Grado en Ingeniería Electrónica Industrial

Escuela Politécnica Superior de Jaén

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior de Jaén

Trabajo Fin de Grado

INCLUSIÓN DEL ROBOT PMB-2 EN LA PLATAFORMA BASADA EN ROS

Alumno: **Enrique Manuel Ortega Vázquez**

Tutores: Prof. D. Pablo Cano Marchal
Prof. D. Diego Manuel Martínez Gila

Dpto: Ingeniería Electrónica y Automática

Septiembre, 2017

Universidad de Jaén
Escuela Politécnica Superior de Jaén
Departamento de Ingeniería Electrónica y Automática

Don Pablo Cano Marchal, tutor del Proyecto Fin de Carrera titulado: Inclusión del robot PMB-2 en la plataforma basada en ROS, que presenta Enrique Manuel Ortega Vázquez, autoriza su presentación para defensa y evaluación en la Escuela Politécnica Superior de Jaén.

Jaén, septiembre de 2017

El alumno: Los tutores:

Ortega Vázquez,

Enrique Manuel

Pablo Cano Marchal / Diego Manuel Martínez Gila

RESUMEN

El presente trabajo de fin de grado trata de la integración de dos robots, llamados Meka y PMB2, en la plataforma de software libre de ROS, con el fin de interactuar y desplazarse en un entorno parcialmente desconocido.

Para ello, se desarrollará una metodología de aprendizaje sobre el hardware y el software previamente integrados en los robots. Además, se creará o modificará el software necesario para compatibilizar ambos sistemas y que funcionen al completo, así como algunos programas para ejecutar el lanzamiento y la operación remota del nuevo sistema. Otro de sus principales objetivos es acoplar la plataforma robótica a nuevos sensores total o parcialmente integrados en ROS, así como su conexión y comunicación.

Finalmente, ordenar el cumplimiento de una tarea específica que englobe todos los sistemas en un trabajo cooperativo, siempre con vistas al futuro en investigación que la plataforma posibilita.

ABSTRACT

This project focuses on integrating two robots, which names are Meka and PMB2 within ROS robotic platform, in order to move and interact in a partially unknown available environment.

To achieve that, a specific methodology, based on learning about included hardware and software previously integrated in those robots will be developed. Furthermore, needed software to coordinate both systems along with some programs to remotely operate will be created. Another main goal is to integrate the robotic platform with new sensors.

Finally, a program to achieve a specified task using both robots and sensors is required. The future of this work is to use it in investigation.

Índice

Listado de figuras

1. Introducción, motivación y objetivos

Un robot es una integración de elementos mecánicos, electrónicos y software para conformar una máquina versátil de propósito general automáticamente controlada, reprogramable y que pueda manipular un entorno (parcialmente) desconocido. Para ello recoge información de dicho entorno mediante diversos sistemas de adquisición de datos; un claro ejemplo puede ser un sistema de visión por computador. Por lo tanto, los robots funcionan en un lazo de control cerrado.

Este trabajo de fin de grado surge de la necesidad de integrar dos robots en una única plataforma interconectada junto con otros sensores para cumplir tareas de propósito general. Los robots son los siguientes:

- **Meka**, del antiguo fabricante Meka Robotics, un robot humanoide con dos brazos de 7 grados de libertad y mano antropomórfica con el fin de interactuar sobre el entorno o con seres humanos.
- **PMB2**, del fabricante Pal Robotics, una base móvil autónoma sobre la cual irá montada el anterior robot para transportarlo en un entorno parcialmente desconocido.

La base móvil autónoma ofrece una ventana al futuro de los vehículos personales autónomos, un campo con gran financiación en la actualidad, ya que la gran mayoría de marcas están apostando por esta nueva rama de la automoción. En el caso de la base autónoma, al igual que un coche autónomo (Figura 1), debe ser capaz de tomar continuamente decisiones sobre cómo comportarse mientras está circulando, de forma coherente con las reglas y límites para los que fue programado, siempre atendiendo a unas leyes establecidas por el hombre y que restringen su comportamiento.

Figura 1: Coche autónomo marca Ford

Los sistemas autónomos de conducción deben ser capaces de monitorizar su propio estado de forma autónoma, advertir posibles fallos y actuar en consecuencia para llevar el vehículo a una posición y estado seguros.

En dichos robots existen algunos sensores, pero especialmente actuadores que funcionarán en conjunción con otros sistemas, como cámaras, sistemas de adquisición de datos o terminales de operación y procesamiento para su aplicación conjunta. La combinación entre ellos permite que la interacción con el entorno sea mucho más completa y dinámica, ya que el PMB2 puede transportar al Meka y detectar obstáculos gracias a los sensores que tiene.

El objetivo de integrar ambos robots en una única plataforma es, como se ha mencionado anteriormente, ***crear un sistema inteligente consciente de su entorno parcialmente desconocido y capaz de actuar en él, incluyendo seres humanos***. Para ello, se cumplirán los siguientes objetivos parciales:

- Unión y conexión de los robots con los demás elementos de la plataforma, ya que inicialmente se encuentran separados.
- Implementación de una navegación autónoma parametrizable por el usuario, capaz de esquivar obstáculos y que llegue a puntos definidos por programación.

- Estudio y realización de una posible interacción del robot humanoide (Meka) con el humano.
- Programación de envío de metas para un caso específico, en este caso, navegar hacia un humano con el fin de interactuar con él.

Para la metodología, el desarrollo de este proyecto y cumplir con los objetivos anteriormente mencionados, se ha realizado una primera toma de contacto con cada uno de los robots, especialmente en el caso del PMB2, ya que se encontraba programado por la empresa que lo construyó y se debía entender el funcionamiento que tiene. Las pautas seguidas para cumplir con los objetivos listados anteriormente son las siguientes:

- Estudio y dominio de la plataforma ROS (Robot Operating System), además de una explicación básica de su funcionamiento.
- Modelado de la unión de ambos robots para que el modelo físico-matemático sea correcto y facilite ciertas funciones de posicionamiento e interacción.
- Estudio del funcionamiento del robot Meka, sus cinemáticas inversas y programación para enviar posiciones de la mano respecto a otros puntos.
- Parametrización de la navegación autónoma acorde a la nueva unión con el Meka.
- Planificar conexión entre los sistemas teniendo en cuenta la movilidad del PMB2 y del Meka, además de su alimentación.
- Compatibilidad con el proyecto “Fusión sensorial” [1] para el caso de estudio.
- Instalación de un terminal que será utilizado para el desarrollo de software, terminal de conexión, visualización y operación entre los ordenadores de los robots.
- Programación de una interfaz gráfica para controlar todas las funciones creadas.

La plataforma que se pretende crear constará principalmente de los robots con los que se trabaja en el presente proyecto. El esquema de funcionamiento de cada uno de los robots se incluirá en los capítulos correspondientes, mientras que los

elementos adicionales se describirán en el capítulo 5: caso de estudio. El número de ordenadores en funcionamiento no está reflejado, ya que se puede variar, pero el mínimo necesario es de 3: uno por cada robot y otro para lanzar y dirigir toda la plataforma además de manejar el funcionamiento de las cámaras perteneciente al proyecto de fusión sensorial [1].

Para dotar de funcionalidad a un robot en cualquier entorno es indispensable el empleo de sensores que recojan información del mismo; no obstante, dicho entorno deberá cumplir unas condiciones de contorno, a definir al inicio del desarrollo de la aplicación, para asegurar que se respetan la mayor parte de las variables en juego.

La estructura del resto del trabajo de fin de grado es la siguiente:

- Capítulo 2: ROS (Robot Operating System), donde se explica la plataforma en la que se han programado y comunicado los robots.
- Capítulo 3: Base móvil autónoma PMB2, donde se realiza una breve descripción del punto de partida, hardware, software, y la implementación de la navegación autónoma ademas de todos los requerimientos necesarios.
- Capítulo 4: Meka, al igual que en el anterior capítulo se explica el punto de partida, las cinemáticas inversas aunque no su implementación y su uso en conjunción con otros sistemas de percepción.
- Capítulo 5: caso de estudio donde se utilizarán todos los elementos de la plataforma conjuntamente, como demostración inicial de la capacidad cooperativa.
- Capítulo 6: GUI (*Graphical User Interface*) o interfaz gráfica de usuario para ejecutar cómodamente los comandos utilizados.
- Capítulo 7: conclusiones de los resultados obtenidos acorde con los objetivos propuestos.
- Capítulo 8: perspectivas futuras y siguientes pasos a realizar para un correcto aprovechamiento de la plataforma.

2. ROS (Robot Operating System)

En este apartado de la memoria se va a explicar la estructura de la plataforma software elegida como medio de comunicación e intercambio de datos entre todos los elementos que conformarán la aplicación, que no es otra que ROS [2]. Los motivos por los que esta plataforma ha sido seleccionada frente a otras, aunque brevemente citados en la introducción, son: carácter *open-source*; compatibilidad y soporte para gran cantidad de sensores y actuadores; facilidad para reutilizar código y escribirlo en varios lenguajes de programación; estructura topológica nodular, que permite segmentar la aplicación en subnúcleos independentes y comunicables entre sí… Además, ROS también resulta ser la mejor elección por estar ya presente como sistema operativo principal en los dos robots que se van a utilizar en el: PMB-2 y Meka M1.

2.1. ¿Qué es ROS?

Para implementar y conectar ambos robots entre sí, se utilizará la plataforma ROS [3] [4], ya que se trata de una infraestructura flexible y abierta pensada para programar el software de cualquier robot, la cual se utilizará como middleware de comunicación entre todos los dispositivos. ROS es un framework para escribir y desarrollar software para robots. Se puede entender como el conjunto de herramientas, librerías y convenios cuyo objetivo es resolver la problemática de crear comportamientos robóticos robustos y complejos, simplificando dicha tarea y extendiéndola a una amplia variedad de robots y plataformas.

2.2. Antecedentes históricos y presente de ROS

ROS fue originalmente desarrollado [3] en 2007 por el Laboratorio de Inteligencia Artificial de Stanford (SAIL en inglés), enfocado al desarrollo del STAIR [5] (Stanford Artificial Intelligence Robot). El objetivo era desarrollar un robot capaz de navegar en entorno de hogar y oficina, manipulando objetos y herramientas e interactuando y ayudando a personas en diferentes tareas. Para ello, se requería un software flexible y dinámico, capaz de adaptarse a las diferentes variaciones requeridas por las

condiciones de trabajo del robot con el mínimo número de modificaciones posible en el código; este concepto se fue ampliando hasta nacer ROS.

La comunidad de ROS está creciendo muy rápido y existen desarrolladores en todo el mundo. Muchas compañías robóticas están desplazando su software para que sea compatible con ROS, como Siemens, ABB, BMW, etc. Esta tendencia es visible en los robots industriales [6], donde las compañías (Figura 2) cambian el software propietario (que además supone un coste elevado) a ROS-Industrial: una solución ideada para empresas de automatización compatible con el ROS usado en el presente proyecto, pero con herramientas específicas para el mundo industrial.

Figura 2: ROS-Industrial en empresas americanas

El movimiento de ROS aplicado a la industria está ganando terreno en los últimos años, con lo que una gran parte del desarrollo se realiza en este campo. El aumento de aplicaciones en ROS [7] puede generar muchas oportunidades de trabajo, por lo que en los próximos años, ROS probablemente será un requerimiento esencial para un ingeniero dedicado a la robótica.

2.3. Razón de uso en el presente trabajo

La razón principal es que ambos robots son compatibles con la plataforma ROS, con lo que una gran parte del hardware (sensores, servomotores, etc.) tendrá los drivers incluidos en dicha infraestructura, además de los modelos vectoriales y parte de las funcionalidades que deben realizar.

Por otro lado, tiene herramientas para ser un middleware de comunicación con los demás dispositivos, facilitando en gran medida la conexión entre los diferentes equipos. En el presente proyecto, el equipo director junto con los sistemas de percepción y los robots, pueden llegar a conformar un total de 4 ordenadores completamente conectados entre sí.

Una lista de ciertas ventajas [7] frente a otros sistemas para la programación e integración de sensores y robots es la siguiente:

- El 100% del software utilizado es libre y gratuito, incluido el sistema operativo sobre el que se ejecuta: Ubuntu.
- Utilidades de alto nivel y algoritmos complejos para una implementación sencilla y compatible, se puede mencionar **SLAM** (Simultaneous Localization And Mapping) y **AMCL** (Adaptive Monte Carlo Localization) utilizados para la navegación autónoma, MoveIt, o **SMACH** (librería de programación) para reproducir tareas complejas.
- Gran cantidad de herramientas para depurar, visualizar y realizar simulaciones robóticas. RViz y Gazebo son algunas de las herramientas a destacar, siendo la segunda un interesante simulador para probar código sin necesidad del robot real.
- Drivers para un repertorio destacable de sensores y actuadores, como la cámara TOF, Kinect, Arduino, LIDAR y servos (ambos incluídos en los robots).
- Operabilidad interplataforma: los nodos se pueden programar en C++ para mayor eficiencia o Python, con total flexibilidad entre ellos.
- Capacidad para gestionar tareas multihilo y concurrentes de forma eficiente. Además, el sistema es descentralizado, por lo que tareas de computación exigentes se pueden realizar en sistemas independientes pero conectados.
- **Independencia y modularidad**: una tarea compleja se divide en distintos programas llamados nodos, reutilizables para otras tareas. Además, muchos nodos siguen ejecutándose si otros fallan, por lo que el sistema puede seguir funcionando.
- Amplia comunidad de desarrolladores, que ofrecen tanto soporte y ayuda como nuevas contribuciones de código abierto y gratuito; esto permite la reutilización de código, lo cual hace que no haya que comenzar desde cero cada proyecto

y permite simplificar la tarea de acometerlos, sino que simplemente aportar nuevas funcionalidades o características que, además, conviene compartir para que la comunidad siga creciendo y evolucionando, en un continuo proceso de realimentación por parte de los usuarios.

Todas estas ventajas hacen que ROS sea la plataforma de programación para robots más utilizada en la actualidad, y la escogida para la realización del proyecto; no obstante, se citan también algunas desventajas [8] del *framework*:

- Curva de aprendizaje muy pronunciada al inicio para comprender el funcionamiento básico.
- Dificultad en empezar simulaciones, ya que requiere muchas configuraciones además de un ordenador potente.
- Dificultad en la modelización del robot. Se usan modelos llamados “URDF” (*Unified Robot Description Format*), que se pueden crear en Solidworks, pero sin dichas herramientas las formas complejas pueden llevar mucho tiempo, ya que se basan en formas geométricas simples. Es por ello, que este proyecto depende de los modelos que ofrece el fabricante.
- Necesidad de un ordenador con Linux para ejecutar ROS. Aunque existe una versión para Windows, no es totalmente compatible, puesto que es experimental. Algunos robots pequeños y simples pueden funcionar exclusivamente con microcontroladores, por lo que ROS sería innecesario en dichos casos. Para los robots que se usarán, las tareas que ejecutan son demasiado complejas para dicha opción y necesitan de un computador.

2.4. Estructura de ROS

ROS se puede representar como una **red topológica** formada por nodos o programas, que realizan tareas específicas y se comunican entre sí para intercambiar información. Este *framework* consta de cuatro [9] componentes fundamentales: el núcleo, la infraestructura de comunicaciones, las utilidades específicas para robots y las herramientas de desarrollo y visualización.

- Núcleo: llamado “roscore”, consta de una serie de programas que son requisito indispensable para el funcionamiento de un sistema basado en ROS. Proporciona la habilidad de comunicación punto a punto (P2P) entre nodos, así como nombre y servicios de registro para cada uno de ellos; además, el core se encarga de establecer un servidor de parámetros general a todo el sistema.

  Una vez que los nodos se registran en el roscore, comunican al mismo la información que suministran, y aquella a la que desean suscribirse, que provendrá típicamente de otros nodos conectados a la red; será el núcleo el encargado de proporcionar a los nodos las direcciones de los otros nodos con los que se desea establecer la comunicación; por lo tanto, aunque los nodos estén conectados entre sí (a través de tópicos, como se explicará más adelante), virtualmente todos ellos están relacionados con el roscore, como se observa en la Figura 3.

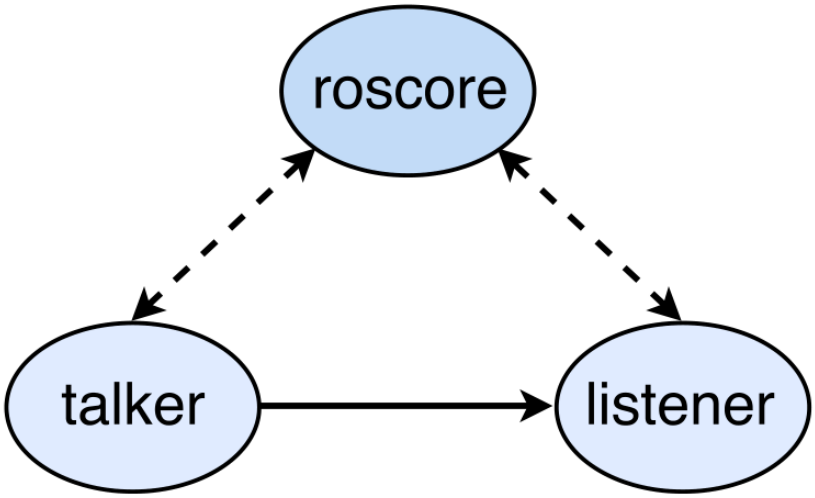

Figura 3: ejemplo conexión entre nodos

- Infraestructura de comunicaciones: el intercambio de información entre programas se realiza a través de mensajes, por medio del método de publicación/suscripción. Dicho método de comunicación facilita la divisón y estructuración del código, lo que permite que éste pueda ser reutilizado en el futuro con facilidad. Los mensajes [11] que los diferentes nodos o programas intercambian entre sí pueden pertenecer a tipos ya definidos por ROS y habituales en las aplicaciones de desarrollo de robots, tales como números,

matrices o estructuras más complejas, como información de sensores en entornos tridimensionales; o también pueden pertenecer a tipos definidos por el propio usuario, que puede así establecer su propio protocolo de intercambio de información entre nodos.

Además, los mensajes pueden transmitirse de forma asíncrona o síncrona, de forma que los nodos simplemente utilicen la información proveniente de otros nodos cuando estos la publiquen (en el caso asíncrono); o permitiendo interacciones del tipo cliente/servidor, habilitándose así la posibilidad de establecer prioridades y permisos entre nodos en el caso síncrono. Finalmente, también existe un servidor de parámetros, que permite modificar manual o automáticamente los ajustes de las diferentes tareas en tiempo de ejecución.

- Utilidades específicas para robots: debido a que ROS es un *framework* enfocado al desarrollo de aplicaciones y software específico para robots, incluye facilidades para la consecución de dichas tareas, tales como:

    - Mensajes estandarizados para robots, que cubren la mayoría de casos habituales en robótica: posiciones, transformadas, vectores… para el caso de los modelos geométricos; nubes de puntos, escaneos provenientes de láseres, imágenes… para adquirir información del entorno a través de sensores; y datos de odometría, mapas y rutas, indispensables para la navegación.
    - Lenguaje específico para la descripción de robots, en formato URDF, que consiste en documentos XML en los que se especifican las propiedades físicas de las partes del robot, así como la forma de interconexión entre cada una de ellas. Esto permite la fácil visualización de los robots en entornos de simulación trimidensionales.
    - Herramientas de ayuda a la navegación, localización y posicionamiento de robots, algunas de ellas incluidas en las herramientas de visualización, como las funciones 2D Pose Estimate y 2D Nav Goal incluidas en **Rviz** [12].

- Herramienta de diagnóstico y detección de errores llamada *roswtf*.

Desarrollo y visualización: entre ellas, son destacables los comandos de consola específicamente diseñados para ROS, que facilitan la creación, compilación y edición de paquetes y nodos, así como la navegación entre ellos y el lanzamiento y puesta en ejecución de los mismos. Además, ROS ofrece varias herramientas de visualización, tales como **Rviz** (para visualizar la información proveniente de sensores y los modelos cinemáticos de los robots, así como la navegación en el entorno de los mismos) y **rqt** (creación de interfaces gráficas y visualización de datos). Por último, hay que destacar que dichas funcionalidades se pueden ampliar gracias al amplio catálogo de software libre disponible en los repositorios de ROS para su descarga y utilización.

2.5. Nivel del sistema de archivos de ROS

Para comprender correctamente cómo funciona ROS, primero se hará hincapié en cómo se organizan los archivos [13] en el disco, como se muestra en la Figura 4.

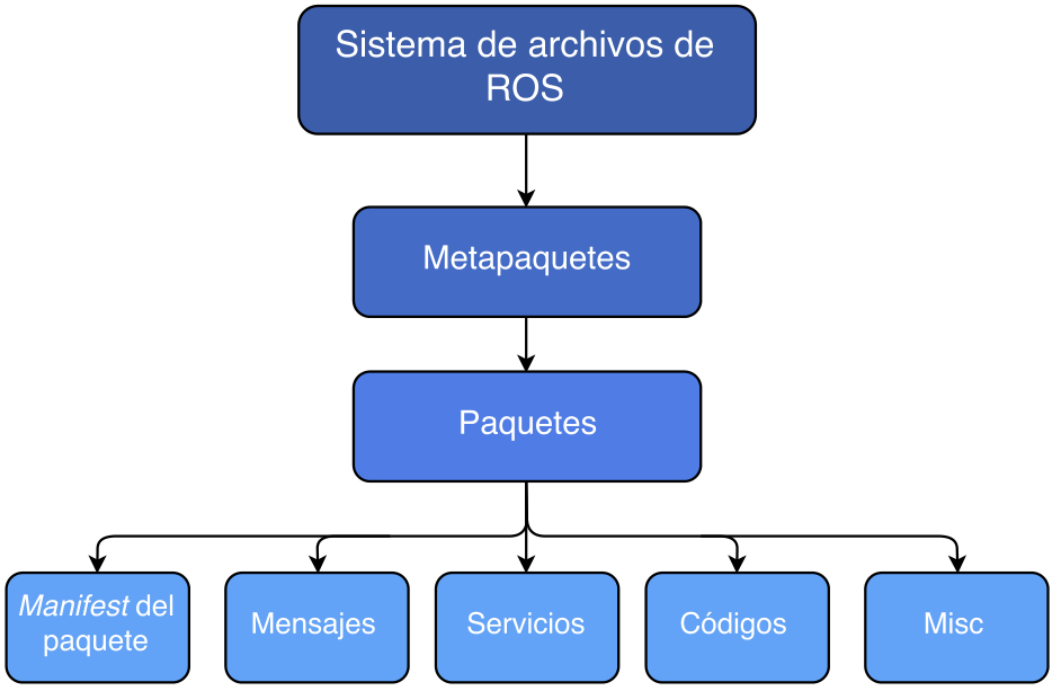

Figura 4: nivel del sistema de archivos de ROS

- Paquetes: son la unidad básica y central de software en ROS. Contiene ejecutables (nodos), librerías, archivos de configuración, archivos de lanzamiento y nuevos tipos de mensajes, servicios y acciones que no sean los incluidos por ROS.
- Manifiesto del paquete: es un archivo que contiene información del propio paquete, autor, licencias y las dependencias de otros paquetes. El nombre que suele tener es *"package.xml"*.
- Meta-paquetes: Es utilizado para un grupo de paquetes con un propósito específico, donde los paquetes internos suelen tener dependencias entre sí, como en el caso del paquete de navegación que se utilizará en futuros capítulos del presente trabajo.
- Mensajes: Los mensajes son un tipo de información que los procesos de ROS se envían entre sí. La extensión es *".msg"* e irán declarados en el manifiesto, además de estar incluidos en la carpeta "msg".
- Servicios: Similar a los mensajes, pero utilizado cuando los procesos utilizan una comunicación del tipo cliente/servidor. La extensión es *".srv"* e irán dentro de la carpeta *"srv".*

Una explicación más detallada acerca de cómo crear, mediante comandos, esta estructura de archivos, se encuentra disponible en el **Anexo I:** Información adicional de ROS.

2.6. Nivel de computación de ROS

La computación [14] en ROS se realiza en procesos llamados **nodos** con una red punto a punto, pero además se basa en otros elementos (Figura 5):

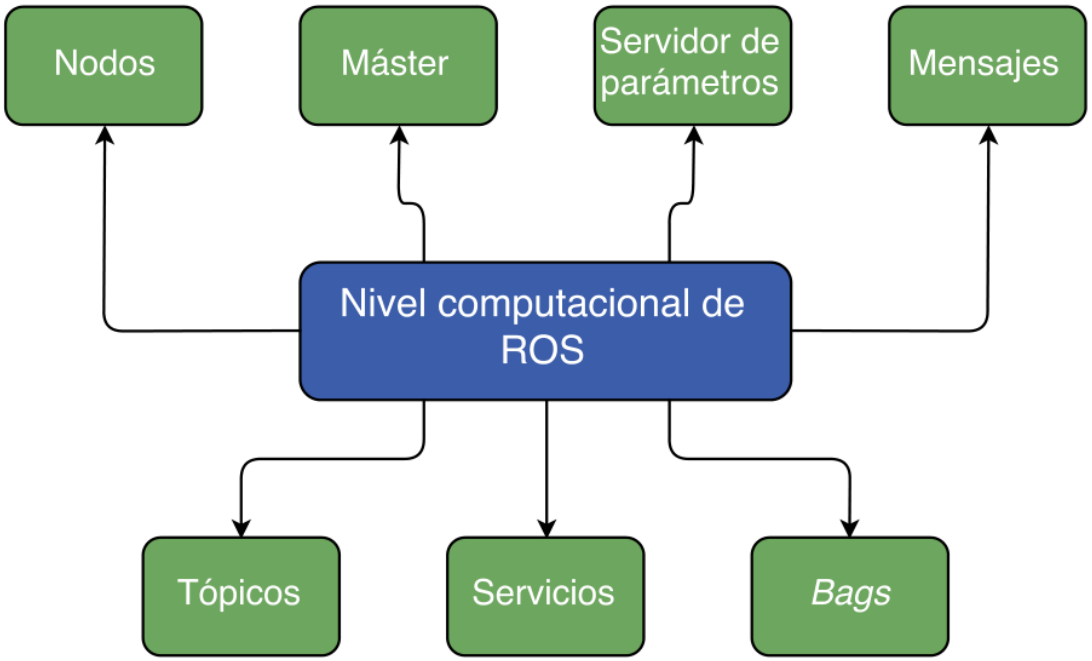

Figura 5: gráfico del nivel computacional de ROS

Nodos: Son los procesos que realizan la computación. Puede estar escrito en C++ o Python, utilizando librerías de ROS como *roscpp* y *rospy*, que se comunican mediante tópicos o servicios independientemente del lenguaje en el que estén programados. Esto permite atomizar los procesos complejos que ejecuta un robot en pequeñas tareas y poder reutilizar el código, con compatibilidad prácticamente total. Se puede gestionar y depurar mediante el comando *rosnode*. Para ejecutar un nodo, se escribe la siguiente línea en la consola de comandos:

```
$ rosrun <nombre_del_paquete> <nombre_del_ejecutable>
```

Si en su lugar se deben ejecutar varios nodos y parametrizarlos, se utilizan los archivos *".launch"* que contiene un paquete, que además lanzan el ROS Máster si no está presente.

```
$ roslaunch <nombre_del_paquete> <nombre_del_archivo.launch>
```

Máster: El ROS Máster también llamado *roscore* como se ha mencionado anteriormente, gestiona la comunicación entre nodos, por lo que es obligatoria la existencia de uno para un correcto funcionamiento. Aunque la ejecución de nodos se haga en ordenadores interconectados diferentes, sólo debe existir un ROS Máster y

debe estar indicado en todos los ordenadores, aunque existe la posibilidad mediante integraciones externas de tener varios [15]. Se ejecuta mediante el siguiente comando, adoptando normalmente el puerto *11311*.

```
$ roscore
```

Servidor de parámetros: permite guardar información en una localización central que pueden utilizar todos los nodos. Un ejemplo es la descripción URDF de cada uno de los robots. Es parte del ROS Máster.

Mensajes: los nodos se comunican entre sí utilizando mensajes de un tipo prefijado, como por ejemplo un vector o una cadena de caracteres. Los tipos más básicos están incluidos, aunque se han creado muchos otros para tareas específicas en la robótica, por lo que se recomienda utilizarlos en lugar de crear nuevos tipos.

Tópicos: Cada mensaje se transporta usando buses llamados tópicos. Cuando un nodo publica un mensaje, se dice que un nodo publica un tópico; mientras que cuando lo recibe de otro, se convencionaliza que el nodo se subscribe a dicho tópico (aunque ningún nodo publique a través de él). Cada tópico tiene un solo tipo de mensaje, pero es obligatorio que el mensaje sea único. El comando [16] para gestionarlos es el siguiente:

```
$ rostopic <comando> <nombre_del_topico>
```

Servicios: ya se han mencionado anteriormente, a diferencia de los tópicos, la comunicación es bidireccional.

Bags o “bolsas”: un formato de guardar y reproducir datos de ROS, útiles para guardar datos de sensores. Al reproducir, para un nodo será exactamente igual recibir de un sensor auténtico o simulado que una reproducción.

```
$ rosbag record <tópico_1> <tópico_2>
```

```
$ rosbag play <nombre_del_bag>
```

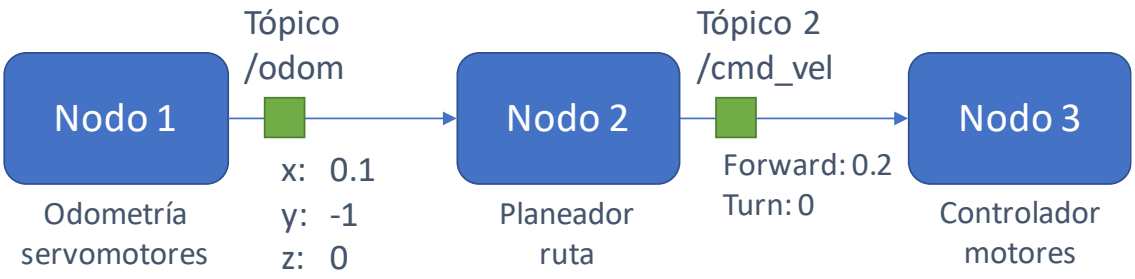

Figura 6: ejemplo de nodos *publisher/subscriber*

En la Figura 6 superior, se pueden observar 3 nodos, que publican y/o suscriben a diferentes tópicos, que incluyen los mensajes señalados en un momento concreto. El nodo 1 es un sensor del encóder, mientras que el nodo 3 es para un actuador (servomotor). El nodo 1 publica la odometría estimada respecto las lecturas de los encóders del servomotor mediante el tópico */odom*. El nodo 2 se suscribe a dicho tópico, procesa la información del mensaje y traza una ruta para la navegación autónoma; al mismo tiempo publica como salida los mensajes de velocidad que debe realizar el actuador mediante el tópico */cmd_vel*. El nodo 3 está suscrito a ese tópico, realiza las cinemáticas inversas y acciona los servomotores para alcanzar dicha velocidad.

3. Base autónoma móvil PMB2

Se trata de un robot construido por la empresa Pal-robotics, con el fin de ser una base móvil autónoma (véase Figura 7). Su objetivo principal en este proyecto es ofrecer movilidad a la plataforma, con lo que permite desplazarse al robot Meka, lo cual incluye la **detección** y **trazado** de obstáculos y trazar rutas para esquivarlo.

Figura 7: fotografía del robot PMB2

En los siguientes subapartados se explica todo el trabajo realizado con éste robot. Primero se describe el estado previo al trabajo, tanto el hardware como el software incluídos, posteriormente su conexión en la primera toma de contacto, implementación de navegación autónoma parametrizable desde otro ordenador con la explicación y uso de los algoritmos necesarios para lograr un correcto funcionamiento. Por último, se explicarán los problemas típicos tanto en la conexión como el uso y parametrización.

3.1. Descripción del hardware existente

El cuerpo tiene forma de cilindro de 54cm de diámetro y 30cm de altura, con una capacidad de carga de 50kg, por lo que podrá transportar fácilmente el robot Meka, su ordenador y baterías asociadas. Dado que el robot sólo tiene 2 ruedas motrices, dispone de otras 4 ruedas pivotantes sobre las que descansa el peso de toda la base. Debida a esta disposición se considera que es un robot diferencial de ruedas, y no de tipo holonómico, muy a tener en cuenta para el desarrollo de la navegación. Además, las cinemáticas inversas para traducir comandos de velocidad lineales o rotacionales se incluyen en el **Anexo II**: Cinemática de un robot diferencial.

En un robot diferencial su movimiento se basa en dos ruedas motrices independientes y a cada lado del robot, como el robot en la Figura 8, para estabilizarlo, dispone de ruedas *caster* o pivotantes que no son motrices. Se trata de un robot no-holonómico.

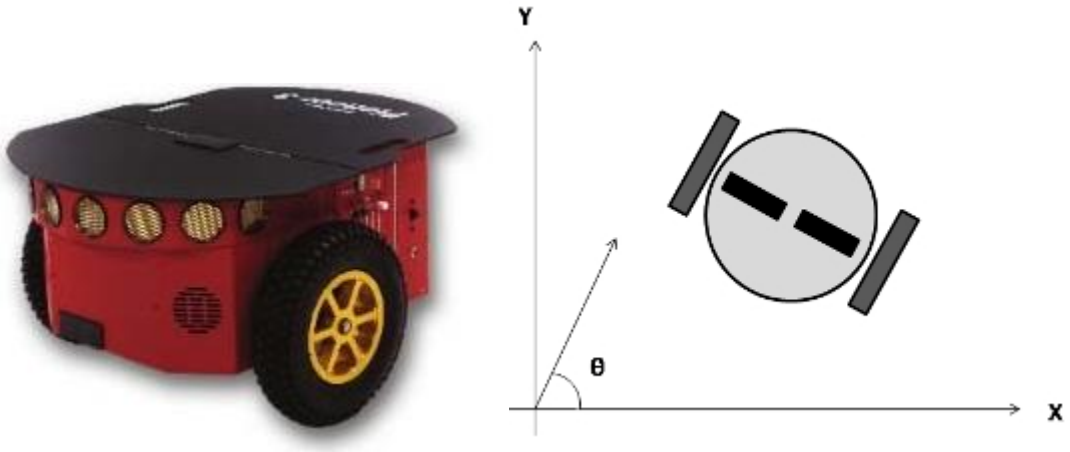

Figura 8: Pioneer 3DX. Posición del robot en x, y, θ

Por otro lado, un robot holonómico [17] tiene tantos grados de libertad controlables como grados de libertad. Por tanto, se puede mover en cualquier dirección desde cualquier orientación, como se observa en la Figura 9. Un coche tiene 3 grados de libertad, pero sólo puede controlar 2, por lo que no es holonómico y requiere de maniobras extra para controlarlo. Aunque los robots holonómicos requieren menos maniobras para su movimiento por el entorno, como inconveniente, son más ineficientes y la odometría es más imprecisa ya que depende de la rugosidad de la superficie en la que se desplaza.

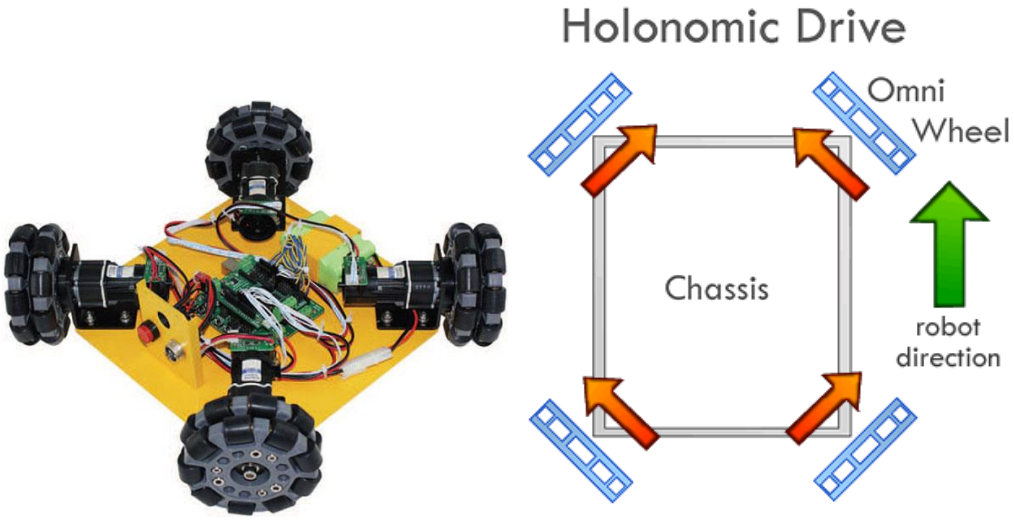

Figura 9: robot holonómico y su movimiento

Los sensores a destacar son:

- **LIDAR (Light Detection and Ranging)**, dispositivo láser que permite determinar la distancia entre el emisor y un objeto o superficie no reflectante o transparente. Aunque existen varias opciones, el que incluye es el Hokuyo URG-04LX-UG1 con un rango de detección de 20 a 5600mm, alimentado a 5V, por lo que no afecta a la toma de alimentación de 12V. Escanea en un área de 240º con una resolución de 0.36º. Está indicado para medir distancias en un plano en 2D, con lo que facilitará el mapeado y la posterior navegación, además de detección de obstáculos.

Figura 10: situación del LIDAR (izquierda) y fotografía sin máscara (derecha)

- **IMU (Inertial Measurement Unit)** o unidad de medición incercial, dispositivo que mide orientación mediante brújula y aceleración en 6 grados de libertad.
- **Servomotores**: en cada una de las ruedas motrices dispone de un servomotor, que serán los que moverán la base además de informar del ángulo de giro en el que se encuentran. Sirven de gran utilidad para proporcionar odometría sobre la que se sustenta la navegación. Se puede observar en la rueda visible en la Figura 7 que muestra el PMB2.

Además, incluye elementos para facilitar la conectividad (WiFi, Bluetooth y Ethernet), así como un servidor DHCP interno, con lo que puede desplegar una red WiFi sobre la que se sustentará toda la plataforma. Incluye dos puertos Ethernet Gigabit, uno para el servidor interno y otro para la conexión con equipos externos o Internet. Como se diseñó para ser una base dedicada a la investigación cuenta con una gran flexibilidad, incluye 4 entradas/salidas digitales a 5V, batería de 8 horas de funcionamiento así como salidas a diferente voltaje, que se utilizarán para alimentar otros sistemas.

Especialmente, la salida de 12V se contempla para alimentar la cámara Kinect colocada encima del Meka, perteneciente al proyecto de fusión sensorial [1]. También podría utilizarse para alimentar el ordenador del Meka, aunque se debe tener en cuenta el límite de 5A máximos; la cámara Kinect consume 1A según las especificaciones, por lo que quedan 4A máximos disponiles para el ordenador, que debe comprobarse por la seguridad eléctrica del PMB2.

3.2. Descripción del software incluido

La empresa fabricante de la base móvil incluye software, tanto el preinstalado en el ordenador interno del robot como uno para instalar en otro ordenador de desarrollo.

El software incluido es el siguiente:

- Linux de kernel propio basado en Ubuntu 12.04.1 LTS “precise” de 32 bits que **no** permite un dual boot con Windows. El software de instalación del SO (Sistema Operativo) no está correctamente programado y no incluye

herramientas de distribución de particiones del disco duro, por lo que podría destruir el sistema operativo previamente instalado en el ordenador.

- ROS basado en la distribución Hydro, incluyendo algunas herramientas en una distribución propia de PAL llamada Cobalt. También incluye una navegación autónoma funcional, pero poco parametrizable de forma permanente por el usuario.

En el robot se ha intentado hacer uso del software facilitado. En concreto se han utilizado utilidades ya existentes, como nodos que ejecutan y dan información tanto de sensores como de actuadores. Dichos nodos se ejecutan en el arranque del ordenador interno del robot, por lo que en muchos casos se partirá del caso en el que se están ejecutando.

Para el desarrollo de software en otro ordenador para el presente proyecto se sopesó si utilizar la opción incluída por el fabricante pero finalmente se descartó, debida principalmente a que Ubuntu 12.04 es una versión prácticamente obsoleta. Este SO incluye una herramienta que podría facilitar la instalación de paquetes en el robot, en esencia, un programa similar a *Synaptics* para instalar paquetes de forma remota en el robot. Sin embargo, no era funcional, por lo que la instalación de paquetes se debe realizar a través de comandos *ssh*. También tiene incompatibilidades gráficas con cualquier tarjeta gráfica que no sea de la marca **NVIDIA**, con lo que la herramienta de visualización Rviz no es operativa, ofreciendo errores como el de la Figura 11.

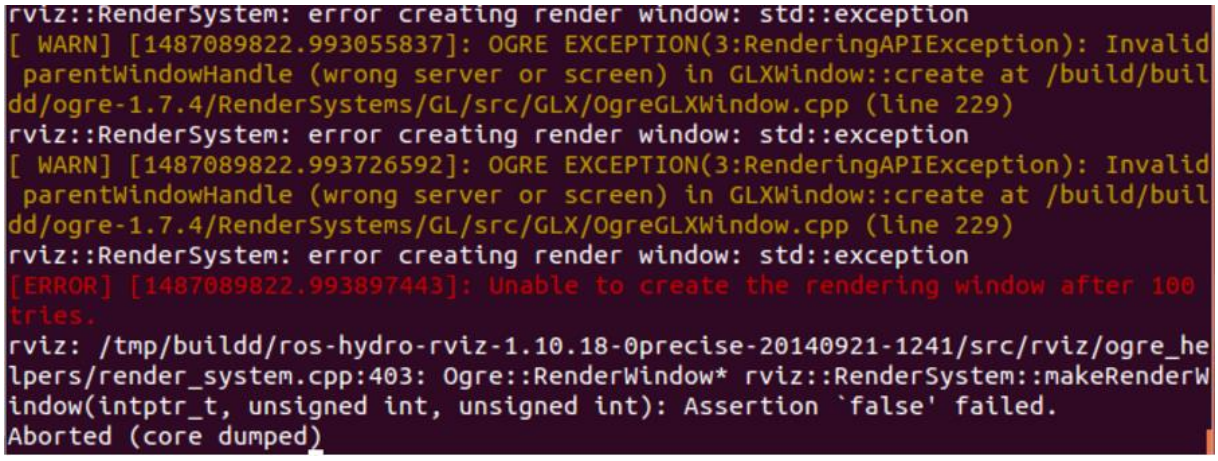

Figura 11: error de funcionamiento de Rviz en PAL-Ubuntu 12.04

Por ello finalmente se eligió:

- Ubuntu 14.04.5 LTS oficial de 64 bits, con lo que los problemas de drivers y soporte quedan resueltos.
- ROS Indigo, la distribución soportada más estable que además tiene gran compatibilidad con ROS Hydro, por lo que se reducen problemas al conectar y programar toda la plataforma, incluyendo fusión sensorial [1].

Dicho SO se instalará en una herramienta de virtualización de sistemas llamada VirtualBox. La justificación de esta decisión y no ejecutar el sistema operativo Ubuntu directamente en el ordenador, que supondría una mejor eficiencia en la computación, es la de una mayor flexibilización para preparar el terminal de operación en cualquier otro ordenador, con la sencillez de tan sólo copiar todos los archivos en la carpeta de VirtualBox en el nuevo sistema: fácil portabilidad. También permite la administración de las redes, las cuales deben estar configuradas en modo “adaptador puente” para que permita una conexión bidireccional con ROS.

Debido a esta decisión, la computación se trasladará a los respectivos ordenadores que controlan directamente los sistemas robóticos, ya que mejorará los lazos de control (al no depender de las redes inalámbricas) y se repartirá de una forma más equitativa. Esto es posible gracias a la descentralización [9] que permite ROS.

3.2.1. Nodos existentes reutilizables

Como se mencionó anteriormente, se intentará aprovechar al máximo las herramientas desarrolladas por el fabricante, ya que facilita en muchos casos la abstracción a un mayor nivel:

Por ejemplo, se envían comandos de velocidad que un nodo convertirá en el accionamiento de los servos.

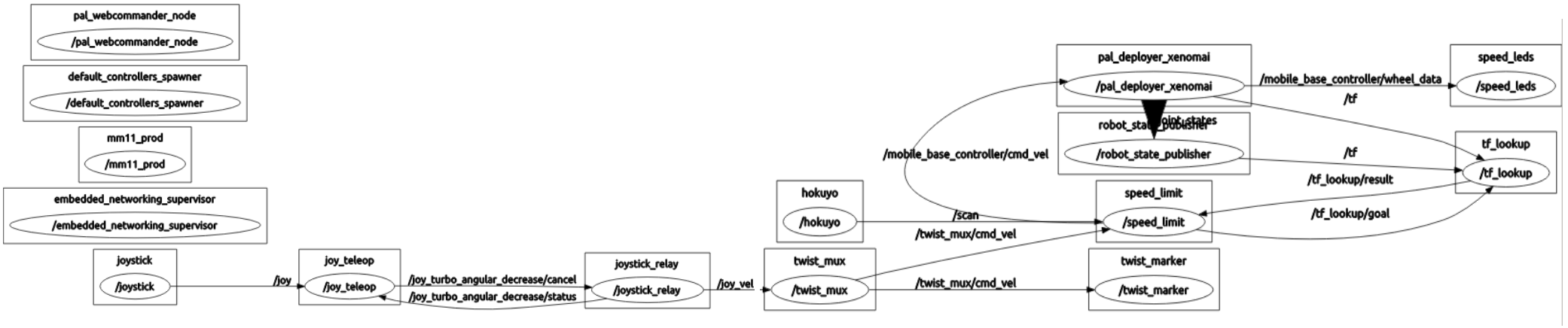

Figura 12: nodos básicos ofrecidos por PAL-Robotics

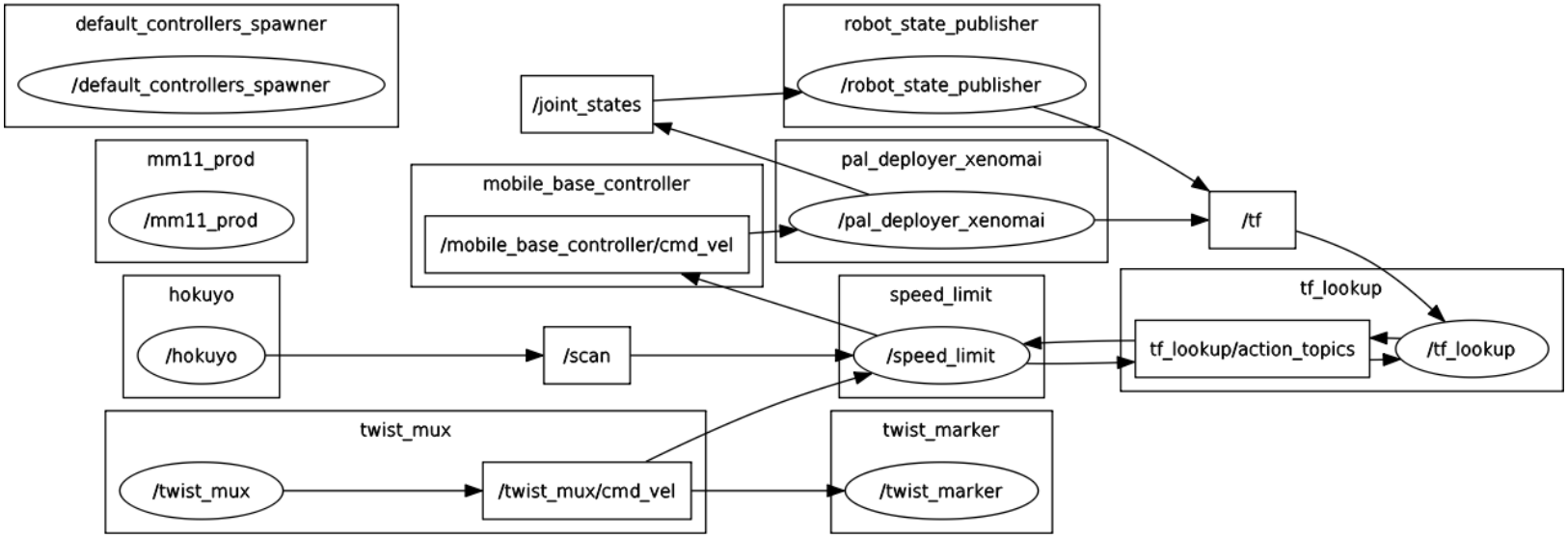

Figura 13: nodos básicos necesarios de PAL

Una lista de nodos ofrecidos por Pal-Robotics a destacar sin tener en cuenta los de la navegación autónoma son los de la Figura 12:

- *Hokuyo*: nodo que gestiona los datos de lectura del LIDAR, dando como tópico */scan*.
- *Mm11_prod*: nodo que gestiona otros sensores, como el IMU o las 4 entradas/salidas digitales. Crea un tópico llamado /gpio (*General Purpose Input/Output*) que incluye un vector de números (0 o 1) que no se puede observar en ninguna de las figuras superiores debido a que ningún otro nodo está suscrito.
- *Joystick, joy_teleop* y *joystick_relay*: manejan el uso y activación del mando joystick utilizado para controlar movimientos del robot a radio-control.
- *Twist_mux* y *speed_limit*: gestionan comandos de velocidad para mover el robot teniendo en cuenta los obstáculos, explicado en detalle en la sección 3.4.1.
- *Pal_deployer_xenomai*: nodo muy importante, ya que gestiona toda la acción e información de los servomotores: recibe comandos de velocidad y actúa en los susodichos (cada uno independientemente), además publica los datos de posición en el tópico */joint_states*, para dar información de giro en el modelo vectorial TF. También publica la odometría, aunque no visible en la figura porque ningún nodo se ha suscrito el tópico */odom*. Está suscrito al nodo */mobile_base_controller/cmd_vel*, que es el tópico que contiene mensajes de velocidad ya filtrados por los nodos anteriormente explicados
- *Robot_state_publisher*: nodo de propósito general de ROS, carga el modelo URDF de cualquier robot y junto al tópico */joint_states* permite conocer la forma y posición de todos los servos para publicarla en TF, además de la forma que tienen. Muy útil para su representación espacial, que se puede visualizar en Rviz como en la Figura 14.

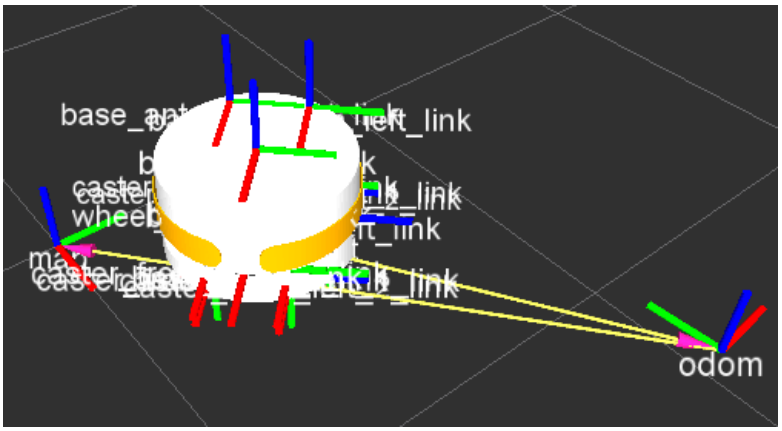

Figura 14: PMB2 con TF visibles usando Rviz

Todos los nodos de los sensores, arranque de TF y actuadores se inician con el paquete ***"pmb2_bringup".*** Los nodos restantes son de diagnóstico o lanzamiento, aunque no serán compatibles con la navegación implementada y algunos como */pal_web_commander*, que se encarga de publicar la página web para gestionar conexiones. En la Figura 13, se puede observar un gráfico detallado de los nodos y tópicos más importantes.

3.3. Conexión con el ordenador integrado

El acceso al ordenador interno podría parecer trivial, pero no tiene acceso al puerto de vídeo de forma sencilla, ya que se necesita desmontar una carcasa deteniendo el funcionamiento del LIDAR. Para ello, la primera vez se conecta directamente por cable Ethernet el ordenador de desarrollo con el pin 12 (Figura 15) si no se ha configurado adecuadamente el punto WiFi.

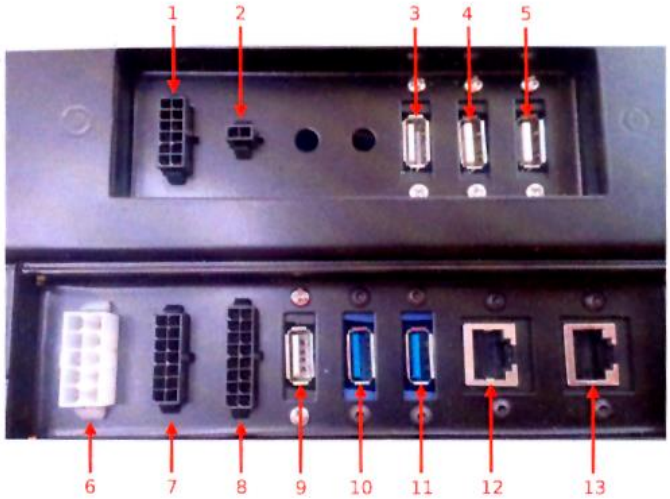

Figura 15: fotografía numerada de los puertos del PMB2

Luego, en la máquina virtual de VirtualBox, activando la conexión puente con la tarjeta de red, al conectarse correctamente debería mostrarse en la ventana Red algo similar a la Figura 16:

Figura 16: ventana de Redes tras conectar el PMB2 a la máquina virtual

La IP del apartado DNS es la del robot. Si se conecta por el servidor interno como en este caso, normalmente será 10.68.0.1. En el apartado “dirección IP” se encuentra la IP del propio dispositivo en la red, necesario conocer para realizar la conexión bidireccional en ROS. Ya que se ha seleccionado como centro conectivo de la plataforma, se utilizará con gran frecuencia en el presente trabajo. El motivo de dicha decisión se describe con detalle en el capítulo 4.1, ya que el ordenador Meka no permite una conexión inalámbrica e irá montado encima del PMB2, por lo que se conectarán mediante un cable Ethenet por el pin 12 (Figura 15).

Se sustituirá por un nombre más apropiado en todos los sistemas:

```
$  sudo gedit /etc/hosts
```

Y se añade la siguiente línea sin comillas:

“10.68.0.1 pmb2-5c”

El software que incluye el robot permite realizar un ajuste de las conexiones mediante una útil herramienta llamada *WebCommander*, se trata de una página web creada por el PMB2. Una vez conectado, se puede conectar mediante cualquier navegador web con la siguiente dirección:

```
http://pmb2-5c:8080
```

Se ajusta de la siguiente manera para que actúe como punto WiFi para el proyecto, pero también se puede ajustar para que se conecte a una red WiFi externa (Figura 17)

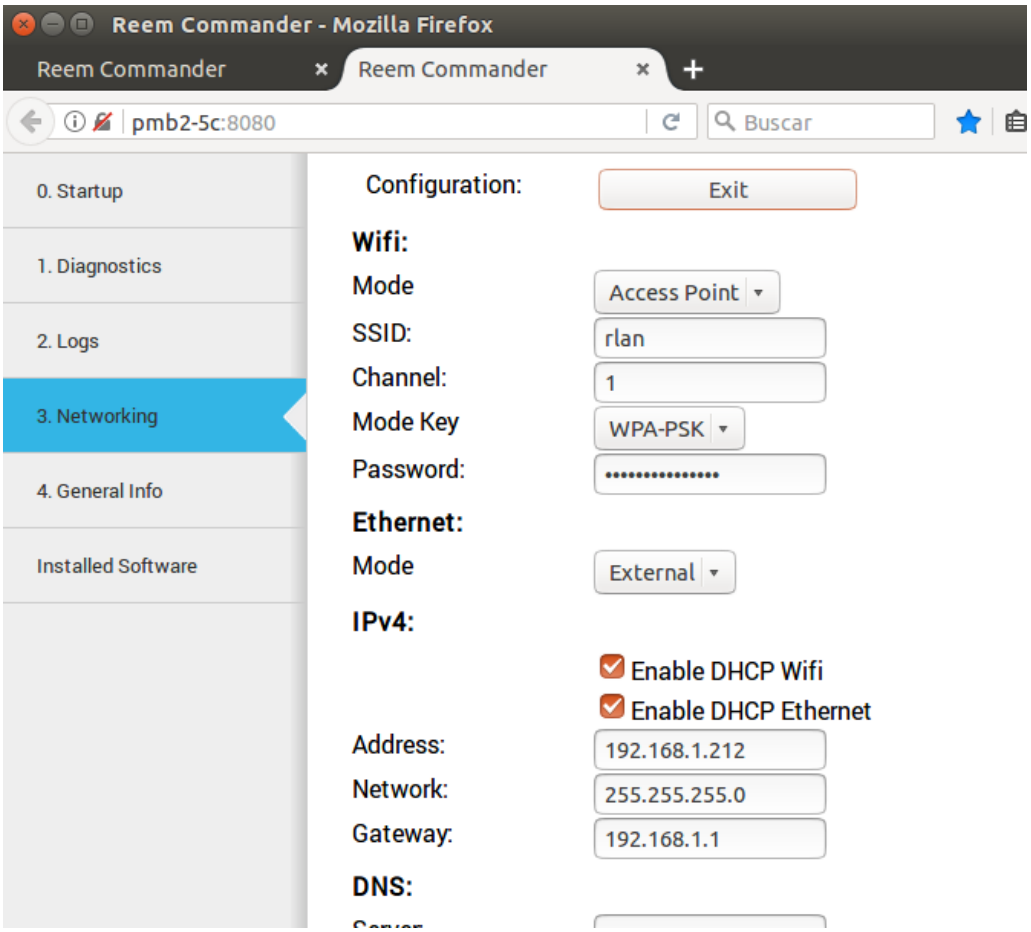

Figura 17: WebCommander de PAL-Robotics, sección de conexiones

Si se desea acceder a la línea de comandos del robot de forma remota, ya que carece de interfaz gráfica, se utilizará el comando *ssh*, donde *<user>* será *root* o *pal* y después pedirá la contraseña:

```
$ ssh <user>@pmb2-5c
```

Para que la conexión con ROS sea completa y bidireccional, teniendo en cuenta que el máster se coloca en el PMB2, en cualquier otro ordenador se debe escribir:

```
$ export ROS_MASTER_URI=http://pmb2-5c:11311
$ export ROS_IP=<IP_del_ordenador>
```

3.4. Navegación autónoma

El objetivo principal de este robot es aportar movilidad a la base, con lo que debe realizar una navegación autónoma. Ésta consiste principalmente en ir a un punto y

esquivar en la medida de lo posible obstáculos, sean fijos o móviles, sin colisionar en ningún modo con ellos. Las tareas involucradas en la navegación son la percepción, localización, planificación de trayectoria y control de movimiento, así como su respuesta ante agentes externos no inesperados, es decir, obstáculos fijos o móviles no previstos en la planificación inicial. Una vez conocido el entorno mediante el mapeado, la localización es esencial tanto para la planificación de una trayectoria libre de obstáculos como para realizar el seguimiento de dicha trayectoria mediante un controlador de movimiento.

En el software incluido existe una navegación autónoma monitorizable, pero su funcionamiento no es fácil de desgranar ni modificar ya que se utilizan muchos paquetes propios de PAL-Robotics en lugar de usar los comunes de ROS. Tampoco permite explorar su funcionamiento sin mapa.

Con este fin, se ha creado un paquete de navegación 2D en el presente proyecto para configurar la navegacion autónoma de propósito general de ROS llamado ***pmb2_grav_2dnav***, que incluirá los archivos de configuración y lanzamiento, además de algunos scripts para enviar metas por programa en lugar de hacerlo mediante la herramienta de visualización personalizable RVIZ.

3.4.1. Requisitos de la navegación

La Figura 18 muestra un esquema del funcionamiento de la navegación, cuyas entradas de información son:

- Odometría: información de la posición ofrecida por los servomotores, aunque no es totalmente precisa y se debe corregir, ya que tendría un error acumulativo [18].
- Información de sensores: bien nube de puntos a partir de una cámara o el escáner de un láser como el Hokuyo incluido.
- Mapa y AMCL: son opcionales en el caso de utilizar una navegación con la ayuda de un mapa previamente creado, se describirán con profundidad más adelante.
- Transformadas TF: a partir de sensores y el modelo vectorizado del robot en el formato URDF, esencial para un funcionamiento correcto en ROS, también se describirá más adelante.

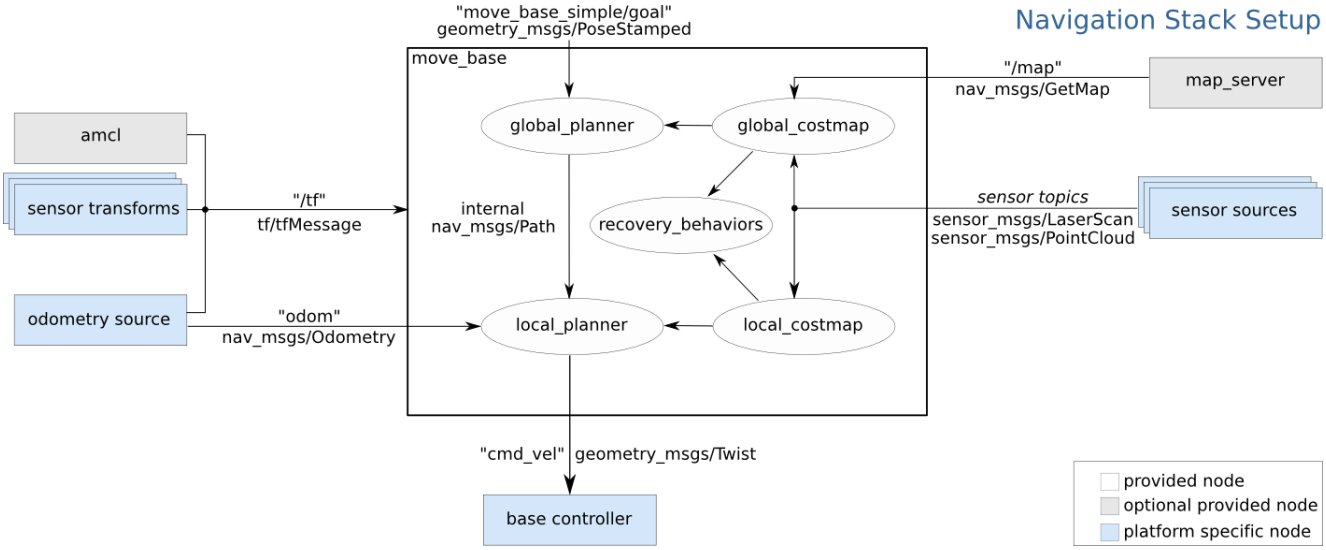

Figura 18: esquema de funcionamiento de la navegación autónoma

La única salida de información es la de un tópico con comandos de velocidad de nombre "*/nav_vel*" (en la Figura 18 [2] se llama *"cmd_vel"*, pero se renombra así para el nodo *Twist_mux*), que se manda a un nodo capaz de transformar esa información en movimiento de los servomotores de tracción.

En este caso se envía al nodo creado por PAL llamado Twist_mux, que actúa como un multiplexor de todos los comandos de velocidad (ya sea por teclado, joystick o navegación) y los filtra priorizando o desactivándolos. De salida da el tópico filtrado *"/twist_mux/cmd_vel"*, que pasa como entrada al nodo *"/speed_limit"*. A su vez, *"/speed_limit"* utiliza también como entrada el propio láser para evitar colisiones frontales (independientemente de la navegación), dando de salida el tópico *"mobile_base_controller/cmd_vel"*, que finalmente pasa al nodo da órdenes a los actuadores e informa del estado de los servos llamado *"pal_deployer_xenomai"*. Se puede observar todos estos tópicos en la Figura 13.

Debido a la importancia que cobra la seguridad ante las posibles colisiones con el entorno o con humanos, estas funcionalidades será reutilizadas.

3.4.2. Conceptos de la navegación

El diagrama de la Figura 18, lo que está dentro del recuadro principal consiste en todos los componentes requeridos que están incluidos por el paquete de navegación para implementarse. Los recuadros azules son componentes obligatorios

para su funcionamiento, que varían para cada robot, ya que depende de los sensores y actuadores: todos ellos están implementados por PAL-Robotics con el paquete "***pmb2_bringup***", y se lanzan automáticamente en el arranque del robot.

Por otro lado, los componentes en gris son opcionales, ya que son necesarios para una navegación con mapa, pero no obligatorios para el funcionamiento.

Los programas y algoritmos utilizados se incluyen en uno de los paquetes más completos y de uso en multitud de robots llamado ***"navigation stack"***. Aunque se utiliza en multitud de robots, deben satisfacer los siguientes requisitos:

- Sólo maneja robots diferenciales o de ruedas holonómicas.
- La forma debe ser un cuadrado o un rectángulo, aunque la forma circular también está soportada.
- Publica información de las relaciones entre la posición de todas las uniones y la posición de los sensores. En ROS, se solventa utilizando las transformadas TF.
- El robot debe aceptar comandos de velocidad linear y angular.
- Un sensor láser o equivalente debe estar presente para el mapeo y la localización

3.4.3. Coordenadas vectoriales TF

Un gran problema en la robótica que además afecta a casi cada tarea posible es la de la localización espacial de todos los elementos en acción. Para ello, se debe manejar adecuadamente las coordenadas 3D en el tiempo, con lo que en ROS se incluye el paquete *"TF"*, básico para cualquier funcionamiento [19]. Por lo tanto, no sólo gestiona la dimensión espacial si no también la temporal. Al igual que ROS, puede operar en un sistema distribuido, en diferentes unidades. Es por ello por lo que la sincronización entre los relojes de todos los sistemas es crucial para evitar problemas de coordinación entre los sistemas.

Es una herramienta muy potente, ya que los *frames* o también llamados objetos de las transformadas se pueden referenciar una tras otra, siempre que conserven una jerarquía de árbol: un "padre" puede tener varios "hijos", pero un "hijo" sólo puede tener un "padre". Así se unifican todas las coordenadas y se referencian a un punto

en común. En el caso de la plataforma del presente proyecto, el padre de todos los objetos será el mapa, que será una representación acorde al entorno real, como se observa en la Figura 19. Dichos gráficos se realizan mediante una herramienta de **rqt** llamada ***rqt_tf_tree***.

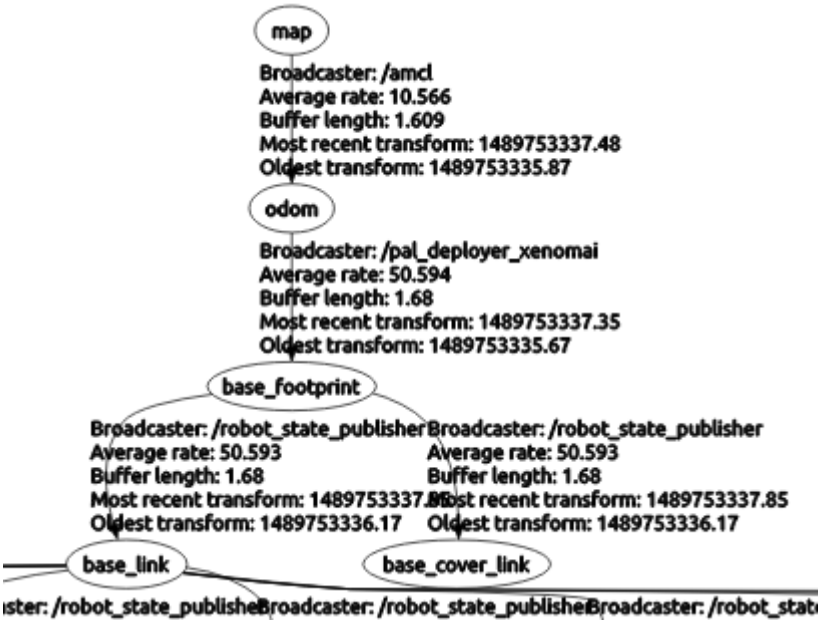

Figura 19: ejemplo parcial de árbol TF en la navegación autónoma del PMB2

Por ejemplo, si no se supiera adecuadamente dónde se sitúa al láser respecto del robot, el mapa sería impreciso, ya que toma en cuenta la posición respecto a las ruedas motrices. Tampoco podría ejercer correctamente la navegación, ni el movimiento de un brazo robótico para coger un objeto o la situación de un humano para navegar hasta él. También se tiene en cuenta la posición dinámica para conocer la trayectoria que siguen los objetos, su velocidad.

Aunque se ha mencionado que es un sistema de referencia en 3D, realmente no sólo se tiene en cuenta la posición, sino también la orientación del objeto. La suma de la posición y orientación se llama *"pose"*, de vital importancia en la navegación autónoma, ya que el robot deberá orientarse correctamente a su objetivo (un humano, una estación de carga, etc.) y conforma un total de 6D.

Algunas relaciones son estáticas, como los componentes internos, estructura y probablemente sensores de un robot. Las demás relaciones serán dinámicas, como

la mano de un robot respecto a sí mismo. Se especifican mediante el URDF (Unified Robot Description Format), en una descripción XML, o en el nuevo formato llamado XACRO, incluyendo importación de objetos, álgebra y lógica básicas. En el presente trabajo, el URDF del PMB2 y del Meka estarán realizados por los fabricantes de ambos robots, ya que es bastante complejo y se necesita información completa de la estructura del robot; pero se modificarán para añadir los nuevos componentes de la plataforma. Habrá nodos que escuchen estos *frames*, mientras que otros generarán algunos, como es el caso de las cámaras, que generarán posiciones de humanos.

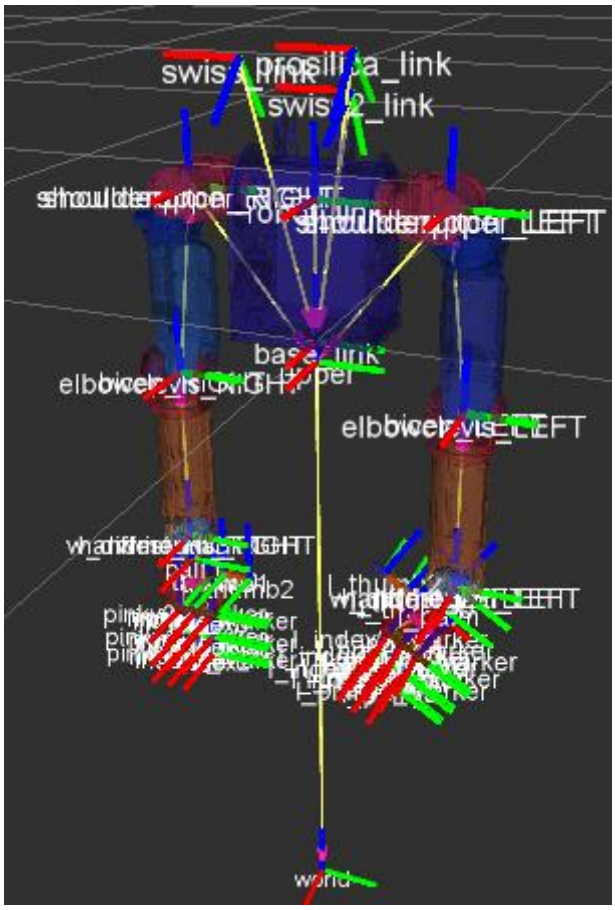

Figura 20: ejemplo del modelo URDF del Meka con los TF visibles

3.4.4. Algoritmo AMCL

AMCL (Adaptive Monte Carlo Localization) es un algoritmo utilizado para la localización [20]. El algortimo AMCL es un sistema probabilístico de localización para un robot cuyo movimiento se ejecuta en 2D y especialmente que utilice un láser como sensor principal, como es en este caso. Usa un filtro de partículas para rastrear la

posición del robot sobre un mapa conocido. Al ser un nodo ROS de propósito general, incluye gran cantidad de parámetros modificables para optimizar la localización, aunque conociendo que el PMB2 es un robot diferencial, el lanzamiento de ejemplo que incluye llamado *"amcl_diff.launch"* funciona correctamente, según se pudo testear.

Su funcionamiento requiere principalmente de cuatro tópicos:

- Scan: tópico al que está suscrito y proporciona datos del láser LIDAR, que compara con los bordes del mapa. También se puede conseguir con cámaras que proporcionen nubes de puntos (es decir, deben medir la profundidad), y hacer uso del paquete "*pointcloud_to_laserscan*", aunque con menos precisión ya que el área escaneada es menor y con menor resolución.
- TF: transformaciones vectoriales que indican la posición de diversos objetos.
- Map: mapa creado por otros nodos, como se comprobará a continuación. Es un tópico que contiene la imagen del mapa.
- Initialpose: es un tópico que señala una posición inicial para reinicializar el filtro de partículas, necesario si se encuentra en una posición muy diferente a la real en el mapa. Se puede referenciar con RVIZ; también se puede incluir como parámetro en el launch, si el robot se inicia en un emplazamiento conocido, como una estación de carga.

Su función en ROS es la de crear una transformada TF entre el mapa y la odometría (un punto al azar que depende de la posición estimada a partir de los servomotores). Aunque no esté perfectamente calibrado en el inicio, el error se corrige según el robot se mueve por el mapa. Por ello, un mapa ligeramente deformado no afecta al funcionamiento de la navegación autónoma, ya que este algoritmo corrige la posición.

Dicha transformada tiene un ligero retraso negativo en el tiempo, es decir, está ligeramente fechada en el futuro. Lo realiza para poder publicar pequeñas correcciones en la transformada y así al nodo AMCL se le podrá permitir actualizaciones a baja frecuencia, por si supone un gran peso computacional y evitar errores al utilizar transformaciones excesivamente antiguas.

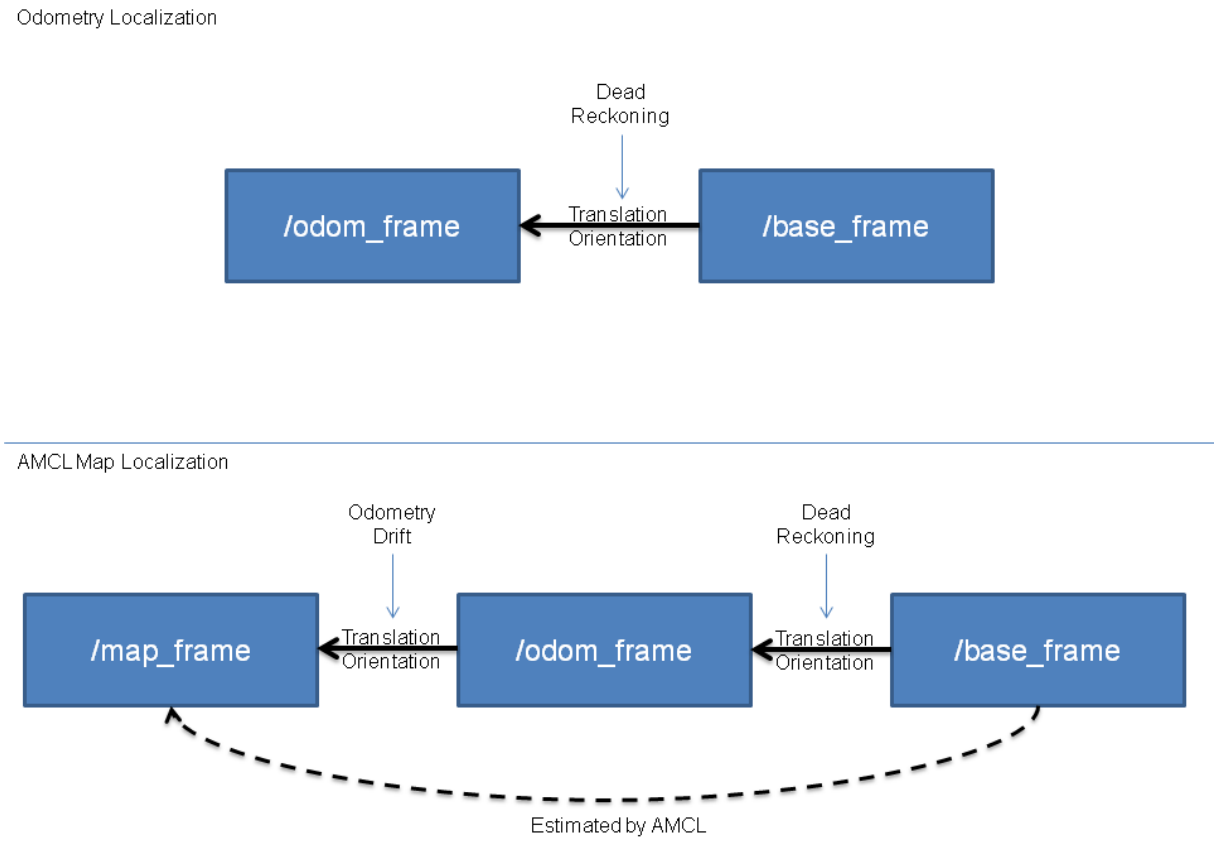

Figura 21: transformada entre el robot y su odometría. Debajo, la misma añadiendo la transformada proporcionada por el AMCL

3.4.5. Mapeando el entorno: SLAM

SLAM [21] (Simultaneous Localization And Mapping) o localización y mapeado simultáneos es "*una técnica usada por robots para construir un mapa de un entorno desconocido en el que se encuentra, a la vez que estima su trayectoria al desplazarse dentro de este entorno*".

Resuelve el problema de comenzar en una posición desconocida en un entorno desconocido e incrementalmente construir un mapa mientras se posiciona en él. La clave de la dificultad se halla en no conocer la posición absoluta ni en un mapa preciso, debido a la imprecisión inherente a cualquier sensor. En este caso no se puede utilizar el algoritmo AMCL, puesto que no hay mapa del que partir para hallar la localización, sino que el algoritmo que se utilice debe proporcionar ambas cosas: localización y mapa.

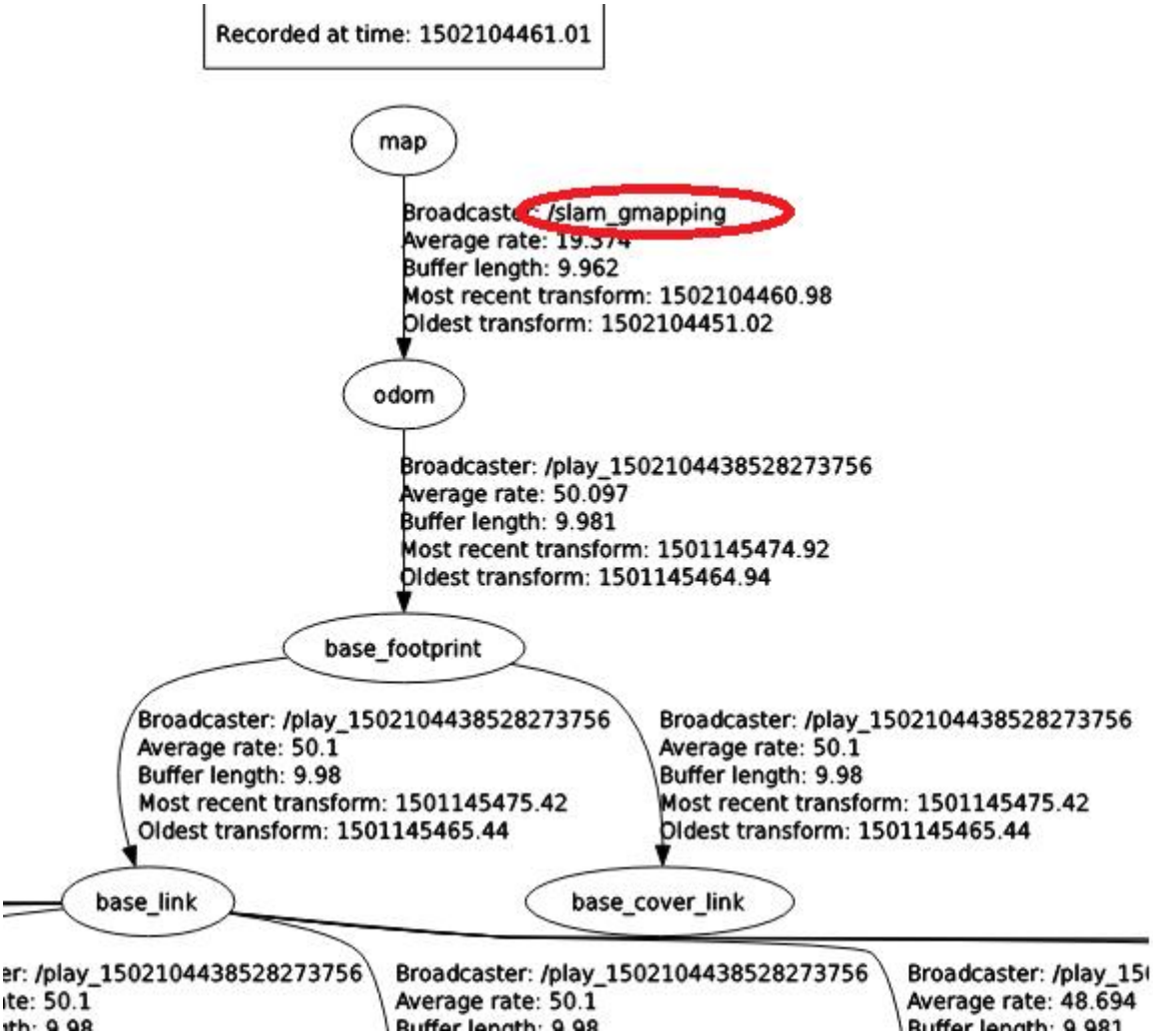

Figura 22: localización proporcionada por SLAM

Históricamente, SLAM fue resuelto hace pocas décadas, ya que se propuso su problema en 1986. El descubrimiento más importante fue que el problema del mapeado y la localización, una vez formulado como un único problema de estimación, era de naturaleza convergente.

Las soluciones más exitosas se basan en la utilización de técnicas probabilísticas, cuya base es la regla de Bayes. Dicha regla establece para dos variables aleatorias x y d la *Ecuación 1*, basándose en probabilidades y grados de confianza.

$$p(x|d) = \frac{p(d|x)p(x)}{p(d)}$$

Ecuación 1

Generalmente, el proceso consta de pasos como extracción de características, asociación de datos, estimación del estado y actualización de las características. También se ha de actualizar la posición del robot, utilizando la odometría pero debida a su imprecisión acumulativa, además debe utilizar sensores de distancia: en el PMB2 es el LIDAR. Algo que tienen en común todas las técnicas de SLAM es la de utilizar puntos de referencia o *landmarks*, además de un filtro que estima la incertidumbre en la posición del robot y dichos puntos.

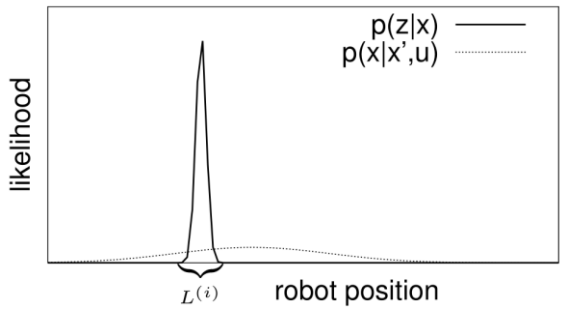

Figura 23: incertidumbre de la posición del robot en SLAM con la regla de Bayes

Los puntos de referencia son los que conformarán el mapa, por lo tanto, obstáculos tales como objetos estáticos o paredes. Debido a este factor, un *landmark* debe permanecer estacionario, por lo que las estimaciones se realizan acorde al tiempo, por si el obstáculo es móvil (como una persona en movimiento) y descartarlo para la construcción del mapa. El problema de la asociación de datos es comparar distintas observaciones de un mismo *landmark*, ya que podrían desaparecer o no situarlos en una posición similar. Por tanto, un buen diseño del algoritmo cobra importancia para la elección de dichos puntos de referencia.

Uno de los métodos más usados es aprender mapas en celdas utilizando mediciones de láser (LIDAR), y es la que se aplica en el presente proyecto. Se utilizan

técnicas adaptativas para reducir el número de partículas en un filtro *Rao-Blackwellized* de partículas para el aprendizaje de mapas por celdillas [22], con lo que se minimiza la incertidumbre de la posición del robot.

Afortunadamente, dicho algoritmo está implementado en un paquete descargable de ROS llamado *"slam_gmapping"*. Se suscribe a */tf* y */scan*, aunque las transformadas necesarias son las de los *frame* pertenecientes al láser, la base y la odometría. Un gráfico de los datos a los que se suscribe y publica se encuentra en la Figura 25, aunque provienen de una bolsa de datos, para el nodo es exactamente igual que recibir los datos directamente del robot.Publica un mapa, que posteriormente se guarda mediante el nodo llamado ***map_server***, un nodo de gran ayuda ya que se encarga tanto de guardarlo en formato PNG como cargarlo posteriormente una vez cumplida la tarea del mapeado. Se guardarán en la carpeta "maps" del paquete *pmb2_grav_2dnav*. Un ejemplo de un laboratorio de la UJA es el de la Figura 24. Aunque aparentemente tiene muchos errores en las paredes, se debe a la presencia de multitud de objetos tras vidrieras trasparentes, que son transparentes para el láser.

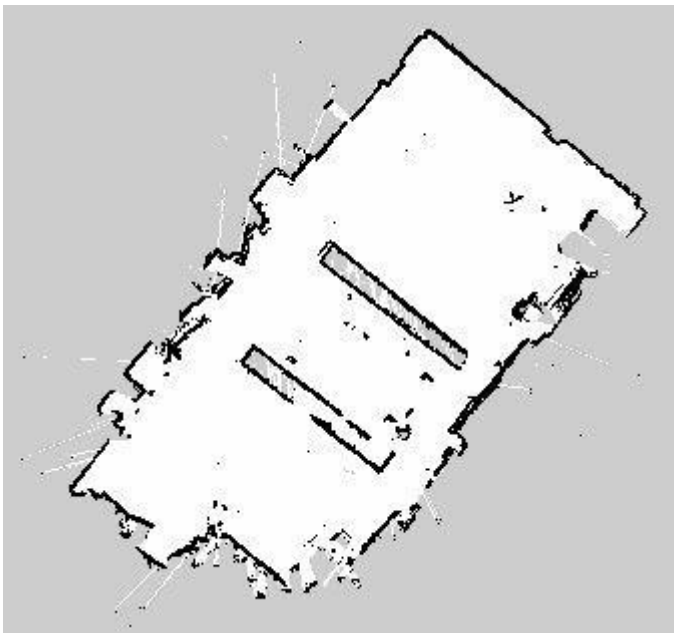

Figura 24: mapa creado gracias a la técnica SLAM

Los parámetros para un mapeo correcto utilizando el paquete anteriormente mencionado se encuentran en la ruta *pmb2_grav_2dnav/config/mapping/gmapping.yaml*, donde se pueden destacar

algunos parámetros como *base_frame*, en el cual se coloca *base_footprint*, que es el *frame* que está en el centro a la altura del suelo. Los demás parámetros son dependientes del láser, ya que junto con la odometría (en TF) es la manera de crear el mapa. Si la capacidad de computación lo permite, el parámetro *transform_publish_period* se coloca un número por debajo de uno, para que emita transformadas al mapa con la mayor frecuencia posible. Se ejecuta mediante el launch llamado *"slam_en_pmb2.launch"*.

Una forma para poder depurar los parámetros utilizados es usar la herramiento *rosbag*, que consiste en guardar tópicos elegidos durante un tiempo específico. Para el caso de SLAM, se realizó así para evitar repetir la tarea de mapeado físicamente y es exactamente igual a suscribirse a los tópicos publicados directamente del PMB2

Para ello, se ejecutan los siguientes comandos antes de proceder a mover el robot a lo largo del entorno a mapear:

```
$ roscd pmb2_grav_2dnav/bags
$ rosbag record -O mapping_03.bag scan tf
```

Después, se puede crear el mapa en cualquier otro equipo, no necesariamente en el PMB2:

```
$ roscore
$ roscd pmb2_grav_2dnav/bags
$ rosbag play --clock mapping_03.bag
$ roslaunch pmb2_grav_2dnav slam_en_pmb2.launch
```

Finalmente, se puede guardar el mapa con la función anterimente mencionada:

```
$ rosrun map_server map_saver
```

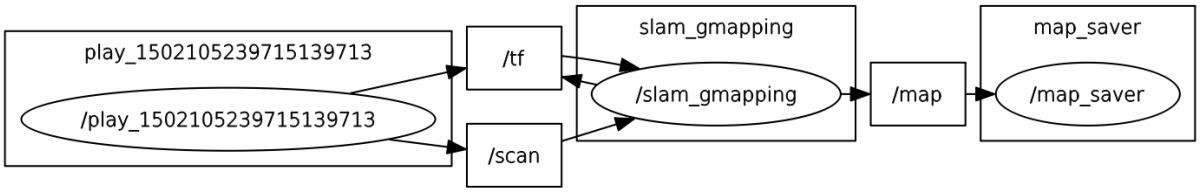

Figura 25: SLAM a partir de una grabación de datos previa

Actualmente, en el proceso de mapeado se requiere la intervención humana para dirigir el robot a lo largo de la zona a explorar. La navegación autónoma puede estar

activa en este proceso y funcionar si se activa el parámetro de navegar más allá de los límites del mapa, pero requiere de alguna funcionalidad extra que ejecute la tarea de explorar la zona, ofreciendo al robot puntos posibles en los que navegar de forma automática. Existe un paquete llamado *"frontier_exploration"* [23], que realiza exactamente lo comentado, pero no funciona correctamente en áreas con multitud de obstáculos y no ha sido incluído en el presente proyecto.

En el caso de tener más de un robot simultáneamente mapeando, se debe utilizar el paquete de ROS llamado *"map_merger"*. En este trabajo sólo el PMB2 se encarga de mapear, por lo que no será necesario.

Existe la posibilidad de crear varios mapas pequeños mapas y pasar de uno a otro, pero por motivos de exactitud y trazado de planes a larga distancia es preferible hacer un mapa de toda la extensión a recorrer. Si es necesario, el mapa se puede editar manualmente con un editor de imágenes como GIMP, con lo que se pueden crear obstáculos virtuales para impedir la entrada del robot a un área específica.

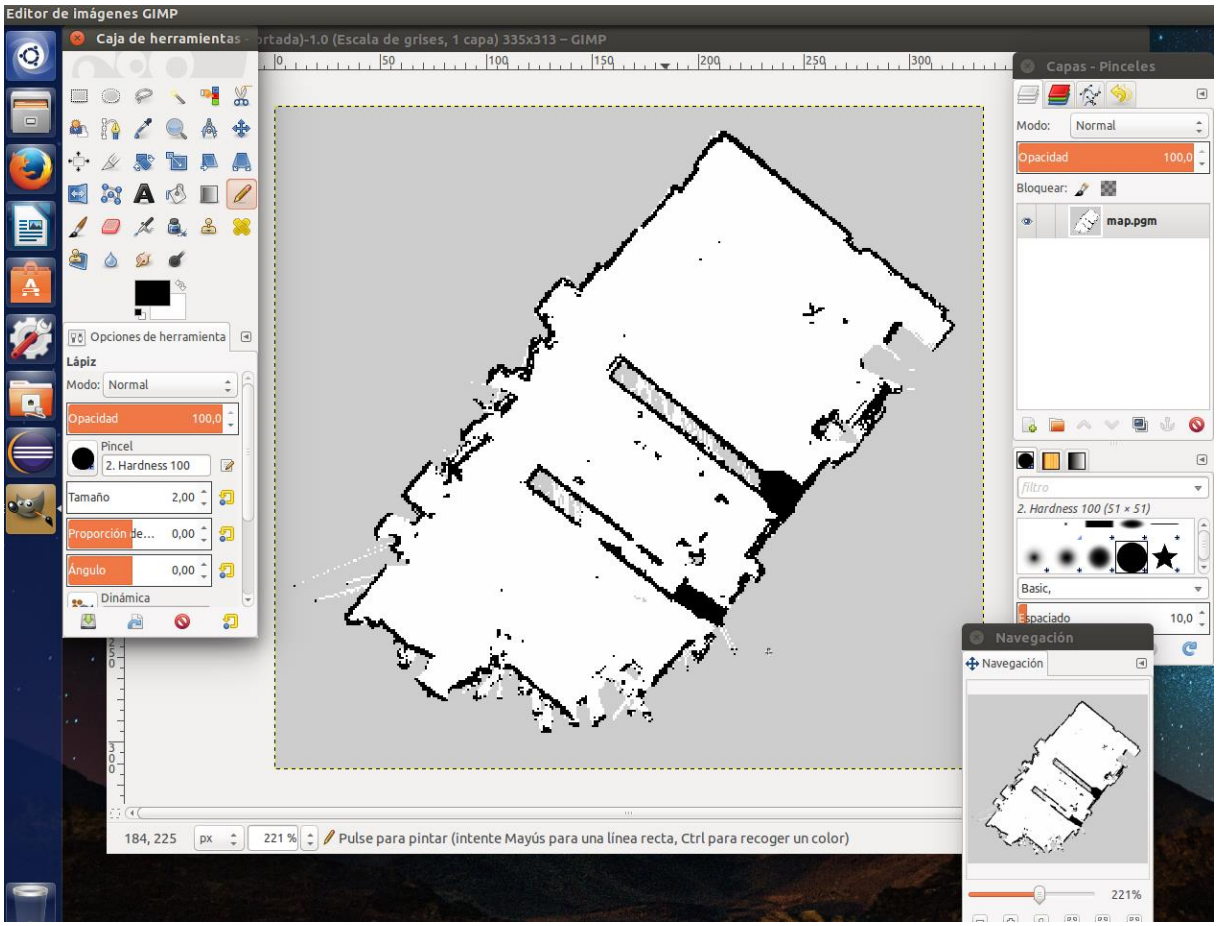

Figura 26: mapa editado con editor de imágenes GIMP

Por ejemplo, en la Figura 26, se ha editado el mapa de la Figura 24 y se han añadido obstáculos en el pasillo derecho, que es excesivamente estrecho. Así, el planeador de rutas lo toma como un obstáculo y el robot nunca pasará por dicha área.

Con esta opción, se podrían actualizar obstáculos conocidos, tales como una puerta automática, aunque requeriría guardar constantemente la posición dada por el algoritmo AMCL. Advertencia: un editado excesivo puede afectar al correcto funcionamiento de la localización, ya que podría deformar excesivamente el mapa respecto al original y no tomar la referencias correctamente.

3.4.6. Navegación y trazado de rutas

La principal razón para incorporar la acción del mapeo del entorno como una fase previa es la de tener un punto de referencia global para toda la plataforma, en especial el posicionamiento de la cámara TOF, tal y como se indicó en el apartado 3.4.3. Por otro lado, aunque la navegación autónoma no requiere específicamente de un mapa para funcionar como se indica en la Figura 18, el trazado de rutas mejora enormemente, ya que podrá trazar rutas más allá del alcance de los sensores integrados, con lo que podrá esquivar eficientemente obstáculos fijos y conocer la distribución de la estancia.

Una vez conocidos y los requisitos para la navegación, se explicará el nodo central sobre el que se basa la navegación, llamado *“move_base”*. Su función principal es mover el robot de la posición actual a una posición final o meta con la ayuda de otros nodos de navegación. El nodo *move_base* enlace con planeadores globales y locales para el planeo de rutas, conectado al mapa de coste global y local para obtener el mapa. Además, se utiliza un paquete de recuperación rotativa en caso de atascamiento. El esquema más básico posible de su funcionamiento a nivel de requerimientos es el de la Figura 27.

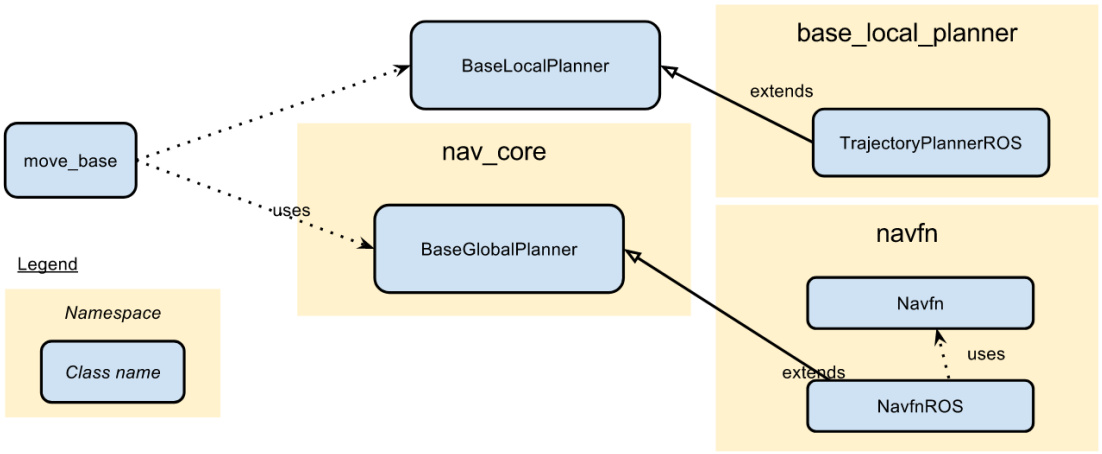

Figura 27: esquema de *move_base*

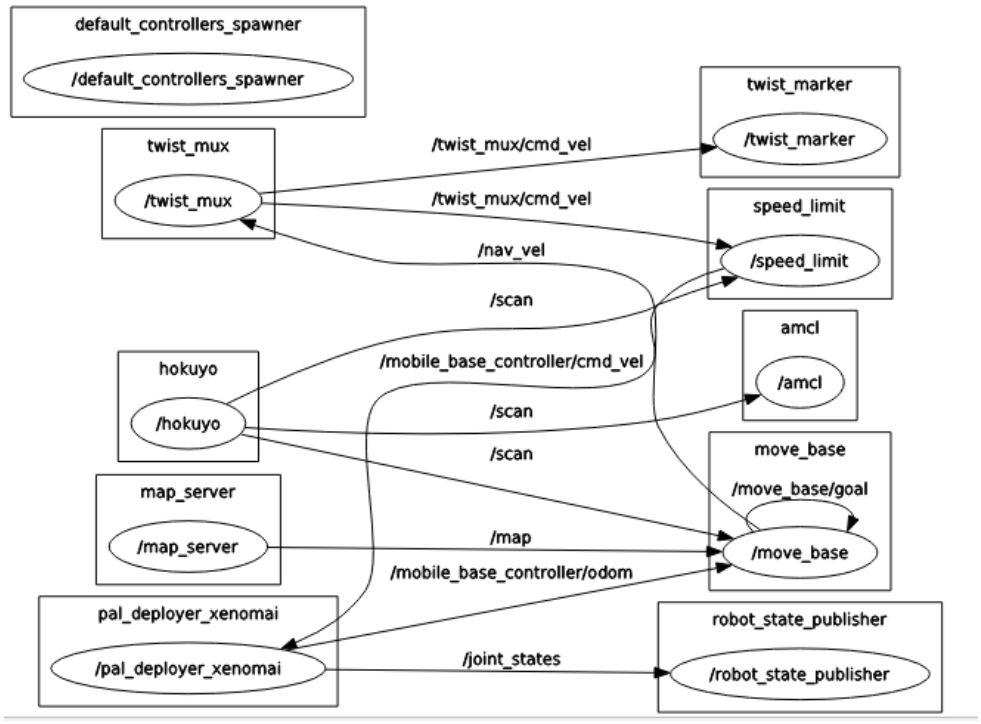

Figura 28: nodos necesarios en la navegación autónoma

Los nodos necesarios para la navegación autónoma realizada en el PMB2 son visibles en la Figura 28, pero ante la complejidad se ha realizado otro gráfico en la Figura 29 minimizando el número de nodos y tópicos mediante un *rosbag*. No se incluye el tópico de envío de metas llamado */move_base_simple/goal*.

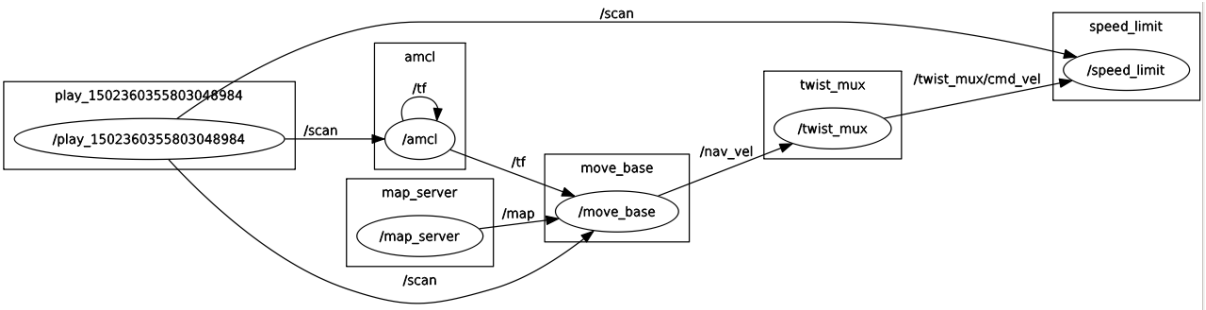

Figura 29: nodos simplificados en la navegación autónoma

ACTIONLIB

Además, *move_base* se provee implementado como un *SimpleActionServer* que contiene las metas de la navegación autónoma en forma de *pose*, incluyendo además el *frame* de TF al que está referenciado. Un *SimpleActionServer* pertenece a ***actionlib***, un protocolo [24] de comunicación de ROS similar al mencionado anteriormente de servidor/cliente. A diferencia del servidor/cliente habitual, soporta el uso de acciones, con lo que intercambia con un *SimpleActionClient* las siguientes especificaciones:

- Meta: Para cumplir las tareas usando acciones, se utiliza la noción de “meta” que puede ser enviada al *SimpleActionServer*. En el caso de la navegación autónoma, consiste en un *pose* que contiene información espacial del lugar al que debe moverse el robot. También se utiliza en el caso de brazos robóticos.

- Retroalimentación: el servidor implementa una forma de comunicar al *ActionClient* el progreso incremental de la acción, que en este caso es la posición actual respecto la ruta trazada.

- Resultado: Finalmente se envía el resultado del servidor al ciente, que sólo se envía una vez. Normalmente es un resultado satisfactorio (ha llegado correctamente a la meta) o un resultado de “abortado” (en el camino ha encontrado algún obstáculo que impide llegar a la meta).

Inflado del mapa de coste

Se introduce el concepto de mapas de coste e inflación. Como se ha mencionado anteriormente, la navegación autónoma utiliza dos tipos de navegación: global y local.

- La navegación global se utiliza para crear rutas entre la posición actual del robot dada por AMCL y la posición final o “meta”.

- La navegación local se utiliza para crear rutas en distancias cercanas para esquivar obstáculos en una zona de unos 4x4 metros que sigue aproximadamente la ruta global.

Para trazar las rutas, se utilizan mapas de coste globales (generado por SLAM) y locales (dado por los sensores de distancia, actualizables a tiempo real). Estos mapas dependen de la forma y tamaño del robot, asignando un valor alto a obstáculos y decreciendo hasta un valor bajo en zonas libres de obstáculos por las que el robot se pueda mover libremente. Son utilizados en los planeadores de rutas, por lo que cobran gran importancia. La asignación de valores acorde al tamaño del robot sige el esquema de la Figura 30.

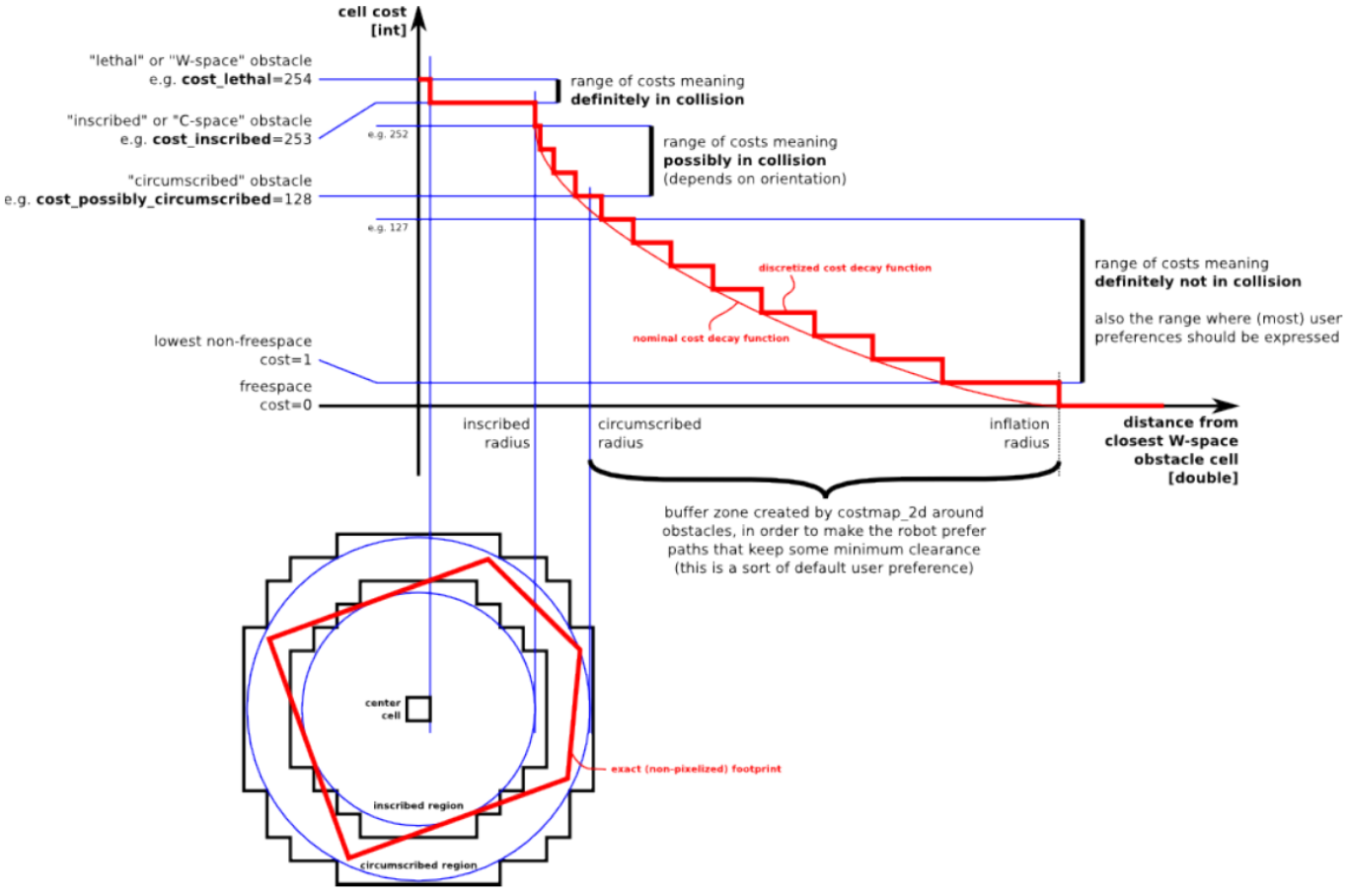

Figura 30: asignación de coste en mapas de coste

En el caso del PMB2, el radio del robot es de 0,28 metros, ya que el diámetro de la base cilíndrica es de 56 centímetros. Por tanto, depende del radio del robot para comprobar si está colisión y del parámetro del radio de **inflación**, que influye en el trazado de rutas y la separación de los obstáculos. Si el parámetro es excesivamente alto sorteará los obstáculos adecuadamente pero evita que pase por zonas estrechas, como puede ser una puerta, por lo que se recomienda un valor cercano o superior a 0,5 o 0,55 metros.

Planeadores de rutas

El nodo *move_base* puede incluir cualquier planeador de rutas global o local, aunque se incuyen diversos algoritmos cuya eficacia puede depender del entorno. Un planeador de rutas global tiene en cuenta el mapa generado por SLAM, con lo que puede esquivar rutas a larga distancia; mientras que el planeador local tiene en cuenta el mapa de coste local generado por lo sensores, efectivo para detectar obstáculos nuevos y poder esquivarlos en la medida de lo posible. Algunos discretizan el espacio en forma de celdillas acorde al mapa de coste (cada celda con un valor de coste) para encontrar una posible ruta entre celdillas, pero impide el movimiento en ciertos ángulos y puede limitar encontrar una ruta óptima a cambio de eficiencia computacional.

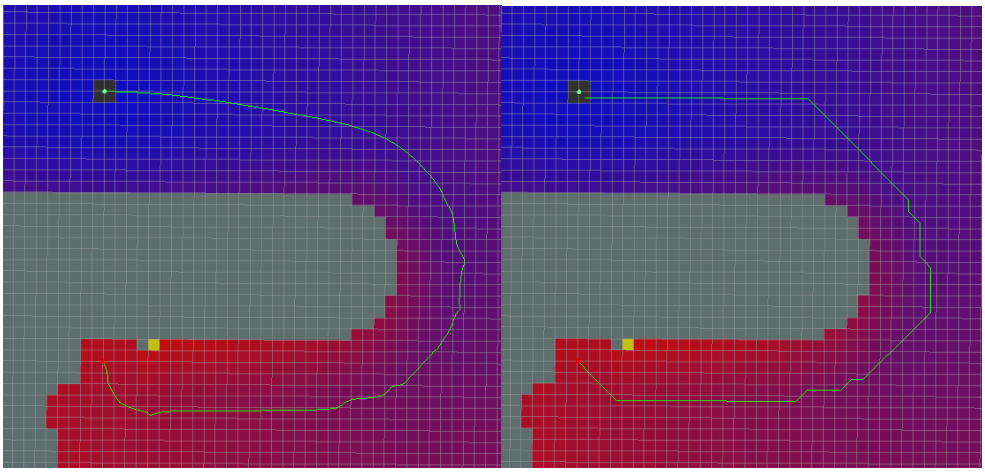

Figura 31: planeador de rutas estándar (izq) o por celdillas (der)

En el caso de los planeadores globales existen algunos algoritmos, como el algoritmo Dijkstra o los algoritmo A* (estrella). Ambos se incluyen en la integración *global_planner*. El planeador A* calcula menos potencial, ya que utiliza asociación de celdillas conectadas de 4, por lo que requiere menos recursos computacionales pero podría realizar trazados menos óptimos, como se puede comparar en la Figura 32.

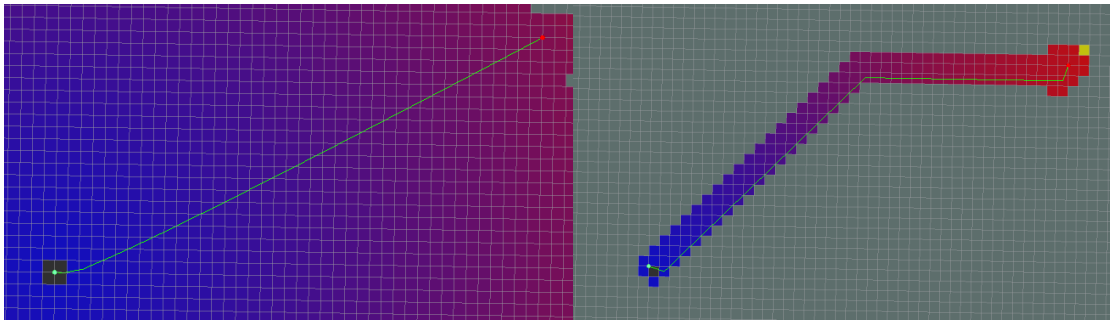

Figura 32: comparativa entre algoritmos Dijkstra (izq) y A* (der)

En el presente proyecto se ha parametrizado el navegador global llamado ***NavFn***, que utiliza el algoritmo de Dijkstra mejorado para bases autónomas circulares, cuya única parametrización remarcable es la de utilizar el mapa y que permita trazar rutas en zonas no planeadas previamente.

El planeador local [25] utiliza el mapa de coste local. Generalmente tienen una complejidad superior a los planeadores globales ya que deben trazar rutas para esquivar posibles obstáculos, con lo que se trazan a relativa frecuencia según el robot va avanzando. Utiliza por tanto, el DWA (*Dynamic Window Approach*) o enfoque de la Ventana Dinámica, es decir, genera el mapa de coste local según el robot va avanzando, centrado en él y actualizándose constantemente. Estos planeadores crean trayectorias cinemáticas para el robot y generando los comandos de velocidad que se envían al controlador de la base mediante tópico de velocidad en dx, dy, dtheta.

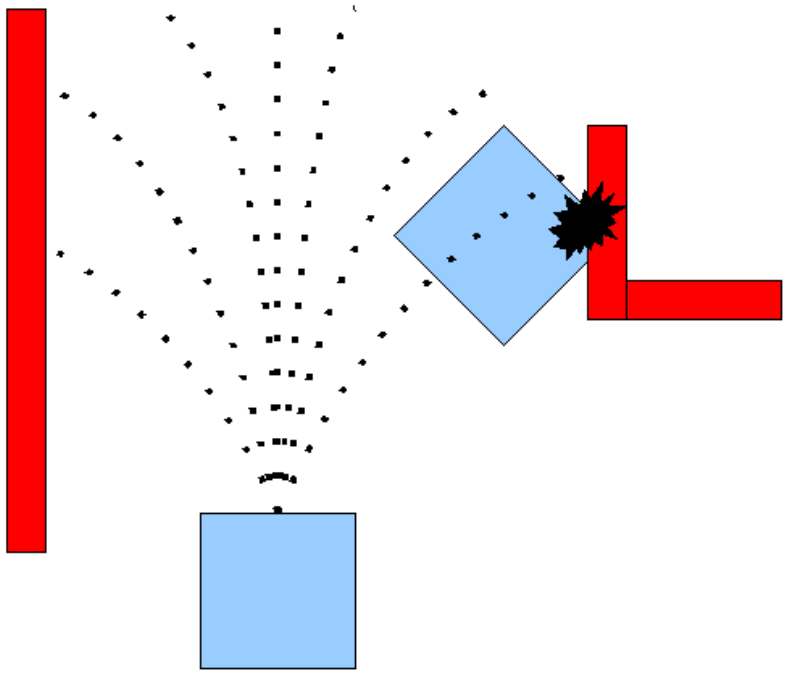

Figura 33: simulaciones que realiza DWA

El enfoque de la ventana dinámica o DWA sigue los siguientes pasos continuamente:

- Discretiza el espacio de control del robot (dx, dy, dtheta).
- Para cada velocidad discretizada, produce una simulación del futuro respecto del actual estado del robot en un período de tiempo parametrizable.
- Evalúa la puntuación de la trayectoria a partir de la simulación, teniendo en cuenta proximidad a obtáculos, proximidad a la meta, proximidad a la ruta global y velocidad. Elimina todas las trayectorias resultantes en colisión.
- Escoge la trayectoria de mayor puntuación y envía la velocidad asociada al controlador de la base.

Aquí la parametrización es clave en el comportamiento correcto de la navegación. Se deben introducir mínimos y máximos tanto en velocidad como aceleración (acordes a la capacidad del propio robot). En el caso de implementar el DWA, ejecuta trayectorias circulares, por lo que un tiempo de simulación excesivamente alto provoca trayectorias sub-óptimas además de un exceso de crga computacional.

También se parametriza la tolerancia en “xy” y la tolerancia en ángulo final respecto de la meta. Una tolerancia baja causa una alta exigencia, por lo que el robot

podría oscilar hasta cumplir los requisitos. Teniendo en cuenta la inexactitud inherente a sensores y el algoritmo de localización AMCL, una tolerancia muy baja puede suponer en una oscilación excesiva o incluso perpetua alrededor de la meta enviada.

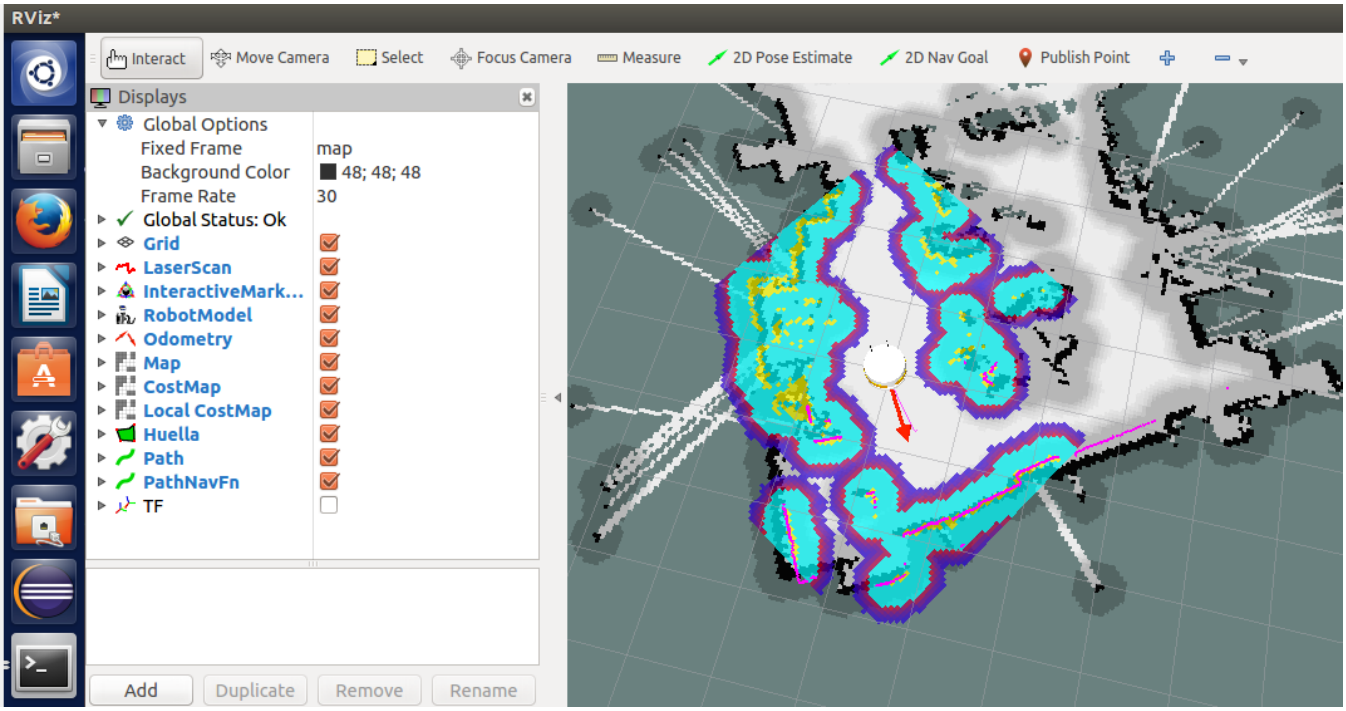

Figura 34: Rviz con el PMB2 realizando navegación autónoma

En la Figura 34, con la herramienta de visualización RVIZ se puede monitorizar el robot PMB2 realizando la navegación autónoma, junto a los mapas de coste (gris si global, en color si local) y la ruta trazada. También se puede observar cómo el láser (en rosa) coincide en gran medida con el mapa realizado por *slam_gmapping*.

```
<launch>

  <arg name="scan_topic" default="/scan"/>
  <arg name="odom_topic" default="/mobile_base_controller/odom"/>
  <arg name="no_static_map" default="false"/>
  <arg name="base_global_planner" default="navfn/NavfnROS"/>
  <arg name="base_local_planner" default="dwa_local_planner/DWAPlannerROS"/>
  <!-- <arg name="base_local_planner" default="base_local_planner/TrajectoryPlannerROS"/>-->


  <!-- Starting move_base node -->
  <node pkg="move_base" type="move_base" respawn="false" name="move_base" output="screen" machine="pmb2-5">

    <param name="base_global_planner" value="$(arg base_global_planner)"/>
    <param name="base_local_planner" value="$(arg base_local_planner)"/>

    <rosparam file="$(find pmb2_grav_2dnav)/config/planner.yaml" command="load"/>

    <rosparam file="$(find pmb2_grav_2dnav)/config/recovery_behaviors.yaml" command="load"/>

    <!-- observation sources located in costmap_common.yaml -->
    <rosparam file="$(find pmb2_grav_2dnav)/config/costmap_common.yaml" command="load" ns="global_costmap" />
    <rosparam file="$(find pmb2_grav_2dnav)/config/costmap_common.yaml" command="load" ns="local_costmap" />

    <!-- local costmap, needs size -->
    <rosparam file="$(find pmb2_grav_2dnav)/config/costmap_local.yaml" command="load" ns="local_costmap" />
    <param name="local_costmap/width" value="5.0"/>
    <param name="local_costmap/height" value="5.0"/>

    <!-- static global costmap, static map provides size -->
    <rosparam file="$(find pmb2_grav_2dnav)/config/costmap_global_static.yaml" command="load" ns="global_costmap"
unless="$(arg no_static_map)"/>

    <!-- global costmap with laser, for odom_navigation_demo -->
    <rosparam file="$(find pmb2_grav_2dnav)/config/costmap_global_laser.yaml" command="load" ns="global_costmap"
if="$(arg no_static_map)"/>
    <param name="global_costmap/width" value="100.0" if="$(arg no_static_map)"/>
    <param name="global_costmap/height" value="100.0" if="$(arg no_static_map)"/>


    <remap from="cmd_vel" to="nav_vel"/>
    <remap from="odom" to="$(arg odom_topic)"/>
  </node>

</launch>
```

Figura 35: lanzamiento del nodo *move_base*

La parametrización para el PMB2 realizada en este proyecto se encuentra en archivos tipo *YAML*, que serán accedidos mediante *roslaunch*, como se observa en la Figura 35 ubicados en la carpeta *config* en el paquete *pmb2_grav_2dnav*. Además, se remapean los tópicos para compatibilizar con el nombre de los tópicos al que otros nodos están suscritos, como en el caso de */nav_vel*, para mantener la prioridad en el nodo multiplexor de comandos de velocidades. El argumento *no_static_map* permite el uso de la navegación autónoma sin mapa si el valor se cambia a *True*, aunque no se recomienda ya que empeora significativamenta la navegación del robot a distancias mayores de 5 metros.

Gracias a la descentralización que permite ROS, se puede implementar la navegación autónoma en otro ordenador externo al PMB2 siempre que la frecuencia del controlador no sea excesivamente alta (mayor de 10Hz). Sin embargo, debido a la gran potencia que el ordenador incluído en el robot se desplazará la computación,

aunque la parametrización se realice desde la máquina virtual; la navegación autónoma apenas supone el 5% de la CPU del PMB2, mientras que en la máquina virtual pasa a ser el 40%. Esto es posible gracias a funciones de **lanzamiento remoto** integradas en *roslaunch*, mediante *ssh*. El requisito imprescindible es que los nodos iniciados remotamente estén instalados en la máquina objetivo. Para ello, se utliza el archivo *robots.machines* incluído en el paquete creado para el lanzamiento de la plataforma del presente proyecto llamado ***platform_bringup*** , con otros archivos de lanzamiento de la plataforma, como los que gestionan el funcionamiento del Meka, la Kinect conectada al PMB2 o incluso scripts propios para el caso de estudio.

El envío de metas se puede realizar mediante Rviz gracias al botón “2DNavGoal” (Figura 34). También se realiza mediante código, tanto en C++ como en Python: se puede referenciar a cualquier frame (TF). Para ello, dentro del paquete *pmb2_grav_2dnav* se incluyen algunos ejemplos simples dentro de la carpeta *scripts* (Python) o *src* (C++). Gracias a la facilidad de enviar metas, se puede programar una patrulla entre puntos de un mapa o incluso “metas relativas” respecto al mismo robot si utiliza *base_link* como frame objetivo (funcional con la navegación autónoma sin mapa). El fin de dichos scripts es la de facilitar el aprendizaje de envío de metas y el testeo de la navegación implementada, que se utilizará en la aplicación del capítulo 5.

3.5. Problemas típicos

Un problema común en la navegación autónoma es la de quedarse estancado en una posición. Normalmente ocurre si está demasiado cerca de obstáculos, con lo que para evitar la colisión se detiene. Aunque se incluyen unas técnicas de recuperación parametrizadas, como por ejemplo girar sobre sí mismo para buscar una zona por donde esquivar el obstáculo; algunas veces no puede escapar por sí mismo por seguridad, con lo que se deberá utilizar un controlador manual (joystick o teclado de ordenador mediante el paquete *key_teleop*) y alejarlo de los obstáculos.

Por otro lado, la navegación autónoma es muy sensible a las transformadas TF. En el caso de utilizar más de un equipo o realizar la computación en un ordenador externo, es imprescindible que los relojes de todos los equipos estén sincronizados.

En cuestión de seguridad, actualmente utiliza el láser para la detección de obstáculos. Ya que es un sensor bidimensional, sólo detecta obstáculos a la altura que está colocado, unos 7 centímetros respecto del suelo. Por lo tanto, se debe procurar que el entorno no tenga salientes en voladizo, ya que podría colisionar con ellos al no ser detectados por el láser. En el futuro se puede integrar el uso de otro sensor, como el RGB-D de la Kinect para visualizar obstáculos a diferentes alturas, aunque no ofrece un rango tan amplio ni tan preciso como el láser.

4. MEKA

Meka es un robot construido por la empresa Meka-Robotics, la cual fue adquirida por Google en 2013. Su función principal es la de actuar con el entorno gracias a los dos brazos humanoides que incorpora.

Figura 36: robot Meka colocado sobre el PMB2

4.1. Descripción breve del hardware

Incluye dos brazos antropomórficos de 7 DOF (*Degrees Of Freedom* o grados de libertad) montados sobre una estructura asemejada al torso de un humano. Como herramienta final, cada brazo dispone de una mano con forma humana de 4 dedos,

simétricas entre sí. Dentro del "torso", se incluye la electrónica necesaria para controlar los servomotores que posee. Aunque los brazos son simétricos, fueron adquiridos en fechas diferentes, por lo que existen ligeras diferencias en el hardware, pero se tratarán por igual en el software, ya que los drivers necesarios están incluidos en el equipo.

Los sensores (encóders) y actuadores (servomotores) son controlados a bajo nivel mediante **EtherCAT** alimentado todo el sistema a 24V. Tiene 7 motores por brazo y 5 en cada mano [28], lo que hace un total de 24 motores. Cada motor se conecta a una placa que lo controla por EtherCAT, como por ejemplo el DSP (unidad *Digital Signal Processing*); las placas entre sí utilizan dicho protocolo a 4kHz.

El protocolo de comunicación del robot es EtherCAT [26], un protocolo informático de código abierto y alto rendiemiento utilizado tanto en entorno industrial como en la automatización de edificios. Supera las limitaciones típicas de los paquetes Ethernet, ya que el telegrama se procesa sobre la marcha al pasar entre esclavos y mandarlo al siguiente. Además, permite estructuras de árbol flexibles, siempre que haya sólo un dispositivo designado como maestro.

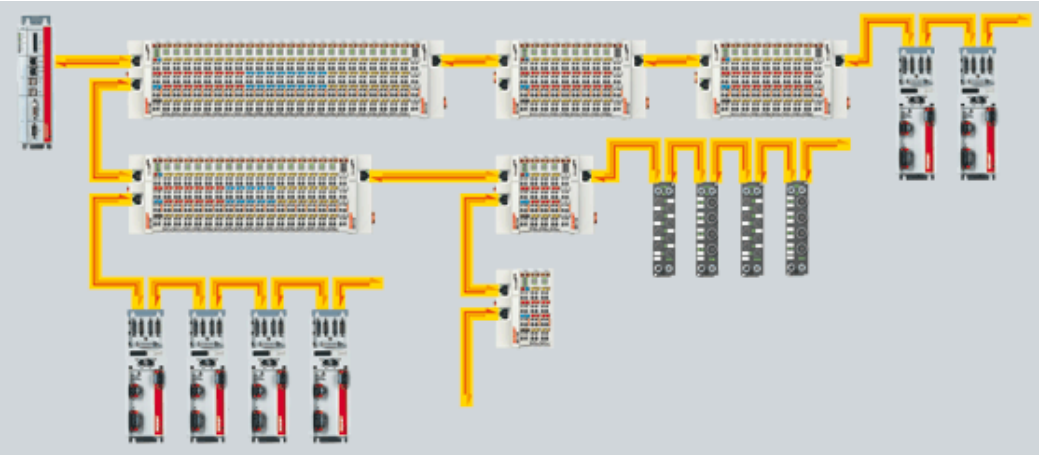

Figura 37: ejemplo de estructura de árbol flexible EtherCAT en la industria

Dicho protocolo supone un retraso de nanosegundos, por lo que permite ejecuciones en *real time* (tiempo real): rápido, deslocalizado y determinístico, claves para un correcto control y funcionamiento. Debido a esta disposición, el lazo de control se ejecuta directamente en el PC al que está conectado mediante Ethernet o cable Firewire. Por lo tanto, el propio ordenador hace de máster EtherCAT, mientras los demás dispositivos actúan de clientes. Los componentes en dicho ordenador deben

cumplir con una frecuencia determinada sin la posibilidad de ralentizarse, al contrario que los ordenadores tradicionales donde el procesamiento es secuencial.

El ordenador se alimenta a 12VDC, mientras que el resto del robot se alimenta a 24VDC, con lo que los sistemas de alimentación deben ser independientes entre sí.

4.2. Descripción del software

Como se ha mencionado en el apartado anterior, el ordenador que controla el robot debe cumplir el lazo de control, con lo que el sistema operativo no es el común entre los terminales tradicionales. Como en el caso de la areonáutica, la computación se debe realizar a tiempo real. Para ello, se creó en 2006 un kernel especial de Linux junto con un programa llamado RTAI [27], el más rápido de los creados hasta la época. Permite controlar el hardware correctamente para controlar las trayectorias de ambos brazos robóticos.

Actualmente, dicho kernel RTAI debe ir incorporado a a distribución de Linux Ubuntu, para que sea compatible con ROS. Utiliza una distribución Ubuntu 12.04 con RTAI 3.9 (sin soporte y prácticamente obsoleto) y distribución ROS Hydro, que es la proporcionada por el fabricante Meka-Robotics, pero compatible con las aplicaciones en ROS Indigo desarrolladas en el proyecto. Sin embargo, no está soportada la función de ***MoveIt!***, una aplicación de propósito general para brazos robóticos de ROS, que ofrece ciertas ventajas al permitir reutilización de código e implementación sencilla de cinemáticas, además de otras ventajas mencionadas en el apartado 4.6.2.

Debido a la personalización requerida para que el hardware cumpla el exigente lazo de control de 1kHz (períodos de 1ms), el CPU no puede entrar en modo ahorro de energía, así como algunos dispositivos como el WiFi o los USB integrados no están habilitados. Por tanto, la única forma de conexión es mediante un segundo puerto Ethernet utilizado para conexiones de red. Ya que el robot Meka irá montado encima del robot móvil PMB2, deberá ir conectado a este último mediante Ethernet a través del puerto Ethernet interno, con el pin 12 de la Figura 15.

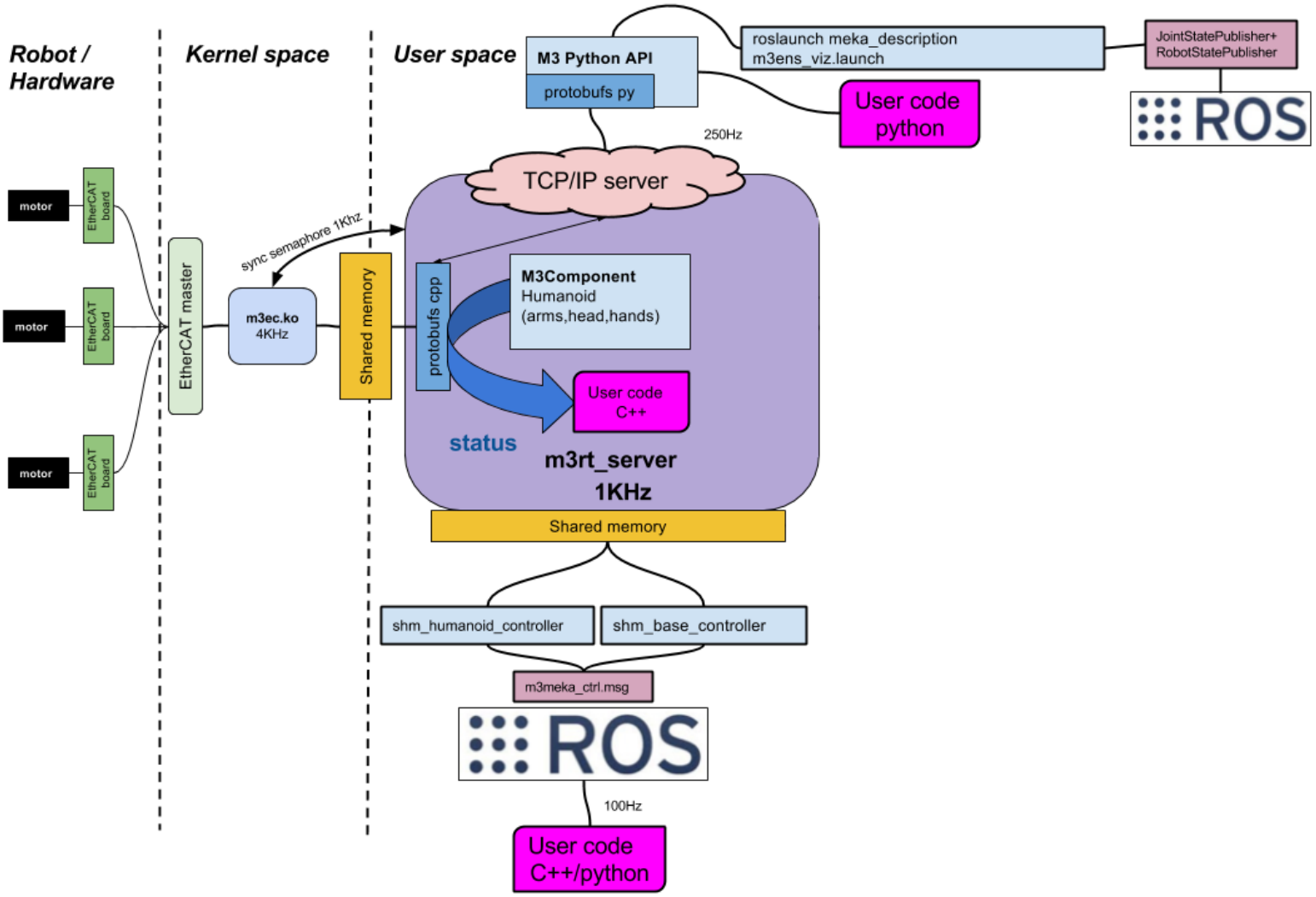

Figura 38: esquema de software del Meka

Una vez conectado el Meka a su ordenador y ambos correctamente alimentados, se ejecutan los siguientes comandos:

```
$ lsec
```

Dicho comando monitoriza todos los esclavos de EtherCAT conectados. Si el número del último cliente es 22 y todos están en estado de “PREOP” o “INIT”, se pueden lanzar los siguientes comandos, que iniciarán el servidor maestro de EtherCAT llamado ***m3rt_server*** y la memoria compartida entre ROS y dicho servidor, con lo que se podrán ejecutar aplicaciones ROS y ambos brazos pueden ser controlados desde ellas. Dicha memoria compartida es visible en la parte inferior de la Figura 38 [28].

```
$ m3rt_server_run –m
$ rosrun shm_humanoid_controller shm_humanoid_controller
```

Una vez ejecutados los comandos, se pueden utilizar aplicaciones de ROS para manipular el Meka.

4.3. Problemática: falta de servicio técnico

Como se mencionó al comienzo de este capítulo, la empresa constructora y distribuidora del software integrado llamada Meka-Robotics fue absorbida por la compañía propietaria de Google el año 2013. La empresa desapareció junto con el servicio técnico, por lo que todo el soporte de hardware y software está a manos del Grupo de Robótica, Automática y Visión por Computador de la Universidad de Jaén o expertos externos. La desventaja es que sólo existe una decena de robots similares al Meka, por lo que hay pocos expertos en hardware y software que se puedan encargar de las dificultades técnicas que puedan surgir.

En el presente proyecto se intentó actualizar el software del PC a Ubuntu 14.04 con kernel compatible con RTAI 5.0 y distribución de ROS Indigo (compatible con las funciones de *MoveIt!*) [28], pero se requiere especificaciones exactas del hardware del PC y una correcta configuración para compilar con éxito dicho kernel, ya que puede provocar fallos de la CPU.

Debido a la necesidad de cumplir con RTAI y además tener una tarjeta Ethernet compatible con determinados drivers para instalar el servidor de EtherCAT, no se puede exportar a otro ordenador con facilidad. Además, el CPU debe ser de altas prestaciones para poder ejecutar la tarea de control en el lazo de control de 1kHz.

Por último, el estado actual de los manipuladores robóticos para la investigación ha cambiado con respecto a la fecha en la que el Meka se fabricó. Actualmente se utilizan brazos robóticos con una fisionomía menos antropomórfica en pro de una mayor manejabilidad. También se han sustituido los actuadores finales por pinzas más sencillas en lugar de las complejas manos antropomórficas, ya que su control es más sencillo y permite realizar tareas con una mayor repetitividad, adecuado para el uso del *machine learning* aplicado a recoger y utilizar una gran diversidad de objetos.

4.4. Unión del Meka y el PMB2

La razón principal de implementar el PMB2 en la plataforma es la de proporcionar movilidad. Para ello, el Meka se colocará montado encima del PMB2, tal y como se puede visualizar en la Figura 36.

4.4.1. Unión física

La unión física consiste esencialmenta en dos piezas:

- Una con forma de “U” justo debajo de “torso” del robot, atornillado en 6 puntos para lograr una sujección segura. Sus medidas, tomado como un prisma son 145x204x234mm.
- La otra es simplemente un perfil de aluminio de 90x45mm con una longitud de 0,55 metros, para dar altura suficiente con la que los brazos no colisionen con el PMB2. En la parte inferior se montan unas cuñas atornilladas a la base móvil en 4 tornillos, visibles en la Figura 36.

4.4.2. Unión lógica (en ROS):

Teniendo en cuenta las especificaciones físicas de la unión, se procede a modelar en URDF (o XACRO, la versión avanzada que permite importar archivos) para que las transformadas TF generadas por el nodo *robot_state*_publisher sean correctas y permita situar correctamente ambos brazos respecto del mundo y unificar todos los TF de la plataforma. Ahora que el robot se desplaza por el mapa debe ser capaz de posicionarse correctamente en el entorno, ya que no es fijo sino móvil. Es indispensable realizarlo con toda la exactitud posible, ya que de ello depende la precisión de interactuar correctamente con el entorno.

Existen dos opciones para realizar la unión de los robots:

- Crear un archivo URDF único, modificando el utilizado en el arranque del PMB2 y que incluya el URDF actual del Meka junto con las modificaciones oportunas para incluir el soporte y sensor adicionales.
- Modificar sólo el URDF del Meka, añadiendo el soporte y el sensor correctamente. La unión entre el Meka (con su soporte) y el PMB2 se puede realizar mediante un *static_transform_publisher*, que es un emisor de

transformadas TF simple y estático. Dicho emisor unirá de forma estática el *frame* padre del Meka a alguno de los *frame* de la base móvil autónoma, con lo que el Meka se desplazará junto con el PMB2 al igual que como lo haría con el URDF único.

Finalmente se eligió la segunda opción, ya que aporta las siguientes ventajas: mayor independencia de lanzamiento, con lo que permite funcionamiento de PMB2 o Meka en solitario y mejor depuración de fallos. Además, se evita modificar el software del PMB2, uno de los objetivos propuestos a la hora de programar la plataforma.

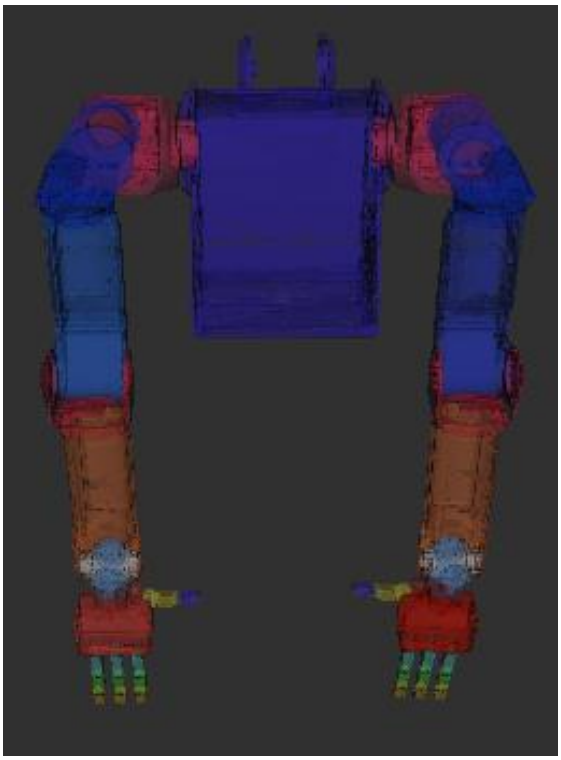

Figura 39: modelo URDF de partida llamado *uja_fixed_hands.urdf.xacro*

Partiendo del URDF llamado *"uja_fixed_hands.urdf.xacro"* visualizado en la Figura 39, ubicado en el paquete ***meka_description****/robots* y cuyo *frame* padre se llama *base_link*, se modificó con el nombre "*uja_acople.urdf.xacro"*:

- Cambio de nombre de *base_link* a *base_old_link*, para evitar problemas de compatibilidad con el *frame* del mismo nombre que contiene el URDF del PMB2.
- Creación de un *link* tipo "box" que representa la "U" del soporte llamado *box_pc_link* con las dimensiones especificadas anteriormente.

- Otro *link* tipo “box”, pero mucho más alargado, que representa el soporte vertical llamado *beam_link*.
- El *base_meka_link* está situado justo en la terminación del soporte vertical, para añadirlo de forma sencilla al PMB2.

La dificultad es la creación de los *joint* o uniones entre dichos *link*, ya que se parte de algunas transformadas que no están perfectamente medidas, por lo que se debe ajustar a prueba y error. Las uniones serán rígidas, al igual que en el soporte físico, por lo que el tipo es *fixe*; a diferencia de los de las ruedas que son *continuous* o los brazos, de tipo *revolute* (con límites). El cambio más importante es el realizado colocando ***base_meka_link*** como padre, para que no haya errores con el del PMB2 al tener *frames* con el mismo nombre y tener un acople más sencillo.

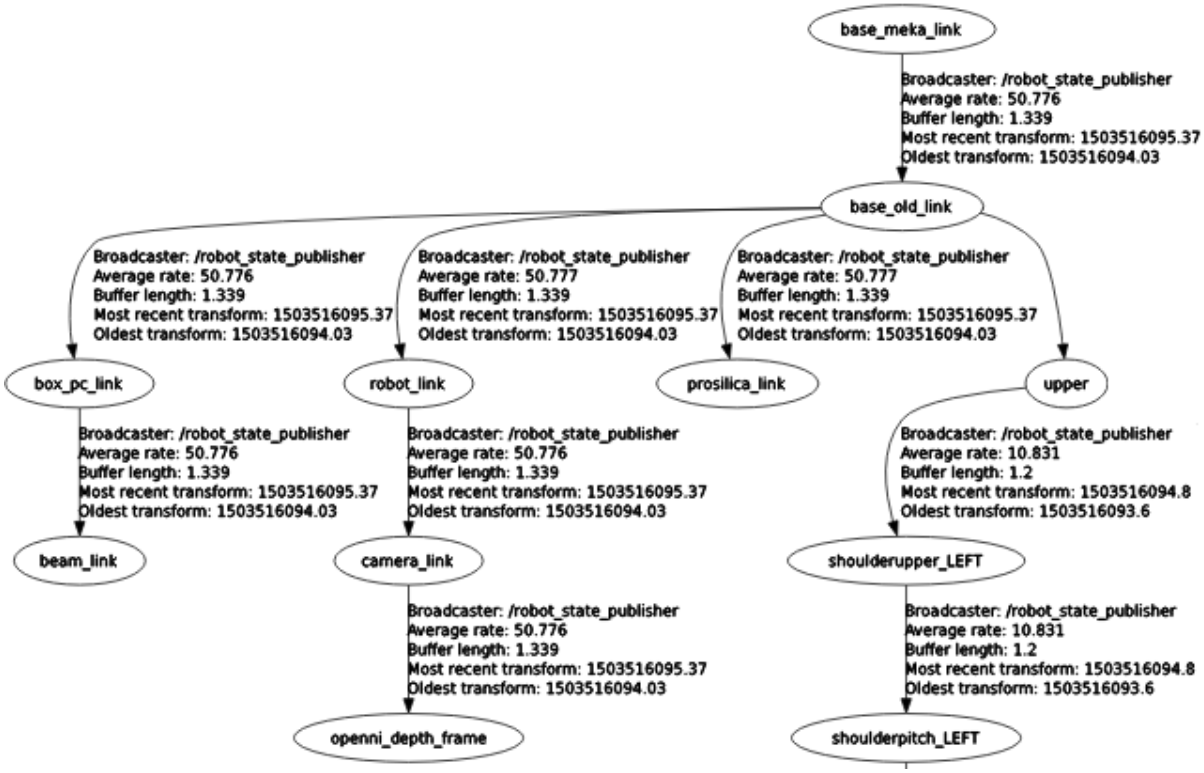

Figura 40: árbol TF parcial con el *frame* padre renombrado y soporte, sensores añadidos

Por otro lado, es importante que el nodo *robot_state_publisher* se llame de otra forma, al igual que el nombre del parámetro del modelo XACRO se debe llamar diferente al nombre por defecto, por ejemplo se llamará *meka_description*, a tener en

cuenta también en el software de visualización Rviz y los demás nodos, ya que se tiene que remapear para que acepten el nuevo nombre del parámetro.

```
<launch>

<!-- Carga el modelo de meka&Kinect -->
  <arg name="meka_model" default="$(find meka_description)/robots/
uja_kinect.urdf.xacro" />
  <param name="meka_description" command="$(find xacro)/xacro.py '$(arg
meka_model)'" />

<!-- Utiliza el modelo para emitir los TF de todos los frame estáticos -->
    <node pkg="robot_state_publisher" type="state_publisher"
name="meka_state_publisher"  >
        <remap from="robot_description" to="meka_description" />
        <remap from="joint_states" to="humanoid_state" />
    </node>

<!-- Unión entre el PMB2 y el Meka-->
    <node pkg="tf" type="static_transform_publisher"
name="meka_PMB2_joint_broadcaster" args="0 0 0.2 0 0 0 base_link base_meka_link 20" /
>

<!-- Lanzamiento del simulador -->
    <node pkg="controlMeka" type="simMeka" output="screen" name="simMeka"
machine="meka_packs">
         <remap from="robot_description" to="meka_description" />
    </node>
</launch>
```

Figura 41: lanzamiento de nodos necesarios para simulación del Meka

Finalmente, queda calibrar la unión entre ambos robots. Como se ha mencionado, se utiliza un nodo *static_transform_publisher* entre dos de los *frame*. Para que la localización funcione correctamente y se vea reflejada la movilidad que el PMB2 da al Meka, un *frame* del PMB2 debe ser padre del *base_meka_link*. Por ejemplo, se utilizó el *base_link*. El ajuste espacial también se debe realizar por prueba y error, de forma que se apoye exactamente en la superficie. Entre los *frame* mencionados, existe una diferencia de 0,2 metros en la dirección Z, como se ve reflejado en el archivo *launch* de la Figura 41, en el argumento proporcionado al mencionado nodo.

4.5. Inclusión de un nuevo sensor en la plataforma

Además de añadir el soporte, se procederá a instalar un nuevo sensor: en este caso se trata de la cámara RGB-D (*Red Green Blue - Distance*) **Kinect**, una cámara perteneciente al proyecto de la fusión sensorial [1], la cual irá montada encima del Meka y proporciona visión a ambos robots. La cámara (Figura 42) fue desarrollada

por Microsoft para la consola Xbox 360 en 2010 y posteriormente añadida para su uso en PC en 2011. Es un sensor barato y bastante fiable que permite la esqueletización a partir de imágenes de personas. Su uso se especifica en el capítulo 5.

Figura 42: imagen de la cámara Kinect (RGB-D)

Para que posicione correctamente lo detectado, un ajuste similar a la ubicación física respecto al Meka es necesaria. Se colocó justo encima del torso del Meka, como se observa en la parte superior de la fotografía de la Figura 36. Modificando nuevamente el archivo URDF.XACRO, en un nuevo archivo llamado *uja_kinect.urdf.xacro*.

Se puede visualizar en Rviz si se importa su modelo. Para ello, se descargó el paquete ***common_sensors***, que contiene diversos sensores, entre ellos el modelo de la cámara Kinect, pero se configura uno de los modelos en el archivo *common_sensors/urdf/sensors/**kinect_modificado.urdf.xacro***, creado en el presente proyecto.

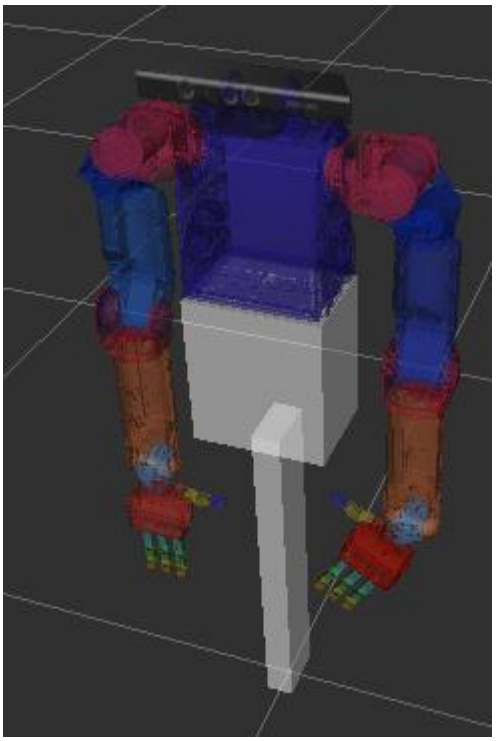

Figura 43: visualización de *uja_kinect.urdf.xacro,* con soporte y Kinect

El *frame "camera_link*" está colocado respecto al *frame* del Meka "*robot_link"* con un desfase de 3cm en el eje 'X' y 9cm en el eje 'Y', además se eliminan otros *frame* que provocan errores con el funcionamiento de los paquetes *openni_launch* y *openni_tracker* para obtener la jerarquía mostrada en el árbol TF de la Figura 40 (la segunda rama). Dicho archivo se importa en el archivo del Meka, con lo que la visualización queda como en la Figura 43.

4.6. Diferentes estrategias para el posicionamiento de los brazos

En la mayoría de brazos robóticos, el control se realiza por servomotores que miden el ángulo de giro o distancia recorrida, según sean articulaciones rotacionales o prismáticas, respectivamente. Cada articulación se traduce en un ángulo de libertad, que permite al manipulador realizar diferentes movimientos. Los movimientos se controlan indicando al controlador del actuador el ángulo de giro objetivo de cada articulación [29]. Mediante dichos ángulos y la fisionomía del robot se puede conocer su posición en el espacio de forma sencilla, ya sea en coordenadas cartesianas, cilíndricas o esféricas. Se denomina cinemática directa.

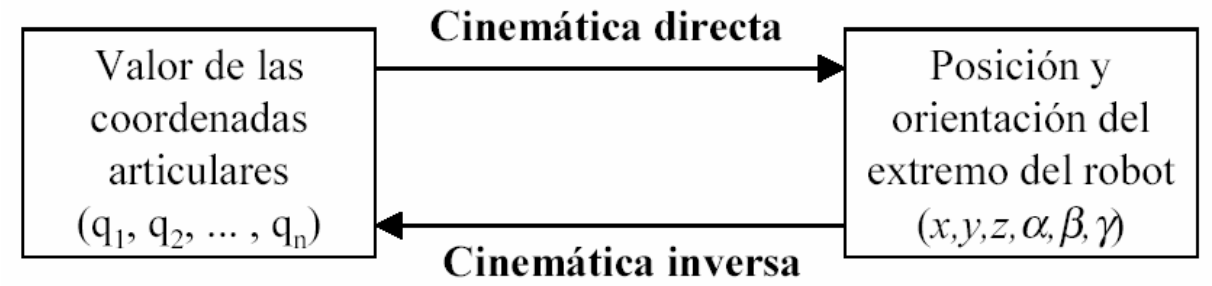

Figura 44: cinemática directa e inversa

Para ubicar al actuador final (en este caso las manos del Meka) en una posición concreta respecto a un punto de referencia, se requiere utilizar la técnica de la **Cinemática Inversa**, la cual no es trivial como la cinemática directa. Dicha técnica es un proceso matemático que determina los parámetros de giro de cada articulación para que el actuador final alcance la posición deseada. La especificación de movimientos sucesivos para alcanzar la meta se llama “planificación de movimiento”.

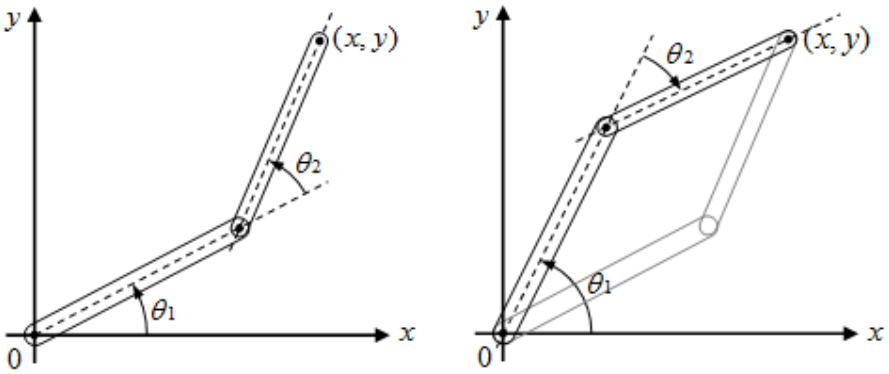

Figura 45: diferentes ángulos en articulaciones para alcanzar el mismo objetivo en un brazo robótico

Se debe tener en cuenta que, para mover el robot de un punto a otro, existe más de una solución. Existen varios algoritmos para hallar dichas cinemáticas inversas, como el método Denavit-Hartenberg, creado en 1955 por Jacques Denavit y Richard Hartenberg con el porpósito de estandarizar la ubicación de los sistemas de referencia de los eslabones de un robot.

Para el cálculo computacional de las variables de articulación, se utilizan diversos métodos aproximados de la pseudo-inversa de la matriz Jacobiana [29]. En ROS se utilliza la biblioteca de código abierto de cinemática y dinámica o KDL (*Kinematic and Dynamic Library*), que forma parte del proyecto de software libre OROCOS (*Open Robot Control Project*). Está programada en el lenguaje de programación C++ para aumentar la eficiencia y puede utlizar dos métodos numéricos

diferentes para hallar las cinemáticas inversas: *Newton-Raphson* y *Levenberg-Marquad*, aplicados al jacobiano del robot. Están especialmente recomendados para brazos robóticos de más de 6 DOF, con lo que su uso está indicado para el Meka, que en cada brazo tiene 7 DOF.

4.6.1. Uso de cinemáticas incluidas

Previa a la realización del presente trabajo de fin de grado, en el ordenador del robot Meka se encuentran unas cinemáticas funcionales que utilizan la librería KDL anteriormente mencionada. Dichas cinemáticas fueron desarrolladas en la tesis doctoral [32] , que se encuentran dentro de la carpeta *asgarcia* en el ordenador del robot. El paquete donde se encuentran los scripts se llama ***controlMeka***, donde se encuentran dos nodos muy similares entre sí:

- *controlMeka*: Este nodo controla la función en ambos brazos, tanto de actuadores, como de la señal de los sensores de los servomotores, como de gestionar las cinemáticas inversas. Publica un tópico llamado "*humanoid_state*", que es igual que el llamado "*joint_state*" que publica el PMB2 con la información de sus servomotores hacia el nodo *robot_state_publisher*, que gestiona las transformadas TF del modelo URDF del Meka. Dicho nodo está suscrito a los siguientes tópicos, duplicados para el brazo izquierdo (*left*) o el derecho (*right*) y que admiten un vector con números de tipo *Float64*:
    - *Command_joint_arm_():* admite hasta 7 números, cada uno maneja en radianes el ángulo de giro de cada articulación.
    - *Command_joint_hand_():* manejan los dedos de la mano antropomórfica.
    - *Command_frame_arm_():* son los que manejan las coordenadas cartesianas de la cinemática inversa de cada brazo. Tras varias pruebas, se ha comprobado que están referenciados al *frame* de los TF llamado ***/robot_link***. Sólo acepta 3 números, ya que ignora las 3 dimensiones de la orientación del actuador final y lo coloca en una orientación predeterminada.
- *simMeka*: se comporta exactamente igual que el nodo *controlMeka*, con el interesante aspecto de realizarlo en simulador, con lo que no necesita estar

conectado al auténtico robot y se puede utilizar en tareas de desarrollo. En la Figura 46 se encuentra un esquema del nodo *simMeka*, que es prácticamente igual para el nodo *controlMeka.*

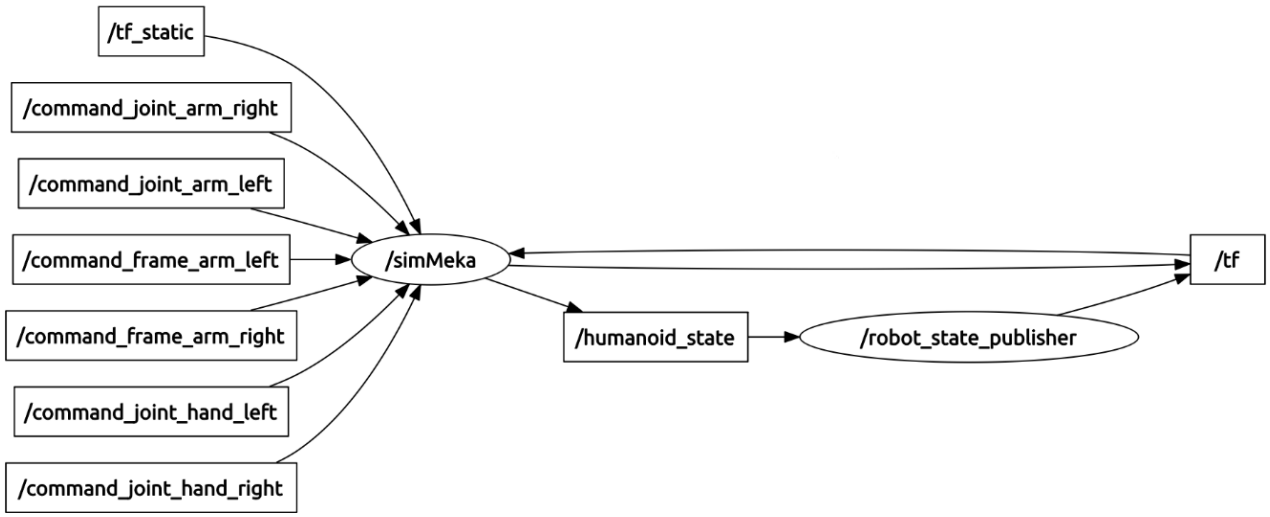

Figura 46: nodo *simMeka* en funcionamiento

Para la inclusión en la plataforma de estos nodos, se renombra el nombe del argumento que contiene la descripción URDF del Meka para que no interfiera con el del PMB2: el nombre por defecto es *robot_description* y se cambia a *meka_description*: se debe remapear para que utilice el nuevo argumento, como se observa en las últimas líneas del archivo de lanzamiento de la Figura 41.

ADVERTENCIA DE SEGURIDAD

En estas cinemáticas no se tiene en cuenta la interacción con objetos o humanos. Por lo tanto, no toma como parámetro de entrada un sensor como la Kinect para detectar posibles colisiones con el entorno en forma de *octomap* (un mapeado probabilístico en 3D que usa como parámetro de entrada un sensor de profundidad 3D). Así como tampoco tiene en cuenta autocolisiones con el soporte, por lo que se debe tener en cuenta que puede causar lesiones a personas o dañar objetos y a sí mismo. Es prioritario tener cerca la seta de emergencia para desactivar el robot cuando está en uso.

4.6.2. Uso de MoveIt

Otra opción para implementar las cinemáticas inversas necesarias para manipular los brazos es la de utilizar ***MoveIt!*** [30], una herramienta integrada en ROS que incorpora los últimos avances en planificación de movimiento, manipulación, percepción 3D, cinemáticas, control y navegación. Es la herramienta de *open-source* más extendida, con un total de 65 robots.

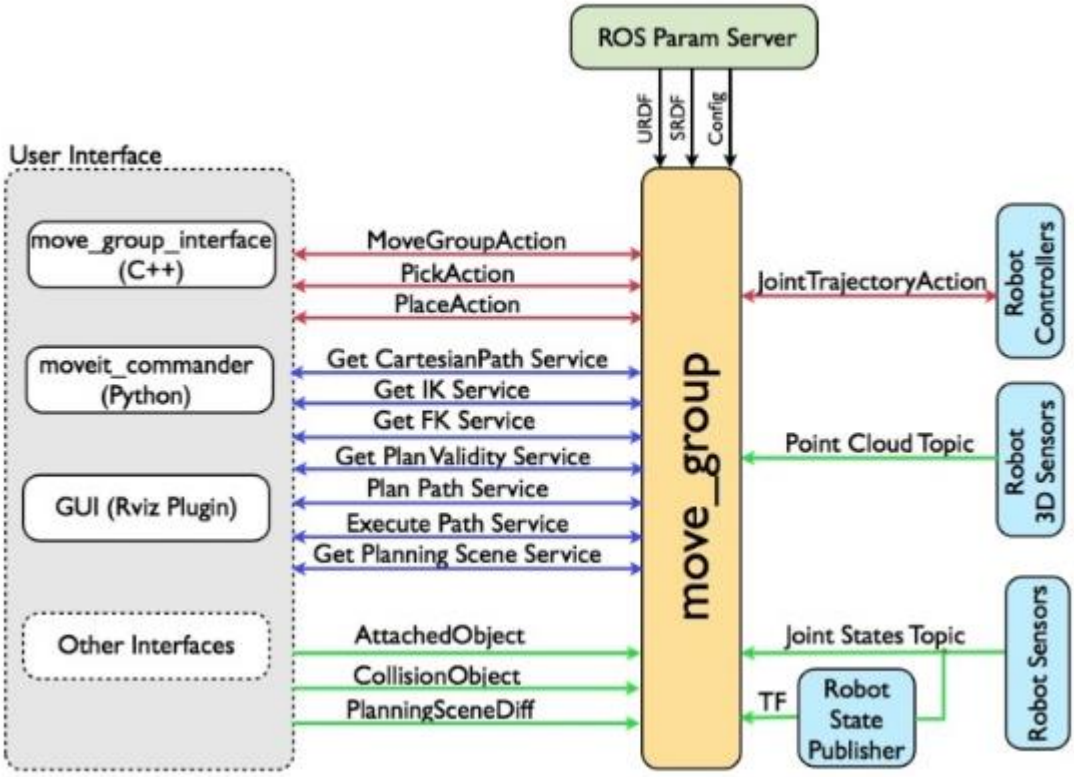

Figura 47: arquitectura de *MoveIt!*

El nodo principal de la plataforma se llama *move_group*, con un nombre y uso de alto nivel similar a la navegación autónoma del nodo *move_base*. Ambos se basan en un *actionlib* para enviar metas, en este caso de la posición del brazo robótico.

Además, incluye una serie de herramientas útiles:

- Cinemáticas inversas basadas en KDL fáciles de implementar, utilizando el modelo URDF del robot. Crea un archivo llamado SRDF.

- Planeador de escenas, que además incluye herramientas para la acción de coger elementos de geometría simple, como cilindros o prismas.
- Comprobador de límites de:
    - Posición: se puede restringir la posición en regiones del espacio delimitadas
    - Orientación: límites de orientación de partes del robot
    - Visibilidad: permanece en un punto donde se el brazo esté dentro de un cono de visión proporcionado por un sensor.
    - Colisión: si está dentro del límite de visibilidad, se incluyen herramientas para preveer si alguna parte del brazo puede colisionar con objetos detectados.
    - Auto-colisión: por defecto y gracias al modelo URDF, comprueba si colisiona con alguna parte propia del robot.

Aunque se exploró el uso de esta opción por las ventajas respecto a la opción de las cinemáticas integradas, requiere la actualización del sistema a Ubuntu 14.04 con ROS Indigo, la cual no fue posible por los argumentos comentados en el apartado 4.3. Además, requiere implementar ***ros_control***, que requiere un conocimiento en profundidad del hardware del robot y su enlace con el software, que salen del alcance del presente trabajo de fin de grado.

4.7. Ejemplos de interacción con personas

Utilizando las cinemáticas ya implementadas del apartado 4.6.1 y el uso de la cámara Kinect en el paquete *openni_tracker* modificado por el trabajo de la fusión sensorial [1], se procede a la programación de un servidor ROS que envía metas en coordenadas cartesianas al nodo del Meka. El servidor programado se encuentra dentro del paquete ***nav_to_people***, cuyo script es *meka_arm_server_duo.py* Por tanto, publica a los tópicos *command_frame_arm_left* y *command_frame_arm_right* como parámetro de salida.

Como parámetro de entrada, necesita un *frame* al que referenciar. En este caso, se utiliza la mano del humano con el que interactúa, cuya posición se proporciona con el paquete *openni_tracker* mediante la esqueletización que realiza. Además, las

coordenadas de meta se referenccian al *frame "robot_link"*, con lo que se debe realizar la conversión de la traslación en cartesianas entre dicho *frame* y el objetivo.

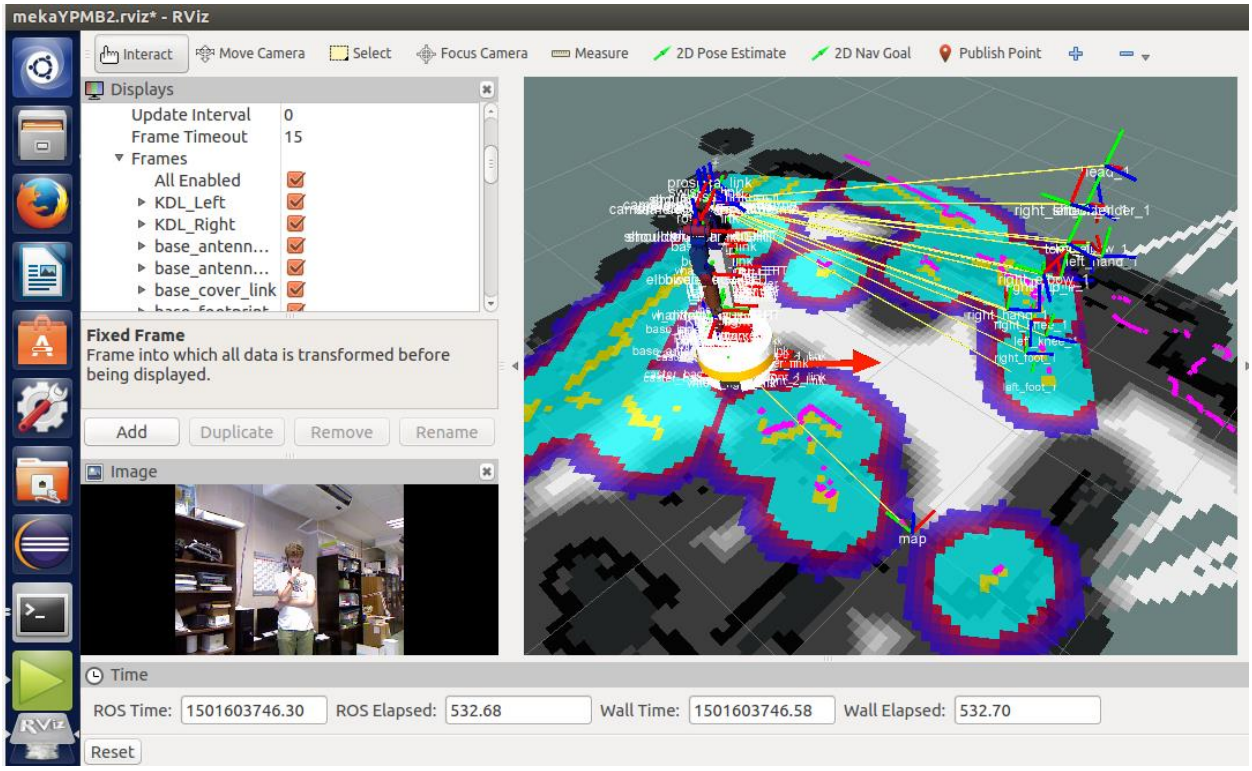

Figura 48: Rviz con ambos robots y la esqueletización de un humano en TF

En la Figura 48, se puede observar la representación en Rviz de ambos robots con el Kinect, que proporciona los *frame* correspondientes al "esqueleto" del humano referenciados a la propia cámara, a su vez referenciados al robot y por lo tanto al mapa. Es por ello que, a los pies del esqueleto, se observan dos manchas rosas, que coinciden con gran exactitud con la detección de los tobillos proporcionado por el láser del PMB2.

Dicho servidor recibirá una petición de un cliente, el cual puede ser un simple script (*cliente_meka.py*) o estar dentro de un nodo mayor. El servicio empleado entre servidor y cliente creado es el siguiente, en la carpeta *srv* del paquete ***nav_to_people***:

```
bool interruptor          #True para activar
string modo               #mimica o dar_mano
string brazo              #left or right = izquierdo o derecho
string frame_objetivo     #el frame objetivo
---
bool success
string message
```

La primera parte corresponde a la petición, que tiene 4 parámetros de entrada:

- Interruptor: valor booleano, que envía “True” para activar o “False” para desactivar el nodo de envío de metas al brazo del Meka.
- Modo: cadena de caracteres, que admite actualmente dos: “mimica” y “dar_mano”, explicados más adelante.
- Brazo: cadena de caracteres, para elegir brazo izquierdo o derecho (del robot Meka).
- Frame_objetivo: el *frame* objetivo para realizar la interacción, cuya posición en TF será ofrecida normalmente por la Kinect.

La segunda parte corresponde a la respuesta que da el servidor al cliente tras recibir la petición. El *message* que devuelve es una cadena de caracteres con toda la información de la petición, para verificar que el servidor ha recibido la petición correctamente.

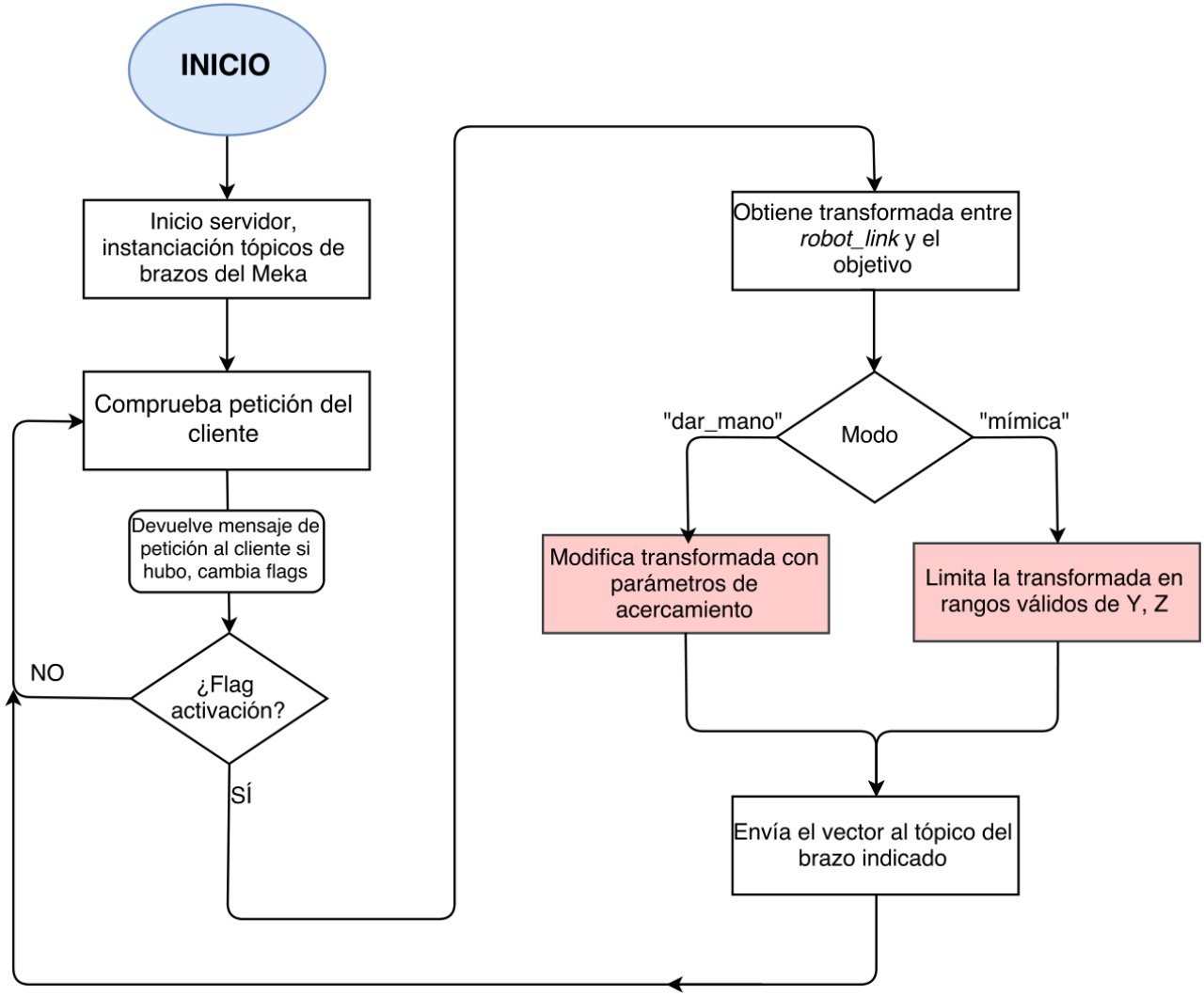

Figura 49: flujograma del script servidor de metas para el Meka

Modo “dar_mano”: un modo en el que el humano acerca la mano y el robot la sigue dentro de unos parámetros, como si diera la mano cerrando una en otra, similar a la Figura 50. Hay que tener en cuenta que la Kinect da los *frame* de la posición de las manos como un espejo, por lo que *left_hand* es realmente la mano derecha de la persona. Los parámetros de proximidad son modificables desde un *roslaunch* con las unidades en metros.

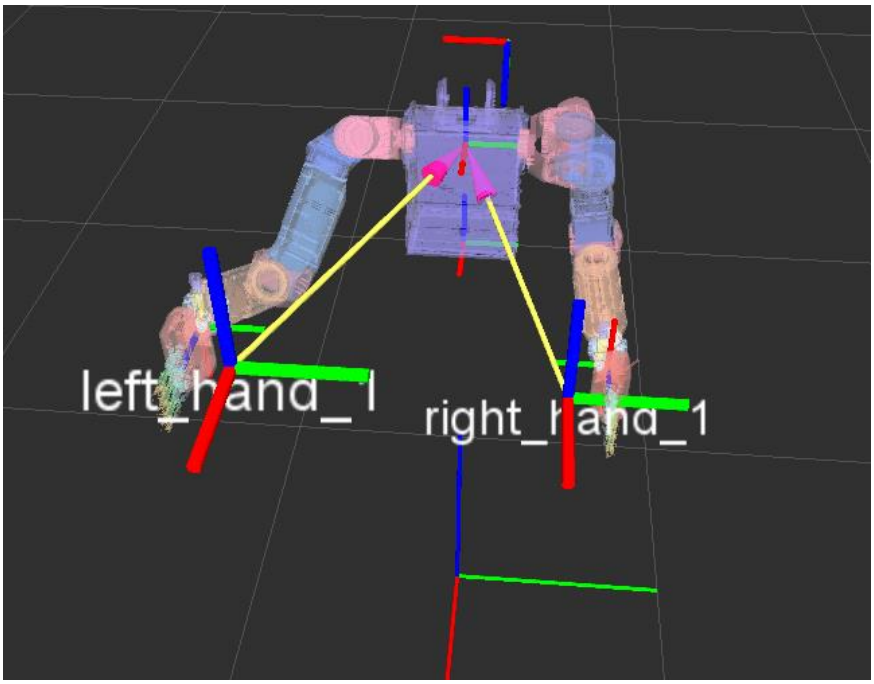

Figura 50: Meka en el modo "dar_mano"

Modo “mímica”: en este modo, el brazo indicado sigue al *frame*, normalmente la mano, dentro de un área especificada, donde el parámetro “X” (lejanía del humano al robot) es fija para evitar colisiones propias y con el humano. Las coordenadas que imita el brazo del robot son “Y” (distancia lateral) y “Z” (altura).

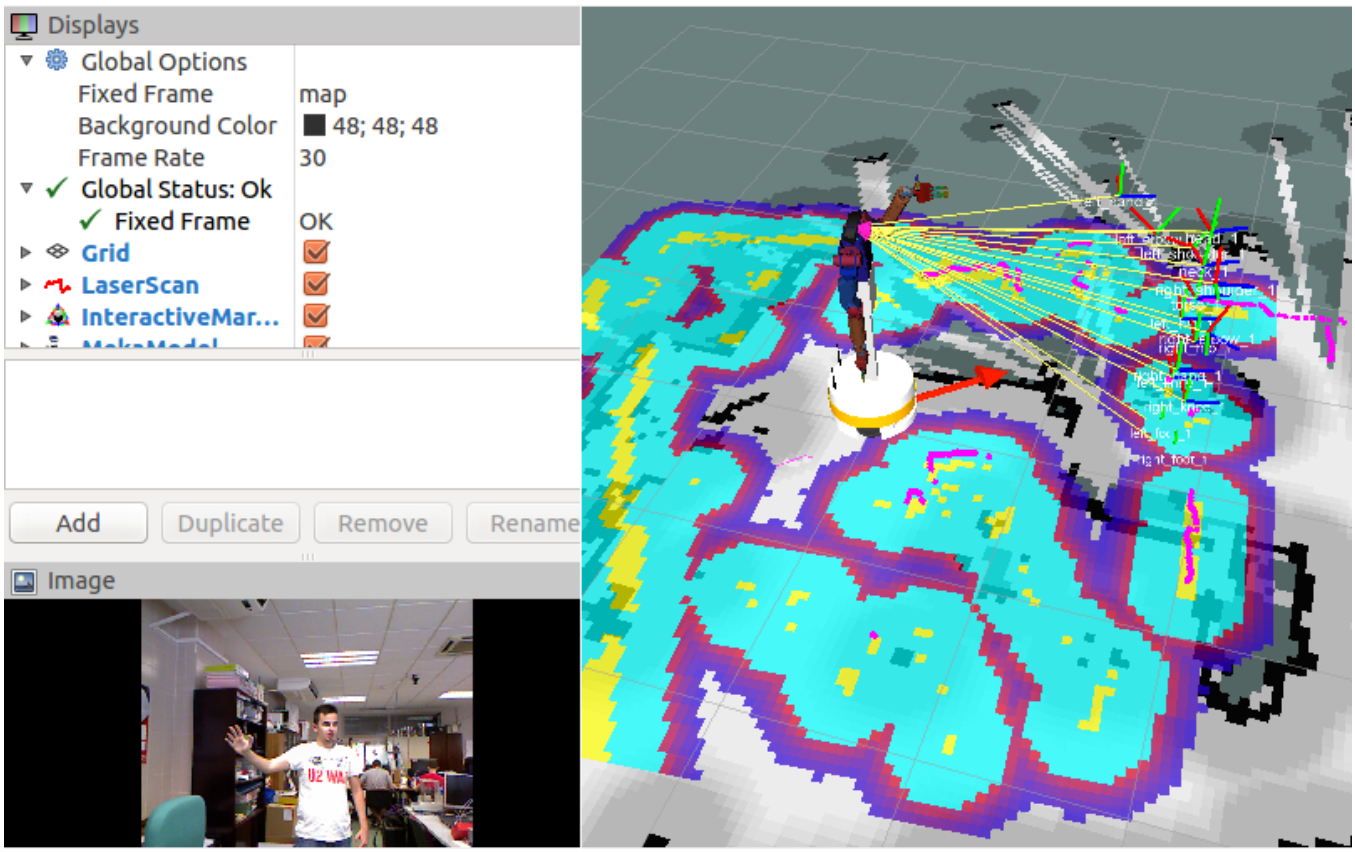

Figura 51: Rviz con PMB2 y Meka en modo "dar_mano", además de esqueletización (derecha)

En la Figura 51 se puede observar la representación de ambos robots:

- PMB2 con la navegación activada (ya que los objetos detectados por el láser están señalados de color amarillo y azul sobre el mapa global).
- La cámara Kinect representando la esqueletización (de la persona de la imagen inferior izquierda) referenciada justo encima del Meka. Dicha persona se encuentra justo enfrente de ambos robots.
- Meka encima del PMB2 y moviendo su brazo izquierdo a la misma altura que la mano derecha de la persona, realizando la “mímica”.

5. Caso de estudio

En este punto de la memoria se pretende exponer un ejemplo práctico (también llamado caso de estudio) de una situación real en la que todas las utilidades anteriormente explicadas y desarrolladas, y que van encaminadas a actuar sobre un entorno parcialmente desconocido utilizando información suministrada por sensores externos, se vean empleadas de forma conveniente. Para ello, se colaborará estrechamente con el proyecto de fusión sensorial [1], fijándose el procedimiento y las pautas a realizar para preparar el escenario de trabajo, así como las tareas concretas que ayuden a ejemplificar los objetivos perseguidos.

Los algoritmos desarrollados no tendrían sentido si no tuviesen una aplicabilidad y objetivos específicos. Para ilustrarlos y facilitar su comprensión, se propone el siguiente caso de estudio, cuyo objetivo principal es ***"detectar a una persona en un área predefinida y movilizar la plataforma robótica hasta su posición para que la identifique e interactúe con ella"***.

El primer paso para definir el caso de estudio es describir adecuadamente el entorno de trabajo, que será un área despejada, de aproximadamente doce metros cuadrados de superficie, en la que tanto robot como personas podrán moverse libremente; se procurará mantener una iluminación para el correcto funcionamiento de las cámaras, así como evitar obstáculos indetectables para la navegación autónoma del PMB2 como los mencionados en el apartado 3.5 que pondrían en riesgo su integridad.

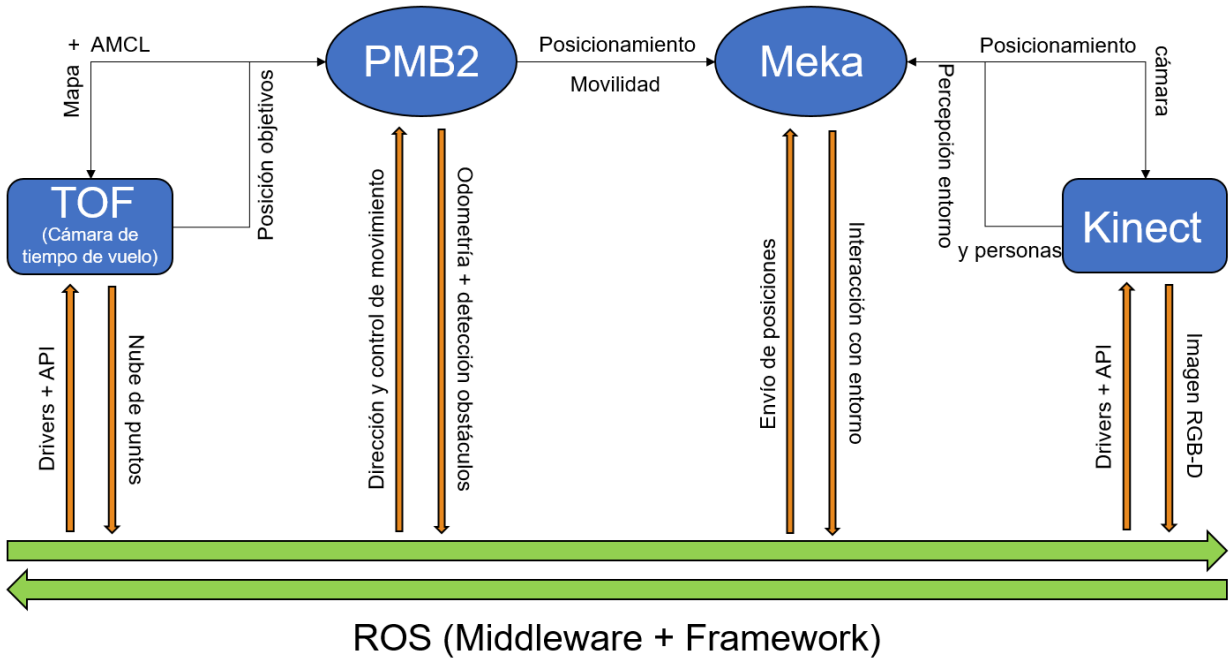

Figura 52: plataforma robótica y sensorial

Los elementos tratados en el presente trabajo son el robot móvil PMB-2, el robot humanoide Meka (y soporte para el sensor **Kinect**) se encuentran formando un único bloque móvil; por otro lado, la cámara **TOF** (*Time of Flight* o tiempo de vuelo), en situación de reposo y fija a una parte de la sala, debe ser capaz de detectar la presencia de personas en el mayor rango posible, con lo que se sitúa en una esquina de la escena, montada sobre un trípode y con un sensor **IMU** convenientemente colocado sobre ella.

Sus relaciones se encuentran descritas en el esquema de la Figura 52, resume brevemente todos los objetivos que deben cumplir cada uno de los elementos con todos sus elementos principales y el flujo de información entre ellos: el PMB2 mapea y proporciona movilidad, el Meka interacciona con el entorno/personas, la cámara TOF posiciona humanos y la cámara Kinect identifica y posiciona humanos a menor distancia, con mayor exactitud.

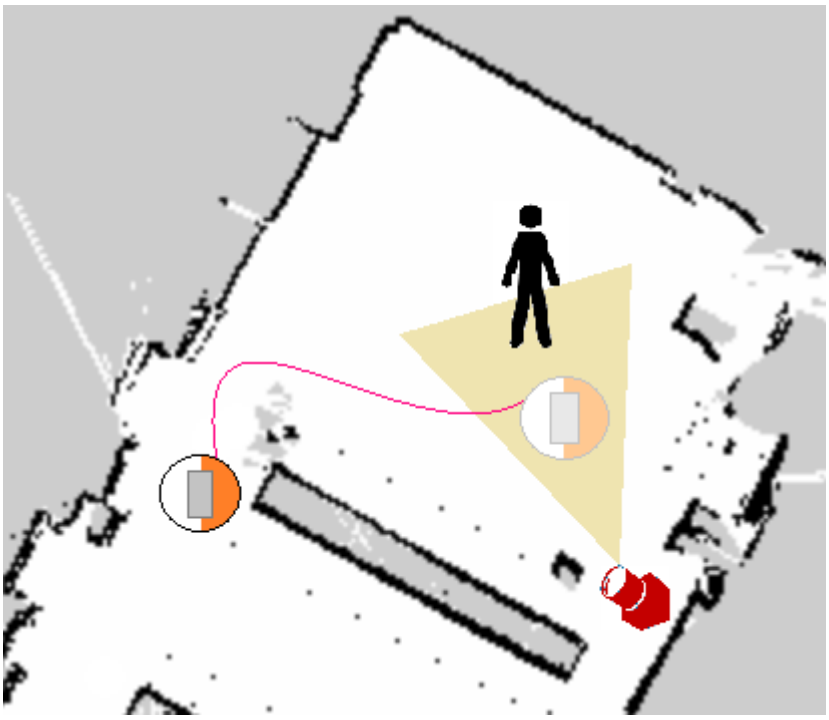

Figura 53: situación en el entorno de trabajo. Robots (círculo), persona y cámara TOF

En la Figura 53 está representado el entorno de trabajo, con la plataforma robótica representada por el círculo, una persona en la escena y la cámara TOF. El haz que en el que la cámara TOF es capaz de detectar una persona está representado por el triángulo amarillo (aunque su rango de acción llega hasta los límites del mapa para la autocalibración). Una vez la persona entra en el rango de detección y se para, se manda una posición cercana a ella a la navegación autónoma del PMB2. Luego será calibrada e identificada por la Kinect, pasando a la interacción con el Meka. Finalmente, vuelve a una posición determinada en el mapa lejos de la zona de detección, para reiniciar el proceso.

5.1. Asistencia al proyecto de fusión sensorial

El requerimiento básico para que el presente trabajo funcione en conjunción con el trabajo de la fusión sensorial [1], para el caso de los sistemas de percepción, es el de referenciar correctamente todos los elementos en el espacio, tanto posición como orientación (*pose* o 6D). Para ello, la herramienta más indicada sobre la que se apoya ROS es TF, como se explicó en el capítulo 3.4.3. Se realizó, por tanto, una asistencia al proyecto para proveer un correcto acoplamiento con la ayuda de los conocimientos desarrollados en la navegación autónoma: especialmente la localización y autoposicionamiento en un mapa determinado.

Para el caso de la cámara Kinect, el problema fue solventado en el capítulo 4.5, ya que está unida al Meka en su archivo URDF. Sin embargo, el caso de la cámara TOF es bastante más complejo. Dicha cámara se situará en una pared o en un trípode, con lo que una medición exacta 3D además de una correcta orientación de los tres ángulos de Euler (*roll, pitch* y *yaw* como se pueden observar en la Figura 54) respecto del mapa es prácticamente imposible, además de imprecisa y cambiante. El apartado de la orientación está cubierto mediante un sensor IMU colocado encima de la cámara TOF, del que la mayor parte de la información se encuentra en el otro trabajo, que trata todos los sensores mencionados y se encuentra adaptado a la plataforma ROS.

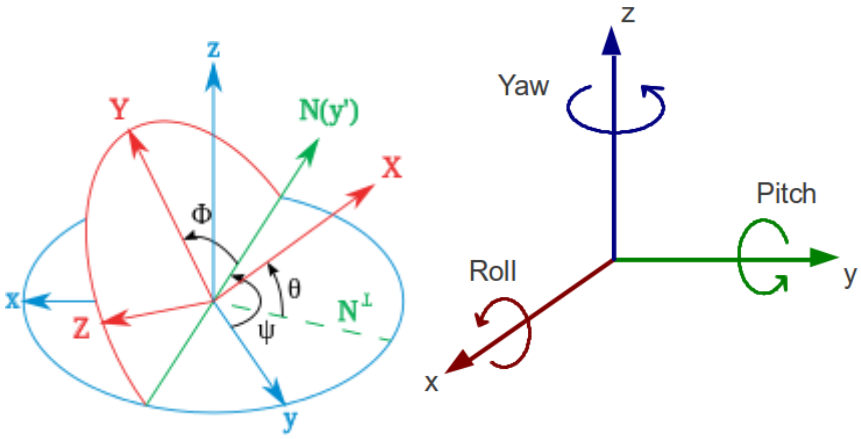

Figura 54: ángulos Euler renombrados como en la aeronáutica y robótica (*roll, pitch, yaw*)

Por lo tanto, se requiere de algo muy similar al caso del PMB2: autolocalización en un mapa creado a partir del entorno. Además, el mapa ha sido generado gracias a dicho robot, por lo que los parámetros como la altura del láser a la que se construyó el mapa son de vital importancia. Se ha de utilizar para ello el algoritmo **AMCL**, detallado en el apartado 3.4.4. Sin embargo, dicho algoritmo está ideado para el uso de LIDAR, no una cámara de profundidad. Gracias al paquete "***pointcloud_to_laserscan***", descargable desde los repositorios oficiales de ROS y que se puede instalar de la siguiente manera:

```
$  sudo apt-get install ros-indigo-pointcloud-to-laserscan
```

Dicho paquete ofrece su utilidad gracias al uso de los archivos de lanzamiento (*launch*), en el cual deben especificarse parámetros como el tópico desde el que se suministra el *PointCloud* de entrada, la región angular que se observará para realizar la conversión, el paso para el ángulo que se tomará como intervalo de muestreo de la nube de puntos (se intentará ajustar este parámetro para que el paso del ángulo

coincida con el que se recorre entre un píxel y otro de la imagen, teniéndose en cuenta que la cámara TOF cumple aproximadamente un ámbito de 45 grados y su imagen tiene una anchura de 176 píxeles) y la **altura** a considerar para efectuar la creación del *LaserScan*, la cual deberá especificarse para que tome valores similares a los de la altura respecto al suelo que tiene el láser del robot móvil PMB-2, teniéndose en cuenta que la cámara se encuentra situada a una cierta altura con respecto al suelo (aproximadamente 1.7m); esta medida se deberá conseguir de forma manual, con ayuda de una plomada (Figura 55).

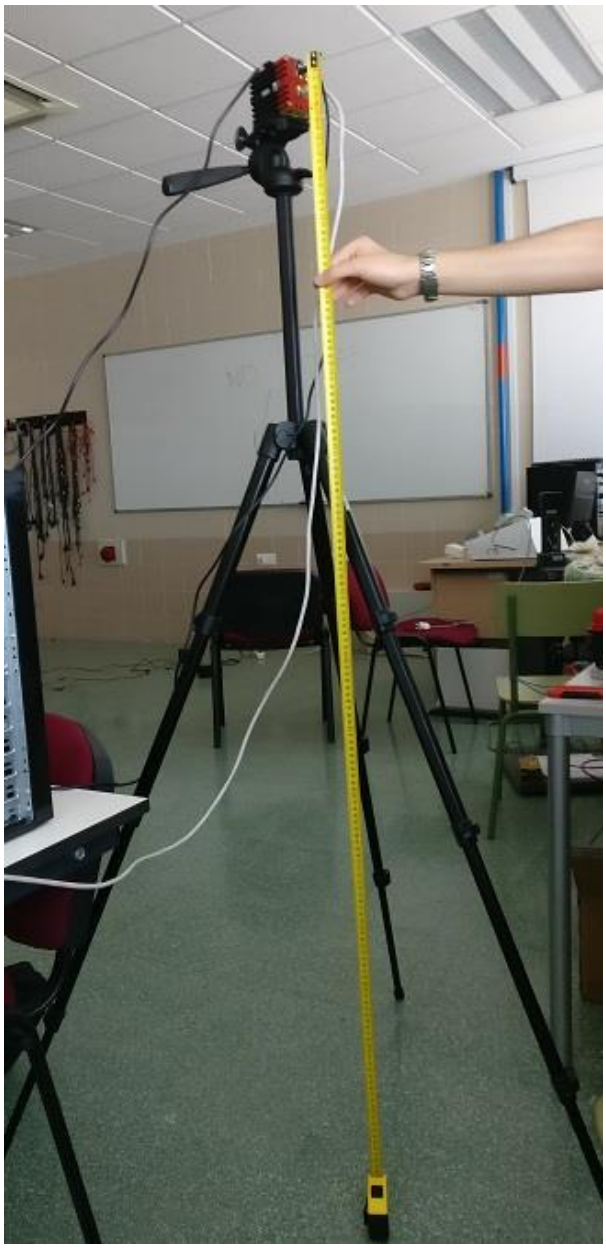

Figura 55: medición de la altura de la cámara TOF

Para solventar la carencia de odometría que necesita el algoritmo AMCL, se empleará una “falsa” odometría, aprovechando que el trípode sobre el que va montado la TOF permite efectuar giros en cualquiera de los tres ángulos de Euler. Los ángulos *roll* y *pitch* adaptarán correctamente la dirección de la nube de puntos, medidos gracias al sensor IMU, así que no son incluidos en la generación de las coordenadas TF. Se crea, según los requerimientos del algoritmo AMCL, el siguiente árbol de transformadas TF espaciales (el nombre de los *frame* ha sido reemplazado para evitar incompatibilidades con los presentes en el PMB2):

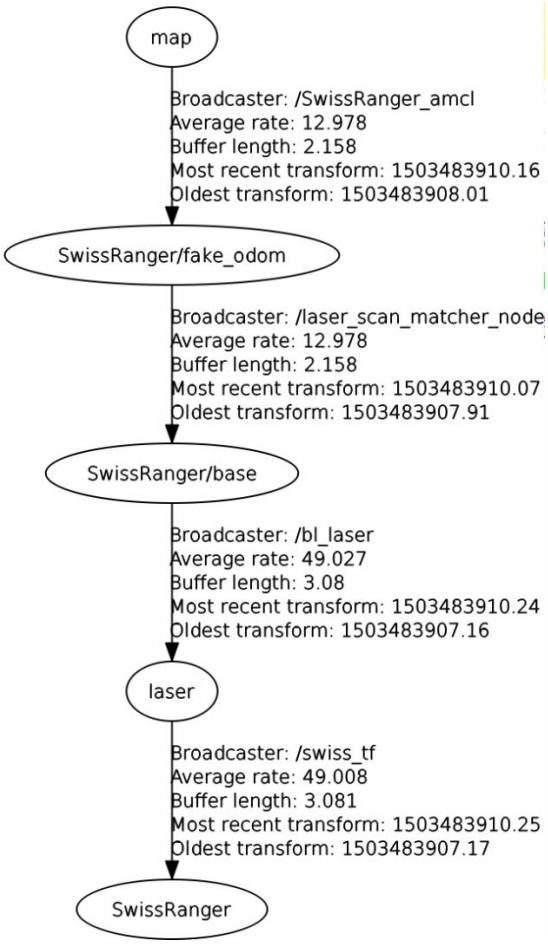

Figura 56: árbol de TF para autolocalizar cámara TOF con algoritmo AMCL

Para ejecutar la localización e iniciar la calibración de la posición de la cámara TOF en el mapa, se ejecutan los siguientes nodos

- Nodos que ejecutan la cámara TOF, el sensor IMU y proporcionan la nube de puntos, especificados en el proyecto de fusión sensorial [1].

- *Map_server:* envía el mapa creado por el robot PMB2
- *Pointcloud_to_laserscan:* convierte la nube de puntos en láser
- *Laser_scan_matcher_node:* proporciona una odometría sólo con el láser, aunque mucho más imprecisa al no depender de una odometría real.
- *SwissRanger_amcl:* algoritmo AMCL con nombre diferente al que usa por defecto
- *Bl_laser* y *swiss_tf:* nodos tipo *static_transform_publisher* que completan el árbol de TF mostrado en la Figura 56.
- *Rviz:* herramienta de visualización, con los tópicos renombrados para que no interfiera con la navegación y autolocalización de los robots.

Con dichos nodos en funcionamiento, se deberá indicar una posición o "pose" inicial de la cámara en el mapa, que debe ser tal que coincida en la medida de lo posible con su posición real (para ello, se debe comprobar la correspondencia entre los límites del mapa y el *LaserScan* creado por la cámara TOF) y **rotarla** manualmente sobre el eje Z (ángulo ***yaw***) para que el algoritmo de autolocalización vaya ajustando la posición de la TOF como en la Figura 57.

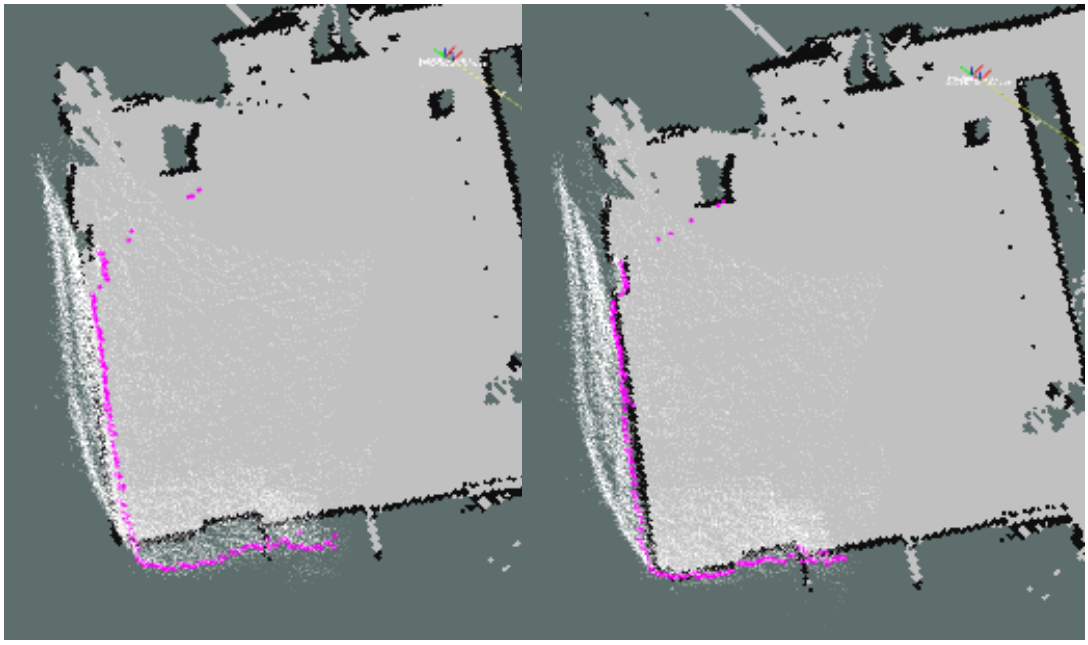

Figura 57: autolocalización de la cámara TOF. Posición inicial (izq) y autocorrección final (der)

Esta posición se guardará en un fichero para ser cargada cada vez que se desee utilizar la cámara como elemento de referencia para localizar a personas dentro del entorno parcialmente desconocido en el cual se esté trabajando. Dicho proceso se encuentra descrito en el apartado 5.2.

Finalmente, y aunque resulta obvio, es conveniente destacar que este procedimiento sólo deberá realizarse cada vez que la cámara o el trípode sean desplazados, por lo que se procurará, en la medida de lo posible, evitar que eso ocurra, salvo cuando sea estrictamente necesario.

5.2. Almacenamiento de archivo de calibración de la cámara TOF

Gran parte del proceso de calibración del posicionamiento de la cámara TOF respecto del mapa ha sido descrito en el apartado anterior. El archivo lanzamiento de los citados además del script para calibrar y guardar dicha calibración en un fichero de texto se encuentra en el paquete "***imu_camera***". El archivo de lanzamiento "*laser_scan.launch*" es el encargado de realizar el arranque de los nodos citados para ejecutar la calibración.

Partiendo de una calibración correcta como en la segunda imagen de la Figura 57, se debe realizar las siguientes tareas:

- Guardar la transformada entre *map* y *SwissRanger/base* en un archivo de texto.
- Apagar el funcionamiento de los nodos para incrementar la eficiencia de las demás operaciones.
- Utilizar dicha transformada para emitirla posteriormente, entre *map* y *SwissRanger*, con lo que la cámara estará posicionada correctamente.

Además, el script que realice estas tareas admitirá dos modos desde el lanzamiento: **guardar** calibración y **leer** calibración. Así, el arranque normal será el de leer la calibración. El flujograma del programa llamado *tof_calibracion_map.py (*con

un archivo de lanzamiento asociado llamado *start_tof.launch* donde se puede especificar el modo), es el siguiente:

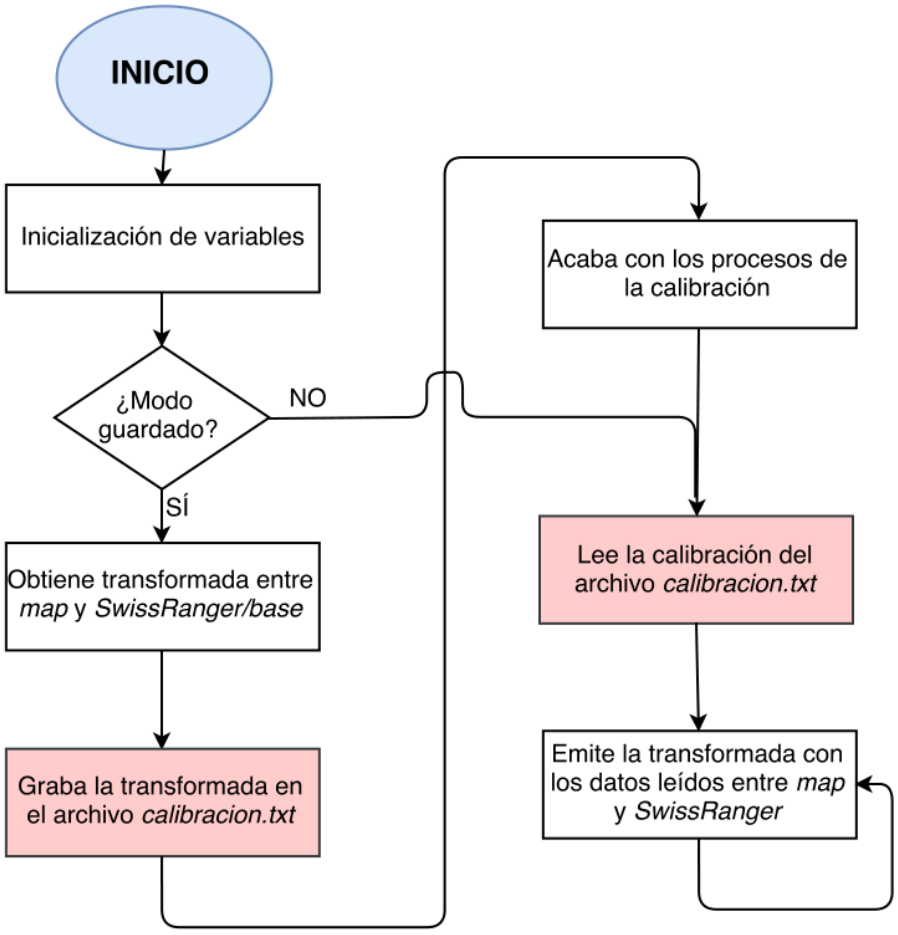

Figura 58: proceso de guardado/carga de la calibración de la cámara TOF

Para realizar las tareas de guardado del archivo de texto se usa la librería *pickle* de Python, teniendo en cuenta que los archivos se deben guardar/cargar de la carpeta *data* del paquete donde se encuentran todas las funcionalidades de este apartado. La librería *Pickle* identifica el tipo de dato guardado, ya sea un vector o una cadena de texto, con lo que facilita la lectura de datos. Se guarda la transformada (7 números tipo *Float* de la transformada: 3 para la posición y 4 para el *Quaternion* u orientación), aunque no es legible si se abre el archivo, con lo que sólo se puede leer mediante el script.

5.3. Adaptación de posiciones humanas para una navegación adecuada del robot

Como se ha mencionado al inicio del presente capítulo, la cámara TOF facilita las posiciones de los humanos, ya que los detecta por su contorno y determina su distancia. Una vez calibrada (posicionada) correctamente respecto del mapa, la cámara ofrece la posición mediante un *frame* en TF llamado *person_tf_0, person_tf_1*… sucesivamente según aparecen más personas en la escena. Aunque el *frame* ofrece correctamente la posición de la persona, no ofrece datos de orientación, por lo que por defecto tienen la misma orientación que el *frame* de la cámara (Figura 59), ya que no tiene suficiente resolución [1].

Figura 59: cámara TOF detectando y posicionando personas mediante TF

Como punto de referencia sobre el cual realizar la navegación autónoma de la plataforma robótica y orientarla hasta la persona detectada, se utilizará la propia cámara. Para ello, se crea un nuevo *frame* llamado "***carrot_X***" (la X se sustituye por el número) respecto de cada *frame* de una persona, que usa la posición dada por la TOF y se oriente en plano XY hacia la cámara. Además, también calcula la velocidad lineal y se publica por un tópico llamado *"**human_X_speed**"*, para saber si la persona se ha parado o sigue moviéndose.

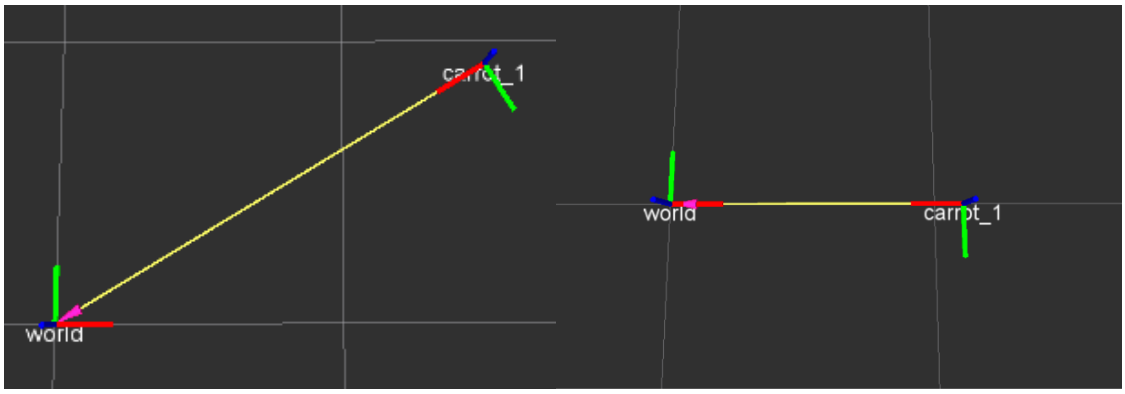

Figura 60: ejemplos de cómo se orienta el *frame* creado hacia uno de referencia

Para orientar correctamente, se utiliza la *Ecuación 2*, que da como resultado el ángulo *yaw* (Euler) de la transformada.

$$yaw = \tan^{-1}\left(\frac{y}{x}\right)$$

Ecuación 2

El cálculo de la velocidad lineal se realiza mediante una función incluida en la librería *tf*. Para gestionar varios humanos en escena, se parametrizan los nombres de los frame, con lo que se lanza el nodo tantas veces como humanos pueda haber. En el archivo de lanzamiento *vel_tof.launch* se gestiona el lanzamiento para 3 personas.

Para comprobar el funcionamiento, se ha creado también el archivo de lanzamiento *vel_turtle.launch*, con lo que en Rviz se puede observar el funcionamiento del TF generado que se orienta y ofrece capturas como las presentes en la Figura 60.

5.4. SMACH: máquinas de estados finitos

Con todos los sistemas conectados, referenciados y todos los *scripts* mencionados en la memoria hasta ahora ejecutándose, falta por realizar el programa que gestione el objetivo final. El flujograma de las tareas correspondientes a cumplir es el siguiente, donde el número de intento se traduce en intentos de navegación hasta la persona: si falla todos, termina la operación.

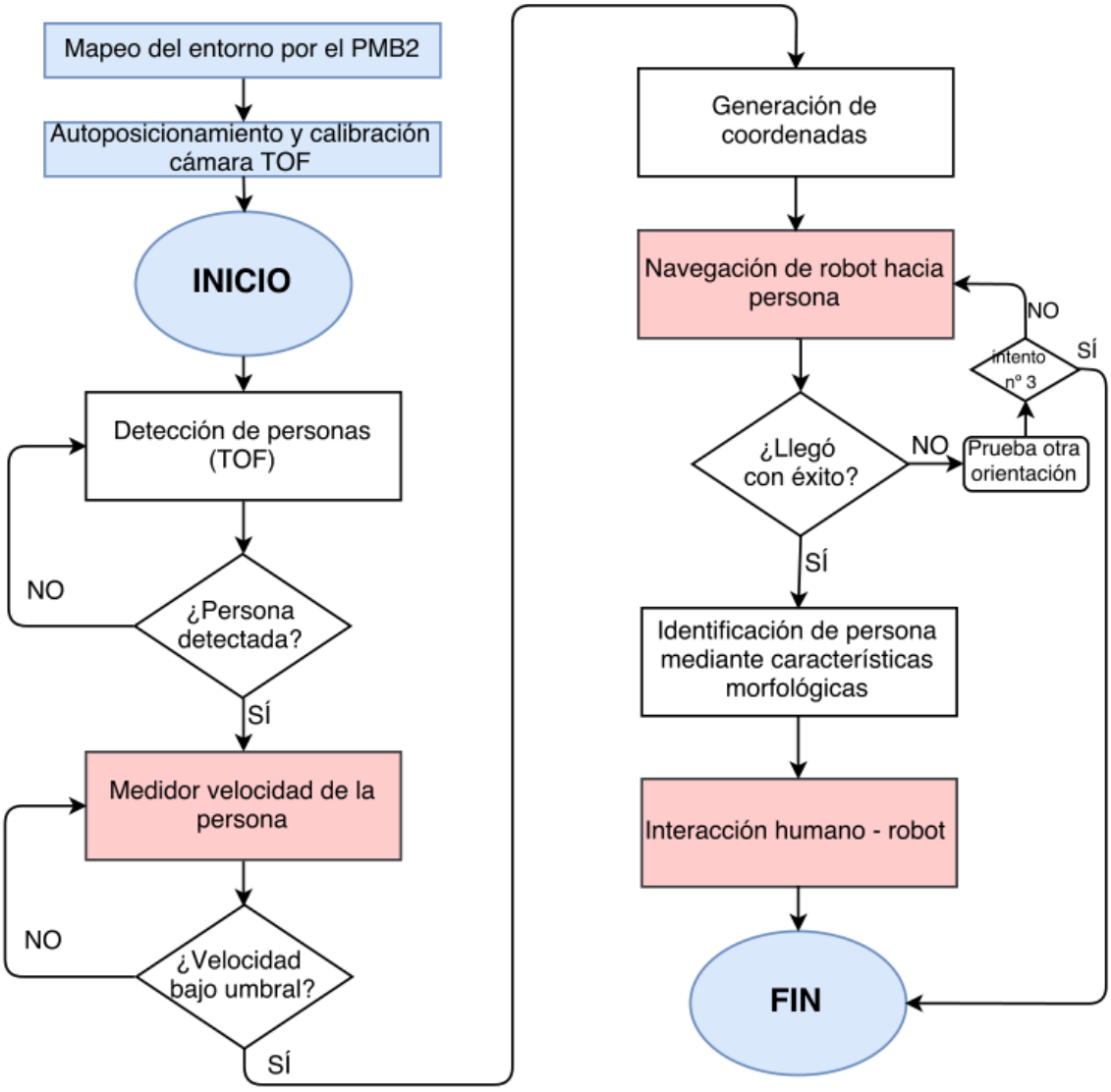

Figura 61: flujograma para alcanzar el objetivo del caso de estudio

Antes de realizar dichas tareas, requiere mapear el entorno gracias al SLAM programado para el PMB2, calibrar la posición de la cámara TOF respecto a dicho mapa y entrenar la red neuronal de identificación de personas del proyecto de fusión sensorial [1]. Para guiar a la persona en el proceso, se dotará al robot de voz artificial.

Voz robótica

Los sonidos son gestionados en ROS mediante el paquete "***sound_play***", que incluye una voz en inglés. Para instalar el paquete además de la voz en español en el robot PMB2, se ejecuta:

```
$ sudo apt-get install ros-hydro-sound-play
$ sudo apt-get install festvox-ellpc11k
```

Dicho nodo actúa como un servidor tipo *actionlib*, por lo que enviar una frase es tan sencillo como enviar una petición en un cliente. En el paquete "***demo_smach***", dentro del script *smach_funciones_demo1.py* se encuentra una función llamada <<habla(frase, espera)>>, que acepta una cadena de caracteres como parámetro de entrada para decirla en español, además de una espera en segundos.

Máquina de estados finitos

El objetivo principal engloba muchas tareas de alto nivel de programación, como son la navegación autónoma, el reconocimiento y posicionamiento de personas, movimiento de un brazo robótico… Todos los procesos ya están programados para recibir órdenes sencillas, con lo que queda unirlas todas en un sólo proceso (Figura 61).

Para gestionar procesos complejos que involucren el accionamiento de muchos sistemas en conjunción que sigan un plan determinado se definen las máquinas de estados finitos [31]. Al ejecutar un proceso completo, se pueden definir todos los estados posibles y sus transiciones explícitamente, desgranando el proceso en tareas sencillas y ofreciendo modularidad. Existe una librería de *Python* para ejecutar tareas de alto nivel llamada **SMACH**, diseñada para coordinar las tareas en "contenedores de estado". Un contenedor de estado puede ser incluso otra máquina de estados finitos, con lo que se van englobando diferentes tareas posibilitando realizar acciones complejas para un robot como abrir una puerta, enchufarse a sí mismo a la corriente eléctrica, etc.

La librería incluye una herramienta útil de visualización y depuración, para observar el estado actual en el que se encuentra y el resultado que produce. Si un archivo programado con SMACH incluye la conexión con un visor y está activo, se puede observar su paso por las diferentes etapas.

Para completar el objetivo marcado realizando las tareas de la Figura 61, se ha realizado el script *final_demo_smach.py*, el cual incluye el uso de:

- Esperar la aparición de una persona escuchando un tópico que pasa el número de personas presentes.

- Script que orienta y calcula la velocidad media de la persona, esperando a que ésta se pare para enviar a la plataforma robótica.
- Navegación autónoma (envía metas gracias al paquete *smach_ros*)
- Escucha varios tópicos con información del sensor Kinect [1].
- Interactúa el Meka con la persona usando el *server_meka* creado para que haga mímica o de la mano.
- Emite frases para guiar al humano en todo el proceso.

El resultado de la máquina de estados en SMACH es visible en la Figura 62.

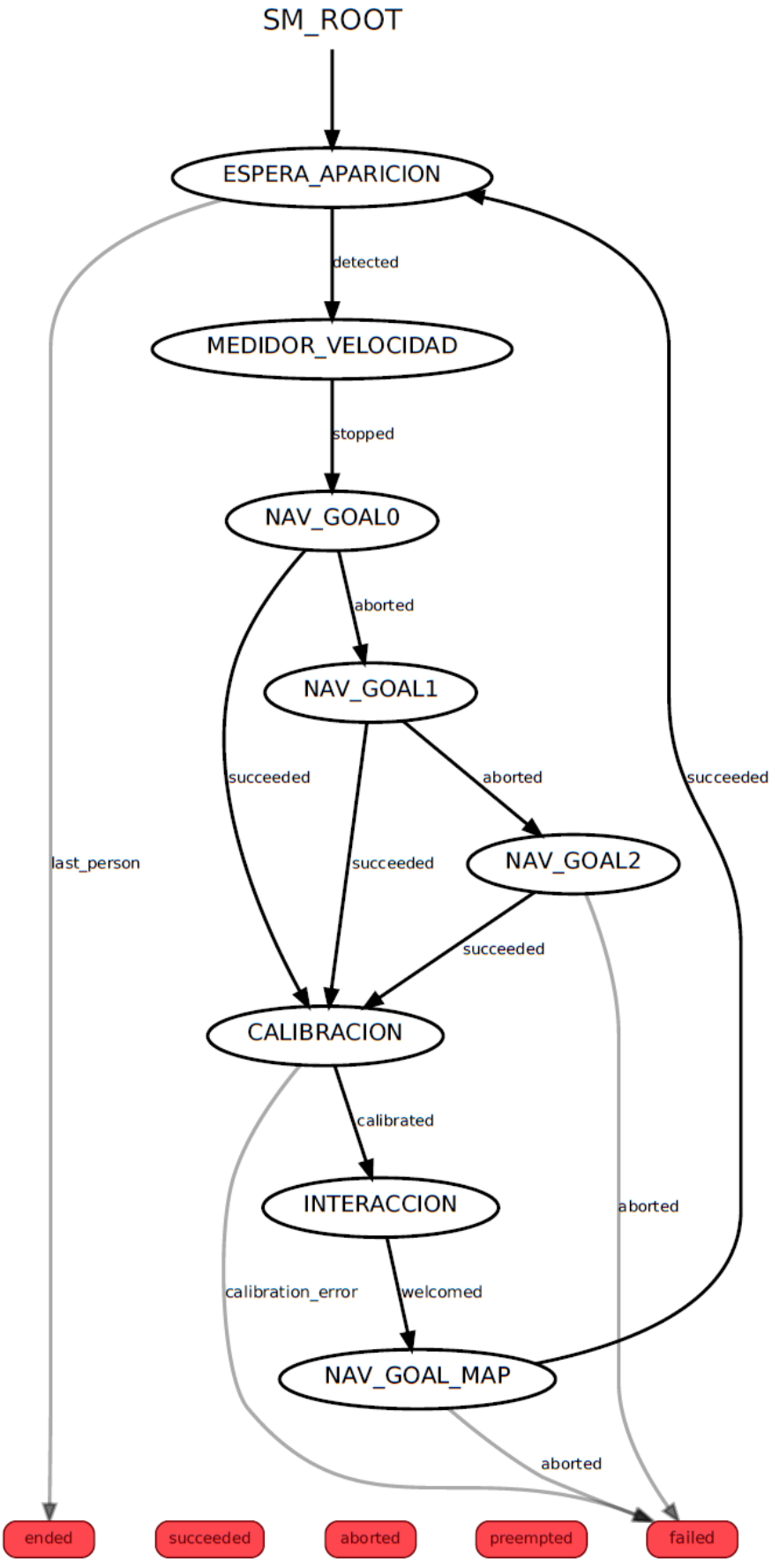

Figura 62: SMACH del caso de estudio con el visor *smach_viewer*

5.5. Conexión y alimentación de la plataforma robótica

La plataforma robótica es la unión del presente proyecto y el de fusión sensorial [1] trabajando coordinadamente. Debido a condiciones anteriormente planteadas, la conexión y alimentación de todos los sistemas sigue el siguientes esquema:

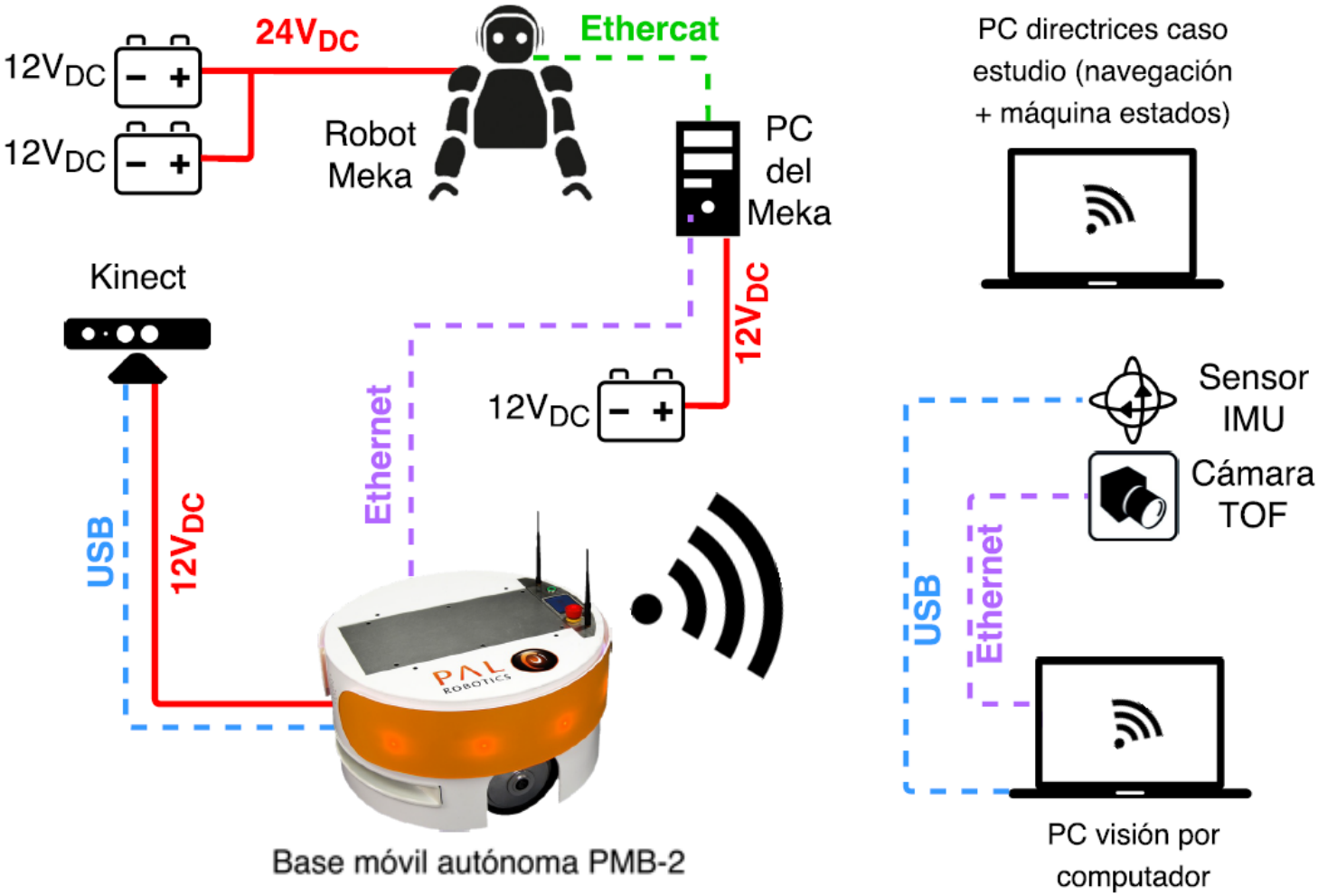

Figura 63: esquema de conexión física y alimentación de la plataforma

De un simple vistazo (Figura 63), se puede observar que el centro de conexiones es la base móvil autónoma PMB-2: conecta por Ethernet al ordenador que controla el robot Meka y por USB a la cámara Kinect, además de ofrecer alimentación a la cámara mencionados gracias a su batería de gran capacidad. Por otro lado, el robot Meka se alimenta con dos baterías en serie de 12V cada una (equivale a 24V); su ordenador requiere una batería de 12V independiente, ya que no era suficiente la salida del PMB2 por la caída de tensión que se producía al alimentar el PC.

La plataforma se conectará mediante conexión inalámbrica WiFi a los demás ordenadores portátiles utilizando su punto WiFi (aunque también puede conectarse a una red externa si es preciso), permitiendo movilidad total.

La conexión Ethernet o WiFi es indiferente en las redes, de acuerdo con el modelo OSI (*Open System Interconnection* o modelo de interconexión de sistemas abiertos [ISO/IEC 7498-1]). Es el protocolo de red de arquitectura en capas, creado en 1980 por la Organización Internacional de Normalización (ISO), que al separar la capa física de las demás permite que la conexión sea indiferente, por lo que la asignación de IP es idéntica, tanto por Ethernet como por WiFi.

Por otro lado, la jerarquía en ROS es diferente. Con todos los equipos conectados a la misma red, es indistinto cuál de los equipos ocupa el lugar de maestro y tendrá en ejecución el *rosmaster*. Según información oficial de ROS [2], se debe elegir el sistema con mayor poder de computación que esté más libre de carga. Ya que el ordenador del Meka debe mantener el lazo cerrado de 1kHz y los portátiles cumplen tareas de virtualización (utilizado en el presente proyecto) o visión por computador, se utilizará el ordenador alojado en el interior del PMB2, ya que coincide con el centro conectivo.

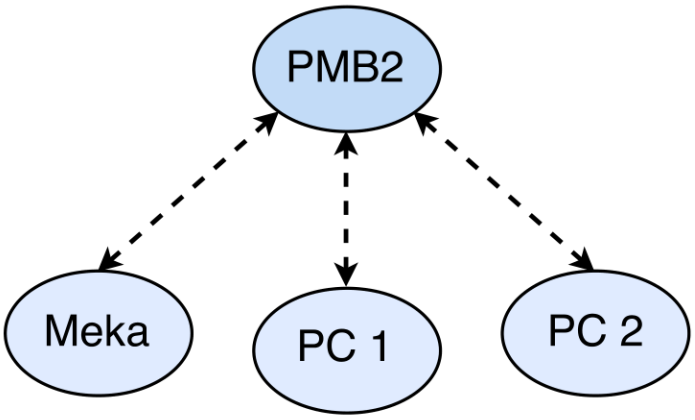

Figura 64: robot PMB2 como maestro ROS de la plataforma

Gracias al software de Pal-Robotics, el robot lanza al arranque el *rosmaster* se puede cambiar editando el archivo, con lo que será el primer equipo que se debe conectar. En el caso de querer utilizar otro ordenador como *rosmaster*, se debe editar un archivo de arranque con permisos de administrador (los primeros comandos son para hacer los cambios persistentes en la memoria no volátil del sistema):

```
# rw
# chroot /ro
# nano /opt/pal/cobalt/bin/pal_startup/launch/palRosSetup.sh

  ###Se añaden las siguientes líneas al final del archivo:
export ROS_MASTER_URI=http://<IP del ordenador>:11311
export ROS_IP=<IP del ordenador>
```

Una vez todos los ordenadores exportan las variables de su propia IP en la red utilizada para conectarse a la plataforma y la IP del maestro, es posible la ejecución de todos los algoritmos. Para cumplir con el flujograma (Figura 61), el procedimiento es el siguiente:

- Entrenamiento red neuronal para identificar personas.
- Mapear el entorno.
- Calibrar la posición de la cámara TOF respecto del mapa y guardarla.
- Iniciar plataforma con archivos de lanzamiento alojados en el PC director:
    - Navegación en PMB2 (mapa + AMCL + *move_base*).
    - Carga del nuevo URDF (Meka+Kinect) en el servidor de parámetros y lanzamiento del nodo unión .
    - Arranque del servidor EtherCAT del Meka junto con los nodos que lo controlan.
    - Iniciar *openni_launch* y *openni_tracker2* a distancia en el PMB2, que gestionan la Kinect. También el nodo *sound_play* para que se escuche en el altavoz colocado en el PMB2.
    - Lanzamiento de la posición de la TOF respecto del mapa.
    - Inicialización de procesos requeridos para que funcione el caso de estudio:
        - Nodo orientador de *frame* y medidor de velocidad
        - Servidor de interacción del Meka
- Lanzamiento de programas del proyecto de fusión sensorial:
    - Lanzamiento cámara TOF junto con el sensor IMU
    - Visión por computador de la cámara TOF para detectar personas por su silueta
    - Red entrenada que identifique a los usuarios
- Finalmente, el script con la librería SMACH que dirige para cumplir con el objetivo del cao de estudio.

6. GUI: interfaz gráfica para controlar los comandos del presente proyecto

Para operar todos los comandos y diferentes paquetes del presente proyecto, debido a la gran cantidad de paquetes y nodos que se necesitan en ejecución, se propuso realizar una GUI (*Graphical User Interface*) o interfaz gráfica de usuario. Esta se encargará de ejecutar los comandos necesarios para poder realizar el caso de estudio con ambos robots (Meka tanto simulación como real).

6.1. GLADE

Herramienta de asistencia a la creación de interfaces gráficas. Aunque la interfaz gráfica se puede realizar directamente desde código, lo más sencillo es diseñarla con otro programa, ya que permite posicionar los elementos fácilmente. Para ello se ha utilizado GLADE [33], una interfaz de diseño especialmente pensada para GNOME (entorno de escritorio muy utilizado en Linux).

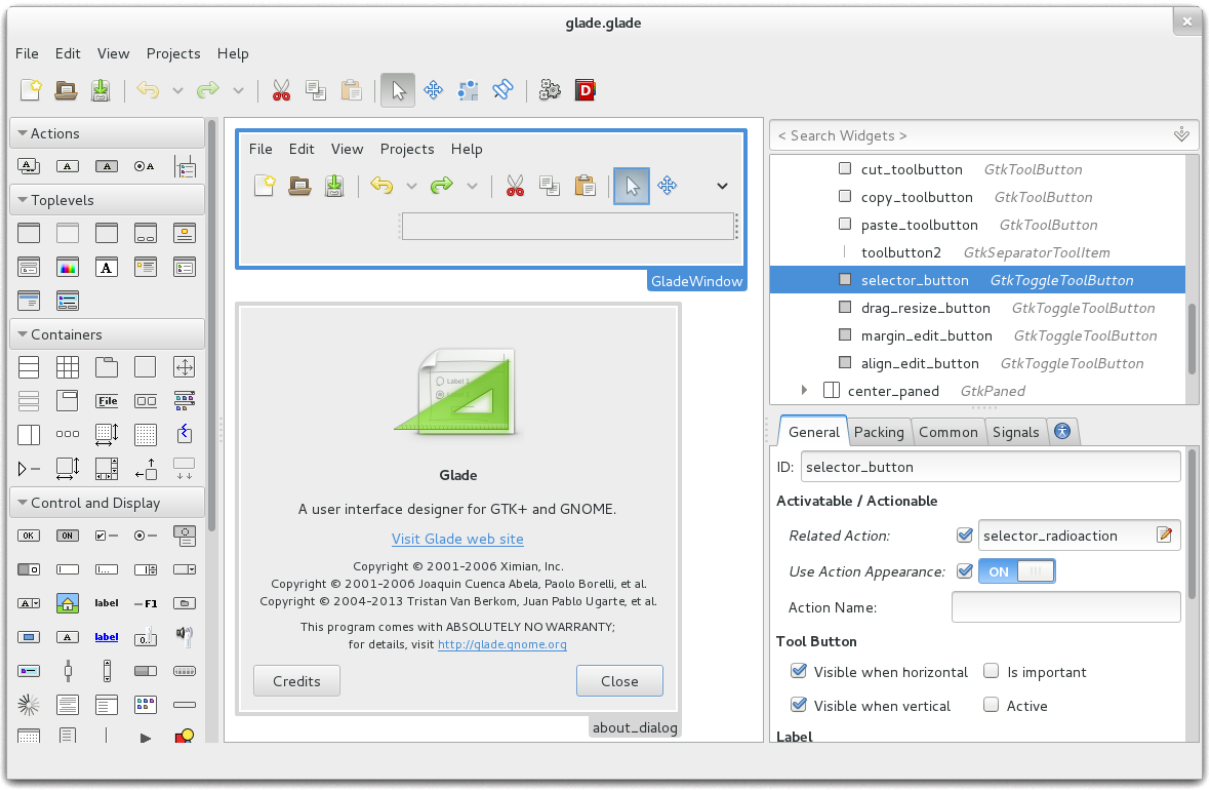

Figura 65: captura de pantalla del programa GLADE

Está planteada para ser utilizada especialmente por la librería GTK, comentado en el siguiente apartado. Lo más importante es definir las señales "*clicked*" en los

botones de la GUI, que luego se utilizarán en el archivo conveniente para definir las funciones asignadas.

GLADE crea un archivo tipo XML, cuyo archivo será referenciado por el *script* que ejecute el programa creado.

6.2. GTK

GTK [34] es una librería multiplataforma para crear interfazces gráficas, la cual pued ser utilizada por multitud de lenguajes de programación: C, C++, Perl y Python. Con un sencillo comando se puede importar la interfaz gráfica diseñada con GLADE y asignar función a los botones.

En el presente trabajo, ambos archivos (script en Python y archivo XML creado por GLADE) se encuentran en la ruta del paquete "platform_bringup/gui", que cumplen las siguientes funciones, divididas en pestañas:

Figura 66: pestaña de arranque de la GUI creada

La primera pestaña tiene dos botones y un seleccionador de modo:

- Botón de "Arranque básico": inicia elementos del PMB2 como la cámara Kinect, el servidor de sonido y sincronización de relojes.
- Seleccionador "Simulador" o "Robot real": Permite seleccionar entre el simulador o el control de los brazos robóticos.
- Botón de "Inicio Meka": a partir de la seleección anterior, ejecuta los nodos necesarios. En el caso de robot real, se debe ejecutar en el ordenador del

Meka mediante *ssh* el comando "*m3rt_server_run -m*" para inicar el servidor maestro de EtherCAT.

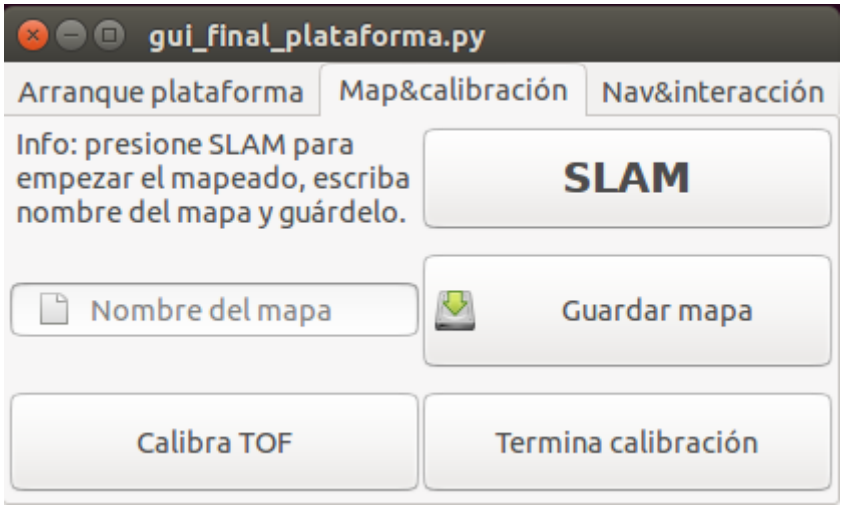

Figura 67: pestaña de mapeado y calibración de la GUI creada

La segunda pestaña está pensada para la primera toma de contacto con el entorno, donde se debe mapear (SLAM), guardar el mapa y calibrar la posición de la cámara TOF:

- Botón "SLAM": ejecuta el mapeado.
- Entrada de texto "Nombre de mapa": permite introducir el nombre del archivo de mapa que se creará.
- Botón "Guardar mapa": una vez mapeado el entorno, este botón utiliza el nombre escrito en el apartado anterior para guardar el mapa. Además, cierra los nodos de mapeado.
- Botón "Calibra TOF", que ejecuta los nodos de calibrado de la posición de la cámara TOF respecto del mapa creado. Dichos nodos se listan en el apartado 5.1.
- Botón "Termina calibración": guarda la calibración ejecutando el programa detallado en el apartado 5.2.

Figura 68: pestaña para iniciar navegación y aplicación final de la GUI creada

La última pestaña permite ejecutar las demás funciones:

- Botón “Cargar mapa”: lanza una nueva pestaña para seleccionar el archivo del mapa de forma gráfica.
- Botón “Inicia Navegación Autónoma”: utilizando el mapa cargado anterior o uno por defecto.
- Botón “Aplicación final”: inicia todos los programas necesarios para ejecutar el caso de estudio satisfactoriamente.

Todos los botones ejecutan archivos de lanzamientos utilizados anteriormente sin necesidad de comandos. El programa de la GUI también comprueba si algunos nodos están funcionando, para actuar en consecuencia y evitar errores de lanzamiento simultáneo.

7. Conclusiones y perspectivas futuras

Se han podido completar la mayoría de objetivos listados en el capítulo 1. Ambos robots están conectados entre sí, tanto física (conexión y alimentación) como computacionalmente (ROS y referencias en URDF). También se han incluído sistemas de percepción de otro proyecto, lo que abre la puerta a la inclusión en un entorno con más sensores.

Se ha explorado y desgranado el funcionamiento de la navegación autónoma, con lo que se puede mejorar la navegación autónoma y el mapeo.

Además, se ha conseguido ejecutar con éxito el caso de estudio, con la posibilidad de manejarlo con la interfaz gráfica creada, para mayor facilidad del usuario final.

Una de las mayores motivaciones de implementar todos los sistemas en ROS es tenerlos conectados con conectividad entre ellos. Esta plataforma está pensada para un futuro de desarrollo e investigación con los equipos mencionados durante todo el trabajo. Además, al explorar el funcionamiento de la navegación autónoma, se ha observado que se podría desarrollar software que cambie las características de la navegación tras el mapeo del medio según las carecterísticas de este, mejorando la integración en diferentes entornos.

Bibliografía

Fernández, E. (2015). *Learning ROS for Robotics Programming Second Edition.* Editorial Packt Publishing.

Quigley, M. (2015). *Programming robots with ROS.* Editorial O'Reilly Media.

Joseph, L. (2015). *Mastering ROS for robotics programming.* Editorial Pack Publishing.

Joseph, L. (2015). *Learning robotics using Python.* Editorial Pack Publishing.

Mullane, J. (2011). *Random Finite Sets for Robot Mapping and SLAM*. Editorial Springer

Referencias

[1] Moral Parras, José Antonio (2017). *Fusión sensorial para realizar tareas robóticas en entornos parcialmente estructurados*

[2] Ros WEB: http://www.ros.org

[3] Fernández, E.; Sánchez, L.; Mahtani, A.; Martínez, A. (2015). *Learning ROS for Robotics Programming,* capítulo 1: "Getting started with ROS".

[4] Quigley, M.; Gerkey, B.; D., William (2015). *Programming Robots with ROS,* prefacio.

[5] Proyecto STAIR, Universidad de Stanford: http://stair.stanford.edu

[6] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 11: "ROS for Industrial Robots".

[7] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 1: "Introduction to ROS and its package management: Why we prefer ROS for robots".

[8] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 1: "Introduction to ROS and its package management: Why some do not prefer ROS for robots".

[9] Componentes básicos de ROS: http://www.ros.org/core-components/

[10] Núcleo de ROS: http://wiki.ros.org/roscore

[11] Mensajes de ROS: http://wiki.ros.org/msg

[12] Herramienta de visualización 3D Rviz: http://wiki.ros.org/rviz

[13] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 1: "Introduction to ROS and its package management: Understanding the ROS file system level".

[14] Lentin Joseph (2015). *Mastering ROS for Robotics Programming*, capítulo 1: "Introduction to ROS and its package management: Understanding the ROS computation graph level".

[15] Hernández, S.; Herrero, F. (2015). *Multi-Master ROS Systems.*

[16] Tópicos de ROS: http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

[17] Fernández, E.; Sánchez, L.; Mahtani, A.; Martínez, A. (2015). *Learning ROS for Robotics Programming,* capítulo 8: "Navigation Stack - Beyond Setups".

[18] Zhang, F.; Simon, C (2013). *Cumulative Error Estimation from Noisy Relative Measurements.*

[19] Foote, T (2013). *Tf: The Transform Library*.

[20] Thrun, S.; Burgard, W.; Foz, D. *Probabilistic Robotics.*

[21] Mullane, J.; Vo, B.; Adams, M. *Random Finite Sets for Robot Mapping and SLAM,* capítulo 1: "Introduction".

[22] Grisetti, G.; Stachniss, C.; Burgard, W. *Improved Techniques for Grid Mapping with Rao-Blackwellized Particle Filters.*

[23] Frontier exploration: http://wiki.ros.org/frontier_exploration

[24] Actionlib de ROS: http://wiki.ros.org/actionlib

[25] Zheng, K. (2016). *ROS Navigation Tuning Guide*.

[26] Protocolo EtherCAT. https://www.ethercat.org/en/technology.html

[27] Interfaz de Aplicación a Tiempo Real (RTAI). https://www.rtai.org

[28] Hoarau, A. *Mekabot Wiki* (feb 2015). https://github.com/ahoarau/mekabot/wiki

[29] *Wiki de Robótica de la Universidad Politécnica de Valencia* (UPV).

[30] *MoveIt!* http://moveit.ros.org/

[31] Quigley, M. (2015). *Programming robots with ROS,* capítulo 13: "On Patrol"

[32] Alejandro Sánchez García (2013). Tesis doctoral: "*Integración sensorial de fuerza, aceleración y visión en robots manipuladores con movimientos restringidos y entornos parcialmente estructurados*"

[33] Glade – A User Interface Designer. https://glade.gnome.org/

[34] The GTK+ Project. https://www.gtk.org/

Anexo I: Información adicional de ROS

Paquetes de ROS

Los paquetes son la unidad básica más importante que maneja ROS. Internamente deben tener una estructura específica para evitar confusión al compartirla con la comunidad para su posterior reutilización. La creación es sencilla con herramientas proporcionadas por la plataforma, ya que con un simple comando se puede crear un paquete y declarar sus dependencias (que son fácilmente modificables)

```
$  catkin_create_pkg <nombre_paquete> <dependencia1> <dependeencia2> <..>
```

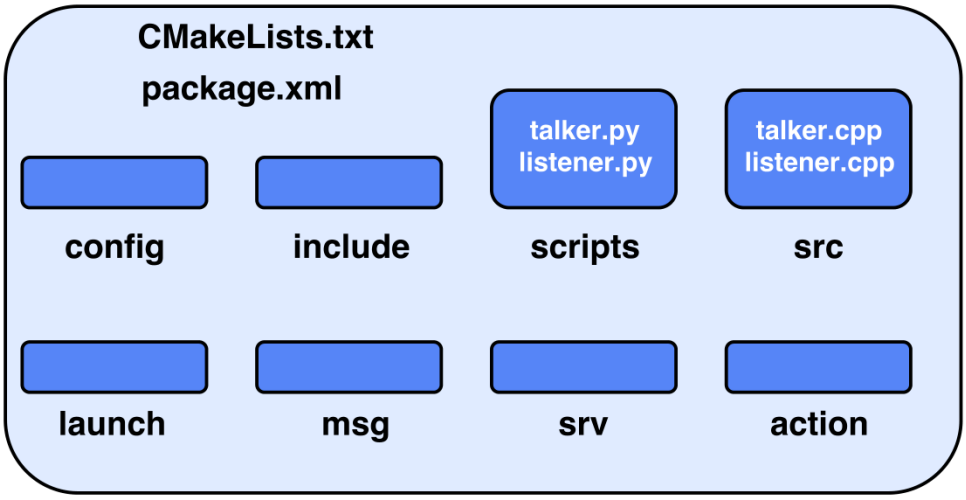

Figura 69: estructura típica de un paquete ROS

La mayor parte del presente trabajo contiene los archivos de programación en paquetes de ROS, aunque no todos incluyen las carpetas visibles en la Figura 69: sólo las necesarias para el correcto funcionamiento de cada paquete.

Workspaces

El primer requisito para crear un paquete ROS es crear un ROS catkin workspace. Es la función principal para compilar, enlazar librerías y ejecutables… a partir de los archivos estructurados como un paquete ROS. Para ello, se ejecutan los siguientes comandos:

```
$  mkdir -p ~/<nombre_workspace>/src
$  cd ~/<nombre_workspace>/src
$  catkin_init_workspace
```

Dentro de la carpeta “src” del workspace ya creado se ubicarán los paquetes de ROS, cuyos nodos se escriben en Python o C++; en el caso de Python, se ubicarán en la carpeta “scripts”, y sólo será necesario dar permisos de ejecución a dichos scripts “.py” una vez, mediante los siguientes comandos:

```
$  cd ~/<nombre_workspace>/src/<nombre_paquete>/scripts
$  sudo chmod u+x *.py
```

Si la programación se realiza en C++, además de la propia edición del código, se deberán editar los archivos “CmakeLists.txt” y “package.xml” del paquete [2], compilando después:

```
$  cd ~/<nombre_workspace>
$  catkin_make
```

Por último, se deben referenciar los workspaces en el “shell” de Ubuntu, para que el sistema de archivos de ROS reconozca los nodos y podamos ejecutarlos; para que la referencia sea permanente, se deben escribir los siguientes comandos cada vez que creemos un workspace:

```
$  echo “source ~/<nombre_workspace>/devel/setup.bash” >> ~/.bashrc
$  source ~/.bashrc
```

Anexo II: Cinemática de un robot diferencial

La documentación externa en inglés se encuentra en el libro *Learning Robotics Using Python*, escrito por Lentin Joseph (2015). La información detallada estáen el capítulo 3, apartado “Mathematical modeling of the robot”, donde explica la diferencia entre un robot diferencial y uno holonómico además de calcular las cinemáticas del robot diferencial.

Anexo III: Instalación de ROS y archivos en incluidos en el CD

Los pasos necesarios para la instalación de ROS en Ubuntu vienen explicados en la web oficial [2] de la plataforma; no obstante, se resumen aquí los pasos a seguir, teniéndose en cuenta que la versión de Ubuntu elegida es la 14.04LTS y la versión de ROS a instalar es la ***Indigo***:

- Configurar el ordenador para aceptar paquetes de *software* de ROS:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_re
lease -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

- Establecer las claves de ROS (sustituir por “hkp://pgp.mit.edu:80” ó “hkp://keyserver.ubuntu.com:80” si el comando falla):

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80
--recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

- Actualizar el índice de paquetes Debian:

```
$ sudo apt-get update
```

- Una vez finalizado todo lo anterior, se procede a instalar ROS. Se recomienda instalar la versión completa:

```
$ sudo apt-get install ros-indigo-desktop-full
```

- Cuando la instalación finalice, y antes de poder utilizar ROS, inicializar el *rosdep*:

```
$ sudo rosdep init
$ rosdep update
```

- Añadir las variables del entorno de ROS automáticamente al *bash* cada vez que se abre una nueva terminal de comandos:

```
$ echo "source /opt/ros/indigo/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

Para finalizar con este punto, y evitar posibles problemas en el futuro, es **importante** destacar el hecho de que las variables de entorno de los ***workspaces*** deben añadirse al *bash* de la terminal en la que se vayan a utilizar paquetes pertenecientes a dichos *workspaces*; para ello, se debe ejecutar el siguiente comando:

```
$ source <ruta_al_workspace>/devel/setup.bash
```

Si se desea automatizar este proceso, para que las variables de un *workspace* determinado se añadan al inicio de cada sesión en terminal, la línea de código anterior deberá incluirse en el archivo "**.bashrc"**, dentro de la carpeta personal; una forma fácil de editarlo en cualquier momento es:

```
$ gedit ~/.bashrc
```

O incluso para añadir directamente, se realiza un comando similar al anterior:

```
$ echo "source <ruta_al_workspace>/devel/setup.bash" >> ~/.bashrc
```

Además, se pueden instalar programas extra incluidos en ROS, pero que no están instalados por defecto. El comando siguiente permite la instalación de los mínimos necesarios para el funcionamiento de los script programados en el presente proyecto y que no están incluidos en el CD:

```
$ sudo apt-get install ros-indigo-openni-* ros-indigo-smach-* ros-indigo-gm
apping ros-indigo-navigation ros-indigo-moveit ros-indigo-laser-filters lib
openni0 libopenni-dev python-termcolor
```

Si durante la instalación de alguno de los paquetes requiere otros, los instalará automáticamente. Si en algún momento se solicita al usuario instalar otros paquetes, sólo debe ejecutar el comando *sudo ap-get install* con el paquete deseado.

La IDE (*Integrated Development Environment*) o entorno de desarrollo integrado utilizado para programar todos los scripts del proyecto se llama ***Eclipse***, una IDE *open source* totalmente gratuita, junto con el módulo *Pydev* (utilizado para programar en Python y que reconozca las librarías de ROS).

Archivos del proyecto incluidos en el CD

Para instalar los paquetes desarrollados en el presente proyecto e incluidos en el CD, se descomprime el archivo llamado "***programas_TFG_EnriqueOrtega***". Dentro existen dos carpetas, una donde se incluyen los paquetes de ROS programados en el presente proyecto y en otra los paquetes de otros autores totalmente externos necesarios para el funcionamiento. También se incluyen *drivers*

para ejecutar la Kinect en el robot PMB2 y se localizan en el archivo comprimido "*drivers_kinect_PMB2*".

Todos los paquetes de ROS se pueden copiar directamente a la subcarpeta "*src*" dentro de un ***Workspace*** creado previamente. Se ejecuta el siguiente comando y se hace *source* a dicho *Workspace*:

```
$ catkin_make
```

La distribución de los paquetes se encuentra disponible como información externa en el CD, en un archivo PDF (*arbol_archivos.pdf*). También se encuentra una carpeta con los PDF para visualizar los árbol TF de ambos robots, tanto unidos como separados.

Anexo IV: Glosario de acrónimos, siglas y definiciones

- **AMCL**: descrito completamente en el apartado 3.4.4.
- **API**: (*Application Programming Interface*) interfaz de programación de aplicaciones, es un conjunto de surutinas, funciones y métodos que ofrece cierta biblioteca para ser utilizado por otro software.
- **Arduino**: plataforma electrónica *open-source* basada en un *hardware* y *software* fácil de utilizar, para realizar prototipos rápidos o proyectos que necesiten de un microcontrolador.
- **Bash**: (*Bourne Again Shell*) es un programa informático cuya función consiste en interpretar órdenes, por lo que consiste en un lenguaje de consola utilizado en sistemas Unix.
- **DHCP**: (*Dynamic Host Configuration Protocol*) protocolo de configuración dinámica de *host* es un servidor que usa protocolo de red tipo cliente/servidor en el que un servidor posee una lista de direcciones IP dinámicas y las asigna a los clientes conforme se van conectando.
- **Framework**: entorno de trabajo, es un conjunto estandarizado de conceptos , prácticas y criterios para enfocar un tipo de problemática particular. Sirve para el desarrollo de software con módulos concretos y asistencia definida. Incluye programar, bibliotecas, herramientas así como un lenguaje propio.
- **Kinect**: sensor descrito en el apartado 4.5.
- **LIDAR**: (*Light Detecion And Ranging*) es un dispositivo que permite determinar la distancia desde un emisor láser a un objeto o superficie utilizando un haz de láser pulsado.
- **Máquina virtual**: software que simula a una computador y puede ejecutar programas como si fuera una computadora real. Es “un duplicado eficiente y aislado de una máquina física”.
- **Meka**: robot construido por Meka-robotics con dos brazos robóticos antropomórficos de 7 DOF cada uno. Es uno de los robots con los que se trabaja en el presente proyecto.
- **Middleware**: lógica de intercambio de información entre aplicaciones, es un software que asiste a una aplicación para interactuar o comunicarse con otras aplicaciones, programas, redes y sistemas operativos. Simplifica la compleja

tarea de definir protocolos de comunicaciones en sistemas distribuidos, como en ROS.

- **Odometría**: estudio de la estimación de la posición de vehículos con ruedas. Se utiliza principalmente como un componente de la navegación autónoma, como aproximación relativa a su posición inicial. Se suele proporcionar por los encoders hayados en las ruedas motrices, conformando un servomotor. Mediante unos cálculos, el cambio de posición angular en cada rueda se transforma en el desplazamiento lineal sobre el suelo. Como los sensores no pueden tener una precisión infinita, se trata de una aproximación, que aunque sea buena a corto plazo, el error irá aumentando conforme se va desplazando, también llamados errores sistemáticos.

  A esto se suma imperfecciones en los cálculos o simplemente deslizamientos de la propia rueda con el terreno, conformando errores no-sistemáticos. Es por ello que se necesita la ayuda de otros sensores para realizar la tarea de la localización adecuadamente.

- **P2P**: (*peer-to-peer*) son redes cuya arquitectura es una distribución de tareas entre pares, los cuales tienen equidad en nivel de importancia .
- **PMB2**: robot construido por Pal-Robotics. Es una base autónoma móvil flexible y personalizable, destinada a mover cargas o incluso otros robots, como en el presente proyecto, que porta al robot Meka.
- **RGB-D**: (*Red Green Blue – Depth*) tipo de imagen estereoscópica a color, que contiene tanto información visual como tridimensional del entorno.
- **ROS**: (*Robot Operating System*), desarrollado en el capítulo 2 y con guía de instalación en el **Anexo III:** Instalación de ROS.
- **SLAM**: desarrollado en profundidad en el apartado 3.4.5.
- **SMACH**: librería de *Python* utilizada para el desarrollo de tareas complejas. Compatible y completamente integrada en ROS.
- **SSH**: comando de Linux utilizado para conectarse a la terminal de comandos de otro ordenador. Su variante *scp* es la copia de archivos basada en *ssh*.
- **TF**: utilidad de ROS descrita detalladamente en el apartado 3.4.3.
- **TOF**: *Time Of Flight* o Tiempo de vuelo, describe una variedad de métodos para medir el tiempo que toma un objeto, partícula u onda en viajar una distancia a

través de un medio. Dichos métodos son utilizados por la cámara TOF, para medir la distancia desde la propia cámara a una nube de puntos de los objetos a los que esté apuntando.

- **URDF**: (*Unified Robot Description Format*), es un archivo escrito en XML que contiene la descripción de un robot tanto para lanzar TF de las partes rígidas como visualizar correctamente en Rviz.
- **XML**: (*eXtensible Markup Language*) es un meta-lenguaje que permite defini lenguajes de marcas, desarrollado por el W3C (*World Wide Web Consortium*) para almacenar datos en forma legible.