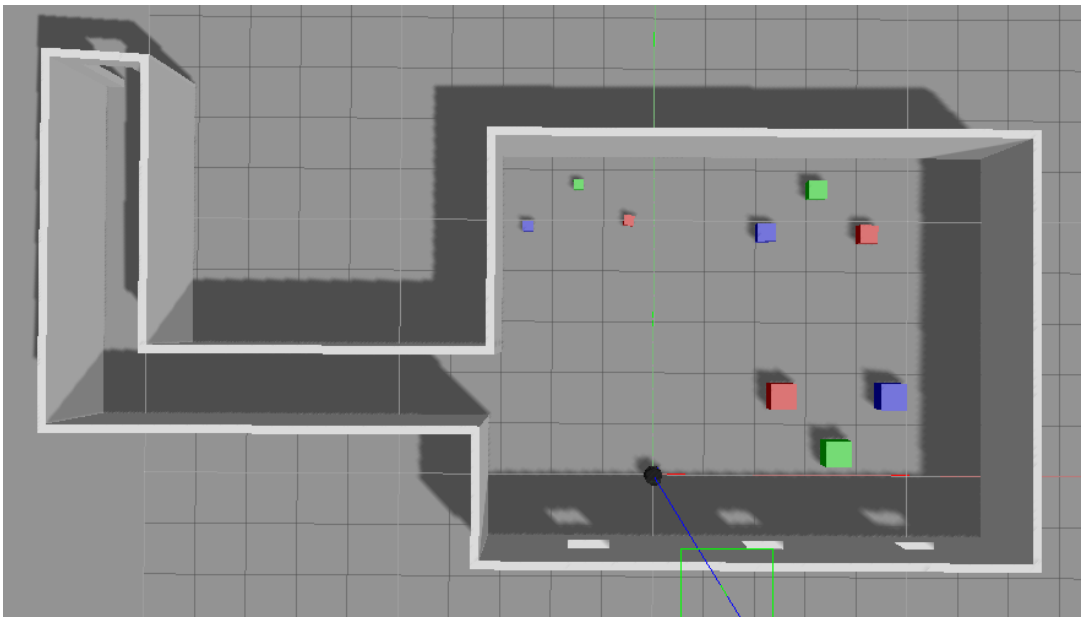
Aplicación real de un robot móvil

NOVELO TÉLLEZ, Yoinel

TORRES CÁMARA, José Miguel



Descripción breve

Se implementará una aplicación de logística a pequeña escala en la que a un Turtlebot se le describirá un objeto en lenguaje natural, lo buscará por visión artificial y lo entregará en un punto de entrega u otro según el objeto

Robots Móviles
Ingeniería Robótica
Universidad de Alicante
Enero de 2019

Índice

1. Introducción

Actualmente la robótica móvil es un campo con innumerables aplicaciones que van desde el ámbito asistencial hasta el de emergencias y rescate, pasando por la investigación de entornos peligrosos o de difícil acceso, así como por entornos domésticos e industriales. Sin embargo, uno de los aspectos más interesantes y donde más creemos que la robótica móvil, así como la robótica en general puede aportar es en la **colaboración con humanos**, de forma que se aprovechen las ventajas de ambos *sistemas* compensando los puntos débiles del otro.

Otro pilar importante que hemos considerado es el de la **visión artificial**. Tras el *boom* del 2012 en este campo, es una disciplina que cada vez se aplica en más entornos y la robótica móvil no es una excepción. Una cámara es un sensor del que se puede extraer una enorme cantidad de información invirtiendo en coste computacional con un coste económico considerablemente reducido. Por esto, nos pareció importante reservarle una parte a esta disciplina en nuestra aplicación para, por simple que fuese dicho espacio, sabernos capaces de llevarlo a otro nivel en futuras aplicaciones.

Evidentemente y como no podía ser de otra forma, la **navegación** es el aspecto esencial de la robótica móvil, por lo que incluir interacciones con el *stack* de navegación de ROS fue una parte imprescindible en nuestra práctica. La integración de este *metapaquete* la realizamos estableciendo ciertos objetivos de alto nivel que se lograban gracias al *path planning* ya implementado por otros miembros de la comunidad de ROS.

2. Descripción de la aplicación

Implementremos un sistema robótico de **logística interna**. Se le pedirá un objeto definido por dos variables (pequeño/mediano/grande y rojo/verde/azul). El robot entrará al almacén, irá a donde se almacenen los del tamaño pedido y, mediante un escaneo basado en visión artificial, cogerá el del color solicitado.

La clasificación por tamaño serán varios grupos de mesas (cada uno formado por una roja, otra verde y otra azul), un grupo para objetos pequeños, otro para objetos medianos y otro para objetos grandes.

Por ejemplo, en el grupo de mesas de objetos pequeños, sobre la mesa roja, habrá objetos pequeños de tipo 1, en la azul, objetos peq. de tipo 2 y en la verde, los peq. de tipo 3.

3. Creación del entorno

El **mapa** que contará con un **pasillo** en L con una puerta que accedía a una **sala** grande donde estarán las **mesas** y unas **ventanas** donde el robot entregará los objetos.

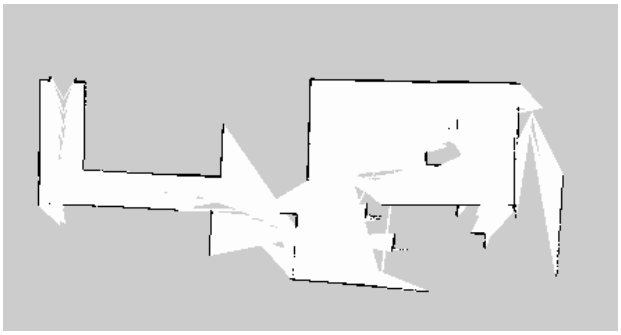
Para crear el mapa hemos desarrollado un entorno 3D en Gazebo (versión 7) del que generamos un mapa 2D utilizando un paquete que descargamos de GitHub (https://github.com/hyfan1116/pgm_map_creator). Este paquete, llamado

pgm_map_creator, se basa en otro, el *st_gazebo_perfect_map_generator* (https://github.com/koenlek/ros_lemtomap/tree/154c782cf8feb9112bc928e33a59728ca2192489/st_gazebo_perfect_map_generator). La principal diferencia entre ambos es la compatibilidad con las distintas distribuciones de ROS ya que el paquete original, a diferencia del utilizado por nosotros, no era compatible con ROS Kinetic.

La instalación del paquete se realiza ejecutando la instrucción *catkin_make* desde el *workspace* Catkin en el que se desee compilar. Aunque es posible que sucedan algunos errores de compilación debido principalmente a falta de dependencias. Sin embargo, estos problemas no tienen relación con la asignatura y son fácilmente solucionables con conocimientos básicos del manejo de sistemas GNU/Linux, por lo que no nos recrearemos en explicar sus causas ni soluciones.

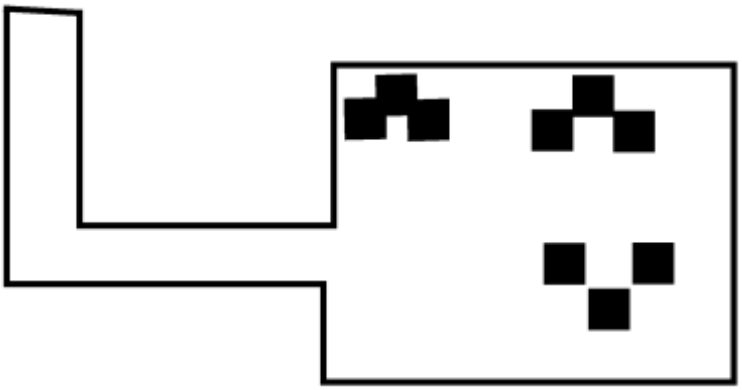
El **mapa 3D** modelado con Gazebo se ha realizado utilizando como base un modelo del mundo demo de Gazebo con iluminación. Mediante la opción *building editor* se han creado las paredes y huecos de estas (ventanas y puerta) y mediante *model editor* generamos las diferentes cajas. Fue necesaria la creación de un modelo específico para cada caja, pues no se podía partir de un modelo general. También hicimos algunos cambios en las propiedades, ya que necesitábamos distintos tamaños y colores.

Tras crear el modelo, intentamos **obtener un mapa 2D** del mismo mediante el paquete Gmapping y teleoperando al Turtlebot. Sin embargo, esto nos proporcionaba resultados muy poco realistas incluso tras intentar modificar algunos parámetros. Por ese motivo investigamos la forma de obtener un *ground truth* del mapa 3D, encontrando el paquete *pgm_map_creator* antes mentado.



Mapa resultado de mapeado con **GMapping** (con *fake laser*)

Para crear el *ground truth* del mapa hemos seguido las instrucciones del repositorio del paquete *pgm_map_creator*. El mapa resultante de la proyección en 2D del entorno tridimensional es el siguiente:



Mapa (**proyección**) 2D del entorno

Cabe destacar que el tamaño de las cajas difiere en la proyección 2D respecto del entorno 3D esto es debido a que en Gazebo cambiar las dimensiones de un modelo no implica cambiar el tamaño realmente, sino que es solo un efecto visual.

Una vez generado el modelo, sólo disponíamos de una imagen que se correspondía con el mundo 3D, pero no de los metadatos que debe contener el ***.yaml***. Por esto, tuvimos que generar dicho archivo manualmente. Para realizar esto, copiamos ese archivo de otro mapa y lanzamos la simulación y la visualización varias veces, ajustando diferentes resoluciones para que el escalado del mapa fuese el correcto y la localización pudiese ser efectiva y eficaz dadas las lecturas del Turtlebot en Gazebo (frente a un mal escalado, NO sería posible hacer *matching* de las lecturas).

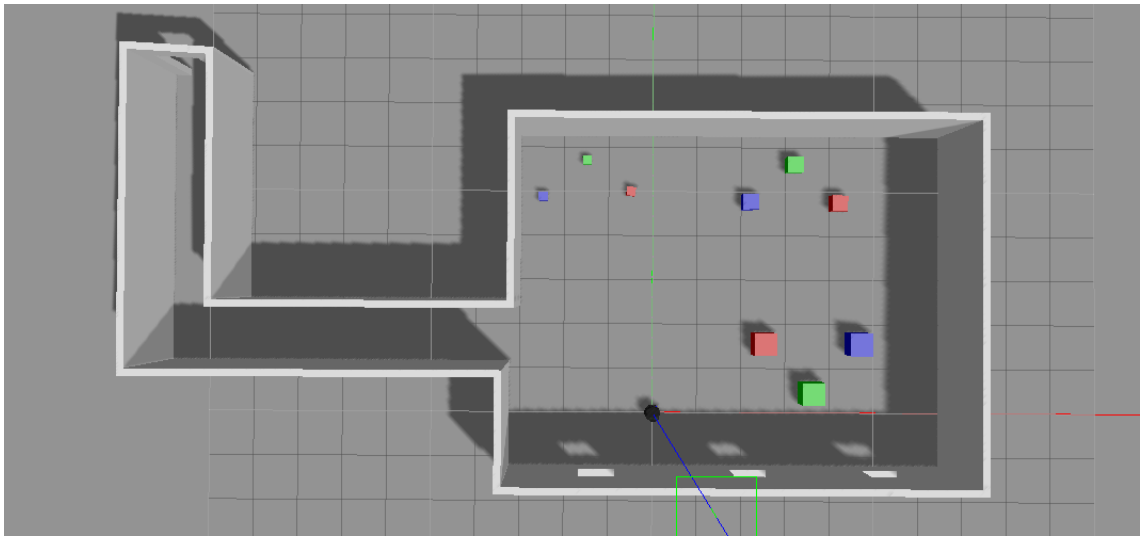
4. Componentes del sistema

4.1. main_process

Se encargará de recibir las órdenes de un usuario que solicite un objeto y publicarlas en los *topics* que convenga para, gracias a los otros nodos (explicados a continuación), lograr el funcionamiento anteriormente descrito.

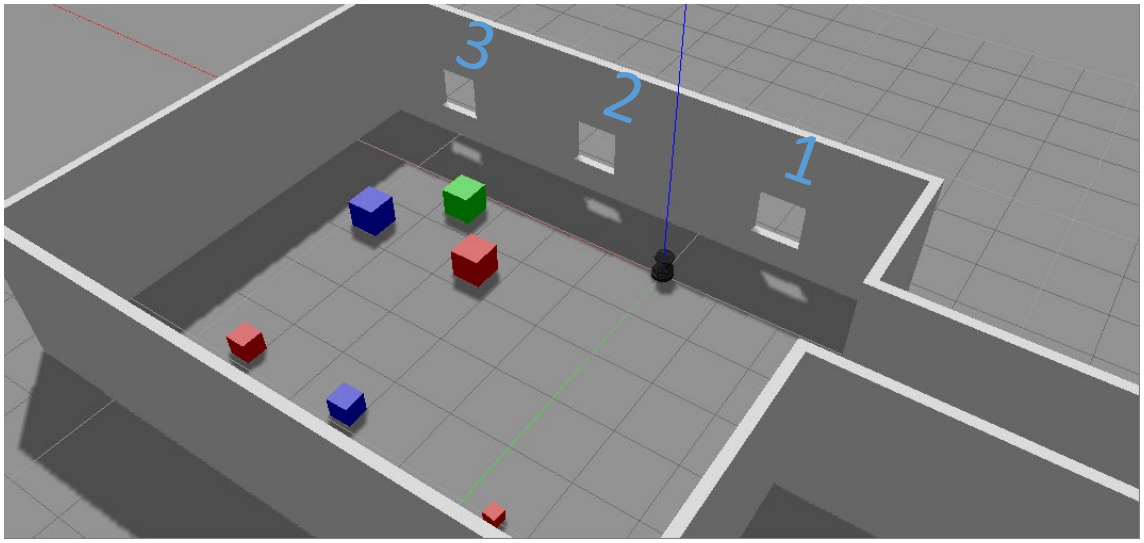
En cada ejecución de este nodo se le permite al usuario pedir un objeto al robot. Esta petición puede hacerse en lenguaje natural, ya que el sistema es capaz de discriminar todas las palabras a excepción de las relevantes (“pequeña”, “mediana” y “grande” respecto a tamaños y “roja”, “verde” y “azul” en lo referente a colores). De este modo, se identificará el objeto deseado extrayendo esas dos características de la descripción que proporcione el usuario. Cabe aclarar que en caso de especificar más de un tamaño o de un color en la misma frase el sistema responderá solo a la primera consigna.

Una vez escogido el objeto deseado, el robot irá a una posición fija donde estarán todas las cajas (roja, verde y azul) de un mismo tamaño (pequeña, mediana o grande). Esto implica publicar un “1”, “2” o “3” por */desiredAction*.



Grupos de **mesas** del almacén (pequeñas (grupo superior izquierdo), medianas (superior derecho) y grandes (inferior))

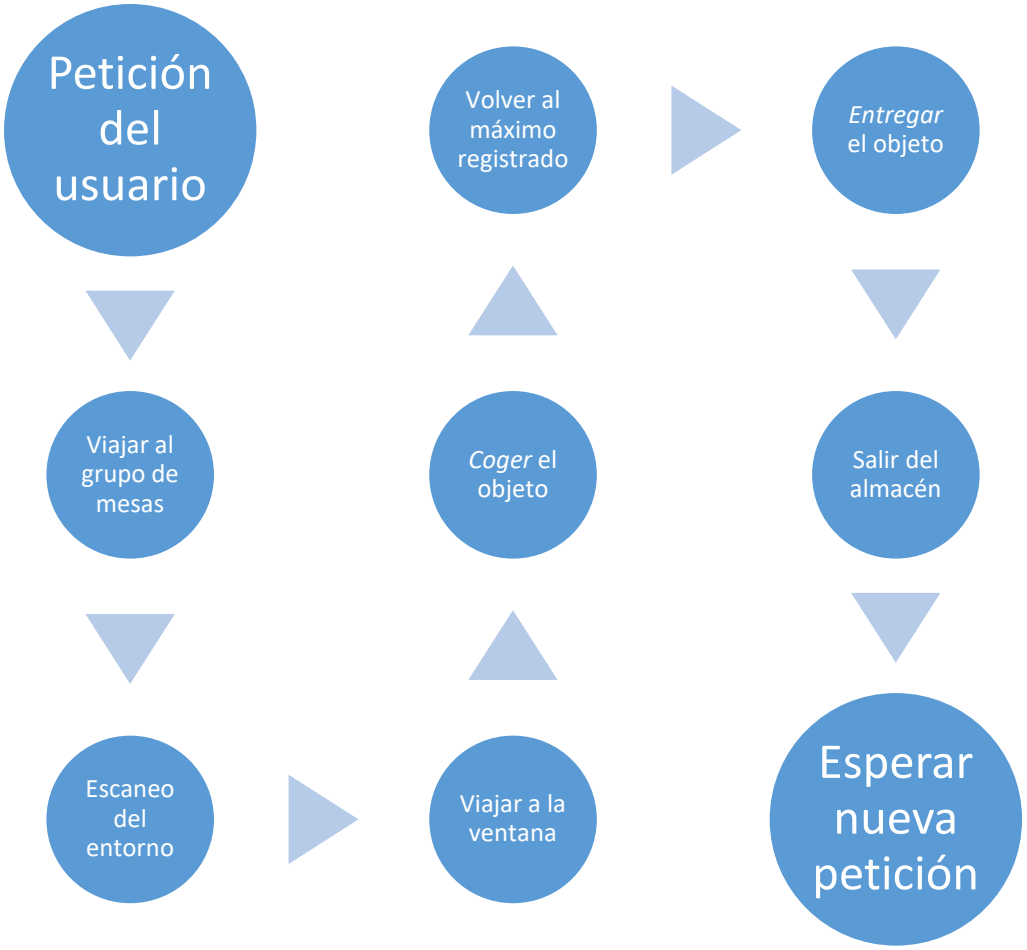
Una vez en medio de las tres cajas del tamaño solicitado, el robot comenzará a girar sobre sí mismo realizando un escaneo visual (enviar "7" por */desiredAction*) para localizar al objeto según el color pedido. Cuando se encuentre, se "cogerá" y se llevará a la ventana donde deba entregarse (la 1 si se ha pedido un objeto pequeño, la 2 si se ha pedido uno mediano y la 3 si es uno grande). Esto corresponde a enviar "4 ", "5" o "6" por el *topic* */desiredAction*.



Ventanas del *almacén* (entrega de objetos pequeños (1), medianos (2) y grandes (3))

Al "entregar" el objeto a través de la ventana, el robot volverá a salir del almacén ("0" por */desiredAction*) para esperar una nueva solicitud (ejecución del nodo *main_process*).

Resulta relevante indicar que para que el nodo *main_process* sea consciente de si se ha finalizado o no una acción, publicamos un *bool* a través de /goal_reached, emitiendo "false" cuando se esté realizando alguna actividad (viajar hasta un punto, escanear, etc.) o "true" si el robot está disponible y, por tanto, el proceso principal puede continuar su ejecución planeando e indicando la siguiente acción a llevar a cabo.



Flujo de ejecución del sistema

4.2. goal_publisher

Su función es suscribirse a un *topic* donde se publiquen mensajes Int32. Según el número que lea (0, 1, 2, 3, 4, 5, 6 o 7), se **llevará a cabo una u otra acción** (ir fuera del almacén, a las mesas pequeñas, medianas, grandes, a la ventana 1, la 2, la 3 o se escaneará el entorno).

La publicación de objetivos o goals la llevamos a cabo mediante la creación de un mensaje ***move_base_msgs::MoveBaseGoal***, del cual (asumiendo una variable de este tipo llamada *goal*) nos interesa su componente *goal.target_pose.pose*, que contiene las dos componentes que definen la pose destino, es decir, la posición (X, Y y Z) y la orientación (X, Y, Z y W), esta última en cuaternios.

Cabe destacar que para extraer las posiciones destino utilizamos la herramienta *Publish point* de RViz, que nos permitió establecerlas con enorme facilidad. Para los cuaternios, a pesar de que, por la posición de los ejes intuíamos su valor ([0, 0, 0, 1] para las poses 0, 1 y 2 y [0, 0, 1, 0] para las 3, 4, 5 y 6), decidimos confirmarlo ejecutando un *rostopic echo* sobre */amcl_pose*, pudiendo así asegurarnos de que estábamos en lo cierto.

En el caso particular de la acción 7, el robot se orienta a 0 radianes, aumentando de 0.2 en 0.2 radianes (aproximadamente 1º) hasta alcanzar los π radianes (180º), momento en el

que debería haber visualizado todas las cajas. Cada vez que avanza, lee el área que se percibe del color deseado y, en caso de ser mayor que la máxima percibida hasta el momento, se almacenan tanto el área como la orientación actual. Al finalizar el escaneo, se vuelve a la orientación para la que se ha detectado una mayor área, simulando que se coge el objeto (NO se coge nada por simplificar la implementación).

Este comportamiento nos resultó especialmente complejo de implementar, ya que nunca habíamos trabajado con cuaternios pero, tras realizar pruebas e investigar en foros y *wikis*, supimos que era necesario normalizarlos antes de enviarlos y que existen de dos tipos: TF y MSG. Mientras el primero sirve para procesarlo internamente en el nodo, el MSG se utiliza para publicarlo. Para crear todos los cuaternios intermedios entre las orientaciones entre -90º y 90º (-π/2 rad y π/2 rad), nos valimos de una función de ROS que implementa dicha conversión (*setRPY()*). Para el paso de cuaternios TF a MSG utilizamos *quaternionTFToMSG()*.

4.3. area_publisher

Un nodo de ROS que se suscribe a un *topic* donde se publican las imágenes captadas por el robot, las pasa a HSV (para aumentar robustez frente a cambios de iluminación), umbraliza por colores (rojo, verde y azul) y **publica** en otro *topic* las **áreas detectadas de cada color** en todo momento. El mensaje publicado es un vector de floats donde el primer elemento (0) será el área detectada de color rojo, el segundo, el área verde y, el tercero, el área azul.

5. Integración y arquitectura

5.1. p2_launcher.launch

Este *launchfile*, situado en *p2_robots_moviles/llaunch/*, hace uso de:

- El paquete turtlebot_gazebo para lanzar tanto la simulación 3D en Gazebo con nuestro mundo (mapa_final_con_mesas) como el *stack* de navegación de ROS con nuestro mapa (map_pr2.yaml y map_pr2.pgm) así como con la posición y orientación deseadas (15, 15 y 270º)
- El paquete *turtlebot_rviz_launchers*. Más concretamente, el archivo ***view_navigation.launch***, que abre un RViz para visualizar nuestro mapa cargado, las lecturas del robot, las partículas del AMCL, las rutas a seguir por el robot y el mapa local

Nuestro *launchfile* también ejecuta los nodos *area_publisher* y *goal_publisher*, explicados anteriormente. El *main_process* se deja sin lanzar por el diseño de funcionamiento que escogimos, según el cual en cada ejecución el robot sólo atendería una petición. Con esto queremos decir que se pedirá al usuario que introduzca un objetivo (escribiendo una frase en lenguaje natural en la consola), de forma que el robot reconocerá lo que se pide, irá al sitio aproximado donde se encuentre (mesas del tamaño pedido), lo buscará escaneando

colores, lo *cogerá*, lo *entregará* en la ventana correspondiente y volverá a la posición de inicio para esperar una nueva ejecución.

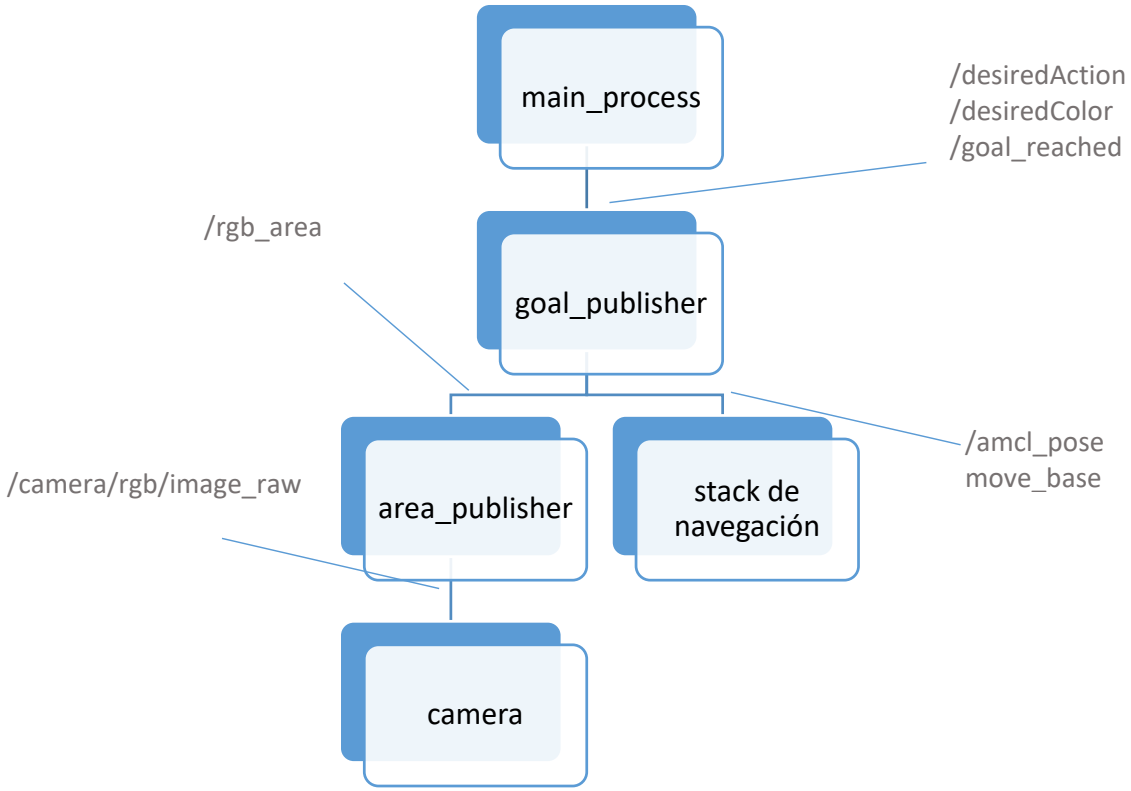
El motivo de haber elegido este funcionamiento es que se permite que el usuario o *algo* tenga el control sobre cuándo se ejecuta y se detiene el nodo *main_process*. Esto hace posible mejorar el sistema global incluyendo una interfaz web o una app que maneje dicha ejecución. También podría hacerse con un botón o cualquier otro dispositivo que sirva de HMI (Human-Machine Interface), como electrodos para bioseñales, pudiendo aplicar este sistema a un entorno asistivo para personas con movilidad reducida o simplemente para mejorar la experiencia de usuario y lograr un manejo más *user-friendly*.

5.2. Arquitectura

Consideramos especialmente relevante la forma de implementar la integración de todos los componentes del sistema al tratar con plataformas que tienen una filosofía bastante clara, como ROS. En todo momento tuvimos en mente que el sistema tenía que ser **flexible, escalable y** muy **modular**. Tratamos de reducir los posibles problemas que cualquier persona pudiese encontrarse incluyendo más *features*, modificando algún componente, sustituyendo alguna parte por otra nueva o incluso utilizando sólo alguno de nuestros componentes para otra aplicación.

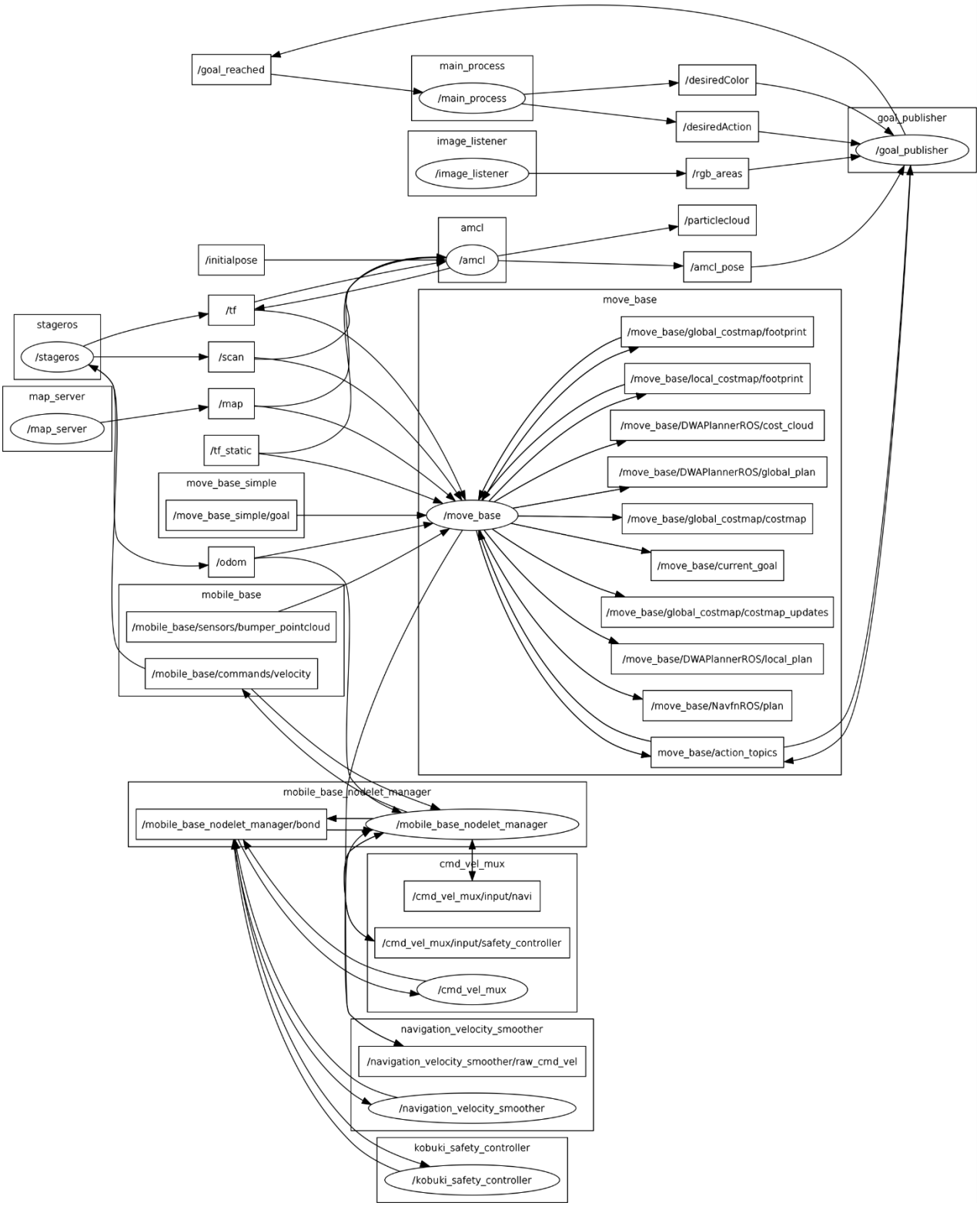
De este modo, tratamos de **adaptar** la clásica **arquitectura por niveles** tan ampliamente utilizada en el desarrollo de *software* **a una** arquitectura más **horizontal** como es la de ROS, permitiendo varios niveles de abstracción en cada capa. De este modo, a *main_process* no le importa cómo haga *goal_publisher* para cumplir las acciones que solicita ni a este último le afecte la forma que tenga *area_publisher* de devolver las áreas de cada color.

Dado este enfoque, puede cambiarse el funcionamiento de cada nivel o incluso la aplicación general y los estímulos a los que se respondan, ya que las formas en las que unas capas envían información a otras son muy versátiles. Por **ejemplo**, si en vez de a áreas se desea responder al porcentaje con el que una red neuronal puntúa la creencia de haber reconocido cierto elemento (de tres posibilidades), puede publicarse ese dato en */rgb_areas* y el resto del sistema seguiría siendo totalmente válido (aunque el nombre del *topic* deje de ser identificativo).



Arquitectura de la implementación

Con tal de clarificar al máximo la arquitectura interna que se origina en ROS como consecuencia de nuestro esquema, mostramos el grafo de computación de ROS que surge al tener todos nuestros componentes activos. Aunque es más confuso, se aprecian algunos detalles de más bajo nivel que resultan interesantes:



Grafo de computación interna de ROS

6. Pruebas y problemas

Las pruebas las comenzamos realizando en un **simulador 2D**, Stage, donde comenzamos comprobando que éramos capaces de publicar, a través de un nodo, **objetivos** a los que el robot iría utilizando el ***stack* de navegación** de ROS.

A continuación, implementamos la **comunicación entre varios nodos** para que uno (*main_process*), organizase los objetivos que serían mandados desde otro (*goal_publisher*).

Tras conseguir esto, nos dispusimos a **definir** todas las **poses** que necesitábamos (1 fuera del almacén, 3 para los grupos de cajas y 3 para ventanas), comprobando que éramos capaces de viajar hasta cualquiera de ellas desde cualquiera otra.

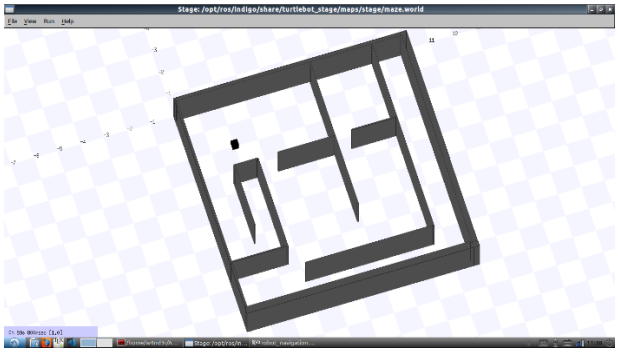
Después introducimos la acción número 7, el **escaneo**, teniendo particulares problemas con la definición de los cuaternios objetivo, ya que nos fue necesario convertir de ángulos RPY a cuaternio TF (de la librería tf), normalizarlo y convertirlo a cuaternio Msg (de la librería geometry_msgs).

Luego implementamos el nodo que recibía una **imagen**, hacía un ***bridge*** para poder utilizarla con OpenCV, y generaba las **áreas** de cada color para publicarlas.

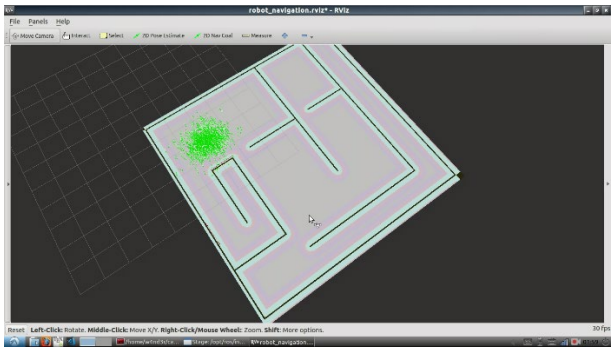
Por último, tras **integrar** todos los **componentes** desarrollados hasta el momento, realizamos pruebas pasando a la **simulación 3D** en Gazebo, donde encontramos algunos problemas en la sincronización entre componentes por la configuración de cuándo se antendían las *callbacks* que solucionamos, finalizando la implementación con éxito.

A modo de que el proceso quede plasmado de forma más visual y aclarar más la parte teórica del sistema, describiremos una prueba genérica del robot yendo a una pose en la simulación 2D:

Para realizar la mayoría de las pruebas, hemos utilizado dos de las herramientas más comunes para el desarrollo en ROS: El simulador bidimensional ***Stage*** y el visualizador ***RViz***. Pueden apreciarse en las siguientes imágenes:



Simulador 2D Stage

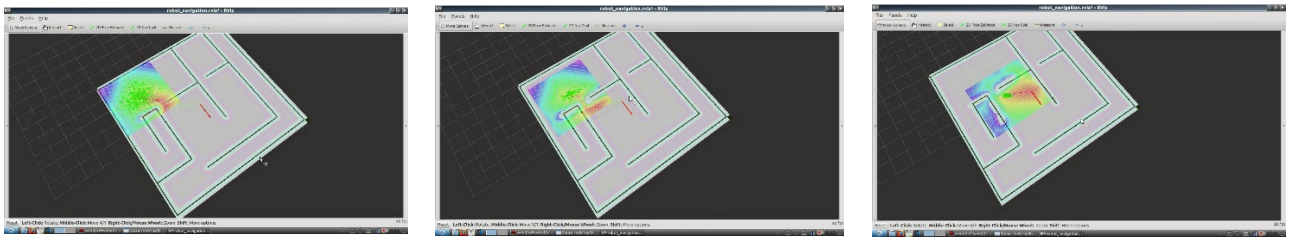


Visualizador RViz

En la ventana de **simulación** se aprecian las paredes del mapa utilizado para las pruebas (en 3D por proyección, en realidad no se pueden insertar elementos con formas o poses que varíen en altura), así como el robot, representado por el cubo negro que aparece en la zona superior izquierda del mapa.

Respecto al **visualizador**, en esa imagen está configurado para mostrar únicamente el mapa que se utiliza para la localización, así como la distribución de partículas con hipótesis sobre dónde está el robot. Durante las pruebas, también le añadimos las configuraciones necesarias para que mostrase las lecturas del láser del turtlebot (emulado utilizando la fila central de datos proporcionados por la Kinect), el *path* resultado de la planificación global para alcanzar el objetivo, el mapa de costes local y el objetivo en sí.

Por lo referente a la navegación del robot, este utiliza un mapa 2D del lugar en el que esté para localizarse utilizando **AMCL** (Adaptative MonteCarlo Localization), un algoritmo de localización basado en filtro de partículas que usa tanto la odometría y las órdenes de movimiento como las lecturas de los sensores en comparación a lo que se debería percibir en cada punto del mapa. Por eso, según se avanza más, se tiene más precisión en la localización, tal y como se aprecia en las siguientes imágenes, donde se ve cómo se van concentrando en un punto las partículas (flechas verdes):



Mejoras en la localización según se acumulan lecturas del entorno

En las imágenes anteriores también se ve un **mapa** de calor **local** (cuadrado de colores entorno al robot). Éste indica tanto las zonas que resultan más *caras* de desplazarse en colores fríos (más lejanas al objetivo o peligrosas por presencia de obstáculos o paredes) como las más *baratas*, en colores más cálidos (alejadas de obstáculos y cercanas al objetivo). El *goal* establecido es el origen de la flecha roja. Su sentido y dirección indican la orientación objetivo.

7. Trabajos futuros

Evidentemente y, a pesar de que consideramos haber logrado nuestros objetivos, nuestro trabajo está lejos de ser ideal o aplicable a un entorno real, por lo que vimos interesante hacer *brainstorming* y reflexionar sobre futuras posibles *features* o, al menos, pasos a seguir para poder seguir desarrollando este proyecto y hacerlo más real, versátil o interesante. Algunos de los puntos que consideramos más relevantes fueron:

- Implementar ***agarre y dejada*** de objetos. La base de nuestra aplicación es la logística, pero nuestro robot no lleva realmente ningún objeto de un lugar a otro. Dejamos esta parte a un lado dada su complejidad técnica y la poca relación con los

contenidos de la asignatura, pero entendemos que es un apartado esencial y debe ser el primer paso a dar para continuar el proyecto

- Mejoras en la ***visión artificial***. Actualmente el procesado visual es bastante básico, por lo que nos gustaría hacer algo más complejo, como reconocimiento de objetos o de códigos QR en lugar de colores y, quizás, combinando con el punto anterior, mejorar el proceso de *pick 'n' place* con datos visuales
- Más opciones de ***interacción humano-robot*** (HRI). En la introducción hicimos hincapié en la importancia que le dábamos a la colaboración para el futuro de la robótica. Un elemento evidentemente esencial en esta colaboración es la comunicación entre las personas y los robots, así que deberíamos ofrecer el máximo de naturalidad y posibilidades para que el usuario pueda escoger la que más le convenga (una GUI, por línea de comandos, por voz, con botones, bioseñales, un programa de más alto nivel, etc.)
- Implementar el ***goal_publisher* como un servicio** en lugar de como un nodo. En la concepción de ROS y sus elementos, las funciones que acarrea este componente se ajusta más a las de un servicio. Sin embargo, decidimos invertir el tiempo en otros puntos en lugar de aprender a utilizar servicios y migrar el desarrollo del nodo a este otro tipo de elemento

8. Instrucciones de ejecución

En el README.md del GitHub que hemos utilizado para realizar las últimas modificaciones de la práctica y organizar los archivos más claramente (subimos las cosas desde una de las dos cuentas, pero trabajamos en ellas los dos) figuran instrucciones de uso en inglés, pero las incluiremos aquí también para mayor claridad:

El proceso de **instalación** es el clásico de cualquier paquete de ROS de distribuciones que usen Catkin. Se debe crear un *workspace* Catkin (o utilizar uno ya creado), luego, en el directorio *src* del mismo, introducir nuestro paquete (*p2_robots_moviles*). Quedando sólo la necesidad de compilarlo ejecutando *catkin_make* desde la raíz del *workspace*.

Para **ejecutarlo**, asumiendo que ya se ha hecho *source* del *workspace,* simplemente habrá que lanzar el p2_launcher.launch ejecutando desde una sesión de terminal

$ roslaunch p2_robots_moviles p2_launcher.launch

Con esto, quedará lanzada la simulación 3D en Gazebo, el *stack* de navegación de ROS y una visualización con RViz así como los nodos *área_publisher* y *goal_publisher*, tal y como se comentó en secciones anteriores. Por último, para cada objeto que se desee pedir, hay que lanzar manualmente el nodo *main_process* de la siguiente manera:

$ roslaunch p2_robots_moviles main_process

Ahora se solicitará que se introduzca cualquier oración indicando que caja se desea, como "Tráeme una caja roja y mediana" o "Por favor, búscame una caja grande que sea de color azul y entrégala". Estos son sólo dos ejemplos, pero el nodo será capaz de entender cualquier oración, tal y como se ha explicado.

9. Conclusiones

Tras realizar esta práctica hemos refinado muchísimo nuestra comprensión de **ROS** como plataforma, así como interiorizado la forma de trabajar con él. También hemos adquirido habilidades transversales por el uso de **Git** como sistema de control de versiones y de **GitHub** como plataforma de desarrollo colaborativo.

Así mismo, hemos apreciado la importancia de conocer las **bases teóricas** de la navegación, así como algunos aspectos del funcionamiento de ROS a **bajo nivel** para depurar errores de nuestro sistema. Sin todo el conocimiento previo adquirido a lo largo de la asignatura nos había llevado mucho más tiempo el **descubrimiento y** la **corrección de** algunos **fallos** que fuimos capaces de detectar y suplir con relativa facilidad.

Respecto a los resultados obtenidos, concluimos que uno de los puntos más relevantes es que permiten una **escalabildad** prácticamente ilimitada (dejando de lado las limitaciones *hardware*), ya que pueden definirse más poses, modificarse las actuales, indicarle que vaya a cualquier lugar del mapa a entregar los objetos, etc. Además, puede utilizarse con cualquier robot, pudiendo tanto añadirle complejidad como reducir costes, según los objetivos de cada aplicación real.

Otra conclusión que extrajimos fue la gran capacidad que ofrece **ROS** y lo versátil que es no sólo para la programación de robots móviles ni robots en general, si no para fomentar el movimiento del ***software* libre**, la **colaboración y** las **comunidades** de desarrolladores.

Por último y, ligeramente en relación a lo que comentábamos en la introducción, consideramos que nuestro resultado tiene un gran potencial para añadirle funcionalidades y esto nos hace ratificarnos en nuestra opinión acerca de la existencia de **infinidad de campos de aplicación de la robótica móvil.**