Escuela Politécnica Superior de Jaén
Máster Universitario en Ingeniería Mecatrónica

UNIVERSIDAD DE JAÉN
Escuela Politécnica Superior (Jaén)

Trabajo Fin de Máster

ROBOT GUÍA PARA LABORATORIOS DE DOCENTES

Alumna: Vargas Monge, Lorna Francini

Tutor: Prof. D. Alejandro Sánchez García
Dpto.: Ingeniería Electrónica y Automática

Julio, 2021

Universidad de Jaén
Centro de Estudios de Postgrado
Dpto. de Ingeniería Electrónica y Automática

Don Alejandro Sánchez García, tutor del Proyecto Fin de Máster titulado: Robot Guía para laboratorios de docente, que presenta Lorna Francini Vargas Monge, autoriza su presentación para defensa y evaluación en la Universidad de Jaén.

Jaén, Julio de 2021

Alumno:

Francinie Vargas Monge

Lorna Francini Vargas Monge

Tutor:

SANCHEZ GARCIA ALEJANDRO - 77321466C

Firmado digitalmente por SANCHEZ GARCIA ALEJANDRO - 77321466C
Fecha: 2021.07.02 14:05:51 +02'00'

Alejandro Sánchez García

RESUMEN

En las últimas décadas los seres humanos se han dedicado a la creación y desarrollo de diferentes máquinas capaces de realizar diferentes tareas con fines diversos; los robots. Cada vez se les otorgan más capacidades similares a las de los humanos como la posibilidad de extraer información del entorno e interpretarla para tomar decisiones. Por lo tanto en este trabajo se muestra el estudio y desarrollo de un programa robótico para la navegación autónoma del robot *PMB-2* utilizando el sistema operativo ROS, así como la capacidad de describir el entorno y entablar una conversación simple con el usuario por medio de una interfaz gráfica que simule el movimiento de una cara. Se presenta una breve descripción teórica de ROS, de los algoritmos a utilizar para la navegación (método *Monte Carlo* para localización, método de aproximación de ventana para generación de rutas, etc), así como los pasos y las pruebas realizadas.

ABSTRACT

In the last decades, humans have created and developed different machines capable of performing different tasks for different purposes; the robots. Over time, they have been given human-like capabilities, such as the ability to extract information from the environment and interpret it to make decisions. Therefore, this project presents the study and development of a robotic program for the autonomous navigation of the *PMB-2* robot using ROS as operating system, as well as the ability to describe the environment and establish a simple conversation with the user through an interface that simulates the movement of a face. It's also presented a brief theoretical description of ROS, navigation algorithms (*Monte Carlo* method for location, Dynamic Window Approach method for route generation, etc.), as well as the steps and tests implemented to achieve the main goal.

ÍNDICE

LISTADO DE FIGURAS

1. INTRODUCCIÓN Y OBJETIVOS

El acelerado crecimiento actual en la tecnología y la investigación han permitido utilizar robots en prácticamente todas las ramas en las que se desarrollan los seres humanos, desde actividades básicas de limpieza hasta acciones más complicadas como la exploración de entornos desconocidos como el planeta Marte. Una de las ramas que actualmente se está explotando en la robótica, es la de otorgar características humanas a los robots, como lo son la capacidad de percibir el entorno, navegar en el, escuchar, detectar objetos por medio de cámaras o sensores, e inclusive comunicarse con las personas por medio de inteligencia artificial. Esto con el fin de utilizarlos en situaciones que requieran una interacción, por ejemplo actividades de servicio al cliente. Entre los robots más famosos que cuentan con estas características se encuentran el robot *Pepper* de *SoftBank Robotics*, *Sophia* de *Hanson Robotics* y el robot *Atlas* de *Boston Dynamics* (Revista de Robots, 2020). Estos robots son muy avanzados pero tienen como inconveniente su elevado costo.

El presente trabajo muestra el estudio y los pasos realizados con el fin de dar al robot comercial *PMB-2* la capacidad de movilizarse en los laboratorios de electrónica de la Universidad de Jaén, a la vez que le comenta al usuario información relevante de estos; es decir un robot guía. Esta aplicación puede ser de gran utilidad didáctica para estudiar y mostrar los diferentes algoritmos de navegación e interacción robot-humano, además de ser un atractivo para mostrar las diferentes instalaciones del edificio de ingeniería. Otro uso importante de los robots capaces de movilizarse autónomamente y capaces de comunicarse con las personas, es que permite realizar actividades en tiempos de pandemia, como los vividos actualmente, sin suponer algún riesgo; en este caso se pueden realizar tours guiados individualizados donde el usuario conocerá las instalaciones del lugar sin estar en contacto con otras personas.

Por lo tanto, el objetivo principal del trabajo de fin de máster es, "*implementar la programación necesaria para que la base robótica PMB-2 navegue de forma autónoma por la cuarta planta del edificio A3 de la Universidad de Jaén, describiendo el laboratorio en el cual se encuentra"*. Para el cumplimiento de este objetivo se tienen los siguientes objetivos parciales:

- Realizar la programación de la navegación autónoma del robot de manera parametrizable, que permita movilizarlo a puntos definidos con respecto al mapa de la planta del edificio, utilizando el sistema operativo ROS.

- Diseñar y programar una “cara” para el robot en un sistema embebido (*Raspberry Pi*) que se encargue de reproducir audios que simulen el habla del robot.
- Implementar una máquina de estados para programar el funcionamiento global del sistema en las diferentes situaciones en las que se encuentre el robot.

Para el desarrollo del proyecto y así cumplir con los objetivos propuestos, la metodología seguida consistió en un estudio de las tecnologías involucradas, el planteamiento de la solución en conjunto con diferentes pruebas y simulaciones con el robot. Más específicamente las pautas seguidas fueron las siguientes:

- Estudio del sistema operativo ROS, su ideología, funcionamiento, comandos, y simulaciones.
- Estudio del uso del robot móvil *PMB-2*, su estructura, manipulación, conexión y su programa ya previamente instalado.
- Estudio, implementación y pruebas de los diferentes algoritmos de navegación automática.
- Conectividad entre sistemas, en este caso robot-computador-*Raspberry* para la realización de pruebas.
- Programación de Raspberry para ser utilizada como interfaz entre usuario-robot.
- Refuerzo de los conocimientos de programación de *Python*, máquinas de estados, interfaces gráficas y otros.
- Pruebas de navegación simuladas y en el laboratorio.
- Programación de la máquina de estados para la unificación de programas y pruebas finales.

La estructura de la siguiente memoria es la siguiente:

- Capítulo 2. Robots y Robótica: donde se describe el campo de trabajo de este proyecto y el robot a utilizar.
- Capítulo 3. ROS: *Robot Operating System;* donde se explica la plataforma utilizada en el proyecto, sus principales comandos y ventajas.

- Capítulo 4. Navegación autónoma: en este capítulo se exponen los diferentes algoritmos que existen en la actualidad para lograr la navegación autónoma con diferentes robots y sensores.

- Capítulo 5. Interacción robot-humano: capítulo en el que se describe los programas desarrollados para crear una cara robótica, su movimiento y como se logra una interacción entre humano y robot.

- Capítulo 6. Configuración utilizada: capítulo donde se muestra las configuraciones realizadas entre el robot, la *Raspberry* y el ordenador externo para cumplir con los objetivos del presente trabajo.

- Capítulo 7. Recorrido guiado: donde se muestra los resultados obtenidos del programa realizado en un caso de estudio.

- Capítulo 8. Conclusiones: capítulo final donde se resumen las conclusiones obtenidas al realizar el proyecto y se describen posibles proyectos futuros que se pueden realizar con la plataforma como continuación o adaptación de este proyecto.

2. ROBOTS Y ROBÓTICA

A lo largo de la historia, los seres humanos se han interesado por crear máquinas, dispositivos o herramientas capaces de imitar las funciones y movimientos de los seres vivos, con el fin de facilitar tareas o simplemente por entretenimiento o decoración. En la antigua Grecia, se utilizaba la palabra autómata para referirse a objetos con la capacidad de moverse por sí solos (Real Academia Española, 2014), y desde entonces se desarrollaron diferentes inventos bajo este término. Sin embargo no fue hasta 1920 que se presenta la primera aparición del término *robot*, por el novelista *Karel Capek* en su obra *Robots Universales Rossum* para denotar una máquina que remplaza al ser humano en sus labores. Esta palabra proviene del checo *robota*, que se traduce literal a trabajo duro o servidumbre. Sin embargo, el impulsor del término fue el escritor de ciencia ficción; *Isaac Asimov*, alrededor de los años 50. A él también se le atribuye el término robótica, y las conocidas y ficcionales leyes de la robótica (Christoforou & Müller, 2017).

En la actualidad, existen diferentes tipos de robots; los hay de todo tamaño, forma y diseño; para diferentes aplicaciones; desde limpiar un cuarto hasta realizar una cirugía médica o explorar Marte. Esta diversidad que presentan los robots, hace difícil encontrar una definición precisa y concisa que encasille lo que realmente puede ser un robot y qué lo distingue de máquinas y otros dispositivos. La palabra robot puede comprender varios niveles de sofisticación tecnológica; por lo que cada persona o entidad lo puede definir de manera distinta, siendo el concepto muy influenciado por el cine, la literatura y la ciencia ficción. Existen múltiples definiciones por diferentes entes, las más destacables son:

-*Robotic Industries Association (RIA):* “Manipulador reprogramable multifuncional diseñado para mover material, partes, herramientas o dispositivos especializados a través de trayectorias variables para la realización de diferentes tareas”

-*Real Academia Española (RAE):* “Máquina o ingenio electrónico programable que es capaz de manipular objetos y realizar diversas operaciones antes reservadas a las personas. Que imita la figura y los movimientos de un ser animado. Programa que explora automáticamente la red para encontrar información “

-*International Federation of Robotics (IFR), basado en la norma ISO 8373:* “Robot Industrial: Manipulador multipropósito programable en tres o más ejes. Robot de Servicio: Que realiza tareas útiles para los humanos excluyendo las aplicaciones de automatización industrial”

-*International Organization of Standardization (ISO):* “Mecanismo accionado programable en dos o más ejes, con cierto grado de autonomía, que se mueve en su entorno, para realizar tareas destinadas. Incluye un sistema de control y una interfaz a este. Se clasifican en robot industrial y de servicio dependiendo de su aplicación”

-*Institute of Electrical and Electronics Engineers (IEEE):* “Máquina autónoma capaz de detectar/sentir su ambiente, llevando a cabo computaciones para tomar decisiones y realizar acciones en el mundo real”

-*Enciclopedia Británica:* “Máquina operada automáticamente que sustituye el esfuerzo de los humanos, aunque no tiene por qué tener apariencia humana o desarrollar sus actividades a la manera de los humanos.”

-*Diccionario Merrian Webster:* “Máquina que se asemeja a los humanos y desarrolla como ellos tareas complejas como andar o hablar. Un dispositivo que desarrolla de manera automática tareas complicadas, a menudo de manera repetitiva. Un mecanismo guiado por control automático.”

Ninguna de las definiciones anteriores es perfecta; la mayoría tienen palabras en común como: autónoma, programable y multifuncional. Sin embargo, estas definiciones se pueden debatir, ya que no diferencian de manera estricta a los robots de diferentes máquinas como una lavadora de platos o un elevador. Se pueden enumerar características que un robot deba tener, como la capacidad sensorial, el procesamiento de datos, los actuadores y los demás elementos necesarios para cumplir una tarea. Pero debido a la diversidad de los sistemas robóticos, algunos serán más simples que otros en estas cualidades. Otros características que se pueden mencionar para distinguirlos de las máquinas son; versatilidad (variedad de funciones ligadas a sus grados de libertad), flexibilidad (capacidad de realizar diferentes trabajos sin modificaciones físicas), y adaptación del entorno (utilizando diferentes sensores).

Aunque no sea posible describir que es un robot englobando todas sus posibilidades y características, en un solo enunciado, es necesario tener en cuenta las diferentes definiciones para así darse una idea realista de que puede llegar a ser un robot en la actualidad. Una frase dicha por uno de los pioneros de la robótica industrial; *Joseph Engelberger;* engloba esta paradoja que tienen las personas a la hora de definir un robot (Guizzo, 2020):

“No puedo definir lo que es un robot, pero se reconocer uno cuando lo veo”.

Por otro lado, se puede definir la robótica como una rama interdisciplinaria de la ingeniería que se encarga del análisis, diseño, construcción, control y procesamiento de datos de robots. La robótica comprende partes de la ingeniería mecánica, ingeniería electrónica/eléctrica, ingeniería mecatrónica, teoría de control automático, ciencias de la computación, matemática, física, entre otras, para realizar tareas que requieren conocimientos como: modelado de sistemas dinámicos, control y realimentación de sistemas, sensores y acondicionamiento de señales, actuadores y electrónica de potencia, hardware e interfaz con el controlador y programación.

2.1 Hitos en la historia de la robótica

Como se mencionó anteriormente, el ser humano siempre ha sentido la curiosidad y deseo de construir diferentes dispositivos automáticos, a continuación se enumeraran algunos hitos históricos en el ámbito de la robótica (Cinemática y dinámica de sistemas mecatrónicos, 2019-2020).

Alrededor del 270 a.C, *Ctesibio* crea *Clépsidra* y un órgano de agua, que se consideran los primeros mecanismos hidráulicos. A partir de esta fecha, se registran numerosas figuras y dispositivos mecánicos dotados con cierta autonomía y movimientos de sus partes, como los son el Gallo de Estrasburgo (1352), o El león mecánico de *Leonardo Da Vinci* (1515). En el siglo XX*, d*ebido al acelerado crecimiento de la industria (ya para 1913 *Ford* fabricaba alrededor de mil vehículos diarios usando cadenas de producción), el desarrollo de las computadoras (concepto introducido por *Alan Turing* en 1936), y el desarrollo de los fundamentos de servo control y automatización en el *MIT* (1939); se empiezan a patentar dispositivos para reproducir movimientos de diferentes máquinas. Para 1948, *Ray Goertz* crea el primer sistema de tele manipulación maestro-esclavo mecánico en el *Argone National Laboratory. Y e*n 1954, *George Devol* solicita patente del *Unimate* (primer robot industrial). Este desarrollo acelerado, permite que en 1956 se lleve a cabo el primer congreso de inteligencia artificial. Alrededor de los años 60, ya se observan significantes avances en este campo; *General Electric* construyó *HandyMan:* sistema de tele-operación con dos manipuladores y diez grados de libertad y *American Machine Foundry* produce el primer robot de configuración cilíndrica; *Versatran.* Además para esta época el uso de robots se empieza a expandir no solo al ámbito de la industria sino que también para suplir necesidades médicas, militares y de investigación espacial. En todos estos avances de la robótica, las universidades jugaron un papel muy importante en el desarrollo de nuevos sistemas y estándares, por ejemplo en 1968 *Standford Research Institute*

construye *Shakey*, un robot móvil con autonomía de movimiento basado en sensores de localización. En los años 70 se empieza a celebrar el simposio nacional americano sobre robots industriales, y destacan robots como el *PUMA* de *Honda* (1978) y el primer robot *SCARA* de *IBM* y *Sankyo* (1979). Para finales de los noventa e inicios de los 2000, se popularizan los robots de entretenimiento y servicio, como los son el perro *Aibo* de *Sony* (1999), y la aspiradora robótica *Roomba* (2002). Todos estos hitos mencionados son una pequeña parte de las múltiples contribuciones que se han realizado en este campo a través del tiempo; conocerlos permite comprender la robótica actual y sus posibilidades. Algunos ejemplos se muestran en la figura 2.1

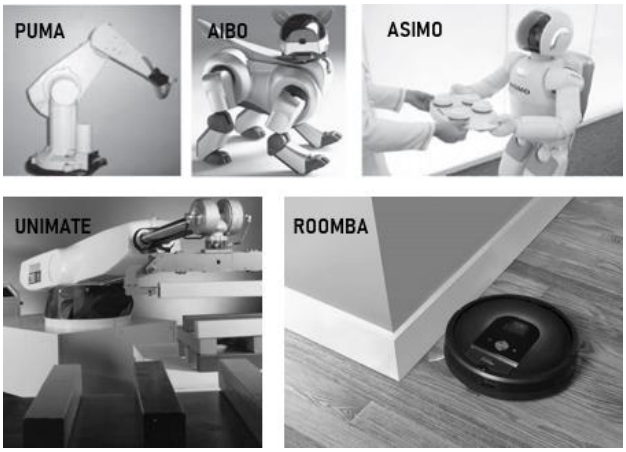


Figura 2.1 Robots a lo largo de la historia.

2.2 Tipos de robots

Existen muchos tipos de robots que varían dependiendo de su funcionalidad, su diseño mecánico, época de creación, etc. Todas estas características permiten clasificar a los robots en distintas categorías, algunas de ellas son:

2.2.1 Según funcionalidad

- Robot de Servicio: Esta categoría comprende a cualquier robot que se utilice fuera de las instalaciones industriales, es decir son los robots que tienen objetivos profesionales, educacionales, uso personal o entretenimiento.

- Robot industrial: Son los robots dotados de articulaciones móviles destinado a tareas repetitivas dentro de las diferentes etapas de la producción industrial para aumentar la productividad, calidad y estandarización de los procesos. A su vez estos se pueden clasificar dependiendo a su estructura mecánica o configuración

cinemática (figura 2.2) en: cartesiano, cilíndrico, polar, *SCARA* o angular (Ben-Ari & Mondada, 2018).

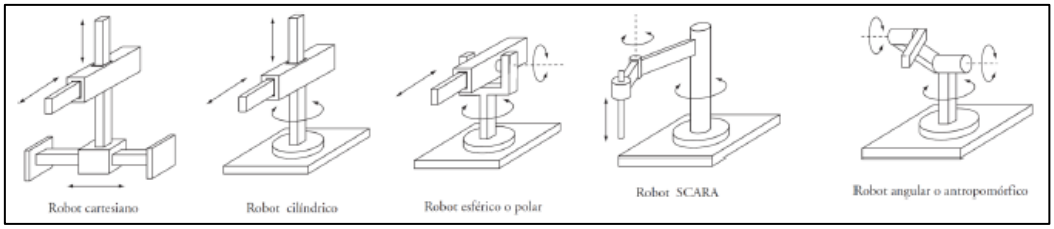


Figura 2.2. Clasificación de robots industriales según su configuración cinemática.

2.2.2 Según funcionalidad específica

Según el Comité Español de Automática (CEA), se pueden clasificar dependiendo de sus características o campo de aplicación, como: robots domésticos, asistenciales, médicos, de entretenimiento, espaciales, educacionales, humanoides, militares, es decir según su funcionalidad o aplicación específica (Kazin, 2014).

2.2.3 Según movimiento

- Robot fijo: Robots que están montados sobre una base estable y se mueven en distintas posiciones alrededor de esta.

- Robot móvil: Estos robots no están sujetos a una superficie fija, por lo que pueden tener un movimiento libre. A su vez pueden clasificarse como: terrestres (similares a los vehículos, con una o más ruedas o extremidades), submarinos (trabajan bajo el agua para asistencia de nado, recolección de materia o imágenes), voladores no tripulados (drones y cuadricópteros) e híbridos (combinación de las características anteriores para aplicaciones específicas), (Ben-Ari & Mondada, 2018).

2.2.4 Según generación

Esta categoría permite clasificar los robots haciendo referencia al momento tecnológico en el que fueron creados:

- Primera generación (inicio de la robótica - hasta los años 80): Repiten tareas programadas secuencialmente, y no tienen en cuenta las alteraciones del entorno. Se utilizan principalmente para mover objetos.

- Segunda generación (años 80 – inicios de los 2000): Estos robots adquieren información limitada de su entorno y actúan en consecuencia a ello. Pueden localizar,

clasificar y detectar esfuerzos y adaptar sus movimientos. Son reprogramables, y son el principio de lo que se conoce como robots inteligentes.

- Tercera generación (actualidad): Posee capacidad para la planificación automática de tareas al captar el entorno en tiempo real. Están directamente relacionados a la inteligencia artificial (Cinemática y dinámica de sistemas mecatrónicos, 2019-2020).

2.2.5 Según tipo de energía de los actuadores

Dependiendo del tipo de energía utilizada por los diferentes componentes del robot, éste puede ser clasificado como: neumático, eléctrico, hidráulico, natural (solar).

2.2.6 Según número de ejes o grados de libertad

Esta característica es aplicable a los robots que tienen cadena cinemática y se clasifican dependiendo de la cantidad de ejes de cada uno de los movimientos independientes con que está dotado el robot (Anusha, 2018). Por ejemplo: 3 grados de libertad (para recoger y colocar un objeto en diferentes posiciones utilizando tres ejes diferentes de coordenadas), 6 grados de libertad (para recoger y colocar un objeto en cualquier lugar y orientación) y otros (diferentes combinaciones de grados de libertad).

2.2.7 Según el tipo de control

Los robots pueden clasificarse según el tipo de control e interacción humana que se deba tener para realizar las tareas:

- Tele operado, controlado o manual: Estos robots requieren la intervención humada para lograr una tarea. Pueden ser controlados remotamente o por controles cableados que los guían para que realicen diferentes actividades.

- Adaptativo o automático: Las direcciones han sido programadas en ciclos predeterminados. En ciertas ocasiones y para tareas más complejas se requiere intervención humana. Estos puede ser secuenciales o por trayectoria.

- Autónomos o inteligentes: Operan con autonomía y mínima intervención humana para realizar las tareas. Suelen tener capacidad sensorial para comprender el entorno y tomar decisiones de acuerdo a los cambios de este.

- Virtuales: Programas de computador diseñados para simular un robot real, por ejemplo los *Chatbot* o los rastreadores web.

2.2.8 Otras clasificaciones

Existen otras clasificaciones para los robots, como lo son: la clasificación según la Asociación Japonesa JIRA (*Japanese Industrial Robot Association*) y según la asociación francesa de robótica AFR (*Association Francoise de Robotique*). Estas los clasifican en: clase 1 o dispositivo de manejo manual, clase 2 o robot de secuencia fija, clase 3 o robot de secuencia variable, clase 4 o robot de reproducción, clase 5 o robot de control numérico, clase 6 o robot inteligente; y tipo A o dispositivos manuales, tipo B o dispositivos automáticos, tipo C o programables y tipo D o con capacidad sensorial; respectivamente (Anusha, 2018).

2.3 Utilidad e importancia de los robots y la robótica

En los últimos años la robótica se ha desarrollado y ha evolucionado en gran medida, logrando así transformar la vida de las personas. La robótica pasó de ser una gran herramienta en el entorno industrial a ayudar a la humanidad en cualquier campo, ya sea social, económico, de salud o entretenimiento. Los robots se utilizan para un sinfín de tareas; cotidianas, repetitivas, estresantes, aburridas o intensas para los humanos, como labores que requieran un arduo entrenamiento, trabajos de gran exactitud, actividades peligrosas o de riesgos elevados. Son capaces de obtener información que los humanos no pueden, consiguen realizar tareas con error mínimo, de manera eficiente y rápida sin la interacción humana. La constante búsqueda de mejoras en los sistemas robóticos ha permitido el desarrollo y avance científico y tecnológico en diferentes áreas de la ingeniería y ha ayudado a resolver problemas actuales. Es necesario comprender que la robótica no vino a sustituir al ser humano o a inhibirle de su trabajo, sino, simplificar las actividades para ahorrar tiempo y dinero, y evitar daños a las personas por actividades riesgosas o repetitivas. A su vez, la robótica genera diferentes trabajos como técnicos y/o operarios, cargos de mantenimiento y de ingeniería. La inteligencia artificial, la visión por computador, las redes neuronales y otros algoritmos así como el *hardware* y el *software,* la creación de nuevos materiales y métodos de fabricación, *e*stán en constante crecimiento e investigación, por lo que en el futuro se puede esperar robots más eficientes que poco a poco irán imitando la inteligencia humana en gran escala.

2.4 Robot *PMB-2*

Para el desarrollo de este trabajo de fin de máster, se utilizó el robot *PMB-2* de la compañía *PAL-Robotics*[1] (figura 2.3). Esta empresa española fue creada en el 2004 y desde entonces se ha dedicado a la investigación y desarrollo de diferentes robots humanoides y bases móviles con diferentes objetivos.

Figura 2.3 Robot móvil *PMB-2*.

El robot *PMB-2* es un robot móvil circular que puede ser utilizado para manejar tareas de logística en plantas industriales o almacenes, hospitales o laboratorios, hoteles u oficinas. Es utilizado en el sector de educación en la investigación de algoritmos de inteligencia artificial/*Machine Learning,* interacción humano-robot, manipulación, percepción, navegación y planeamiento de movimientos (PAL Robotics, 2020). En este caso se utilizará para realizar recorrido guiado en los diferentes laboratorios del departamento de ingeniería electrónica y automática, por medio de algoritmos de navegación autónoma y algoritmos de comunicación para robots.

Este robot se considera un robot diferencial de ruedas, es decir su movimiento se basa en dos ruedas laterales independientes conectadas a los motores, y varias ruedas pivotantes o secundarias utilizadas para estabilizar al robot. El movimiento del robot es controlado por medio de las velocidades de cada rueda; la velocidad del robot será ortogonal al eje que conecta las ruedas, lo que provoca una restricción no holonómica. Para cambiar de dirección, se varía la tasa de rotación; si las dos ruedas se manejan en la misma dirección y velocidad, el robot se moverá en línea recta, si se manejan en direcciones opuestas, el robot rotará sobre su punto medio, y para lograr una rotación entre el centro de rotación y la posición recta, dependerá de la combinación de la velocidad y dirección que se le indique a cada rueda (Robins & Somashekhar S, 2016).

[1] *Pal Robotics:* http://pal-robotics.com/es/

Un robot no holonómico es un robot que tiene menor cantidad de grados controlables que de grados de libertad, lo que provoca que no se pueda mover en cualquier dirección (figura 2.4) y se necesite un tipo de control más extenso para moverlo según deseado, utilizando las ecuaciones de cinemática del robot.

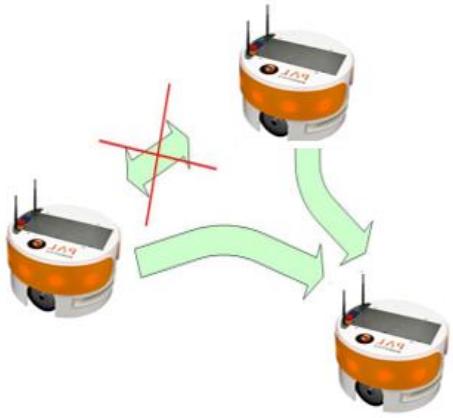


Figura 2.4 Movimiento de robot no holonómico.

Entre las principales características de este robot se encuentran: su capacidad de ser completamente personalizable, los sensores disponibles y adaptabilidad a nuevos sensores, su capacidad de carga, y además que es controlado mediante el sistema operativo robótico ROS. Las especificaciones técnicas de los componentes de este robot se muestran en el siguiente apartado.

2.4.1 Especificaciones técnicas

Tal y como se observa en la figura 2.3, este robot tiene forma cilíndrica y es considerado un robot diferencial de ruedas. El resto de especificaciones se relatan en la siguiente tabla (PAL Robotics, 2020):

Tabla 2.1. Especificaciones técnicas del robot móvil *PMB-2*

Características generales	
Peso robot + baterías	30 Kg
Capacidad de carga	50 Kg
Máxima velocidad	1 m/s
Cantidad de ruedas motrices	2
Cantidad de ruedas pivotantes	4

Cantidad de amortiguadores	2
Diámetro	54 cm
Alto	30 cm
Panel de usuario	
Cantidad de luces LED	4
Audio	1 altavoz
Expansión	10 GPIO
Energía	12V-5A / 5V-5A
Conectividad	
WiFi	802.11 a/b/g/n/ac
Bluetooth	Smart 4.0 Smart Ready
Ethernet	2x Gigabit
USB	2x USB3 / 4x USB2
Computador	
CPU	i5 Haswell
Disco duro SSD	60 GB
Sistema operativo	Ubuntu LTS + ROS
Características eléctricas	
Batería	36 V 20 Ah Li-Ion
Elementos integrados	
Sensor láser (*Hokuyo* URG-04LX-UG1), IMU de 6 grados	

2.4.2 Sensores y actuadores integrados

Entre los principales dispositivos integrados con los que cuenta el robot *PMB-2*, los cuales serán utilizados para la navegación autónoma, se encuentran:

- Sensor láser *(Hokuyo URG-04LX-UG1*): sensor 2D tipo LIDAR (*Light Detection and Ranging o Laser Imaging Detection and Ranging*) para determinar la distancia de los objetos al láser midiendo el retraso entre la emisión de un haz pulsado y la

detección del reflejo. Este modelo posee un rango de detección de 5600 mm, escaneando un área de 240º con una resolución de 0.352º y una precisión de ± 30 mm para distancias entre 60 a 1000 mm, y de ± 3% para distancias entre 1000 a 4095mm. Es de bajo consumo (2.5 W) y funciona con un voltaje de 5V. Es diseñado para trabajar en interiores y su frecuencia de muestreo es de 10 Hz (ROS Components, 2016). Este sensor se encuentra en la parte frontal de la base, mide las distancias en el plano horizontal y publica las medidas en el tópico */scan* (En el siguiente capítulo se explicará a mayor detalle los tópicos en ROS)

- *IMU* (*Inertial Measurement Unit*) o unidad de medición inercial: es el dispositivo que permite la medición de la orientación y la aceleración en seis grados de libertad. Este se encuentra montado en el centro del robot y publica las medidas obtenidas en el tópico */base_imu*

- Sensor ultrasónico: se utiliza para medir las distancias a diferentes objetos al emitir una onda de sonido y convertir la señal reflectante que vuelve al sensor en una señal eléctrica. El robot lo utiliza para medir las distancias cortas o medias, para así evitar colisiones locales. Las medidas de este sensor son publicadas en el tópico */sonar_base.*

- Motores en las ruedas del robot: encargadas de mover la base y brindar los datos de odometría para conocer el posicionamiento de las ruedas y por ende del robot.

2.4.3 Software instalado

El ordenador interno que se incluye en el robot, contiene ya instalado por parte de la compañía fabricante, el siguiente software con las siguientes características:

- Sistema Operativo *Ubuntu* 12.04.1 *LTS* de 32 bits

- ROS: distribución *Hydro* con herramientas propias de *PAL Robotics*.

- Programación de los nodos para el funcionamiento de los sensores y actuadores en ROS: estos nodos se reutilizarán para lograr el movimiento de la base robótica, y se explicarán más adelante cuando los comandos de ROS hayan sido explicados.

- Programación de algunos algoritmos de navegación automática en ROS: estos algoritmos se programarán nuevamente, ya que como objetivo del trabajo se tiene estudiar e implementar diferentes algoritmos para lograr la navegación autónoma de la base.

3. ROS: *ROBOT OPERATING SYSTEM*

La creciente escalabilidad y alcance de la robótica en la actualidad ha generado la necesidad de crear diferentes estándares o ambientes de desarrollo, que permitan manejar la complejidad de los sistemas robóticos y facilitar la implementación del software de estos. Cada uno de estos estándares o *frameworks* han sido diseñados en respuesta a un propósito particular; por lo que es importante destacar que no existe un sistema de desarrollo perfecto para todo la robótica, sino que se debe estudiar cada caso para encontrar la mejor solución que se adapte al objetivo específico del robot a implementar (Quigley, y otros, 2009).

El Sistema Operativo Robótico (ROS™ por sus siglas en inglés; *Robot Operating System*) se define como un *framework* para desarrollar software con el fin de controlar diferentes tipos de robots en diferentes escenarios reales o simulados, con énfasis en la integración robótica a gran escala y la investigación colaborativa. Se puede describir como una colección de herramientas, bibliotecas y convenios que permiten simplificar la tarea de controlar el comportamiento de una gran variedad de sistemas robóticos y plataformas. ROS fue creado en el 2007 por el laboratorio de inteligencia artificial de *Stanford* (*SAIL*) durante el desarrollo del robot *STAIR* y fue continuado como proyecto por el grupo *Willow Garage*, el cual extendió sus conceptos e implementaciones. Actualmente, ROS se encuentra mantenido bajo la compañía *Open Robotics*, quien se encarga de realizar los lanzamientos de las nuevas versiones cada año desde el 2013. Los principales objetivos de su creación fueron; implementar una infraestructura flexible y abierta para programar software, ser el middleware de comunicación entre los diferentes dispositivos, así como animar el desarrollo de software robótico de manera colaborativa (Ortega, 2017).

Actualmente existen tres tipos de entorno; ROS 1 (R1), ROS 2 (R2) y ROS industrial, y diferentes versiones para cada uno de ellos, como lo son: Box Turtle R1 (2010), C Turtle R1 (2010), Diamondback R1 (2011), Electric Emys R1 (2011), Fuerte Turtle R1 (2012), Groovy Galapagos R1 (2012), Hydro Meduda R1 (2013), Indigo Igloo R1 (2014), Jade Turtle R1 (2015), Kinect Kame R1 (2016), Lunar Loggerhead R1 (2017), Ardent Apalone R2 (2017), Melodic Morenia R1 (2018), Bouncy Bolson R2 (2018), Crystal Clemmys R2 (2018), Dashing Diademanta R2 (2019), Eloquent Elusor R2 (2019), Noetic Ninjemys R1 (2020) y Foxy Fitzroy R2 (2020).

A pesar de su nombre, ROS no se considera estrictamente un sistema operativo (provee los servicios estándar, como lo son la abstracción del hardware, control de dispositivos de bajo nivel, comunicación entre procesos, etc.), sino una estructura de

capas de comunicación sobre un sistema operativo anfitrión, usualmente tipo *Linux*. Durante la etapa de diseño, la filosofía de funcionamiento que definió la arquitectura de ROS se puede describir en las siguientes cinco características (Quigley, y otros, 2009), (Pérez, 2017):

- Topología *peer-to-peer*: Los programas principales del sistema funcionan bajo una red entre iguales. Estos se conectan entre sí, y continuamente se encuentran intercambiando mensajes bajo el concepto de cliente-servidor sin necesidad de un servicio de enrutamiento, sino un mecanismo de búsqueda denominado *master*. Como principal ventaja de esta característica es que se obtiene la posibilidad de escalar el sistema de una mejor manera cuando la cantidad de datos aumenta.

- Basado en herramientas: ROS contiene una serie de programas genéricos, pequeños e independientes para facilitar su uso. Se incluyen diferentes herramientas que permiten navegar por el árbol de código fuente, configurar diferentes parámetros, visualizar las conexiones, graficar datos y generar documentación. Esto implica que ROS, no tiene un ambiente integrado de desarrollo (*IDE*) ni un sistema en tiempo de ejecución (*runtime enviroment*), sino que las funciones se desarrollan por medio de los diferentes módulos. Al tener la ejecución de las diferentes funcionalidades por medio de programas separados, se alienta a la creación de nuevas implementaciones para mejorar la adaptación de cada tarea en particular, obteniendo un sistema menos complejo y más estable.

- Multilenguaje: A la hora de programar los diferentes eventos que definen el comportamiento que tendrá el sistema robótico, se debe definir cuál es el lenguaje más apropiado a utilizar dependiendo de los requerimientos establecidos, y conocimientos técnicos del desarrollador. Por esto, ROS cuenta con flexibilidad en cuanto al uso de lenguajes de programación, permitiendo la escritura de los diferentes módulos en *C++* y *Python*. Para describir los mensajes de información que se utilizan, ROS utiliza lenguaje neutral de definición de interfaz (*IDL*). Estos se transforman a *Python* o *C* automáticamente al ser utilizado por los módulos, por lo que se facilita la creación de nuevos tipos de mensajes, además de ahorrar tiempo y disminución de errores. Actualmente se pueden contener bibliotecas escritas en *Ruby*, *Java*, *Matlab*, *Octave* y *LISP*; envolviendo las librerías soportadas.

- Ligereza: Las construcciones del sistema se realizan de manera modular por bibliotecas independientes dentro del árbol de código fuente utilizando archivos *cmake*; permitiendo con facilidad la extracción de código. ROS permite reutilizar *software* de numerosos proyectos de código abierto como los son los *drivers* de

sistemas de navegación, algoritmos de visión, algoritmos de planeamiento de rutas, entre otros. Además otra ventaja de esta característica es la continua mejora a través de los aportes de la comunidad; los paquetes de ROS pueden ser actualizados desde repositorios externos por medio de la aplicación de parches.

- Gratis y código abierto: Como se ha mencionado anteriormente, existen otros frameworks robóticos que presentan grandes atributos y ventajas, sin embargo estos suelen ser de paga o no tienen su código completamente disponible. Para facilitar la depuración de errores en todos los niveles del software, ROS presenta la característica de ser una plataforma completamente abierta. ROS se encuentra bajo una licencia *BSD* que permite el desarrollo de proyectos comerciales y no comerciales y se puede obtener de manera gratuita.

A pesar de que la curva de aprendizaje de ROS puede llegar a ser difícil y lenta debido a la cantidad de notaciones, conceptos y configuraciones necesarias para simular y modelar distintos robots, ROS cuenta con bastantes ventajas que facilitan el desarrollo de *software* robótico. Entre las principales ventajas se tiene que: es un sistema gratuitito de código abierto, cuenta con una gran comunidad de desarrolladores permitiendo así la reutilización de código proveniente de universidades, compañías y desarrolladores independientes, además del soporte proveniente de sus tutoriales, foros, y repositorios disponibles, presenta compatibilidad con diferentes robots (para el caso de este trabajo con el robot *PMB-2*), cuenta con flexibilidad de programación en diferentes lenguajes, presenta una topología modular con herramientas que permiten depurar y visualizar el código ejecutado, herramientas de simulación y la posibilidad de ejecución de tareas multi-hilo. Además su filosofía de funcionamiento está basada en comandos por terminal tipo *Unix*; lo que lo hace cómodo para los desarrolladores experimentados (Gallart del Burgo, 2013).

La creación y desarrollo de un sistema robótico completo es una tarea difícil que contempla bastantes segmentos. Implementar automáticamente tareas que son triviales para los humanos, implica una gran cantidad de pasos y algoritmos en los robots. ROS permite concentrarse en el *software* que definirá el comportamiento del robot; ya que ya cuenta con módulos capaces de leer datos de los sensores, enviar comandos a los actuadores y toda la infraestructura computacional que se necesita para poner a funcionar el sistema. Con ROS se pueden implementar gran colección de algoritmos que permiten desde construir mapas de entorno, navegar, manipular objetos, hasta algoritmos de visión de computador y toma de decisiones (Quigley, Gerkey & Smart,

2015). Para implementar estos algoritmos es necesario conocer cómo funciona ROS, lo cual se explica en la siguiente sección.

3.1 Organización de ROS

Topológicamente ROS funciona como una red de programas que realizan tareas determinadas e intercambian información de los procesos realizados. Para una mayor comprensión de cómo es la estructura de trabajo de ROS, se explica dividiéndola en cuatro secciones; nivel de manejo de archivos, nivel computacional, herramientas, programas y otros, y por último, la comunidad de ROS (Quigley, y otros, 2009), (ROS: Documentation Wiki, 2007-2020), (Gallart del Burgo, 2013).

3.1.1 Nivel de manejo de archivos

Para utilizar ROS, es necesario comprender como se organizan los archivos en el espacio de trabajo donde se ha instalado. Esta organización se presenta en la figura 3.1 y sus principales componentes se describen a continuación:

-*Workspace* o espacio de trabajo: es un conjunto de directorios donde se almacena el código y los archivos necesarios para empezar a utilizar ROS. Para habilitar este espacio, se utiliza la herramienta de compilación *catkin*, que consiste en un set macros tipo *CMake* y *scripts* de *Python* que permiten construir ROS en un directorio específico. En un mismo ordenador se pueden tener varios *workspaces*, pero solo se puede trabajar en uno a la vez, ya que solo se puede acceder a los datos generados por el espacio actual. Los comandos para inicializar este espacio son:

```
mkdir -p /workspace_test/src
cd /workspace_test
catkin_init_workspace
catkin_make
```

Los comandos anteriores se encargan de crear el espacio de trabajo bajo el nombre *workspace_test.* En este directorio, una vez inicializado se encontrarán las carpetas: *build*, *devel* y la previamente creada *src.* En el directorio *build*, se almacenan las librerías y ejecutables necesarios para utilizar el lenguaje *C*. El directorio *devel* contiene archivos de configuración. Cada vez que se necesite utilizar ROS, en este espacio de trabajo es necesario cargar el archivo de configuración localizado en esta carpeta, con el siguiente comando:

```
source /devel/setup.bash
```

Por cada terminal nueva que se vaya a utilizar es necesario repetir el comando anterior para poder ejecutar el código que se encontrará en el espacio de trabajo o modificar el archivo *.bashrc* del sistema para que esto se realice automáticamente. Finalmente en la carpeta denominada *src*, se almacena el código a utilizar mediante la estructura de paquetes.

-Paquetes: Se denominan como la unidad básica y central donde se organiza el *software* en ROS. Su directorio contiene el código fuente, bibliotecas, ejecutables, archivos de lanzamiento y de configuración, entre otros. Siguiendo la ideología modular, en un espacio de trabajo se suelen tener diferentes paquetes, cada uno con un objetivo distinto, por ejemplo; un paquete con el código necesario para realizar la navegación, otro para realizar el reconocimiento por visión por computador, otro para generar la ruta óptima, etc. Existen varios paquetes ya implementados por otros usuarios que se pueden descargar de repositorios tipo *git*, para ser incluidos en un proyecto e interactuar con paquetes creados por el usuario. Cada paquete tiene una serie de archivos denominados; manifiesto; estos describen su contenido, dependencias e interacción a la hora de la compilación con *catkin.* Para crear un paquete, se ejecuta el siguiente comando, en el directorio *src* del respectivo *workspace:*

```
catkin_create_pkg <nombre_paquete> <dependencias>
```

Las dependencias pueden ser otros paquetes u otras bibliotecas (por ejemplo *rospy;* librería para utilizar *Python*), sin embargo si no se listan cuando se ejecuta el comando anterior, pueden ser agregadas más adelante, modificando los archivos de manifiesto del paquete. Como recomendación los archivos se suele organizar como se muestra en la figura 3.1, donde dentro del paquete se crean directorios específicos para los diferentes tipos de archivos: *launch, config, scripts, src, msgs,* entre otros.

-Meta paquetes: Conjunto de paquetes que combinados ejecutan un propósito específico. Como se mencionó anteriormente, un proyecto de robótica en ROS puede estar compuesto por varios paquetes, debido a que los diferentes objetivos del robot se dividen de manera modular para facilitar su resolución. Los meta paquetes o *stack* pueden estar compuestos por paquetes descargados y creados, que juntos ejecutarán funcionalidades a alto nivel.

-Manifiesto: Archivos que contienen la información acerca de la configuración y dependencias a utilizar. Describen la información general de cada paquete específico,

licencias y dependencias con otros paquetes. Estos archivos son *package.xml* y *CMakeLists.txt*.

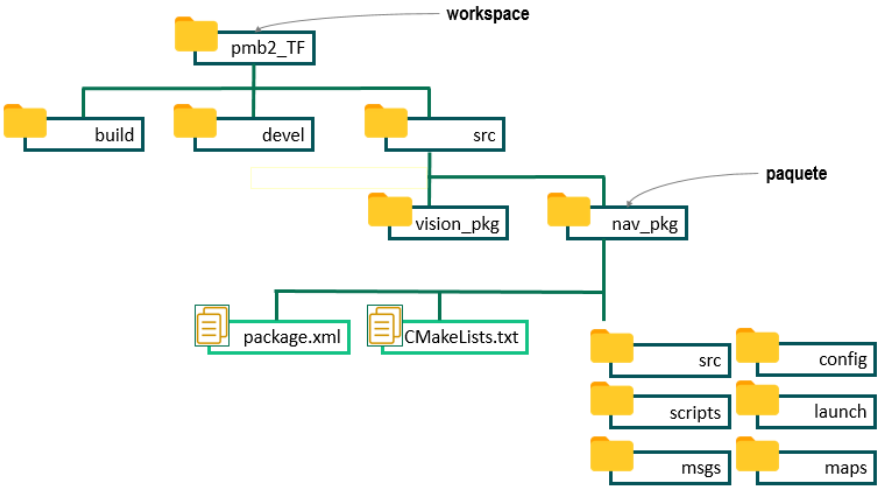


Figura 3.1 Nivel organizacional de archivos.

3.1.2 Nivel computacional

El sistema computacional de ROS se basa una red de nodos con un componente fundamental denominado núcleo, que se encarga de establecer la comunicación entre dos puntos. El intercambio de información entre los nodos se realiza por medio de mensajes principalmente con el método de publicación–suscripción. Los principales componentes para comprender el funcionamiento de la red (*peer-to-peer*) de ROS, son:

-Núcleo/*core/master*: Denominado *roscore;* este elemento se encarga de habilitar la red ROS usando el protocolo *XMLRPC*. Es el primer proceso que debe ser ejecutado ya que registra todos los nodos, tópicos y servicios existentes, los interconecta virtualmente entre sí y monitorea las unidades del sistema. Cuando un nodo es configurado, se conecta automáticamente a este núcleo (sabiendo su localización en la red; generalmente puerto 11311), el cual le provee la información de conexión de los otros nodos para que estos puedan transmitir mensajes entre ellos. Sin este elemento, los nodos no podrían encontrarse entre ellos, ya que forma la conexión directa tipo *p2p* con los otros nodos que publican o se suscriben a los mismos tópicos de mensajes (Figura 3.2). Este tipo de arquitectura se conoce como híbrida entre un sistema clásico de cliente-servidor y un sistema completamente distribuido. El núcleo también provee un servidor de parámetros que usa todo el sistema, funcionando como diccionario multi variable accesible a través de la red de nodos para almacenar y recuperar parámetros en tiempo de ejecución.

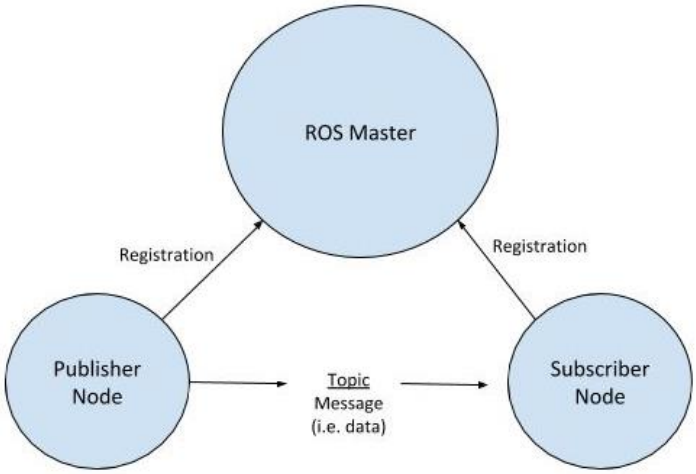


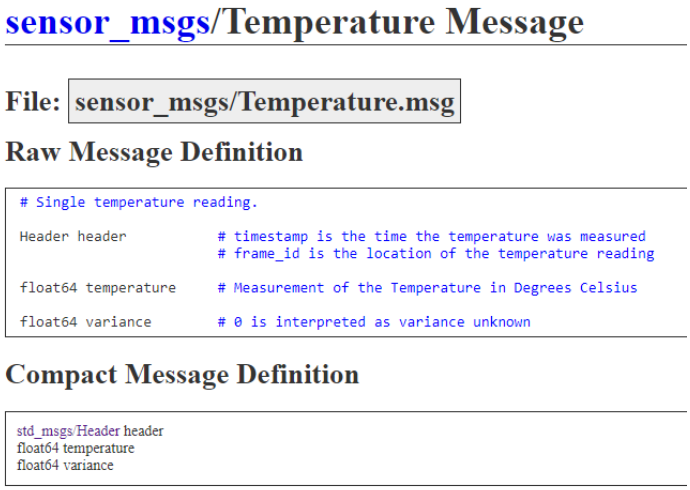
Figura 3.2. ROS *master.*

-Nodo: Módulo de *software* ejecutable que realiza la computación de las diferentes tareas planteadas para el robot. Al ser ROS modular, el *software* robótico se compone de varios nodos interactuando entre sí. Esta separación de las funcionalidades a través de distintos módulos simplifica el sistema, lo que facilita la depuración de fallos y errores. Los nodos se escriben en *C++* o *Python*, utilizando las bibliotecas *roscpp* o *rospy* respectivamente. Para la comunicación de los mensajes, los nodos utilizan tópicos, servicios o parámetros de servicio. Es decir; los nodos se pueden suscribir a diferentes tópicos para recibir mensajes de otros nodos, o bien publicar datos e información. Al crear un nodo es importante que este tenga un nombre único y característico. Para ejecutar uno de estos nodos, se puede ejecutar en la consola el siguiente comando:

```
rosrun <nombre_paquete> <nombre_ejecutable>
Ejemplo:
rosrun nav_pkg localization.py
```

-Mensajes: Estructura de datos con tipo definido que utilizan los nodos para transmitir información por medio de los tópicos. Estos pueden ser de tipos primitivos como enteros, boléanos, punto flotante, arreglos, entre otros. ROS cuenta con una gran cantidad de tipos de mensajes como los mencionados anteriormente y combinaciones de estos, sin embargo, si el usuario desea crear su propio tipo de mensaje, deberá crear un archivo con extensión *.msg* en el directorio correspondiente del paquete y deberá construir nuevamente el espacio de trabajo, actualizando previamente los archivos de manifiesto para incluir los nuevos mensajes. Un ejemplo de un mensajes ya provisto por ROS se muestra en la figura 3.3, denominado *sensor_msg/temperature*; el cual se

compone por un mensaje tipo *header* (que a su vez engloba dos variables de diferente tipo), y dos variables de tipo flotante para almacenar la temperatura y la variancia. Por lo tanto si se tiene un nodo encargado de realizar las mediciones de temperatura con un sensor, este nodo comunicará los resultados a otros nodos utilizando este tipo de mensaje, y así en cada caso según se requiera.



sensor_msgs/Temperature Message

File: sensor_msgs/Temperature.msg

Raw Message Definition

```
# Single temperature reading.

Header header           # timestamp is the time the temperature was measured
                        # frame_id is the location of the temperature reading

float64 temperature     # Measurement of the Temperature in Degrees Celsius

float64 variance        # 0 is interpreted as variance unknown
```

Compact Message Definition

```
std_msgs/Header header
float64 temperature
float64 variance
```

Figura 3.3. Ejemplo de mensaje en ROS. (ROS: Documentation Wiki, 2007-2020).

Los comandos para observar la lista de mensajes disponibles en el sistema, observar los mensajes de un paquete en específico o conocer las características de un mensaje son respectivamente:

```
rosmsg list
rosmsg <nombre_paquete>
rosmsg show <nombre_mensaje>
```

-Tópicos: Los tópicos se definen como los buses que transportan los mensajes entre nodos. Un nodo envía un mensaje al publicarlo a un tópico con un nombre específico, y si un nodo desea recibir cierto dato específico se suscribirá al tópico apropiado que provee tal información. Para un mismo tópico, pueden haber varios publicadores y suscritores y un nodo puede publicar y suscribirse a múltiples tópicos para obtener y tratar diferentes datos según se necesite, simplemente se necesita coincidir con el tipo del mensaje. La creación de datos, y el consumo de estos es un proceso desacoplado, es decir si no estaba suscrito cuando se publicó un mensaje este se pierde y se tendrá que esperar el siguiente mensaje. Para evitar esto, y lograr que un nodo reciba el último mensaje, es necesario que al suscribirse al tópico, se defina este como *latched*. Los tópicos utilizan el protocolo *TCPROS*, aunque en ciertos casos se

utiliza *UDPROS* el cual es de baja latencia útil para casos de tele-operación. Entre los comandos relevantes que se pueden utilizar con los tópicos, se encuentran: el comando *list;* para conocer los tópicos activos del proceso, y el comando *echo* para obtener el mensaje que está siendo publicado en ese tópico específico:

```
rostopic list
rostopic echo </nombre_tópico>
```

Un ejemplo que resume los elementos explicados; nodos, mensajes y tópicos, se observa en la figura 3.4. En este ejemplo, existen varios nodos con diferentes objetivos, uno para computar la odometría del robot según los datos de los sensores de las ruedas, otro para planear la ruta a seguir y por último, un nodo para controlar el motor. En este caso el primer nodo publica una serie de datos (mensaje que contiene la posición en los ejes x, y, z) en el tópico 1 denominado *odom.* El nodo que planea la ruta está suscrito a este tópico, por lo que recibirá los datos, para computar con ellos y planear la ruta óptima. Además, publicará en un segundo tópico los comandos de velocidad necesarios (mensaje que contiene la velocidad lineal y angular). El tercer nodo al estar suscripto al tópico 2, recibirá estos datos para así poder trasladar estos comandos al motor y mover el robot.

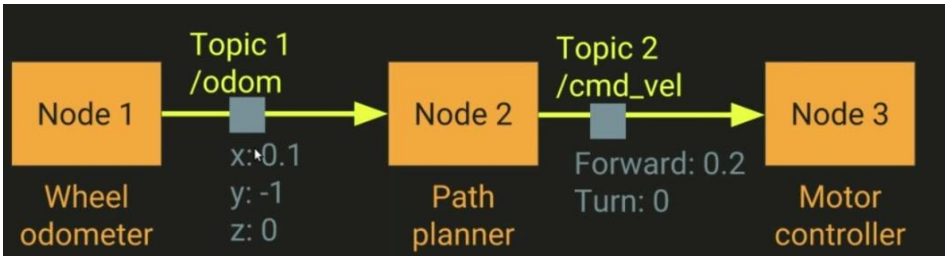


Figura 3.4. Ejemplo de comunicación entre nodos en ROS.

-Servicios: Los servicios permiten que dos nodos se pueden comunicar utilizando el método de petición y respuesta. En este proceso solo dos nodos pueden interactuar, permiten a un nodo cliente llamar a una función (*request*) que se ejecuta en otro nodo servidor y esperar la respuesta obtenida (*response*). Un ejemplo de un caso de uso de servicios, es el uso de una cámara, la cámara se encuentra disponible pero solo se necesita capturar una imagen en ciertas situaciones. En estos casos se envía una petición al nodo de control de cámara para que capture la fotografía y este responderá con el resultado es decir con la imagen. ROS cuenta con una serie de servicios ya definidos que combinan diferentes tipos de mensajes y también permite la creación de nuevos. Para crearlos, al igual que los mensajes, se debe guardar el archivo de

definición (con una o varias variables definidas para la petición, y una o varias variables definidas para la respuesta, separadas por los caracteres: “---”) en el directorio especifico del paquete, modificar los archivos de *cmake* y *package,* y reconstruir *catkin* nuevamente. Para observar los servicios disponibles u obtener información un servicio específico, se usan los siguientes comandos:

```
rossrv list
rossrv show <nombre_servicio>
```

-Acciones: Las acciones cumplen un papel similar a los servicios solo que de manera asíncrona. El código continúa ejecutándose aunque no se haya obtenido el resultado de la acción. La estructura de las acciones viene definido por un objetivo (*goal*), un resultado (*result*) y estado actual o realimentación de como la petición está progresando (*feedback),* como se observa en la figura 3.5. Por ejemplo, si se desea mover rotacionalmente una rueda, se envía al servidor de la acción el objetivo con la magnitud deseada de giro en radianes, el resultado se dará una vez que se termine el proceso con el desplazamiento angular en radianes desde la posición inicial, y en el mensaje de *feedback* se obtendrá constantemente la rotación restante en radianes antes de completar la petición.

```
# The desired heading in radians
float32 theta
---
# The angular displacement in radians to the starting position
float32 delta
---
# The remaining rotation in radians
float32 remaining
```

Figura 3.5. Estructura de definición de una acción.

El mensaje de realimentación es relevante para decidir si se desea cancelar la acción antes de cumplir con la meta, como es deseado en determinados casos, o para llevar el control del progreso de la acción (Figura 3.6). Al igual que los mensajes y los servicios, se pueden crear acciones definidas por el usuario siguiendo el mismo proceso ya explicado. Cada una de las partes de una acción se despliega en ROS por medio de tópicos, por lo que los comandos son los mismos vistos en la sección de tópicos.

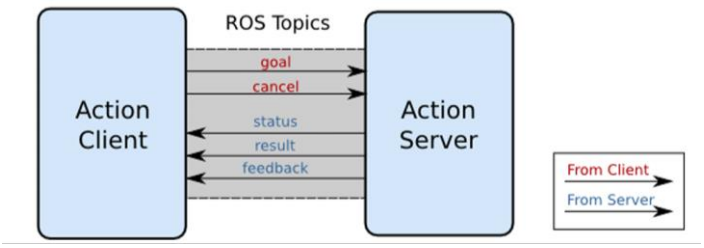


Figura 3.6. Interfaz de funcionamiento de las acciones en ROS.

-Bolsas*/bags*: Los *bags* son la herramienta para grabar y reproducir datos en ROS. Son archivos que almacenan información de mensajes, tópicos y servicios para utilizarlos después y realizar otras tareas con esta información, como pruebas de algoritmos, simulaciones o del programa. Los comandos para grabar, obtener información y reproducir los datos, son respectivamente:

```
rosbag record <nombre_topicos>
rosbag info <nombre_bag>
rosbag play <nombre_bag>
```

3.1.3 Herramientas, programas y otros

ROS ofrece un conjunto de herramientas y comandos para facilitar la implementación del software robótico, simular en diferentes entornos, lanzar varios componentes a la vez, visualizar el flujo de datos entre los módulos, entre otros. A continuación se explicarán algunas de estas herramientas.

***-roslaunch*:** Como ROS se considera un sistema modular, donde un programa se compone de varios nodos ejecutándose simultáneamente, no es práctico tener que ejecutar los nodos uno por uno con el comando *rosrun*. Por esto existe la herramienta *roslaunch*, la cual permite lanzar varios nodos al mismo tiempo para así realizar tareas más complejas. Además permite configurar parámetros globales del sistema, establecer opciones predeterminadas, iniciar automáticamente el *roscore*, y terminar todos los procesos a la vez en caso de finalizar por medio del comando *roslaunch*. Para ejecutarlo, se utilizan archivos de configuración descritos en *XML* de extensión *".launch"* que especifican que parámetros, nodos y demás archivos utilizar. El comando a ejecutar en la consola para utilizar esta herramienta es:

```
roslaunch <nombre_paquete> <nombre_launchFile>
Ejemplo:
roslaunch nav_pkg map_localization.launch
```

***-Namespaces*:** En ROS, es necesario que todos los elementos tengan nombres únicos ya que es su principal identificador. En ciertas ocasiones se tienen varios dispositivos iguales, como por ejemplo las ruedas de un robot móvil, o un sistema con dos cámaras iguales. Si se tienen dos o más dispositivos iguales lo ideal es reutilizar código. Esto se logra utilizando los comandos de *namespaces* los cuales permiten separar objetos idénticos con un *slash* en los archivos de configuración como los *.launch,* o los archivos de definición del robot como los *.urdf.* Por ejemplo si se tienen dos cámaras¸ se pueden definir dos nodos en el archivo de configuración utilizando el mismo código y el *namespace*; *left/image* y *right/image* (figura 3.7). Si un programa está esperando recibir la imagen a través de un tópico y recibe imágenes de los nodos anteriores, habrá un re-direccionamiento al tópico correspondiente; sin tener que modificar el código fuente, con la información acerca de la cámara que tomó la imagen.

```
<node name="image" pkg="cam_pkg" type="camera_code.py" ns="left" />
<node name="image" pkg="cam_pkg" type="camera_code.py" ns="right" />
```

Figura 3.7. Extracto de archivo *.launch* para definición de nodos.

-TF: Usualmente, un robot está compuesto por varios subsistemas como pueden ser; la base, los sensores, las cámaras, los brazos, las pinzas, etc. Cada uno de estos subsistemas se denominan *frames* y su *pose* (posición (*x,y,z*), y orientación (*roll,pitch,yaw*) con respecto a su origen (usualmente el centroide)) va cambiando respecto al tiempo mientras se estén ejecutando diferentes acciones de movimiento en el robot (figura 3.8). Por esta razón, existe un paquete denominado *tf,* el cual se encarga de llevar el seguimiento de las coordenadas de los diferentes *frames* a lo largo del tiempo de ejecución. Este paquete mantiene la relación entre las partes del robot en un árbol estructurado (figura 3.9) que se actualiza constantemente dependiendo de la posición en la que se encuentra el robot y permite al usuario conocer la transformada entre dos puntos. Esta herramienta es muy importante para interpretar correctamente los datos. Por ejemplo si se tiene un sensor láser, es necesario conocer donde se encuentra este colocado con respecto a la base, para poder identificar la ubicación del robot con los datos medidos. Este es uno de los paquetes más importantes, ya que permite relacionar las dimensiones espaciales y temporales, unificándolas en un sistema de referencia común.

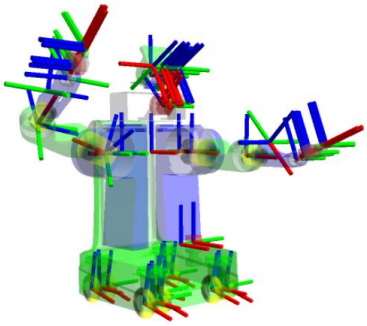


Figura 3.8. Robot compuesto por diferentes *frames*.

Para obtener el árbol estructurado con las relaciones del robot en un documento *pdf*, se debe ejecutar el siguiente comando:

```
rosrun tf view_frames
```

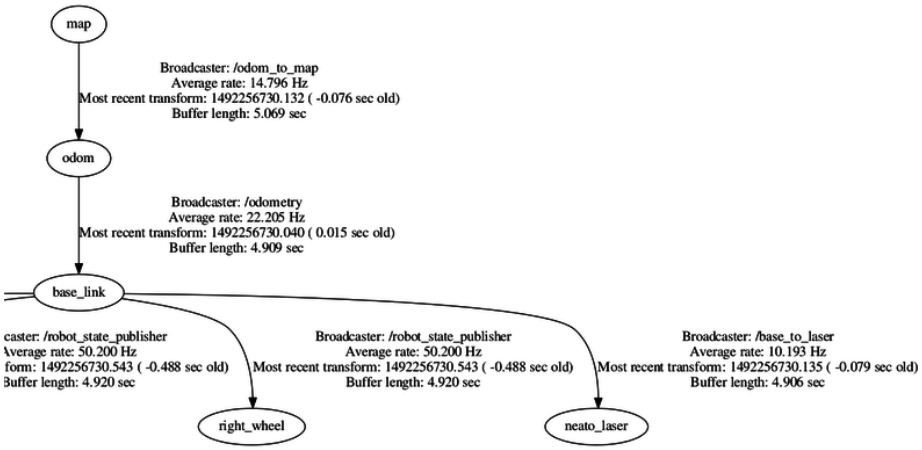


Figura 3.9. Extracto del árbol de transformadas de un robot.

-Archivos URDF: A la hora de simular robots en ROS, es necesario tener uno o varios archivos que definan los parámetros cinemáticos y dinámicos de los diferentes componentes que conforman el robot. Para esto ROS utiliza archivos tipo *URDF;* un archivo de formato específico basado en *XML* que contiene la descripción del diseño del cuerpo del robot, su apariencia, información acerca de las uniones y otras características tales como: material, inercia, peso, restricciones de velocidad y otras. Este tipo de archivo también permite cargar archivos tipo *CAD* para conformar diseños más complejos y visualizarlos en simulaciones. Un ejemplo de cómo crear diferentes figuras y su respectiva unión se muestra en la figura 3.10. Para robots más grandes y complejos se suele utilizar el macro lenguaje *Xacro*, que permite construir archivos *XML* más pequeños y más entendibles al agrupar expresiones y reutilizar definiciones.

```xml
<?xml version="1.0"?>
<robot name="multipleshapes">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>

  <link name="right_leg">
    <visual>
      <geometry>
        <box size="0.6 0.1 0.2"/>
      </geometry>
    </visual>
  </link>

  <joint name="base_to_right_leg" type="fixed">
    <parent link="base_link"/>
    <child link="right_leg"/>
  </joint>

</robot>
```

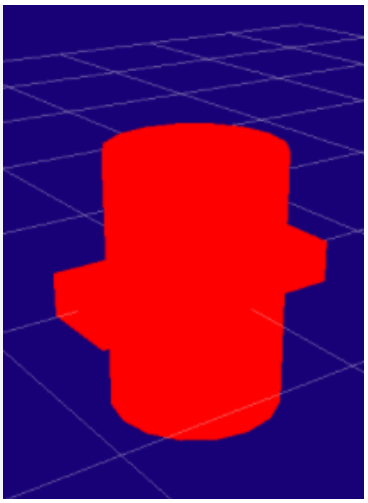


Figura 3.10. Archivo de definición URDF y su representación en simulación.

-Rviz: Herramienta de visualización 3D que permite visualizar el robot, diferentes tipos de mapas, objetos, datos de los sensores, imágenes provenientes de las cámaras, marcadores, entre otros. Cada uno de estos elementos se configura con un tópico a través de su nombre, y esto permite visualizar la información obtenida de manera más significativa. Posee una interfaz simple para seleccionar la información que se desea observar y configurar los diferentes parámetros. Para utilizar *Rviz,* se puede ejecutar llamándolo junto con los nodos correspondientes en un archivo *.launch con un* archivo configurado previamente o se puede ejecutar en consola directamente el comando:

```
rosrun rviz rviz
```

***-Gazebo*:** Simulador que permite emular el entorno de un robot y los datos sensoriales provenientes de este mundo virtual. Se integra con ROS, utilizando los mensajes y los servicios y puede ser cargado desde los ficheros *.launch* por medio de archivos tipo *.world,* que describen las propiedades geométricas y parámetros del entorno. Este simulador permite probar algoritmos, diseños de diferentes robots, realización de pruebas de regresión, escenarios con inteligencia artificial, y demás. Es una interfaz gráfica robusta que utiliza la ingeniería dinámica para la física de cuerpos rígidos para definir el comportamiento en diferentes situaciones.

***-roswtf*:** Es una herramienta que se utiliza para diagnosticar problemas en el sistema de ROS que está siendo ejecutado. Al ejecutar el comando *roswtf,* este examinará la configuración y las variables de ambiente para tratar de determinar si existe algún problema. Esta examinación se realizará sobre el directorio sobre el cual

fue ejecutado el comando y sus dependencias, aunque también se puede ejecutar el comando junto con un archivo de lanzamiento (tipo *launch*). Esta herramienta produce un reporte conteniendo los mensajes de advertencia y errores encontrados para facilitar al usuario resolver los diferentes problemas.

***-rqt_graph*:** Esta herramienta permite generar un gráfico que representa las conexiones de nodos y tópicos presentes en un sistema (figura 3.11). El comando se puede ejecutar en cualquier instante, y cada vez que se ejecute, se actualizará para representar el estado actual del robot.

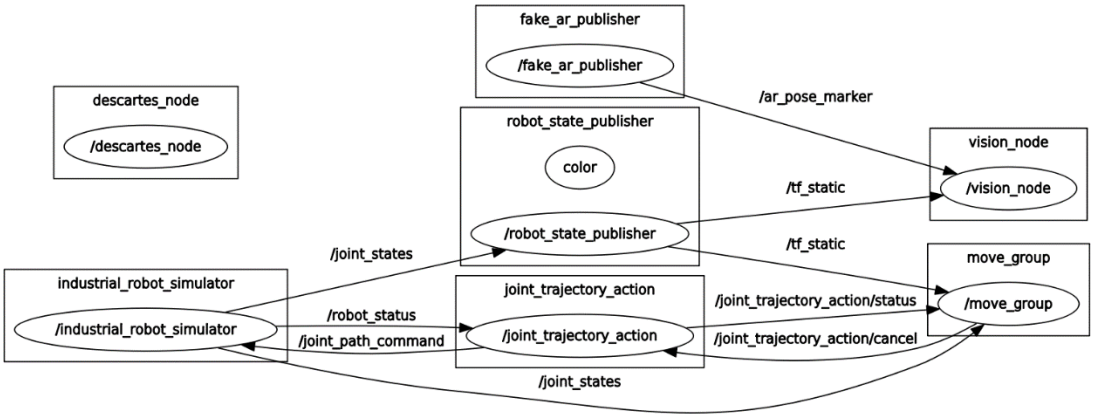


Figura 3.11. Gráfico generado con el comando *rqt_graph.*

3.1.3 Comunidad

Como se ha mencionado previamente, una característica importante de ROS, es su comunidad; que permiten el intercambio de *software* y conocimiento. En el campo de la robótica, la comunidad de ROS, es de las más grandes y más utilizadas. Sus principales recursos son:

-Distribuciones y repositorios: Existen múltiples repositorios en *git* y *SVN* donde se pueden encontrar paquetes listos para instalar; mantenidos y actualizados según las diferentes versiones de ROS.

-Wiki: La página principal de ROS, cuenta con una wiki, mantenida por los usuarios, donde se puede encontrar infinidad de información, documentación y tutoriales de los conceptos, comandos y diferentes paquetes.

-Blog y foro: ROS cuenta con distintos foros para resolver las dudas de los diferentes usuarios, lo que promueve un ambiente de compañerismo y solidaridad por parte de la comunidad robótica. Además se cuenta con una lista de distribución de

correos para comunicar los lanzamientos de nuevas versiones y existe un sistema de tiquetes para resolver diferentes problemas dependiendo de su criticidad.

Desde su lanzamiento, se han compartido alrededor de 2000 paquetes, los cuales son mantenidos actualmente por personas en todo el mundo. Además, más de 80 robots comerciales son soportados por esta plataforma y se pueden encontrar cerca de 1850 *papers* académicos relacionados con ROS. Esta gran cantidad de información permite que no sea necesario escribir el *software* desde cero a la hora de realizar un proyecto utilizando esta infraestructura de desarrollo (Quigley, Gerkey, & Smart, Programming Robots with ROS: A practical introduction to the robot operating system, 2015).

4. NAVEGACIÓN AUTÓNOMA

Como se mencionó en el capítulo 2, los robots se pueden clasificar en tres grandes ramas como lo son los operados, los automáticos y los autónomos. En los operados, su control depende de la intervención de los seres humanos para realizar las tareas; como por ejemplo un brazo robótico tele operado utilizado para realizar cirugías. Los automáticos, son capaces de repetir actividades sin tanta intervención pero en ambientes controlados, un ejemplo de estos son los brazos utilizados en las líneas de producción. Finalmente los robots autónomos, en contraste a los anteriores, son capaces de realizar tareas en ambientes cambiantes, y tomar sus propias decisiones para cumplir así los objetivos propuestos. Por ejemplo, los robots móviles de navegación autónoma que reparten los medicamentes en los hospitales o los vehículos sin conductores. Al inicio de la robótica, las investigaciones y desarrollos estuvieron más enfocados en los robots industriales tipo brazos o pinzas montados sobre una base fija. Con el paso del tiempo, estas investigaciones se han ido diversificando, permitiendo así de dotar a los robots con nuevas y diferentes características, entre ellas la movilidad en un ambiente desestructurado (es decir un ambiente que no fue adaptado para facilitar las acciones del robot); o la capacidad de mantener conversaciones con las personas. Estas características permiten al robot suplir un gran rango de aplicaciones tales como limpieza de edificios, distribución o entrega de paquetes, exploración de terrenos, guías en museos o bibliotecas, control de inventario o activos en empresas o supermercados, rescate, transporte y muchas más (Gallart del Burgo, 2013).

Para cumplir con el objetivo del presente trabajo de realizar diferentes recorridos en los laboratorios docentes, es necesario que la base robótica *PMB-2* pueda movilizarse de manera autónoma de un punto a otro evitando la colisión con objetos que se encuentre en el camino, ya sean estáticos (como muebles) o móviles (como personas), con mínima intervención humana. Debido a que se trabaja de entornos desconocidos y cambiantes y hay muchas variables a tener en consideración, se subdivide el problema en diferentes tareas las cuales se pueden definir en cinco secciones; percepción, trazado del mapa, localización, planeamiento de rutas y control de movimiento. Estas etapas y su relación se muestran en la figura 4.1 (Siegwart, Scaramuzza, & Nourbakhsh, 2011).

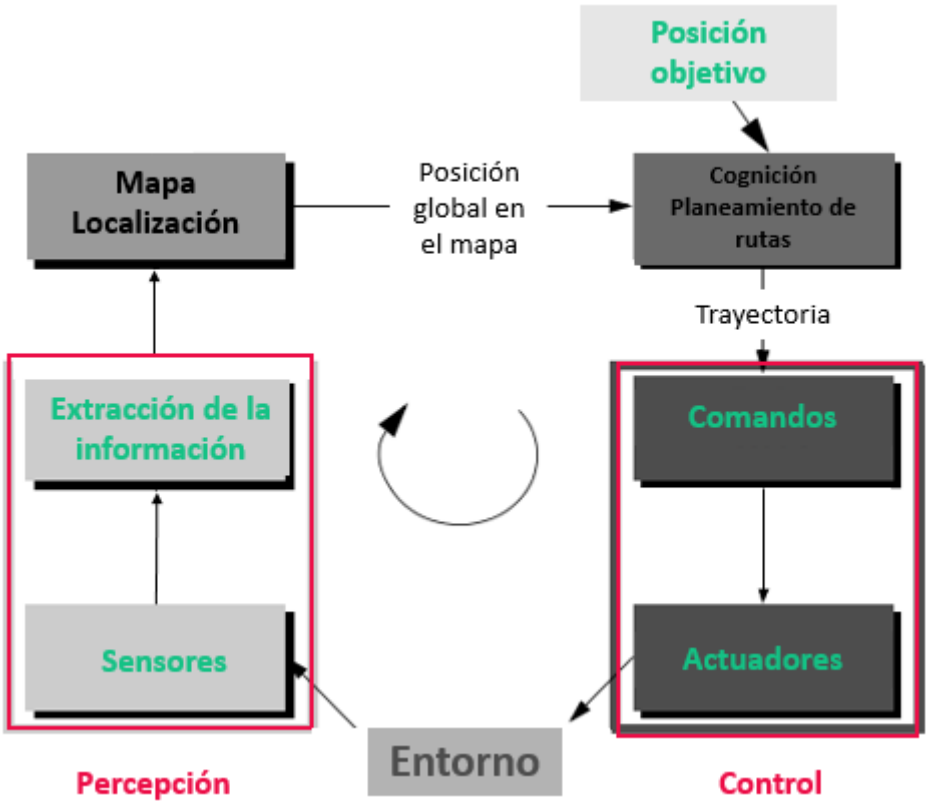


Figura 4.1. Etapas para navegación autónoma.

4.1 Percepción

La etapa de percepción de la información, es la encargada de obtener los datos necesarios para conocer el entorno; reconocer objetos, lugares y eventos que lo componen; para esto el robot debe tener algún tipo de sensor que le permita obtener la información para así poder interpretar los datos. Dependiendo de la aplicación, así serán los sensores necesarios, por ejemplo; puede ser una cámara, un sensor láser, sensor ultrasónico, etc. Para este caso específico, se utiliza el sensor láser *LIDAR* presente en el robot. La base robótica ya cuenta con los controladores y nodos responsables para obtener la información necesaria del sensor de manera que se pueda interpretar y usar en las otras fases necesarias para la navegación. Este nodo tiene como nombre *Hokuyo* y se encarga específicamente en gestionar los datos de lectura del *LIDAR* y publicarlos en el tópico */scan*, el cuál proporciona la siguiente información:

- Marca de tiempo de la primera medida realizada por el sensor y el identificar del *frame* correspondiente al sensor.

- Ángulo mínimo y máximo de escaneo en radianes; es decir rango de escaneo del sensor.

- Distancia angular y tiempo entre medidas en radianes y segundos respectivamente.

- Arreglo de datos de distancia obtenidos por el sensor en cada uno de los ángulos que conforman el rango de medición del láser, en metros.

- Valor mínimo y máximo de los datos en metros.

- Arreglo de datos con el valor de intensidad de los rayos reflejados por el láser. Suele utilizarse para conocer ciertas características de los materiales, sin embargo, esta información no suele ser muy utilizada en navegación.

En la siguiente imagen se puede observar cómo se visualiza este tópico en *Rviz,* en una simulación del robot *PMB-2* al encontrarse frente diferentes objetos.

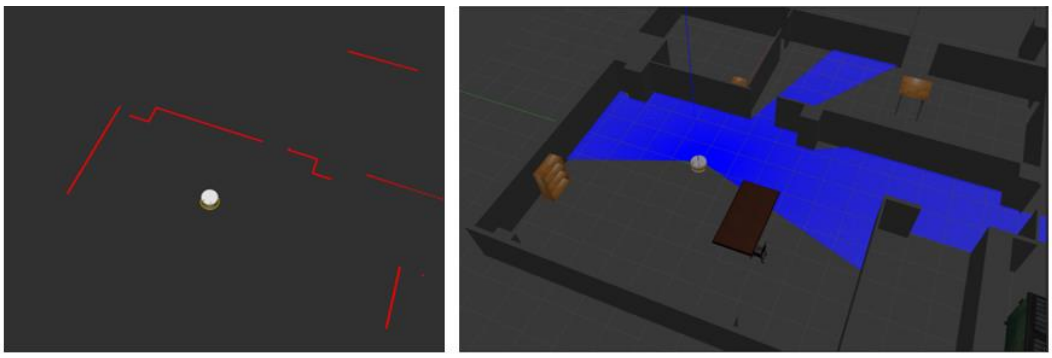


Figura 4.2. Datos obtenidos del tópico /scan en *Rviz*.

4.2 Mapa

Si se desea dotar al robot con la capacidad de navegación es elemental saber dónde se encuentra y a donde se quiere ir, por lo tanto se necesita un modelo matemático que represente el entorno que rodea al robot; un mapa (Zamora, 2015).

Para poder obtener el mapa es necesario saber dónde se encuentra el robot mientras se realiza el modelo, pero a su vez para saber la localización de este, se debe tener un mapa. Por lo tanto se necesitan solucionar dos tareas simultáneamente; obtención del mapa *(mapping*) y la localización del robot. *Mapping* es el problema de integrar la información reunida por los sensores e interpretar los datos en una representación, respondiendo a la pregunta de ¿cómo luce el mundo? Localización, es el problema de estimar la posición y orientación del robot relativa al mapa, respondiendo a la pregunta ¿dónde estoy? La resolución simultánea de estas dos tareas se conoce

como el problema *SLAM* (*Simultaneus Localization And Mapping*), y existen varias soluciones con diferentes algoritmos y enfoques probabilísticos propuestos desde los años 80. Actualmente esta área es uno de los principales focos de investigación en el campo de la robótica y en las últimas décadas se han buscado diferentes maneras de obtener el modelo espacial del ambiente físico a través de lo que percibe el robot.

La mayoría de las soluciones se basan en la probabilidad, ya que es la mejor manera de hacer frente a las dificultades presentadas en el proceso como los son el ruido y la incertidumbre de los sensores y actuadores, y el entorno dinámico, donde los objetos cambian constantemente de posición. El teorema de *Bayes*, permite minimizar las incertidumbres de la solución, tomando la posición del robot y de su entorno como variables aleatorias; para determinar la configuración más probable de acuerdo a las medidas que se van adquiriendo (Thrun, 2002).

El teorema de *Bayes*, establece que para dos variables aleatorias; *x,d*:

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

Se vincula la probabilidad de *x* dado *d* con la probabilidad de *d* dado *x*. Para este caso *x* es el mapa del robot que se basa en la información que nos otorga la variable *d;* medidas del sensor. Por lo que se puede resolver recursivamente conociendo la probabilidad de obtener una medida *d* bajo la hipótesis del estado *x*, y el grado de confianza de ese estado antes de recibir los datos. En robótica, tendremos dos tipos de medidas en este problema; las proporcionadas por los sensores*,* y las de los codificadores de los motores (odometría)*,* medidas que son variantes en cada instante de tiempo y tienen un error o incertidumbre asociada.

Para poder integrar este esquema con datos variantes en el tiempo se suele usar diferentes algoritmos recursivos que se encargan de extraer las características del entorno por medio de la detección y asociación de objetos de referencia conocidos como *landmarks*, estimación del estado y actualización de estas características. Es decir, cuando la odometría cambia por que el robot se mueve, se actualiza el filtro que estima la incertidumbre de la posición, seguidamente la información del ambiente es extraída y se asocian los objetos observados a los que ya se habían observado previamente. Los nuevos *landmarks* son utilizados para actualizar la posición del robot en el filtro y así continuamente. Si el punto de referencia no es estacionario con respecto al tiempo (una persona moviéndose en el entorno) es descartado del mapa. Con esta información se puede ir construyendo lo que se conoce como un mapa de celdillas o de ocupación; en

el cual se discretiza el espacio en unidades pequeñas que tendrán un valor de ocupadas o vacías con cierto nivel de confianza.

A continuación se describirán algunos algoritmos para la obtención del mapa, su uso en ROS y los resultados obtenidos con cada uno.

4.2.1 *Gmapping*

Fue desarrollado en 2007 y es uno de los algoritmos más utilizados para las aplicaciones robóticas que necesiten un mapa de ocupación 2D (Filipenko & Afanasyev, 2018). Este algoritmo utiliza el filtro de partículas *Rao-Blackwellized*, en el que cada partícula individual transporta un mapa del entorno, y utiliza técnicas adaptativas para reducir el número de partículas necesarias para crear el mapa de celdillas. Este enfoque computa la distribución tomando en cuenta no solo el movimiento del robot sino también sus observaciones más recientes, disminuyendo la incertidumbre de la posición del robot en la etapa de predicción del filtro (Grisetti, Stachniss, & Burgar, 2007).

En ROS, se cuenta con un paquete que implementa este algoritmo para la creación de mapas del entorno; denominado *slam_gmapping.* Para la correcta utilización de este paquete es necesario obtener información de los siguientes tópicos:

- */tf*: las transformadas necesarias para relacionar las diferentes partes del robot; el *frame* correspondiente a la base, al sensor y a la odometría. Es preciso conocer donde se encuentra montado el láser con respecto al robot, por ejemplo si un obstáculo es detectado a 20 cm, esta distancia será con respecto al láser, por lo que para saber a cuánto se encuentra el obstáculo realmente del centro del robot es necesario conocer la transformada del sensor a la base.

- */scan*: las medidas obtenidas por el sensor láser *LIDAR* como se explicó en la etapa de percepción, ya que este nodo toma cada lectura del sensor y la transforma al *frame* de la odometría. Es muy importante tener buenos datos del sensor y de odometría para la correcta formación del mapa.

Con estas información, el algoritmo será capaz de ir creado el mapa del entorno, por lo que publicará la información en dos tópicos; /*map*; mensaje de tipo *OccupancyGrid* donde el mapa obtenido será publicado y actualizado periódicamente; y el tópico */entropy;* mensaje tipo *Float64*, donde se publica un estimado de la entropía

de la distribución sobre la posición del robot (entre mayor sea este número; más incertidumbre se tiene de la posición).

Para utilizar este paquete, es necesario modificar algunos parámetros que dependen de las características del robot y el láser a utilizar. Estos parámetros se pueden configurar en un archivo tipo *YAML,* que se carga en conjunto al nodo encargado de ejecutar el algoritmo. Entre los principales parámetros a configurar se encuentran: el nombre de los *frames* referidos a la base robótica (comúnmente llamado */base_link*), el mapa y la odometría; el intervalo en segundos para actualizar el mapa, el tiempo de muestreo del sensor, los rangos máximos y mínimos del láser a utilizar, las medidas y resoluciones deseadas del mapa, la distancia lineal y angular y el tiempo necesario para realizar una lectura de datos, el número de partículas del filtro, entre otros (ROS Wiki, 2013). La configuración utilizada para este proyecto se encuentra en el repositorio de git (Anexo I).

4.2.2 *Hector mapping*

Se desarrolló en 2011, como una optimización de los algoritmos de *SLAM* en la Universidad Técnica de *Darmstadt*, especialmente para entornos no estructurados como escenarios de búsqueda y rescate. Es muy similar al algoritmo presentado anteriormente, con la salvedad que para este, la odometría no es necesaria. Por lo que para ciertos proyectos donde no se tiene esta información puede resultar bastante útil. Para el uso de este, se requiere que la tasa de refrescamiento del sensor sea alta. Como ventajas de este algoritmo se tiene que consume muy pocos recursos computacionales para resolver las ecuaciones necesarias, por lo que se puede integrar en sistemas de bajo costo o bajo consumo (Kohlbrecher, von Stryk, Meyer, & Klingauf, 2011).

Para implementar este algoritmo en ROS, se cuenta con el paquete denominado *hector_mapping,* el cual al igual que *gmapping,* necesita suscribirse a los tópicos de */tf*, y */scan*, además de un tópico para reiniciar el mapa y la posición del robot si fuera necesario; */syscommand*. Como resultado de usar este paquete se obtendrá el mapa en el tópico */map*, además de la posición estimada del robot; */slam_out_pose*, y la posición estimada del robot con una incertidumbre gaussiana; */poseupdate*. Este paquete además cuenta con una serie de servicios para reiniciar el mapa con la posición actual, pausar el procesamiento de los escaneos del láser o reiniciar todo el proceso incluyendo la ubicación del robot.

Al igual que el paquete anterior, se pueden configurar diferentes parámetros con un archivo de tipo *YAML*. Entre los parámetros modificables se encuentran la resolución y tamaño del mapa a crear, el periodo de publicación del mapa, así como los valores del sensor como su distancia y rango máximo y mínimo (ROS Wiki, 2013).

4.2.3 *Karto mapping*

Karto es un algoritmo gráfico de resolución de *SLAM* desarrollado por *Karto Robotics* en 2010. Este algoritmo trabaja organizando la información de posición del robot en un gráfico para optimizarlo y minimizar el error de la función. La construcción del gráfico es dependiente de los datos del sensor, sin embargo las optimizaciones realizadas requieren muchos recursos computacionales. Para la construcción del gráfico se utiliza la posición del robot como un nodo, cuando el robot se mueve se representa con flechas que conectan a los nodos. Si el robot vuelve a una posición conocida, se crea un lazo cerrado que conecta el grafico al último nodo conocido. Para esto se utiliza la información de odometría corregida con las medidas del sensor (Vanelli, 2019).

Para usar este algoritmo en ROS, se cuenta con el paquete *slam_karto* con sus parámetros configurables a través de un archivo *YAML*. Para el uso de este paquete es necesario suscribirse a */tf* y */scan.* Y los resultados obtenidos se obtienen en los tópicos */map* y */visualization_marker_array* (gráfica de posiciones que se actualiza periódicamente) (ROS Wiki, 2013).

4.2.4 Gestión del mapa

La información publicada en el tópico */map* para el caso de los tres paquetes explicados anteriormente, proporciona una serie de valores para cada celda que se encuentran entre 0-100 conocido como *Occupancy Grid Map* o *OGM*; donde un valor de 0 significa que el espacio se encuentra completamente libre, un valor de 100; completamente ocupado y como caso especial un valor de -1, que indica que se desconoce el estado de ese espacio. Con esta información se puede generar un mapa que se puede guardar para ser utilizado por otros nodos posteriormente. Para generarlo y publicarlo (para poder tener acceso a esta información en todo momento), es necesario tener un nodo encargado de gestionar los servicios del mapa; paquete conocido como *map_server*.

Una vez instalado este paquete, y ejecutado alguno de los nodos anteriores, se puede guardar el mapa con el siguiente comando:

```
rosrun map_server map_saver -f <nombre>
```

Al ejecutarlo se genera un archivo *YAML* que contiene los datos obtenidos del tópico (figura 4.3), además de la imagen que tiene la información decodificada del *OGM*, ambos archivos tendrán el nombre que se haya seleccionado en el comando anterior. El archivo contiene el nombre de la imagen, la resolución, el origen del sistema de coordenadas, el umbral de ocupación, umbral libre, y parámetros de configuración de colores y modos como *negate* y *mode*, respectivamente. La imagen es un archivo de tipo *PGM* (*Portable Gray Map*), en escala de grises, donde por defecto los espacios blancos se consideran libres para navegar, las zonas negras son ocupadas por los objetos presentes, y las zonas grises o intermedios como zonas desconocidas; tomando estos colores dependiendo de los umbrales establecidos. Esta imagen generada se puede modificar para agregar zonas restringidas, donde no se quiere que el robot transite, pintándolas como zonas negras. Es necesario tener en cuenta que el mapa generado es estático, no cambia con el tiempo, por lo que si el ambiente cambia drásticamente lo mejor es generar nuevamente el mapa. Y además el mapa generado representa tan solo dos dimensiones, por lo que a la hora de utilizarlo hay que tener en cuenta objetos que por sus dimensiones puedan ocasionar colisiones no previstas en este tipo de mapa (The Construct, 2017).

```
image: lab_hector.pgm
resolution: 0.075000
origin: [-76.837502, -76.837502, 0.000000]
negate: 0
occupied_thresh: 0.65
free_thresh: 0.196
```

Figura 4.3. Ejemplo del contenido de archivo mapa.yaml

El mapa generado será necesario para realizar la localización del robot y calcular la ruta hacia los diferentes puntos objetivos. Por esta razón, y por el hecho de que en el presente trabajo el robot deberá navegar siempre en la misma planta del edificio de ingeniería de la Universidad de Jaén, el mapa se creará previamente. Una vez creado, es necesario publicarlo en un tópico para que este se encuentre disponible para los nodos que lo necesitan. Para esto se utiliza el siguiente comando; disponible con el paquete *map_server:*

```
rosrun map_server map_server mapa.yaml
```

Para la creación del mapa, se ocupa realizar un recorrido para obtener los datos del sensor, de odometría y de las transformadas de los *frames*, para así reconstruir el entorno. Se puede crear el mapa moviendo el robot en tiempo real por los laboratorios, o se puede hacer uso del recurso de bolsas/*bags.* Es decir se puede hacer el recorrido, mientras se graban todos los tópicos. Una vez finalizado se utiliza la bolsa grabada para reproducir los datos y construir el mapa. Se hará esto para crear el mapa con los tres paquetes mencionados anteriormente y así compararlos bajo las mismas circunstancias.

4.2.5 Comparación de los paquetes para creación del mapa.

Primeramente se realizó la comparativa con un *rosbag* obtenido de simulaciones de un laboratorio en la herramienta *Gazebo,* y seguidamente con datos obtenidos haciendo un recorrido en el laboratorio 402. Los mapas obtenidos en simulación se muestran en la figura 4.4, y los obtenidos con el recorrido realizado en el laboratorio en la figura 4.5.

Figura 4.4. Mapas simulados con a) *Gmapping* b) *Hector mapping* c) *Karto mapping.*

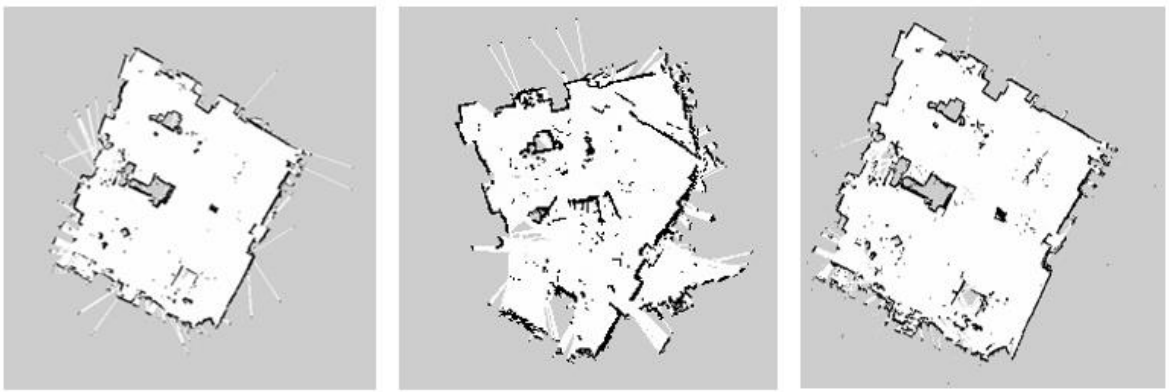


Figura 4.5. Mapas del laboratorio con a) *Gmapping* b) *Hector mapping* c) *Karto mapping.*

Como se puede observar, para ambos casos, el mapa obtenido con el algoritmo de *Hector* no es el más adecuado, se cuenta con pérdida de zonas en el caso simulado y traslape de regiones para el caso real. Para la simulación, los errores se deben a que el entorno simulado poseía muy pocos elementos (muebles, objetos, personas) por lo que se tenían menos puntos de referencia; lo que provocó perdida de información a la hora de construir el mapa, problema que no se encuentra con los otros dos algoritmos usando los mismos datos. Y en el caso real, los errores se deben a la frecuencia del láser que se tiene, que no es lo suficientemente alta para utilizar el algoritmo de este paquete. Por lo tanto, para este proyecto se descarta el uso de este nodo, y se analizaran los otros dos. En cuanto a la computación requerida por cada algoritmo, se utilizó un *script* capaz de determinar el porcentaje de *CPU* utilizando durante la creación del mapa en cada caso; en la figura 4.6 se muestran los resultados para la simulación y en la 4.7 para el caso real.

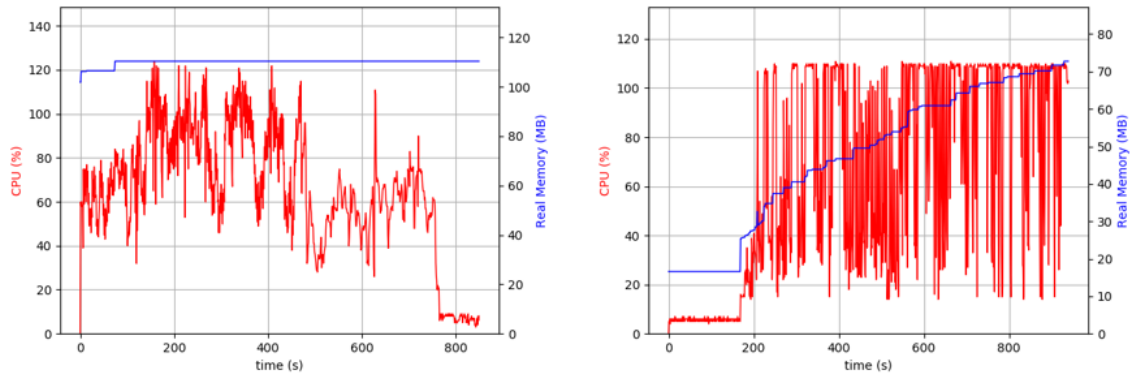


Figura 4.6. Recursos computacionales utilizados al crear el mapa simulado con a) *Gmapping* b) *Karto mapping*.

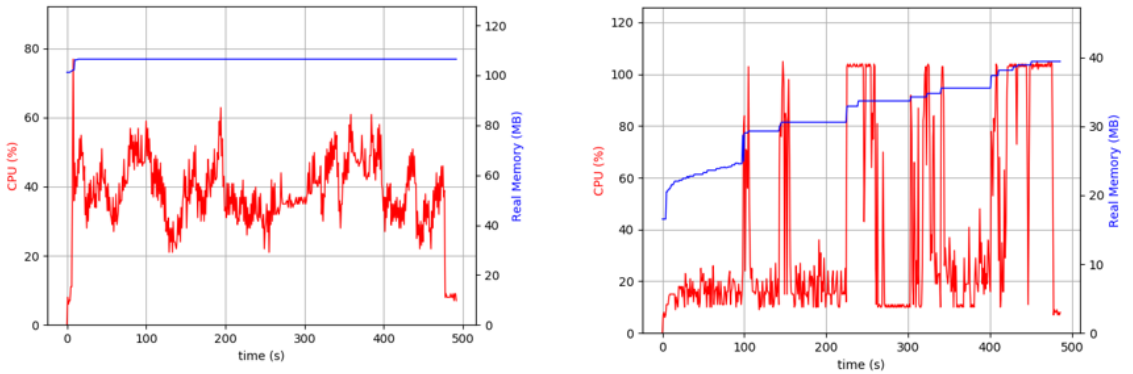


Figura 4.7. Recursos computacionales utilizados al crear el mapa con a) Gmapping b) Karto mapping.

De los gráficos anteriores se puede observar que tanto para *gmapping* como para *Karto*, hay una fluctuación del porcentaje de *CPU* utilizado. En el caso de *gmapping* hay una variación entre el 40%-120% (más de un *Core* utilizado) para el caso de la

simulación, y entre 30%-60% usando el *rosbag* creado en el laboratorio. Para el caso de *Karto,* hay una variación mayor en ambos casos entre 20%-120% y 15%-105% respectivamente, además de observarse más fluctuaciones a lo largo del tiempo. Con estos datos y observando la calidad de los mapas creados; se decide utilizar *gmapping* en el presente trabajo.

4.3 Localización

Una vez que se tenga el mapa, es necesario estimar la posición relativa del robot con respecto a este, para poder planear y ejecutar los movimientos deseados. Esta localización no solo es para la posición inicial, sino una vez que el robot empiece a moverse debe conocer su situación en todo momento, lo que se conoce como seguimiento de la posición. Cuando se habla de localización, se piensa en soluciones simples como lo son *GPS*, sin embargo para navegación de robots pequeños, su resolución en metros no es apropiada, además de que presentan incapacidad de operar con exactitud en el interior de edificios la mayor parte del tiempo. Por esto los sensores utilizados juegan un papel primordial en la localización; es necesario tener en cuenta el ruido y la resolución que presentan ya que estos factores pueden afectar esta etapa.

Cuando el robot comienza a moverse, puede llevar un seguimiento de su movimiento usando la odometría, pero esta suele no ser exacta y tener cierta incertidumbre asociada que se irá acumulando, lo que provoca que después de cierto tiempo, el robot no sepa con exactitud donde se encuentra. Por esto es necesario que se localice con relación al mapa, y sea capaz de usar los sensores para observar el entorno y compararlo con el modelo creado en el mapa. La información de la odometría en conjunto con la información de los observadores o sensores se combina para dotar al robot con la capacidad de localizarse lo mejor posible en el mapa.

Cuando se presentó la teoría del mapeo, se introdujo brevemente el problema de localización, ya que como se mencionó la creación de la representación del entorno se hace de manera simultánea con la estimación espacial del estado del robot en ese momento. Como el mapa se creó anticipadamente y estará disponible a lo largo del tiempo, es necesario utilizar algún algoritmo para determinar la localización mientras se requiera de navegación autónoma. Para resolver este problema se suele utilizar técnicas probabilísticas; ya que aportan una solución en tiempo real, han sido los más estudiados y con los que se obtienen las mejores respuestas. Entre las técnicas más robustas

destacan: *Markov,* filtro de *Kalman* y el método de *Monte Carlo*, las cuales se explican brevemente a continuación.

Para comprender el funcionamiento de estas técnicas se debe conocer que al principio se tiene una distribución con un número de partículas esparcidas en todo el espacio, al obtener más información de los sensores, se pueden ir prediciendo los nuevos estados y así actualizar la distribución con diferentes pesos de importancia. Después de múltiples iteraciones, las partículas convergen a un único valor en el espacio y la idea principal de las técnicas probabilísticas es limitar el error introducido (Robotics Knowledgebase, 2020).

4.3.1 *Markov*

Esta técnica se encarga de representar el estado de creencias usando una distribución de probabilidad arbitraria explícita sobre todas las posiciones posibles del robot. Si se conoce la posición inicial, la distribución se encuentra centrada en la posición correcta. Si no se conoce este estado inicial, la distribución se encuentra uniformemente distribuida para reflejar la incertidumbre global del robot. Para resolverlo se utiliza la regla de *Bayes* y las cadenas de *Markov,* para actualizar la creencia conforme el robot obtiene datos de los sensores o cuando se comienza a mover. Entre las suposiciones que presenta este modelo, están que los errores de odometría están distribuidos normalmente y que las lecturas pasadas de los sensores son condicionalmente independientes de las futuras lecturas, dada la posición actual.

Este algoritmo permite empezar de una posición desconocida y permite que el robot se recupere de situaciones ambiguas, sin embargo tiene mucho requerimiento computacional y de memoria, lo que provoca perdida de exactitud si las medidas obtenidas no son procesadas en tiempo real (Fox, Burgard, Dellaert, & Thrun, 2017).

4.3.2 Filtro de *Kalman*

Utiliza una representación de densidad gaussiana unimodal de la incertidumbre de la posición del robot. No considera cada posición posible, y es el resultado del axioma de *Markov* si la incertidumbre de la posición se asume como una distribución gaussiana. Sigue el posicionamiento del robot desde una posición inicial conocida. Es robusto y eficiente, sin embargo si la incertidumbre se vuelve muy grande debido a alguna colisión se pueden tener fallos y perder la localización completamente. Además, no puede lidiar

con densidades multimodales que son típicas en los problemas de localización (Siegwart, Scaramuzza, & Nourbakhsh, 2011)

4.3.3 Método *Monte Carlo*

El método *Monte Carlo* es un filtro de partículas que representa las creencias anteriores como un conjunto de partículas aleatorias con un peso asignado análogo a la probabilidad discreta. Tiene como ventajas en contraste con el filtro de *Kalman,* la capacidad de representar distribuciones multimodales y capacidad de localizar el robot globalmente sin necesidad de conocer la posición inicial. Y en comparación con *Markov*, reduce la memoria requerida y permite integrar mediciones a una frecuencia considerablemente más alta, además de ser más exacto y fácil de implementar (Dellaerty, Fox, Burgard, & Thrun).

Como ejemplo de la resolución de este algoritmo se presenta un caso en la figura 4.8 (Wang, 2017). Primeramente se inicializa la distribución de partículas uniformemente; es decir la posición del robot se considera igual de probable en cualquiera de las posiciones posibles (a). En el momento en que se obtiene información de los sensores y se detecte algún objeto, por ejemplo una puerta, se le asigna un valor de peso a cada una de las partículas, donde sea más probables a obtener esas medidas del sensor se les asigna un peso mayor; en este ejemplo las partículas que representan la posición cercanas a puertas (b). A partir de ese momento se tiene un nuevo set de partículas, donde la mayoría de ellas se generan alrededor de las partículas anteriores con mayor peso (c). Si el robot se mueve nuevamente, las partículas previas se moverán con el y a su vez el ruido será introducido al problema (d). Nuevamente se le asigna un peso a las partículas de acuerdo con la probabilidad de obtener las nuevas medidas del sensor (e) y se genera otro set de partículas alrededor de las anteriores (f). Con esto se va acotando las posibilidades, la posición puede encontrarse pero con cierta incertidumbre, puede haber dos o más probabilidades por ambigüedad y simetrías encontradas en el mapa. Conforme el robot continúe moviéndose se siguen ejecutando estos pasos, por lo que después de varias iteraciones se converge a una sola posición (g), (h) y (i).

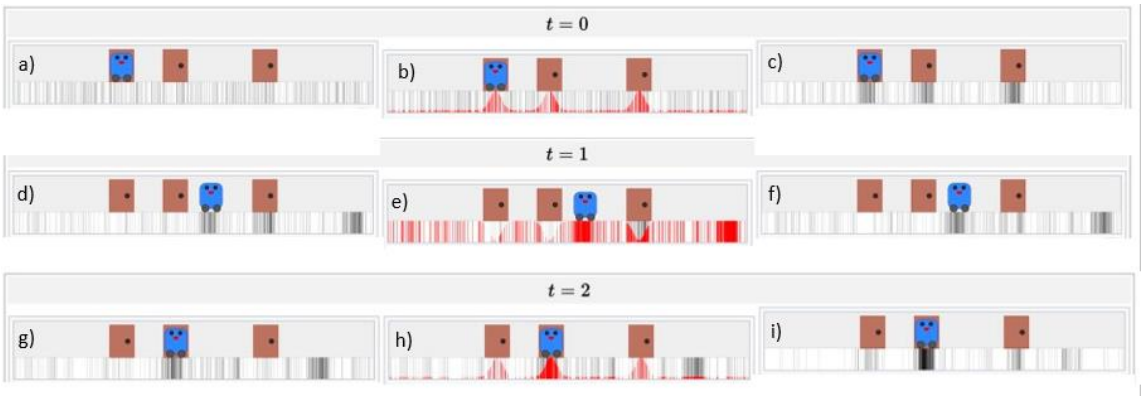


Figura 4.8. Ejemplo de localización con el método de *Monte Carlo*.

Por sus características y ventajas presentadas, en el presente trabajo se utiliza el método de *Monte Carlo* para la resolución de la localización del robot.

4.3.4 Paquete AMCL

Para implementar el algoritmo de *Monte Carlo* en ROS, existe un paquete llamado *amcl (Adaptive Monte Carlo Localization),* el cual implementa un nodo que trabaja con las medidas del sensor láser y el mapa previamente creado para que las partículas que representan las conjeturas de las posibles posiciones converjan en la posición más probable en la que se encuentre el robot, descartando partículas conforme la base se mueve y obtiene información del entorno. La palabra adaptativo en este paquete proviene del hecho que es posible configurar diferentes parámetros utilizados en el algoritmo dependiendo del tipo de robot y sensores utilizados.

Los tópicos a los que necesita suscribirse el nodo son: */scan*; para obtener los datos de los sensores, */tf;* al igual que en la creación del mapa, se necesita conocer las transformadas de las diferentes partes del robot y el entorno para poder dar significado real a los datos obtenidos, */map;* mapa creado con anterioridad e */initialpose;* valor para indicar la posición inicial del robot, si no se indica este valor; el estado inicial se considera una nube de partículas centrado sobre las coordenadas (0,0,0) definidas en el mapa.

A su vez este nodo genera diferentes datos que son publicados en los siguientes tópicos: */amcl_pose;* posición estimada del robot en el mapa con la covarianza asociada, */particlecloud;* set de posiciones estimadas en forma de nube de partículas que utiliza el filtro y */tf:* actualización de la transformada de la odometría al mapa.

Además, cuenta con diferentes servicios como: *global_localization*; servicio que permite reiniciar la localización, dispersando todas las partículas aleatoriamente en todo el espacio libre disponible en el mapa, *request_nomotion_update;* servicio para

manualmente actualizar y publicar las partículas actualizadas y el servicio *set_map;* para configurar manualmente un nuevo mapa y su posición respecto a el.

Para utilizar este paquete es necesario configurar diferentes parámetros, los cuales pueden ser establecidos con un archivo *YAML,* que es cargado cuando se ejecute el nodo *amcl*. Entre los parámetros principales se tiene; los nombres de los *frames* de la odometría, base del robot y mapa, el mínimo y máximo de partículas por utilizar en el filtro, el error máximo permitido entre las distribución real y la estimada, movimiento mínimo requerido para actualizar el filtro tanto lineal como rotacional, posición inicial, características del sensor como rango mínimo y máximo, frecuencia, tipo de modelo del robot, ruido esperado en la odometría, entre otros (ROS Wiki, 2013). La configuración utilizada se presenta en el repositorio de *git*, los parámetros que no se encuentran configurados en este archivo, utilizan el valor por defecto del nodo

Como ejemplo de como se observa el funcionamiento de este nodo, se presenta en la figura 4.9 una simulación de localización utilizando *Rviz.* En esta primeramente se observa como al inicializar el nodo las partículas se encuentran en las coordenadas cero del mapa (a). En el segundo caso se ha utilizado el servicio de *global_localization* para distribuir las partículas a lo largo de todas las posiciones posibles (b). A medida que se mueve el robot en el entorno, las partículas se van filtrando y convergiendo hasta obtener la posición en la que se encuentra (c) y (d). Finalmente se muestra un ejemplo del comportamiento de las partículas al publicar una posición en el tópico */initialpose* (e).

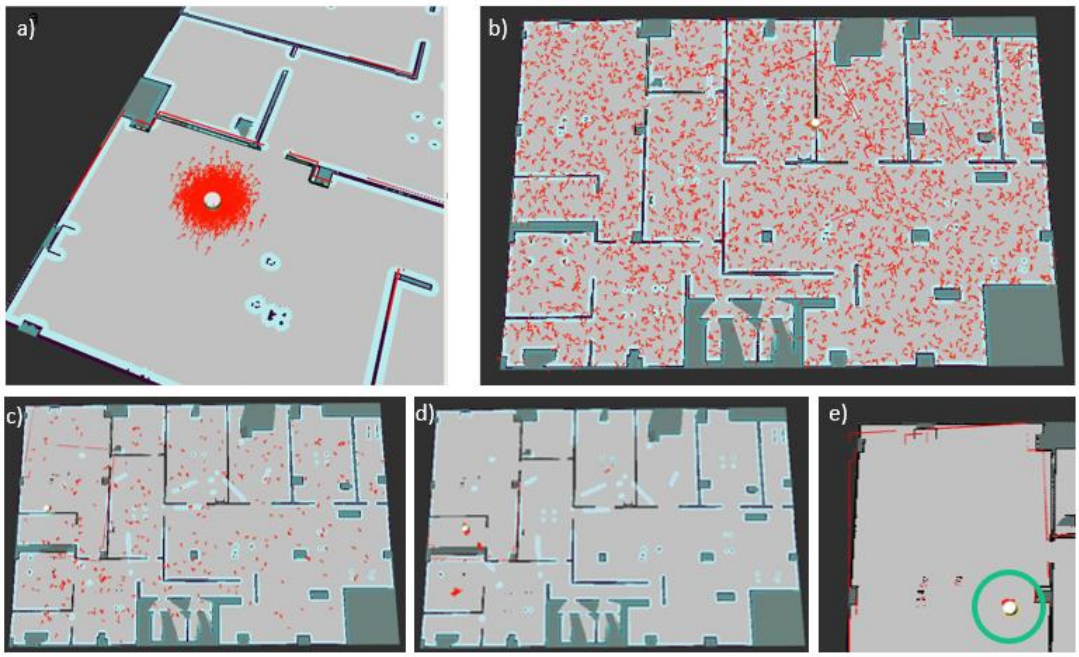


Figura 4.9. Ejemplo del nodo *amcl* en *Rviz.*

4.4 Planeamiento de trayectoria

Esta etapa se encarga de computar la ruta a seguir por el robot para lograr el objetivo, evitando colisionar con objetos, en un tiempo y distancia eficiente. Cabe destacar que esta ruta no es concretamente estática e irá cambiando dependiendo de la información que reciba de los sensores y de la localización actual.

Para poder realizar esta tarea se utiliza el paquete denominado *move_base*, el cual se encarga de unir todos los elementos involucrados en el proceso de navegación. Su función principal es mover el robot de su posición actual hacia una posición objetivo definida por el usuario o un programa por medio de una acción de ROS llamada *move_base/goal.* Este servicio consta de cinco tópicos para: publicar la posición deseada, cancelar la petición, obtener la posición actual del robot en el entorno, obtener actualización del estado del objetivo y resultado de las acciones realizadas.

Viéndolo de manera general, como una caja negra, este nodo se representa en la figura 4.10, donde la información obtenida de los nodos vistos anteriormente permite generar los comandos necesarios para mover el robot como se desea. Si se detalla más el funcionamiento de este, se observa en la figura 4.11 que los conforman diferentes etapas como lo son los planificadores (locales y globales), los comportamientos de recuperación y los mapas de costo, para generar la salida; que consiste en comandos de velocidad, con el fin de mover el robot al lugar deseado (The Construct, 2017), (ROS Wiki, 2013). Estas etapas serán detalladas a continuación.

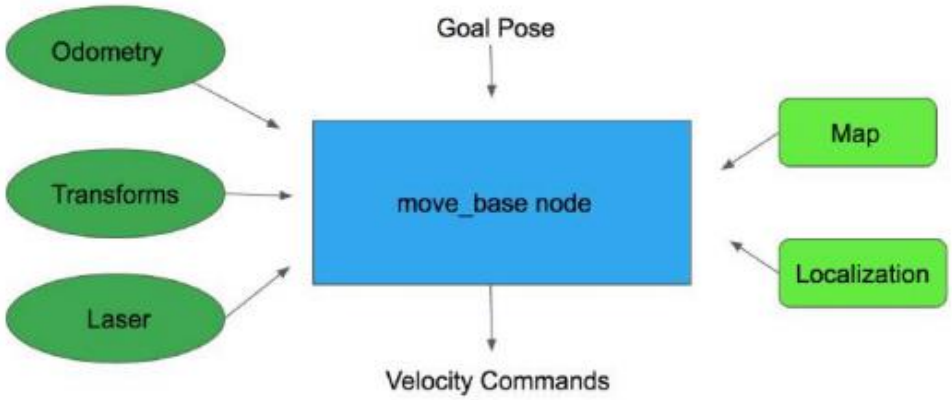


Figura 4.10. Nodo *move_base* generalizado.

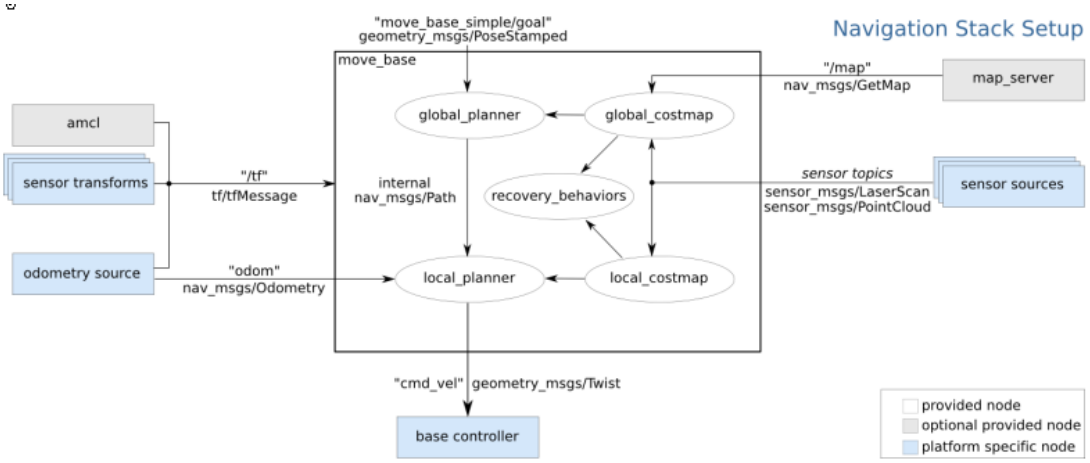


Figura 4.11. Etapas requeridas para el funcionamiento del nodo *move_base.*

4.4.1 Planificador global y local.

Los planificadores son los encargados en generar la trayectoria que debe seguir el robot y definir las velocidades necesarias para lograr esta ruta. Existen dos planificadores de rutas; global y local.

Planificador Global: Cuando el nodo recibe una nueva solicitud de posición objetivo, esta es enviada inmediatamente al planificador global. Este se encarga de calcular un camino seguro para que el robot llegue a su destino definido, es decir forma una trayectoria entre el punto inicial y el punto objetivo. Este camino se calcula antes de que el robot se empiece a mover por lo que no toma en cuenta las medidas de los sensores. La trayectoria calculada se publica en el tópico llamado */plan.* Existen diferentes algoritmos que permiten calcular el recorrido a seguir, este paquete presenta los siguientes (ROS Wiki, 2015):

- *Navfn:* Es el planificador global más utilizado en navegación con ROS. Este usa el algoritmo de *Dijkstra* para calcular el trayecto con menor coste entre el punto inicial y el punto deseado.

- Planeador *Carrot:* Este planificador toma la posición objetivo y verifica si esta es un obstáculo, si lo es; mueve el robot hasta una posición cercana, y pasa el objetivo al planeador local para que se encargue del trayecto final. Es útil cuando se requiere acercarse a una posición que realmente es inalcanzable o cercana a un objeto que se considere obstáculo, aunque realmente no se recomienda para entornos complicados.

- *Global planner:* Es un planificador similar a *navfn* pero más flexible; ya que incluye el funcionamiento del algoritmo A*. El cual requiere menos recursos computacionales ya que le da prioridad a los nodos con mejores resultados en vez de explorar todos los caminos posibles. En la figura 4.12 se observa una comparativa de estos dos planificadores.

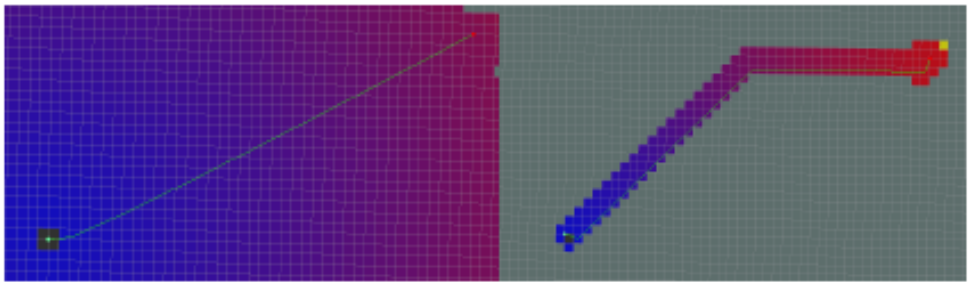


Figura 4.12. Planeador global a) *navfn (Dijkstra)* b) *global_planner (A*).*

Para el planeamiento de rutas globales se necesita el mapa de costos global que será explicado más adelante. Después de realizar varias pruebas en el laboratorio con los diferentes planificadores y sus parámetros se decidió utilizar *global_planner* en el presente trabajo. Una vez definido este método, sus parámetros pueden ser configurados en un archivo *YAML,* como lo son: tolerancia en metros de exactitud al objetivo, factor de multiplicación para el mapa de costos, orientación de la ventana, entre otros (Zheng, 2016).

Planificador local: Una vez el planificador global haya calculado el camino a seguir, este trayecto es enviado al planificador local. Este se encargara de ejecutar cada segmento del plan global. A diferencia de este, el local toma en cuenta no solo el mapa, sino los datos provenientes de la odometría y los sensores láser y computa un plan local que se va actualizado para evitar colisiones. Este planeador computa la trayectoria del robot en cada momento para evitar que pase por lugares donde hay obstáculos o personas, y así llegar a su destino. Lo obtenido en esta etapa, se publica en el tópico denominado */local_plan*. Existen diferentes algoritmos que permiten calcular la ruta local, como lo son:

- *dwa_local_planner:* provee una implementación del enfoque dinámico de ventana (*Dynamic Window Aproach; DWA*) que simula todos los posibles caminos que puede llevar a cabo el robot en las condiciones que se encuentra. Evalúa estas opciones, descartando las opciones no posibles por proximidad de obstáculos o

lejanía con el objetivo, hasta quedarse con la ruta que mayor puntaje de evaluación obtenga. Es una optimización del planificador local denominado *base_local_planner.*

- *teb_local_planner:* implementa el método de banda elástica cronometrada; a partir de la trayectoria ya calculada por el planificador global, se optimiza el camino para ejecutarla en el menor tiempo posible. En caso de encontrarse con un obstáculo, se recalcula la trayectoria pero se tiene un comportamiento de banda elástica en la que el obstáculo tira con fuerza de ella para no alejarse mucho y mantener una ruta similar a la previamente calculada. En la figura 4.13 se muestra el comportamiento de estos dos métodos (González, 2015).

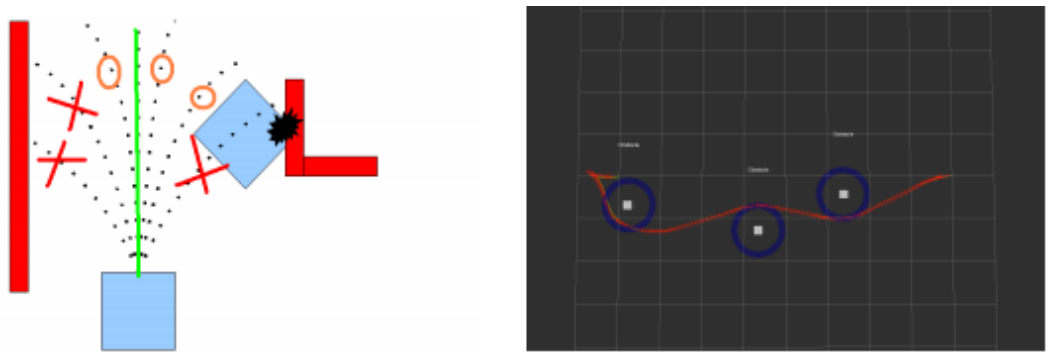


Figura 4.13. Planificadores locales a) *dwa_local_planner,* b) *teb_local_planner.*

Para el planeamiento de rutas locales se necesita el mapa de costos local que será explicado más adelante. Después de realizar varias pruebas en el laboratorio con los diferentes planificadores y sus parámetros se decidió utilizar *dwa_local_planner* en el presente trabajo. Sus parámetros pueden ser configurados en un archivo *YAML,* como lo son: los límites de aceleración del robot en las diferentes coordenadas, los valores máximos de velocidad lineal y rotacional, la tolerancia en radianes y metros para el controlador cuando se esté acercando a la posición objetivo, frecuencia del controlador, tiempo entre simulaciones, entre otros (ROS Wiki, 2013).

4.4.2 Mapa de costos global y mapa de costos local

Un mapa de costos es una representación de los lugares a los que el robot puede navegar de manera segura. Estos mapas, permiten establecer un margen de seguridad al inflar los obstáculos con un parámetro de inflación, donde después de ser escalados representan los espacios libres para navegar o los lugares donde se puede presentar una colisión. Usualmente en estos mapas, cada celda posee un valor entre 0-255,

donde 255 es el valor para las celdas que no cuentan con la suficiente información, 254 para obstáculos letales, 253 para zonas que no presentan en si un obstáculo, pero que si se mueve el centro del robot en esta posición puede resultar en choque por lo que se toma en cuenta el alrededor y 0 para representar las zonas completamente vacías. Existen dos tipos de mapas de costos (ROS Wiki, 2013):

- *Costmap* global: Creado a partir del mapa estático creado con anterioridad, por lo que toma en cuenta los obstáculos fijo como lo son paredes o muebles, y los infla; es decir les da un valor mayor al que tienen para evitar que el robot pase muy cercano a ellos. Es necesario definir ciertos parámetros para configurarlo como lo son el *frame* del mapa, la inflación del mapa, rango de los obstáculos, radio del robot, frecuencia de actualización, entre otros.

- *Costmap* local: Creado con las lecturas de los sensores. Al estar leyendo los sensores en cada instante, se detectan objetos que no se encontraban en el mapa, como personas caminando en el entorno. A diferencia del global, si el entorno cambia, el mapa de costos local también lo hará. Se pueden configurar parámetros como; el nombre de los tópicos donde se obtienen los datos de los sensores, la frecuencia de actualización y de publicación, el tamaño y la resolución del mapa.

4.4.3 Comportamientos de recuperación

En algunas ocasiones puede suceder que mientras se está ejecutando una trayectoria, el robot se atasque. Por lo tanto el paquete de navegación tiene métodos para sacar el robot de esta situación y que pueda continuar navegando, esto es lo que se conoce como comportamientos de recuperación o *recovery behaviors,* que son comportamientos programados que el robot puede realizar si se encuentra en una situación ambigua.

Para utilizar estos, es necesario usar un archivo *YAML* para configurar cuales comportamientos aplicar en caso de atascarse, entre los más comunes se encuentran: rotar 360° para tratar de encontrar un espacio libre alrededor del robot donde pueda salir para continuar navegando y reiniciar los mapas de costos; ya que en caso de haberse perdido, estos mapas pueden no coincidir y generar falsos obstáculos. En este archivo se puede configurar el orden de prioridad a seguir e intentos hasta determinar que el robot no puede recuperarse y ocupa intervención humana (The Construct, 2017), (ROS Wiki, 2013).

Una vez visto todas las etapas que componen el nodo *move_base*, se hace evidente que es necesario llamar diferentes archivos de configuración para cada etapa. Además de tener activo el servicio que nos proporcione el mapa, y el nodo que nos de la información de localización del robot. Para enviar una posición objetivo al robot, una vez se tengan todos los nodos funcionando, se usa el siguiente comando:

```
rostopic pub /move_base_simple/goal geometry_msgs/PoseStamped '{header: {stamp: now,
```

```
frame_id: "map"}, pose: {position: {x: 1.0, y: 0.0, z: 0.0}, orientation: {w: 1.0}}}'
```

A continuación en la figura 4.14 se muestra un ejemplo en *Rviz* de los diferentes elementos explicados anteriormente; trayectoria global y local, mapas de costos globales y locales, posición inicial y posición objetivo del robot.

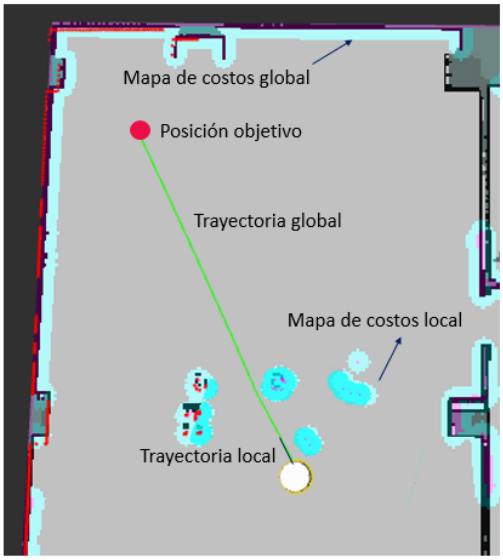


Figura 4.14. Ejemplo en *Rviz* de los componentes de navegación.

4.5 Control de movimiento

En etapa se toma la información de los bloques anteriores para así enviar las señales de velocidad y dirección a los controladores de los motores para realizar la trayectoria calculada. Algunos de los nodos ya pre configurados en el robot relacionado con el movimiento de este son:

- *joystick*, *joy_teleop* y *joystick_relay*: encargados del uso y activación del mando *joystick* para para controlar los movimientos del robot a través de este. Útil para

mover el robot en el laboratorio mientras se graba el *rosbag* necesario para generar el mapa.

- *Twist_mux*: nodo multiplexor de los comandos de velocidad (teclado, *joystick*, o provenientes de la navegación).

- *Pal_deployer_xenomai:* nodo que gestiona las acciones e información de los servomotores: recibe comandos de velocidad y actúa sobre ellos. Y además publica los datos de odometría. A este le llegan los valores de velocidad calculados por los planificadores de rutas.

4.6 Paquete de navegación: *Navigation Stack*

Para las etapas anteriores se han estado utilizando diferentes nodos de diferentes paquetes. Pero existe un meta paquete denominado *Navigation Stack* el cual contiene a los anteriores con el fin de tener en conjunto todas las herramientas necesarias para la navegación autónoma, es uno de los paquetes de navegación más completo y utilizado en ROS. Este paquete está dirigido para robots diferenciales y holonómicos, a los cuales se pueda controlar su velocidad de manera lineal y angular y que tienen un sensor láser para poder realizar las tareas de construcción del mapa y localización. Además se recomienda utilizar este paquete solo para los robots que tengan una base cuadrada o circular. La figura 4.10 muestra el panorama de todo lo que incluye el paquete; que es lo necesario para realizar la navegación autónoma (ROS Wiki, 2013). La base robótica *PMB-2* ya cuenta integrado este paquete, pero en el presente trabajo se configuraran y controlaran con otros programas creados para comprender y estudiar mejor su uso y características. Los paquetes a utilizar específicamente son; *gmapping, amcl, map_server* y *move_base* para así poder configurarlos y probar su funcionamiento de acuerdo al caso de estudio de este trabajo. Para los nodos encargados de leer los sensores y actuar sobre los motores si se utilizarán los implementados en el robot previamente de fábrica, los cuales se inicializan al encender el robot en cada ocasión.

En el presente trabajo, estos nodos y parámetros de configuración son cargados a través de un archivo tipo *launch* denominado *pmb2_mov.launch* que se encuentra bajo el paquete creado *pmb2_lab_nav.*

5. INTERACCIÓN ROBOT-HUMANO

Como el objetivo del presente trabajo es realizar recorridos guiados en la cuarta planta del edificio A3, proporcionándole información al usuario de en los laboratorios docentes, es necesario dotar al robot no solo con la capacidad de navegar de manera autónoma sino también con la capacidad de comunicarse e interactuar con los usuarios. Esta interacción con las personas se hará por medio de una cara que tendrá el robot que será capaz de moverse, escuchar y hablar dependiendo de los diferentes casos en que se encuentre la base móvil. En este capítulo se explican los algoritmos y programas implementados para otorgarle al robot estas características necesarias, bajo un paquete de ROS creado, denominado *pmb2_face*.

5.1 Hardware necesario

Para mostrarle la cara robótica al usuario, con las características mencionadas anteriormente, se utilizan los siguientes componentes (figura 5.1):

- *Raspberry* Pi 4: ordenador reducido de bajo coste con sistema operativo *Raspbian* (basado en *Linux*), con 4 GB de *RAM*, puertos *HDMI*, *USB*, *Ethernet*, *WiFi*, *Bluetooth*, y 40 *GPIO*. Este dispositivo será el encardo en computar las tareas relacionadas a la interacción con los humanos, así como de inicializar todo el proyecto, comunicándose con los nodos y tópicos del *PMB-2* usando ROS. Al ser su sistema operativo una distribución de *Linux*, fue posible instalarle la versión *Kinetic* de ROS, su conexión y configuración con la base robótica será explicado en detalle en el siguiente capítulo. Para la alimentación de este, se utilizaran los puertos disponibles en la base robótica.

- Pantalla táctil *HDMI* de 7 pulgadas: utilizada para agregar capacidad visual e interactiva la *Raspberry*; para observar la cara que simulará los movimientos de una cara al hablar y para que el usuario pueda ejecutar el proyecto al inicio por su característica táctil.

- Altavoz de 8Ω con conector *JST*: se necesita para reproducir los diálogos del robot a lo largo del recorrido.

- Tarjeta de sonido *ReSpeaker 2-Mics Pi HAT*: tarjeta de expansión que permite controlar la salida y entrada de sonido de la *rapsberry*, además de contar con dos micrófonos integrados para desarrollar aplicaciones con voz e inteligencia artificial.

Además de los micrófonos, cuenta con 3 *LEDs* de colores, un botón para usuario y dos salidas de audio (3.5mm Audio *Jack* y *JST* 2.0).

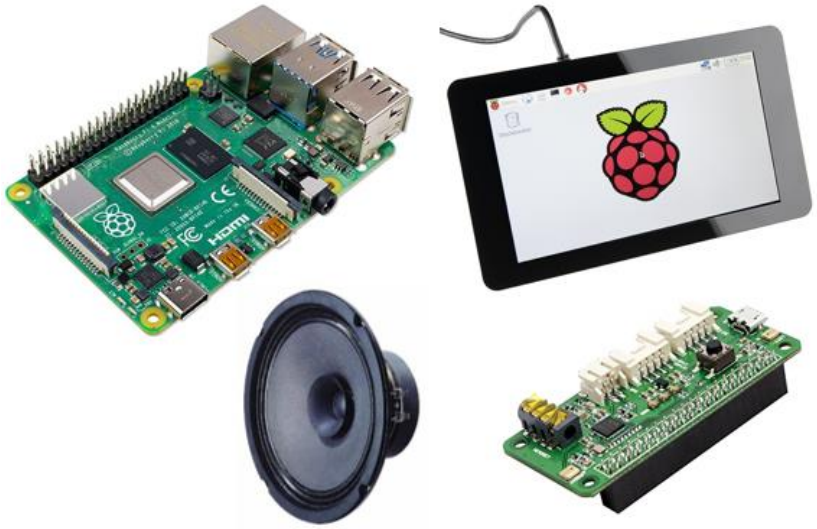


Figura 5.1. Hardware utilizado para la etapa de interacción robot-humano.

5.2 Cara del robot

Para la implementación de la cara del robot, primeramente se desarrolla en *Python* un *script* capaz de realizar tres animaciones que simulan que este se encuentra hablando, se encuentra inactivo pero parpadeando, o se encuentra sonriendo. Para esto se utilizó la biblioteca denominada *Pygame*; la cual cuenta con varios módulos que permiten crear videojuegos, programas multimedia o interfaces graficas de manera sencilla. A gran escala, el funcionamiento del programa realizado se basa en cargar dos imágenes del robot y alternarlas a un número de *frames* por segundo (*fps*) específico que provoque el efecto de movimiento. Como se requieren tres modos, se utilizan cuatro imágenes que se muestran en la figura 5.2:

- Para el estado inactivo con parpadeo; cara con la boca cerrada y ojos abiertos (a) y cara con la boca cerrada y ojos cerrados (b).
- Para el estado de habla: cara con la boca cerrada y ojos abiertos (a) y cara con la boca abierta y ojos abiertos (c).
- Para el estado sonriendo: cara con la boca abierta y ojos cerrados (d).

Además se programó un evento que recibe una entrada del usuario para elegir el modo a utilizar; indicándole la palabra *idle, talk* o *smile* según sea el caso. Cada vez que

se recibe alguna de las tres palabras, el evento se activa, cargando las imágenes correspondientes y reflejándolas en pantalla como animación.

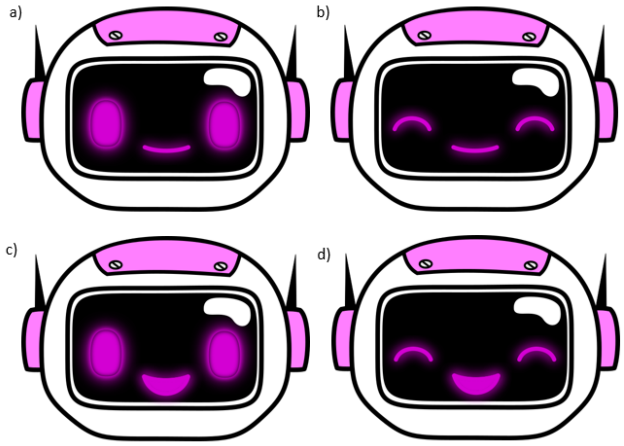


Figura 5.2. Cara del robot con diferentes características.

Seguidamente este programa se incluye en un nodo ROS denominado *face_node* para poder ser integrado con otros nodos del programa global. Al inicializar el nodo, se mostrará la cara del robot en modo inactivo. Este nodo se encuentra suscrito al tópico denominado */face_action* del tipo *string.* Cada vez que se publique algo a través de otros nodos en este tópico, el programa identificará si es alguna de las tres palabras que activan los modos de la cara. Si coinciden se ejecuta el evento respectivo dando como resultado la animación correspondiente (figura 5.3). De esta manera a través de otros nodos se determinará cuando el robot está en silencio o cuando está hablando y se sincronizará con lo observado en pantalla.

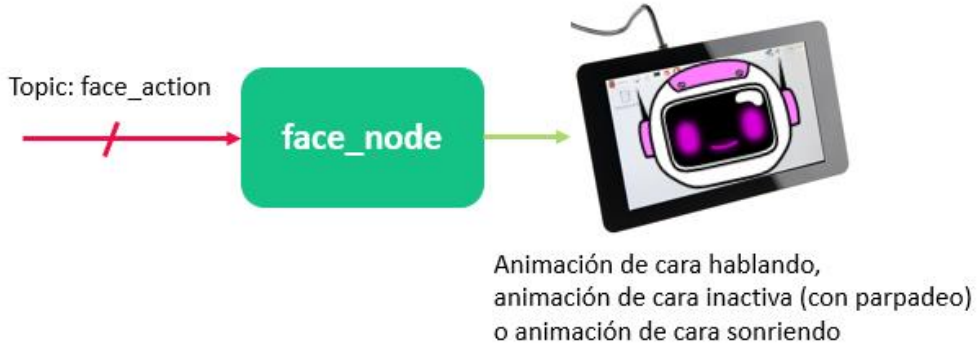


Figura 5.3. Nodo *face_node.*

5.3 Capacidad de habla del robot

Como se ha mencionado previamente, uno de los principales objetivos del trabajo es darle al robot la capacidad de reproducir diálogos para poder brindarle información al usuario durante el recorrido de los laboratorios; es decir la capacidad de "habla". Estos diálogos pueden estar previamente grabados; ya que lo que se mencionará al usuario en cada sitio siempre será lo mismo, pero también es interesante darle la habilidad al robot de mantener una conversación simple con las personas. Para esto último son necesarios varios módulos que se irán explicando más adelante, pero esta sección, se centras en la capacidad de reproducir diferentes palabras que sean ingresadas a través de texto.

Para lograr que el robot hable; ya sea con audios previamente grabados, o que convierta texto a voz, se crearon dos nodos en el paquete *pmb2_face,* y estos se explican a continuación:

5.3.1 Nodo de reproducción de audios

La reproducción de audios pregrabados, se logra creando un nodo denominado *talk_node.* Este nodo funciona como servidor del servicio creado *talk_srv,* es decir ejecuta una función (en este caso la reproducción del audio), y envía una respuesta al cliente que solicitó el servicio al finalizar la acción.

La solicitud del cliente tiene dos mensajes; *talk_req* y *pose_req,* de formato *string* y entero respectivamente. Estos mensajes son para indicar la acción que se desea y el número de identificación del audio a reproducir. Por ejemplo, si el cliente envía una solicitud con los mensajes; "*talk, 5*", el nodo se encargará de reproducir el audio almacenado bajo el nombre 5.ogg y una vez finalizado responder al cliente con un mensaje tipo entero (*talk_resp*) que representa el código de retorno dependiendo del resultado (0 para finalización exitosa, 1 si ocurre un error reproduciendo el audio o 2 si los datos enviados por el cliente no son válidos (otra acción que no sea *talk* o un *id* no válido)).

La función que se ejecuta cuando se llama a este servicio, se encarga de abrir el audio especificado en formato *ogg* (formato de audio comprimido), y de reproducirlo utilizando nuevamente módulos de *Pygame.* Además una vez que se empieza a reproducir dicho audio, se publica en el tópico */face_action* explicado anteriormente para indicarle a la cara que debe moverse simulando que se encuentra hablando. Una vez se termine de reproducir el audio, se publica nuevamente en este tópico para que la

cara se detenga, y se da respuesta al cliente a través del servicio. La figura 5.4 muestra el diagrama de este nodo.

Figura 5.4. Nodo *talk_node.*

5.3.2 Nodo de transformación de texto a voz

Para que el robot pueda comunicarse con los usuarios con conversaciones simples, es necesario que el robot pueda reproducir oraciones que no necesariamente tienen que estar pregrabadas sino que provengan de otro nodo (el nodo de lenguaje *AIML; sección 5.5*). Es decir debe ser capaz de tomar texto y transformarlo en un archivo de audio para ser reproducido.

La conversión de texto a voz se realiza utilizando el servicio en la nube *Amazon Polly* de *AWS,* el cual permite crear diferentes tipos de aplicaciones con mayor impacto en los usuarios al tener la capacidad de habla artificial. Este servicio soporta varios lenguajes y variedades de voces para adaptarse a los diferentes lugares donde se puede utilizar. Entre los principales beneficios son la calidad del sintetizado de habla con gran precisión de pronunciación, la respuesta rápida que permite utilizarlo en sistemas de baja latencia, gran cantidad de lenguajes y voces disponibles tanto en femenino como masculino, el coste, y que al ser una solución basada en la nube, las aplicaciones requieren menos recursos computacionales. El coste de este servicio es de $4 por cada millón de caracteres cuando se excede la capa gratuita, la cual consta de 5 millones de caracteres por mes durante un año (1 millón de caracteres es equivalente a 24 horas de audio continuo) (AWS, 2016). Se realizaron pruebas con un par de sintetizadores gratuitos, sin embargo la respuesta obtenida no era muy buena en cuanto a tiempo de respuesta, exactitud en la pronunciación y entonación de las palabras; lo que provocaba sonidos molestos que dificultarían la interacción con las personas.

Amazon Polly en *Python* requiere primeramente configurar los credenciales de la cuenta de *AWS* previamente creada, al sistema a utilizar; en este caso la *Raspberry*. Seguidamente se debe instalar el *SDK* (kit de desarrollo de *software*) de *AWS* para *Python*, conocido como *boto3*, el cual facilita la integración de los servicios en la

aplicación a crear. Con esta biblioteca ya se puede crear un objeto del tipo *Polly,* al cual se le indica el texto que se desea convertir a voz, y otros parámetros como el idioma, el formato, la voz y el directorio donde se quiera almacenar el audio generado.

Para que el robot pueda tener esta capacidad, se crea un nodo denominado *text2speech_node.* Este nodo se suscribe al tópico denominado */tts;* tópico donde se publicará el texto que se desea convertir a voz por otros nodos. Cada vez que se publique algo en este tópico, se ejecuta la función que se encarga de realizar la petición de conversión en la nube y de reproducir el audio obtenido por medio de *Pygame,* similar a como se realiza en el nodo anterior. Cuando se empiece a reproducir el audio, se publica en el tópico */face_action* el comando necesario para que la cara se empiece a mover, y así sincronizar la cara con la voz y que se tenga la sensación que el robot está hablando. Y finalmente se publica nuevamente en este tópico cuando el audio termine de reproducirse para detener el movimiento de la cara. La figura 5.5 muestra el diagrama de este nodo.

Figura 5.5. Nodo *text2speech_node.*

5.4 Capacidad de escucha del robot

Ya se ha explicado cómo darle el robot la capacidad de hablar o reproducir diálogos, pero para lograr una conversación simple con las personas debe ser capaz de escuchar lo que estas digan para así responderles lo que corresponda. Por lo tanto es necesario un módulo inverso al anterior, en este caso que convierta la voz a texto para poder ser procesado por otros módulos o nodos.

Primeramente es necesario configurar el micrófono a utilizar, instalando el software necesario proveniente del distribuidor de la tarjeta de sonido *ReSpeaker 2-Mics Pi HAT*. Una vez instalado, se puede verificar que este sea reconocido por la *Raspberry* con el comando "*arecord –l*", el cual listará los dispositivos de entrada disponibles. El micrófono de la tarjeta deberá aparecer bajo el nombre "*seeed2micvoicec [seeed-2mic-voicecard]"* y con un número de dispositivo definido. Una vez configurado el micrófono, podrá ser usado en *Python* por medio de la biblioteca *Pyaudio* para grabar archivos de sonido.

Con el fin de lograr el reconocimiento de voz se utiliza el servicio en la nube ofrecido por *Google*; *Speech Recognition,* el cual por medio de algoritmos de redes neuronales de aprendizaje profundo es capaz de reconocer la voz en más de 125 idiomas y variedades lingüísticas. Este servicio no presenta coste para audios de menos de 60 minutos por mes, en caso de necesitar más intervalo de tiempo por procesar, se cobra $0.006 por cada 15 segundos (Google Cloud, 2007). A pesar de que este servicio y el proporcionado por *AWS* sean de pago, se consideran buenas opciones para esta aplicación, ya que las conversaciones entre el robot y el humano no serán muy extensas (se utilizará más que todo para saludar al usuario, conocer su nombre y darle una bienvenida más personalizada) por lo que no superaran los valores gratuitos permitidos y son de las que mejor respuesta tienen en el mercado.

Para utilizar *Speech Recogniton* en *Python* se utiliza la biblioteca del mismo nombre previamente instalada. Con esta es posible crear un objeto al cual se le indica el audio que se desea reconocer, el idioma a identificar y un parámetro para configurar la reducción de ruido del ambiente. Y como respuesta se tiene una variable del tipo *string* con los caracteres identificados en el audio.

La implementación de esta sección en el robot, consiste en la creación de un nodo denominado *speech2text_node.* Este nodo funciona como servidor del servicio creado *listen_srv*, es decir cuando llega una solicitud, ejecuta la función de activar los micrófonos, escuchar durante cinco segundo y transformar el audio de lo que dice el usuario utilizando el objeto encargado de obtener el texto, y envía una respuesta al cliente que solicito el servicio al finalizar la acción. La solicitud del cliente tiene un mensaje de tipo *string; listen_req,* el cual es comparado y si coincide con la palabra *conversation,* lo obtenido en el nodo es publicado en el tópico */stt* para que sea procesado por el nodo de *AIML* (sección 5.5). Si es otra palabra la publicada en la solicitud, el resultado obtenido solo será devuelto a través de la respuesta (tipo *string*) para ser utilizado por otros nodos en el robot (por ejemplo, el nodo de control). La figura 5.6 muestra el diagrama de este nodo.

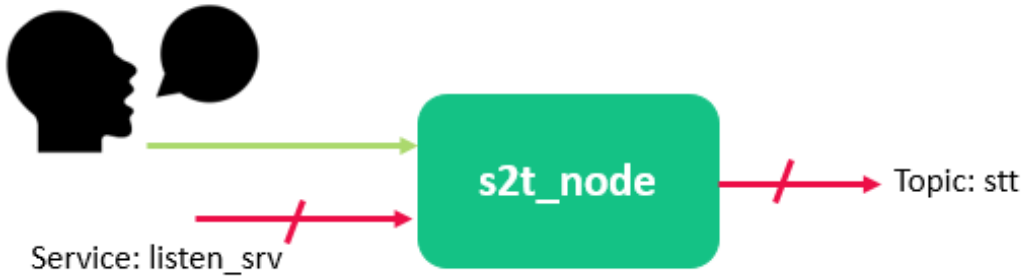


Figura 5.6. Nodo *speech2text_node.*

5.5 Lenguaje *AIML*

Para que el robot pueda entablar una conversación con las personas es necesario que este escuche lo que el usuario le dice, lo analice con algún módulo de inteligencia artificial, y así le responda algo relacionado. En las secciones anteriores se explicó como el robot será capaz de escuchar a la persona y convertir lo que dice a texto, y por otro lado recibir un texto y convertirlo en voz para comunicar un mensaje. Por lo que en esta sección se explicará el módulo intermedio entre estas dos, que es capaz de recibir un texto; interpretarlo y tener como salida otro texto de respuesta. Este módulo intermedio utiliza el lenguaje marcado basado en *XML; AIML* (*Artificial Intelligence Mark-up Language*).

AIML fue desarrollado por *Richard Wallace* y la comunidad de código abierto entre 1995 y 2002 para el desarrollo del robot *ALICE* (*Artificial Linguistic Internet Computer Entity*). Este lenguaje es utilizado para crear robots tipo *chatbots* o *bot* conversacional; es decir una aplicación *software* capaz de mantener una conversación al tener respuestas automáticas dependiendo de la entrada del usuario. Estas respuestas son previamente establecidas por medio de un archivo *XML* de clasificación (Lentin, 2017). Por lo tanto, para crear una aplicación con estas habilidades, se deben crear los archivos *AIML* y se debe integrar un interpretador para cargarlos, y buscar lo dicho por el usuario para proporcionarle la respuesta más adecuada.

El vocabulario admitido en *AIML* consiste en palabras, espacios y caracteres especiales como asteriscos y guiones. Y se compone de diferentes etapas definidas por:

- <*aiml*> Etiqueta para iniciar un documento *AIML*.
- <*category*> Etiqueta para iniciar una unidad de conocimiento.
- <*pattern*> Etiqueta que define el patrón de entrada del usuario.
- <*template*> Etiqueta que defina la respuesta, en caso de presentarse el patrón definido.

Un ejemplo del uso de estos archivos se muestra en la figura 5.7, donde se define como patrón la frase “*Hello Alice*” y como respuesta a esta “*Hello User*”.

```
<aiml version = "1.0.1" encoding = "UTF-8"?>
   <category>
      <pattern> HELLO ALICE </pattern>

      <template>
         Hello User!
      </template>

   </category>
</aiml>
```

Resultado:

```
User: Hello Alice
Bot: Hello User
```

Figura 5.7. Ejemplo de archivo *AIML* y su respuesta ante entrada.

Existen otros comandos que permiten ampliar la versatilidad de las respuestas para que se muestre de manera más natural; como lo son (AIML Foundation, 2014):

- <*star*>: Permite tomar una palabra del usuario para utilizarla en la respuesta. Por ejemplo si de patrón se tiene “Mi nombre es *” y de respuesta “Hola <stair>”, el robot será capaz de responderle con el nombre o la palabra que se disponga después de la frase “Mi nombre” (Usuario: Mi nombre es Francini, Robot: Hola Francini).

- <*srai*>: Permite definir diferentes objetivos para las mismas respuestas, permite reducir patrones gramaticalmente complejos, resolución de sinónimos o detección de palabras clave.

- <*random*>: En el archivo se pueden definir diferentes respuestas para un mismo patrón, y utilizando este comando se indica que de manera aleatoria se responda con alguna de estas respuestas definidas, lo que permite que para la misma frase se obtengan respuestas diferentes en cada ocasión.

- <*set*> y <*get*>: Estos comandos se usan para almacenar y utilizar variables en el archivo.

- <*that*>: Utilizada para agrupar unidades de conocimiento y así responder de acuerdo al contexto.

- <*topic*>: Sirve para almacenar el contexto de la conversación anterior, se utiliza comúnmente después de las preguntas de tipo si/no.

- <*condition*>: Es un comando condicional que sirve para responder una cosa u otra dependiendo de la condición que se tenga.

Con la combinación de todos estos comandos y otros, se pueden crear archivos con información variada capaz de responder en infinidad de temas. Inclusive se le puede otorgar cierta personalidad al robot, al contestar de cierta manera a las preguntas o comentarios; por ejemplo cómico, sarcástico o emocional. Cabe destacar que el interpretador es capaz de cargar más de un archivo, por lo que en la web se encuentran estos clasificados por temas (siempre personalizables) para ser utilizados dependiendo de la aplicación. Si se desean añadir más archivos de este tipo, se agregan en el directorio *aiml* bajo el paquete *pmb2_face.*

Para realizar este módulo en ROS, se crea un nodo denominado *aiml_node.* Este nodo se suscribe al tópico /stt; es decir lo que nos proporciona el bloque encargado de transformar lo que diga el usuario en texto. Cada vez que se publique en este nodo, se ejecuta la función que se encarga de ejecutar el interpretador de archivos *AIML* para *Python*. El interpretador tendrá como entrada el *string* proveniente del tópico /stt y nos devolverá otro *string* con la respuesta según lo que se haya definido en los archivos. Esta respuesta será publicada en el tópico /tts para que así el nodo que transforma el texto en voz; reproduzca la respuesta. La figura 5.8 muestra el diagrama de este nodo:

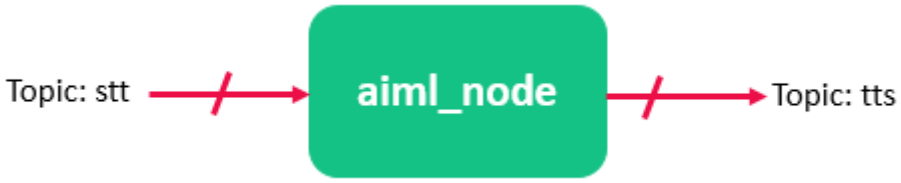


Figura 5.8 Nodo *aiml_node.*

Es necesario tener en cuenta que a la hora de dar respuestas se tiene que elaborar frases más completas para evitar ambigüedad, por ejemplo; si el robot pregunta el nombre al usuario, sería ideal responder con una frase tipo: *"Mi nombre es X"* o "*Me llamo X*" en vez de responder solo con el nombre, esto con el fin de darle más contexto al robot y que se encuentre la respuesta más adecuada.

5.6 Conexión de los módulos de interacción creados

Todos los nodos explicados anteriormente, se conectan entre sí, ya que están suscritos y publican en los mismos tópicos según correspondan. En la figura 5.9 se observa la conexión de funcionamiento que se tiene, la cual se resume en que hay dos maneras de obtener un audio con la voz del robot; primeramente cuando se active el nodo *speech2text_node* mediante el tópico */conv_req* (se activa el micrófono y se

convierte lo dicho por el usuario en texto, que será la entrada para el nodo *aiml_node,* este dará una respuesta de acuerdo a los archivos configurados, y será reproducida por el nodo *text2speech* además de mover la cara del robot para simular el habla), o cuando se active el servicio */talk_srv* para que el nodo *talk_node* reproduzca un audio previamente grabado y también mueva la cara del robot mientras se reproduce.

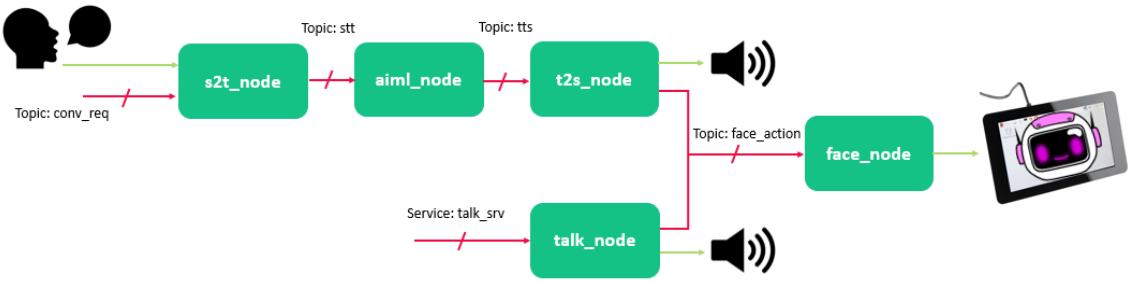


Figura 5.9. Conexión de los nodos de interacción robot-humano.

En el presente trabajo se utilizarán los audios pregrabados para darle información al usuario de las diferentes zonas en las que se encuentra durante el recorrido, y el módulo de conversación al inicio para darle una bienvenida personalizada. Todos estos nodos y parámetros de configuración son cargados a través de un archivo tipo *launch* denominado *pmb2_talk.launch* que se encuentra bajo el paquete creado *pmb2_face.*

6. CONFIGURACIÓN UTILIZADA

El siguiente capítulo describe las configuraciones necesarias en el robot y la *Raspberry* para la puesta en marcha del proyecto, la comunicación entre dispositivos, así como los principales programas desarrollados para cumplir con los objetivos.

6.1 Configuración del Robot *PMB-2*

El ordenador del robot a utilizar cuenta con tres usuarios diferentes para realizar diferentes tareas; estos son:

- Usuario: *root*. Contraseña: *palroot*; este será el usuario principal a utilizar.
- Usuario: *pal*. Contraseña: *pal*.
- Usuario: *aptuser*. Contraseña: *palaptuser*.

Cuando se enciende el robot, y después de iniciar sesión con alguno de los usuarios mencionados, el robot creará una red *LAN* llamada *rlan* con contraseña *palrobotics.* Esta red podrá ser configurada accediendo al puerto 8080 del ordenador (http://pmb2-5c:8080), donde se obtendrá la visualización de una página web que muestra una herramienta denominada *WebCommander.* En esta aplicación se puede observar el estado de los nodos, *logs*, la configuración de las conexiones y el software instalado.

Una vez encendida, la base inicializa ROS (*roscore*), y carga todos los paquetes instalados para que se encuentren listos para utilizar. Además de esto, lanza los nodos de navegación previamente instalados por fábrica. Sin embargo, como se desea iniciar los paquetes de navegación con configuraciones propias, se debe terminar el proceso de estos nodos antes de iniciar el programa creado en este proyecto. Para finalizar estos nodos se utiliza un *script Shell* creado (*initNodes.sh*) para terminar cada uno de los nodos que no son necesarios con el comando *rosnode kill*. Igualmente hay otros nodos iniciales que son necesarios mantener para asegurarse el buen funcionamiento del robot, estos son:

- *default_controllers_spawner:* encargado de cargar e inicializar los controladores
- *hokuyo:* gestiona los datos del sensor LIDAR.
- *pal_deployer_xenomi:* gestiona las acciones a realizar por los servomotores.

- *mobile_base_controller:* nodo que gestiona los comandos de velocidad.
- *robot_state_publisher:* nodo general de ROS, utilizado para actualizar en todo momento las transformadas entre las diferentes instancias del robot (*/tf)*.
- *rosout:* nodo general de ROS, usado para suscribir, registrar o re publicar mensajes.
- *tf_lookup:* nodo que permite hacer seguimiento de las coordenadas de los diferentes *frames* a lo largo del tiempo.
- *twist_mux:* multiplexor para manejar los comandos de velocidad desde las posibles entradas; mando *joystick*¸ teclado por medio del paquete (*key_teleop*) o desde los algoritmos de navegación.
- *pal_web_commander:* nodo encargado de publicar en la página web la información y estado actual del robot para visualización del usuario.
- joystick: utilizado para la activación y funcionamiento del mando para mover el robot.

En la figura 6.1, se muestra cómo estos nodos se conectan entre sí a través de los diferentes tópicos en los que se suscribe y publica la información necesaria para asegurar el correcto funcionamiento del robot.

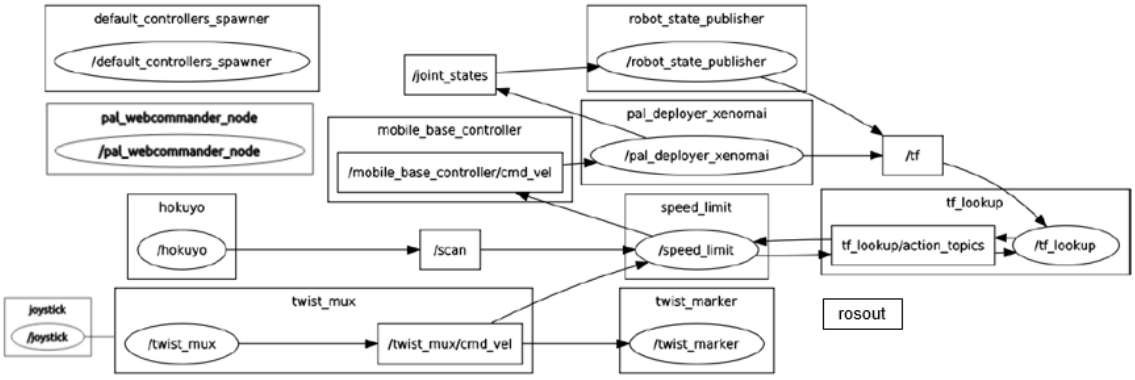


Figura 6.1. Diagrama de conexión de nodos iniciales del robot PMB-2.

6.2 Comunicación *PMB-2*/ *Raspberry* Pi / Ordenador para pruebas

Para utilizar el ordenador del robot, es necesario conectar los periféricos como una pantalla HDMI y un teclado para poder tener acceso a su consola y poder ejecutar los comandos necesarios. Sin embargo para poder realizar las pruebas, se facilita su

manipulación si se trabaja desde un ordenador externo conectado a la red del robot, además de que así se puede utilizar la herramienta *Rviz,* que no era posible en el ordenador del robot por incompatibilidades gráficas. Además también es necesario realizar una conexión entre el robot y la *Raspberry*, ya que el proyecto se ejecuta en ambas plataformas para realizar la navegación y la interacción con los usuarios respectivamente.

Para poder utilizar ROS en los tres ordenadores (Robot, *Raspberry* y ordenador personal), es necesario realizar una conexión completa y bi-direccional entre los pares de máquinas. Cada ordenador debe tener su nombre e *IP* única. El robot será nombrado como *pmb2-5c* y al ser el dispositivo central y el que lanza la red para las conexiones posee la *IP* fija: *10.68.0.1* La *Raspberry* se conectará al robot por medio de un cable *Ethernet* en conjunto con los cables de alimentación, ya que así se asegura una comunicación más directa y además que se necesita la red *WiFi* en este ordenador para las funciones de interacción (por los servicios en la nube de reconocimiento y conversión de voz). Y el ordenador externo se conectará inalámbricamente en la red creada por el robot. A la *Raspberry* y al ordenador, se les definió un nombre y se les asignó una *IP* estática en el mismo rango de la del robot, por lo que se tiene lo siguiente:

- *IP* robot *PMB-2*: 10.68.0.1 Nombre: *pmb2-5c*
- *IP Raspberry*: 10.68.0.128 Nombre: *raspi*
- *IP* ordenador: 10.68.0.129 Nombre: *Fran-5574*

Para añadir estos nombres e *IPs* en los tres ordenadores con el fin de ser identificados entre ellos se modifica en cada uno con permisos de administrador, el archivo */etc/hosts*, añadiendo las líneas con la *IP* y el nombre separados por espacio. Al estar los tres ordenadores conectados en la misma red, se pueden hacer pruebas de conexión entre ellos usando el comando *ping* (verifica el envío de paquetes *ICMP*) o *netcat* (verifica que las máquinas se puedan comunicar a través de todos los puertos).

Una vez verificada la correcta conexión entre los dispositivos, se puede acceder al ordenador del robot por medio del ordenador externo para facilitar su uso. Esto se realiza por medio de la herramienta *ssh* para acceso remoto, que permite acceder al ordenador indicado por nombre o *IP*, y al usuario deseado, en este caso *root*:

```
ssh root@pmb2-5c
```

Después de ejecutar este comando, será necesario introducir la contraseña para hacer uso de la consola. Igualmente para transferir archivos entre máquinas se utiliza esta comunicación *ssh* utilizando el comando *scp* de la siguiente manera:

```
scp <ruta del archivo a copiar> <usuario>@<nombre/IP>:<ruta destino>
Ejemplo:
scp /Docs/test.py root@pmb2-5c:/Pruebas/
```

Para mejorar la seguridad y más adelante ejecutar nodos entre las computadoras, se recomienda generar los certificados *ssh* y copiar las llaves locales en el directorio *~/.ssh* del robot para facilitar la autentificación y conexión.

El robot cuenta con una serie de particiones diferentes, entre ellas:

- */:* montado en el directorio */ro* con la característica de solo lectura.
- */home, /var/logs* y */mnt_flash:* particiones con permisos de lectura y escritura.
- */opt:* partición con solo permisos de lectura.

Es necesario mencionar que para realizar cambios en el robot, que se mantengan entre cada encendido y apagado, hay que montar el directorio con permisos de lectura y escritura, por lo que se deben ejecutar una serie de comandos explicados en el manual de usuario del *PMB-2* antes de realizar los cambios que se desean guardar, y al finalizar, configurarlo nuevamente como directorio de lectura.

```
rw
chroot /ro

<realizar cambios necesarios>

exit
ro
```

En el robot se creó un *workspace* para almacenar los paquetes utilizados en el proyecto en la ubicación: */home/pal/pmb2_UJA/*. Si se desea instalar un paquete en el robot, ya sea de los disponibles en la web (se descarga los archivos de *git*) o uno propio, se transfieren los archivos y carpetas correspondientes en este espacio y se ejecuta el comando *catkin_make.*

Con el objetivo de conectar los dispositivos no solo en la misma red (como se ha explicado anteriormente), sino también a través de ROS es necesario definir cuál será el centro computacional central y conectar el resto de dispositivos a este para poder acceder a la información proporcionada por los nodos. En el presente trabajo, el robot

será la unidad central que se encarga de ejecutar el entorno de ROS a través del *roscore.* Por lo que es necesario modificar el archivo *.bashrc* en la *Raspberry* y ordenador externo, para indicarles que el *roscore* será ejecutado en el robot. Los comandos que son necesarios agregar son:

```
export ROS_MASTER_URI=http://10.68.0.1:11311
export ROS_IP=10.68.0.128
```

Una vez realizado esto, ya se encuentran los dispositivos conectados bajo el mismo ROS, por lo que se puede observar los nodos iniciales lanzados en el robot, en la *Rapsberry* o el ordenador al ejecutar el comando *"rosnode list".* Además se puede publicar, suscribir, crear propios nodos o paquetes, inclusive mover el robot publicando en el nodo respectivo.

6.3 Nodo controlador

Hasta el momento se ha hablado de dos grandes etapas del proyecto; la navegación y la interacción robot-humano, donde los nodos explicados están programados para recibir órdenes sencillas a través de los tópicos o servicios. Es necesario implementar un bloque capaz de controlar los diferentes eventos que se requieran y unir ambas etapas, englobando estos nodos en un proceso más complejo con un fin determinado. Para definir todos los posibles estados y las acciones por realizar en cada instante, y trabajar de manera modular, se crea una máquina de estados finitos, con el uso de la herramienta *SMACH* (*State MACHine*) de *Python*. Esta máquina de estados se inicializa una vez se tengan todos los nodos ejecutándose, tanto los de navegación como los de interacción (a través de sus respectivos archivos *launch*) para gestionar el programa.

SMACH una biblioteca de *Python* que puede ser utilizada en ROS a través del paquete que posee el mismo nombre, que permite crear máquinas de estado jerárquicas para crear comportamientos robustos, modulares y fáciles de mantener para el robot. Es un contenedor de estados; se engloban tareas o inclusive otras máquinas de estado para la realización de las diferentes acciones. Además esta biblioteca cuenta con herramienta de programación y depuración en ROS (ROS Wiki, 2013).

Para la ejecución de la máquina de estados, se crea un nodo denominado *controlSM_node.* Este nodo utilizando el formato definido en el paquete de ROS crea los diferentes estados para controlar el programa y que este siga los pasos mostrados a gran escala el diagrama de flujo de la figura 6.2; primeramente se da una inicialización

del programa; donde se verifica que los nodos necesarios se encuentren activos y se le indica al robot que haga una serie de movimientos para localizarse mejor (rutina inicial), seguidamente el robot le dará una bienvenida personalizada al usuario (preguntándole su nombre), cuando el usuario indique al robot que está listo, se iniciará una segunda máquina de estados para controlar el movimiento y finalmente cuando se haya concluido con todos los puntos definidos, se finaliza el programa.

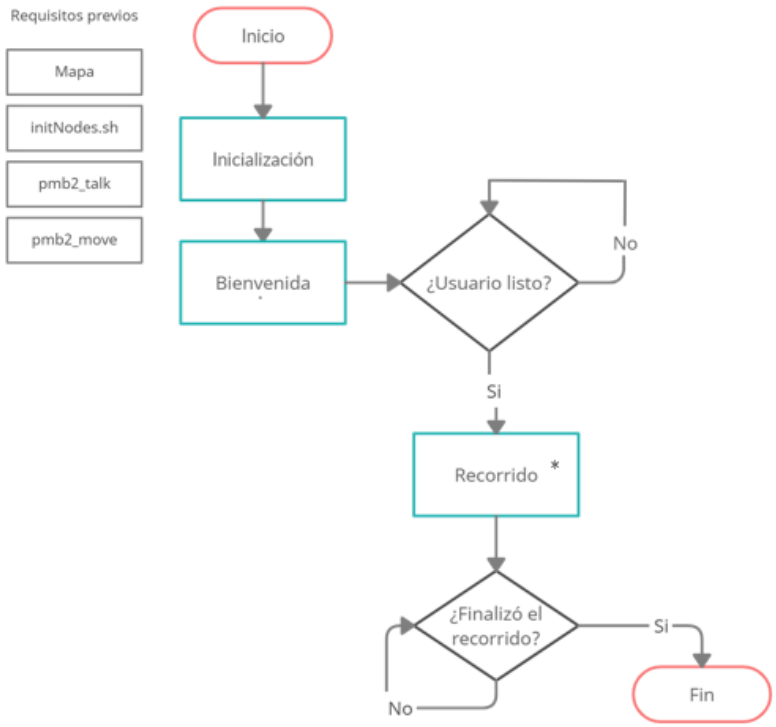


Figura 6.2. Diagrama de flujo principal.

En la figura 6.3 se muestra el diagrama de flujo que describe la máquina de estados encargada del recorrido de *n* cantidad de puntos definidos. Inicialmente se procederá a publicarse la primera meta por medio de un nodo creado (*movePoint_node*: encargado de leer el archivo de configuración que define los puntos relevantes y enviarle la posición objetivo al nodo *move_base*), una vez que el robot llegue a esta posición, se comenzará a reproducir el audio respectivo. Cuando este audio finalice, si no es el último punto del recorrido se le pregunta al usuario el siguiente punto (*p*), y se repite el proceso hasta haber recorrido todos (cuando queda solo un punto por recorrer, no se le pregunta al usuario); en este caso, el robot se despide del usuario y finaliza el programa. Como requisitos previos, se necesita haber creado el mapa, detener los nodos no necesarios que ejecuta el robot, y lanzar los nodos implementados de navegación y comunicación.

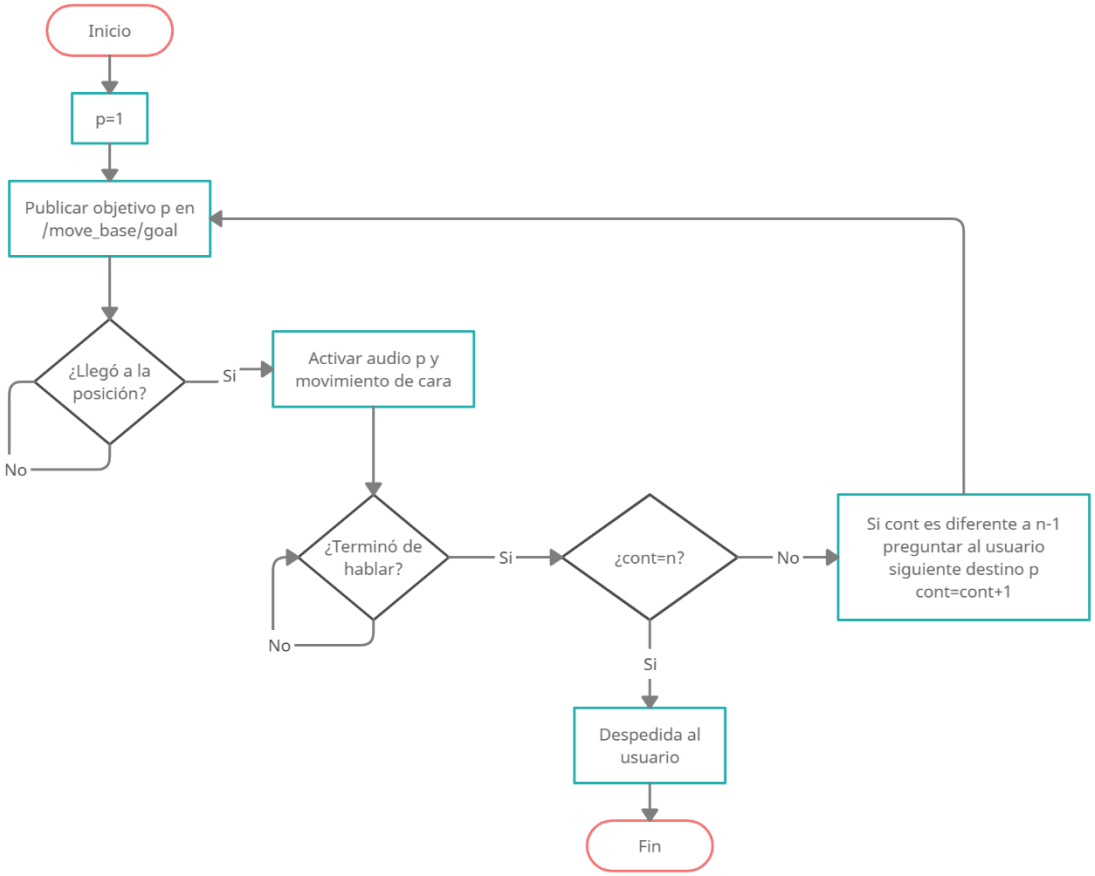


Figura 6.3. Diagrama de flujo del recorrido.

Prácticamente la máquina de estados se suscribe a la mayoría de tópicos utilizados, por lo que la suscripción y publicación a estos se gestiona en cada estado específicamente según sea el caso. El diagrama de la máquina de estados creada con *SMACH,* usando la herramienta de visualización *smach_viewer*, se muestra en la figura 6.4. En esta se muestra como cada estado tiene sistemas de reconocimiento de fallos, o datos inválidos, para que si se presenta alguno, se repita el estado hasta obtener los datos adecuados. Por ejemplo si cuando el robot le pregunta al usuario si está listo, la respuesta es “no”, o no responde de manera adecuada o no se escucha correctamente, el robot esperará un tiempo y volverá al mismo estado; preguntándole nuevamente hasta obtener la respuesta afirmativa para empezar.

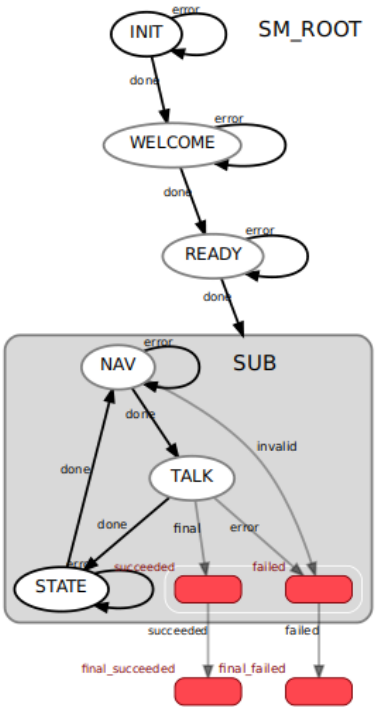


Figura 6.4. Nodo *controlSM_node.*

La cantidad de puntos por recorrer (n) y la posición de estos puntos se pueden configurar por medio del archivo: *pointsConfig.txt* localizado en el directorio */config* del paquete *pmb2_control*. El formato a seguir es el número *ID* del punto, la posición de este (x,y,z, orientación x, orientación y, orientación z, orientación w (formato en cuartenios)), y el nombre que se le desea dar a cada estación, separado entre espacios, como se muestra en la figura 6.5.

```
1 0.79,1.89,0,0,0,0.06,0.99 Robots
2 -4.02,2.03,0,0,0,-0.08,0.99 PLC
3 0.22 -4.68 0,0,0,-0.07,0.99 Ordenadores
```

Figura 6.5. Formato de archivo de configuración para agregar o eliminar puntos al recorrido.

En caso de querer añadir más puntos, para conocer las posiciones que se deben poner en este archivo, se puede cargar el mapa creado en el ordenador y utilizando la herramienta *Rviz,* identificar las posiciones nuevas deseadas o moviendo el robot a la posición deseada y leyendo el tópico que indica la posición actual. También en caso de modificar las posiciones, agregar nuevas o querer actualizar los audios descriptivos de cada posición; se debe añadir el audio deseado con nombre igual al *ID* colocado en el archivo de configuración, en el directorio */audios* del paquete *pmb2_face,* en formato *ogg.*

7. RECORRIDO GUIADO

En este capítulo se muestra el uso y los resultados obtenidos del proyecto, realizando un recorrido en el laboratorio en tres estaciones (Punto 1, Punto 2 y Punto 3), como prueba de funcionamiento de todos los programas previamente creados. Los resultados y figuras mostradas serán sobre las simulaciones ejecutadas en el ordenador externo (Anexo II: Archivos de simulación proporcionados por *PAL Robotics*), utilizando los mismos programas instalados en el robot, y el resultado real se muestra en un vídeo almacenado en el directorio *git* del proyecto.

El objetivo principal de este experimento es que el robot en conjunto con la *Raspberry* (mostrado en la figura 7.1) se inicialice con todos los paquetes necesarios, se comunique con el usuario e inmediatamente se disponga a realizar el recorrido a la primera estación con su respectiva narración, y seguidamente se dirija a la siguiente posición definida por el usuario, y así hasta finalizar los tres recorridos.

Figura 7.1. Montaje final *PMB2-Rasberry*

El entorno donde se desarrollará la prueba será una simulación de una oficina con el programa *Gazebo* (figura 7.2) para las simulaciones y el laboratorio 485 para el caso real. Como se desea comprobar que al ejecutar las trayectorias el robot sea capaz

de esquivar obstáculos; se situarán objetos que no se encuentren en el mapa dentro de la simulación para observar el comportamiento de la base móvil ante estos.

Figura 7.2. Simulación de oficina en *Gazebo*.

7.1 Estructura final del proyecto

Para sintetizar la estructura final del proyecto se presenta la figura 7.3. En esta se muestran los tres grandes bloques del proyecto; movimiento, interacción y control. Como ya se ha mencionado, *roscore* se ejecuta en el ordenador del robot para habilitar la red ROS, los nodos relacionados con la navegación autónoma también serán ejecutados en este ordenador. Los nodos relacionados al habla, escucha y cara del robot serán gestionados por la *Raspberry*, al igual que la máquina de estados y el programa principal del proyecto. Para lanzar nodos en una máquina y que sean ejecutados en otra, se utiliza el parámetro *machine* en los archivos *launch.* Este parámetro se define al inicio del archivo con los atributos como: nombre de la máquina, *IP*, usuario, archivo de ambiente, contraseña en caso que no se tenga configurado los certificados *ssh*, y tiempo máximo de espera de conexión (por defecto 10 segundos):

```
<machine name="pmb2-5c" address="10.68.0.1" user="root" env-loader="/home/pal/pmb2_UJA/scripts/env.sh" />
```

Cada vez que se haga el llamado de un nodo se le asigna en que máquina se requiere su ejecución. Es importante tener en cuenta que el respectivo paquete y nodo

deben estar instalados en la máquina objetivo, por ejemplo, a pesar de que los nodos de navegación se lancen desde la *Raspberry*, al ser el robot quien los va a ejecutar, estos deben de estar instalados en la base robótica.

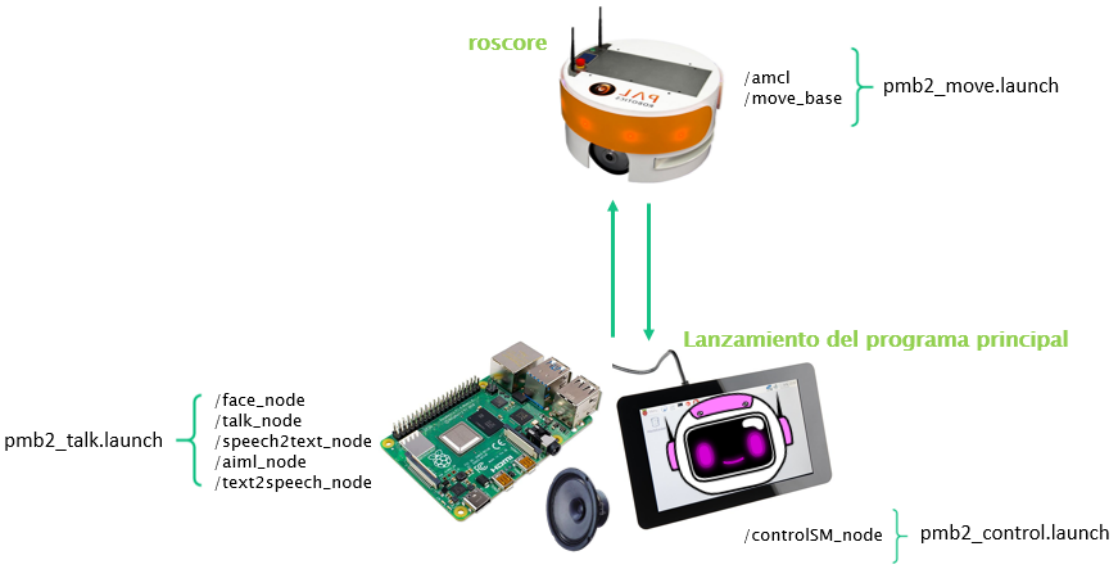


Figura 7.3. Estructura del proyecto.

Las conexiones de los nodos que se obtienen al ejecutar los algoritmos encargados de crear el mapa se muestra en la figura 7.4. En esta se observa lo marcado en recuadro verde como los comandos de velocidad provenientes del mando externo para realizar el recorrido necesario. Y en los recuadros rosa se muestra la obtención del valor del sensor y las transformadas en el caso simulado (a) y utilizando la plataforma real (b).

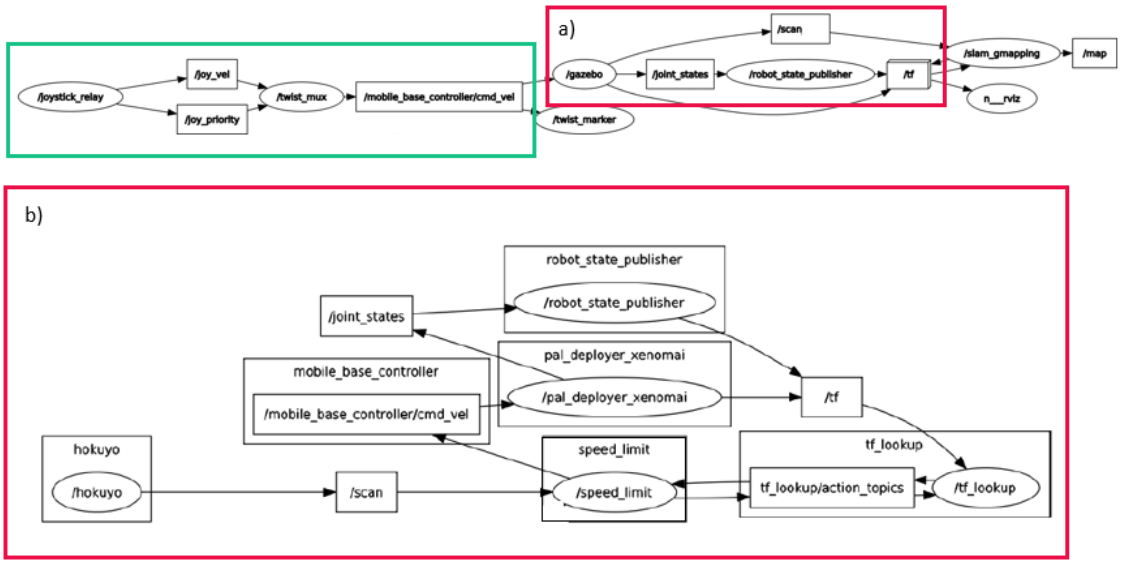


Figura 7.4. Gráfico de conexiones de los nodos al crear el mapa.

Y las conexiones de los nodos al ejecutar todo el proyecto se muestra en las figuras 7.5, donde se puede observar las interacciones de los nodos a través de las suscripciones y publicaciones en los tópicos. Sin embargo esta figura generada con la herramienta *rqt_graph* solo muestra las conexiones realizadas a través de tópicos y acciones, los servicios no se ven reflejados, por esta razón no se ve una línea de conexión entre el nodo de control (máquina de estados) y los módulos encargados del movimiento del robot.

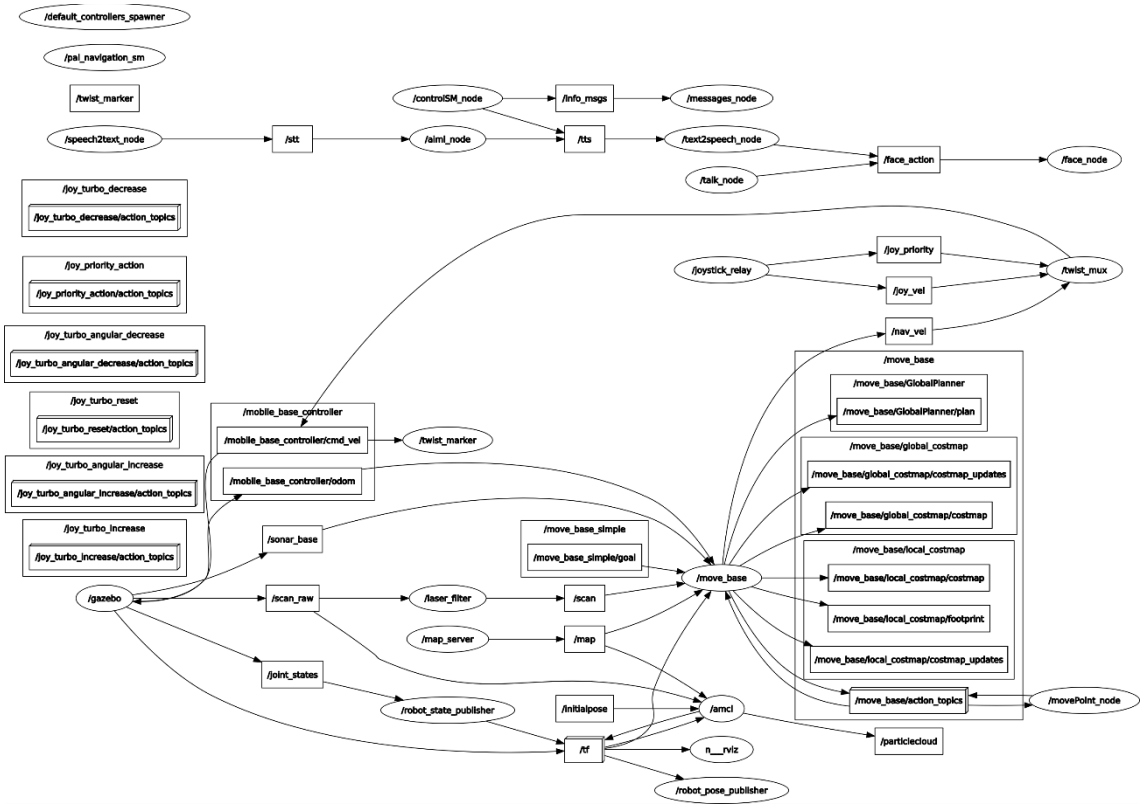


Figura 7.5. Gráfico de conexiones de los nodos del proyecto.

En la figura 7.6 se muestra una sección del árbol de transformadas (diagrama completo en anexo III) del sistema obtenido con el comando *"rosrun tf view_frames"*, donde se observa que el padre de todos los *frames* es el mapa, ya que todas las posiciones son referenciadas a partir de este. Además existe un camino entre el mapa y la base del robot por medio de la odometría, que es de lo que se encarga el nodo de localización *amcl.* Y seguidamente se tienen las conexiones y transformadas entre las distintas partes del robot definidas en el archivo *URDF* creado por el fabricante.

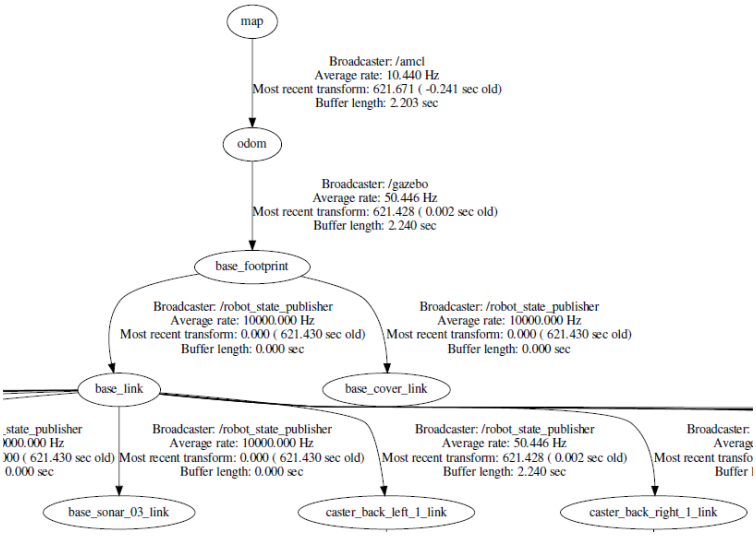


Figura 7.6. Sección del árbol de transformadas del robot *PMB-2.*

Finalmente, los tres paquetes creados y sus respectivos archivos (*pmb2_lab_nav, pmb2_face* y *pmb2_control*) se encuentran almacenados en un repositorio de *git* *https://github.com/lfvm0001/pmb2_UJA*.

7.1 Ejecución de la aplicación

La aplicación final mostrada para el usuario consiste en un archivo *launch* que se encarga de cargar primeramente el *script* que termina el proceso de los nodos que no se desean utilizar, y seguidamente carga los otros ficheros *launch* mencionados anteriormente. Con esto se tendrán cargados todos los nodos necesarios, así como la máquina de estados lista para funcionar, y la cara de robot en la pantalla de la *Raspberry*. En conjunto con la cara se lanza una pequeña interfaz gráfica de usuario (*GUI*) para mostrar mensajes de las diferentes etapas, y un botón para detener o reiniciar el proceso en caso que se requiera (figura 7.7).

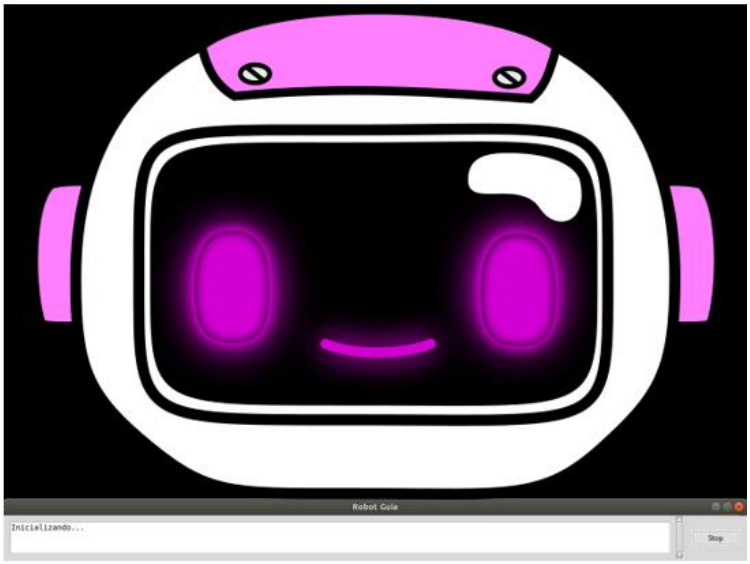


Figura 7.7. Interfaz mostrada al usuario al iniciar aplicación.

La interfaz fue creada con la biblioteca *Tk* para *Python*, y los mensajes son obtenidos a través de un tópico */info_msgs,* en cada una de las etapas donde se requería dar un mensaje informativo se publicó en este tópico, para ser leído y mostrado en la interfaz.

Al inicializarse estos nodos, el robot realiza una pequeña rutina con el fin de mejorar su localización inicial, la cual consiste girar sobre su propio eje y movilizarse una pequeña distancia hacia delante y hacia atrás. En la figura 7.8 se observa como la nube de partículas converge a un valor específico de posición al finalizar la rutina.

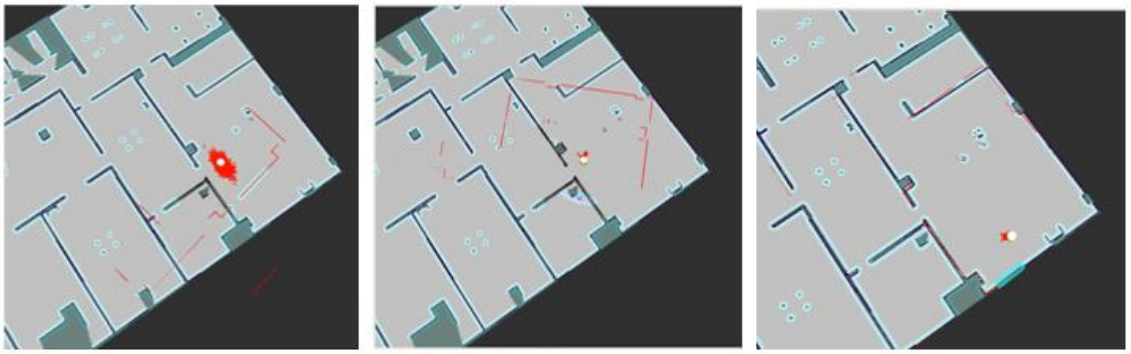


Figura 7.8. Rutina de localización inicial.

Seguidamente el robot lanza saludo al usuario y le pregunta su nombre, para darle la bienvenida y preguntarle si está listo para comenzar, al ser la respuesta sí, el robot genera una trayectoria hacia el primer punto definido en el archivo de configuración. Esta trayectoria generada se observa en la figura 7.9. Mientras el robot llega a su

destino, se modifica el archivo de *Gazebo* para agregar un obstáculo en medio de la ruta definida, para simular el posicionamiento de una persona u objeto no definido previamente en el mapa (figura 7.10). Se observa como la trayectoria generada con el planeador local se ajusta a este nuevo objeto observado con el sensor láser y se logra evitar para continuar el camino hacia el punto destino (figura 7.11).

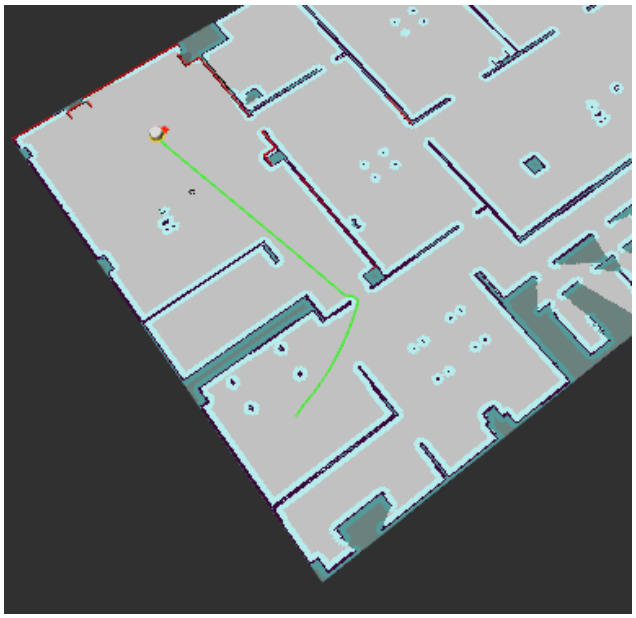


Figura 7.9. Trayectoria generada para el punto 1.

Figura 7.10. Objeto agregado al entorno.

Figura 4.11. a) Trayectoria recalculada, Robot en punto 1 b) *Rviz* c) *Gazebo* d) Mensaje en consola.

Una vez en este punto el robot se mantiene en esta posición, mostrando la cara en movimiento y el audio con la información del punto 1. Al finalizar la descripción, el robot le pregunta al usuario si desea continuar al punto 2 o el punto 3 definidos en la figura 7.2. En este caso se responde el punto 3 y se observa cómo se genera una nueva trayectoria hacia este punto (figura 7.12).

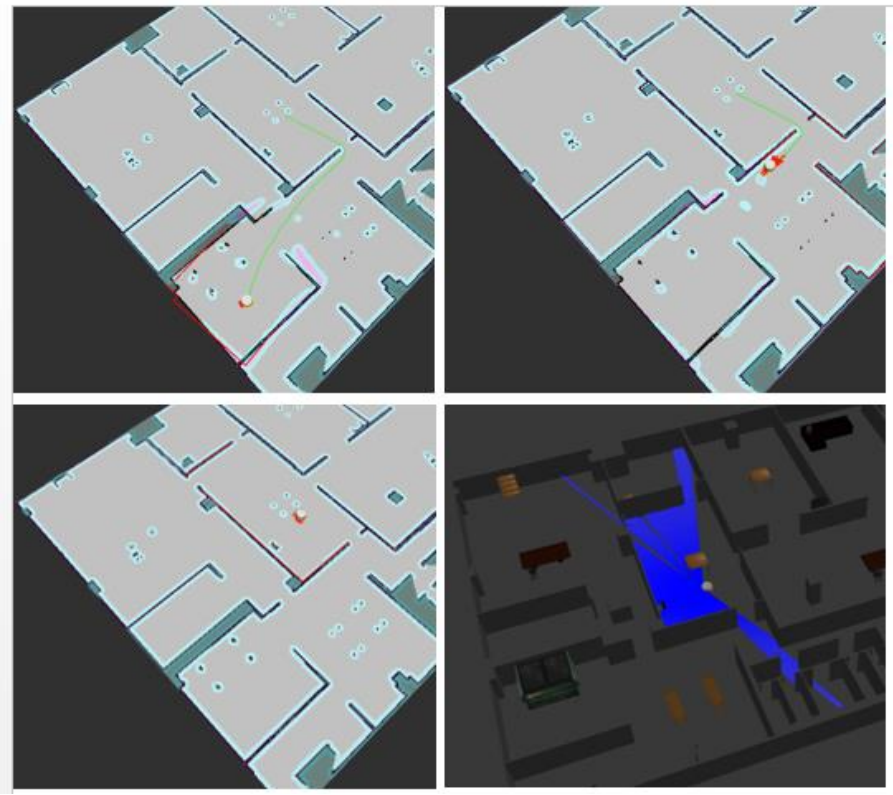


Figura 7.12. a) Trayectoria generada b) Robot siguiendo trayectoria, robot en punto 3 en c) *Rviz* d) *Gazebo*.

Una vez que llega a este punto se reproduce el audio correspondiente y finalmente se genera la ruta hacia el punto 2 (figura 7.13) donde se llega con éxito y se finaliza el programa después de la descripción correspondiente.

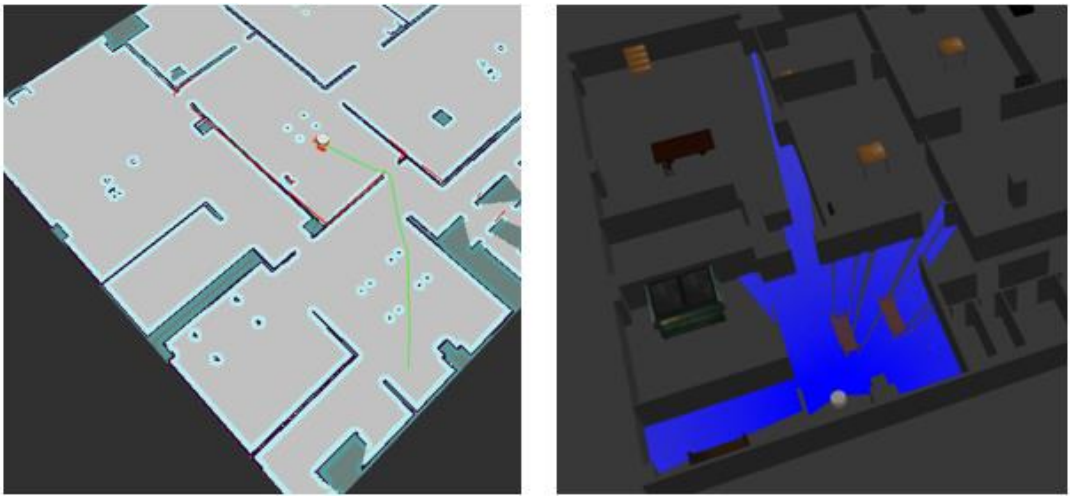


Figura 7.13. a) Trayectoria generada hacia punto 2 b) Robot en punto 2 en *Gazebo.*

A pesar que algunas trayectorias generadas se observan muy cercanas a los obstáculos; por ejemplo las paredes, el robot no llega a colisionar en estas gracias al efecto de inflación de los mapas de costes. El robot se ve cercano al objeto pero este se ve más grande de lo que realmente es para darle ese margen de seguridad y evitar así choques con obstáculos fijos o móviles.

Una pequeña demostración del proyecto se muestra en el video almacenado en el directorio */results* del repositorio de *git.* Es necesario recalcar que cuando se use el programa en el robot real, se debe tener cerca el mando *joystick* o el ordenador externo capaz de mover el robot, para moverlo en caso de un atascamiento del que el robot no se pueda recuperar inclusive después de ejecutar los comportamientos de recuperación.

8. CONCLUSIONES

Se logró completar el objetivo principal al realizar un programa que permite a la base móvil PMB-2 navegar en un entorno previamente mapeado, y darle la capacidad de interactuar con el usuario por medio de audio y visualización de una cara animada.

Tras el estudio del funcionamiento de ROS, fue posible hacer pruebas con diferentes algoritmos para generar el mapa del entorno y así elegir el más adecuado para el problema planteado, en este caso *Gmapping*; por su resultado gráfico y sus resultados computacionales. Se determinó que antes de comenzar la navegación, se requiere realizar una rutina de movimientos para mejorar la localización inicial, ya que el algoritmo de localización se va realimentando del movimiento del robot para obtener resultados más precisos. Además, fue posible la configuración de los archivos necesarios para realizar la navegación autónoma con capacidad de esquivar obstáculos, característica que fue comprobada por medio de simulación y con el robot real. Sin embargo es necesario tener algún sistema externo de control de movimiento en caso de que el robot se encuentre en un estado que no se pueda recuperar a través del programa implementado.

Se logró la conexión entre el robot, la *Raspberry* y el ordenador externo, tanto como para hacer pruebas, como para lanzar los diferentes nodos utilizados en el proyecto y que estos se comuniquen entre sí. Se diseñó una interfaz que permite que haya una conexión más personal entre el robot y los usuarios, por medio de una cara, un micrófono y un altavoz y utilizando servicios en la nube y el lenguaje de inteligencia artificial *AIML.* Los archivos *AIML* se pueden ampliar para tener conversaciones más complejas con el robot, ahora solo se utilizan para el saludo, la despedida y preguntas básicas como a cual punto se desea continuar. El funcionamiento de todo el programa por medio de la máquina de estados se comprobó mediante un recorrido guiado de tres puntos en simulación. Para observar el funcionamiento real se presenta un video en el repositorio. El programa puede ser configurable agregándole o quitándole puntos destino al recorrido por medio de un archivo de configuración.

Se comprueba el gran poder que se tiene al utilizar la plataforma ROS en sistemas robóticos ya que permite implementar programas haciendo uso de algoritmos implementados, de una manera estructurada, modular y fácil de manejar con gran cantidad de nodos.

Usando el mismo concepto de este trabajo de navegación y conversación con las personas, este proyecto da pie a trabajos futuros como lo pueden ser sistema de contabilización de aforo por medio de un sistema de visión por computador (útil en tiempos de pandemia), o contabilización de inventario. También puede ser un sistema de limpieza y desinfección del edificio al realizar un recorrido a lo largo de la planta. Otro proyecto que puede hacer uso de lo implementado es un robot guía en una biblioteca donde se cuente con una base de datos de la ubicación de los libros, por lo que si una persona necesita encontrar un libro el robot lo guíe hasta el estante respectivo.

BIBLIOGRAFÍA

- AIML Foundation. (2014). *AIML Docs*. Obtenido de http://www.aiml.foundation/
- Anusha. (2018). *Type of Robots.* Electronics Hub.
- AWS. (2016). *Amazon Polly Documentation*. Obtenido de https://docs.aws.amazon.com/polly/index.html
- Ben-Ari, M., & Mondada, F. (2018). *Elements of Robotics: Chapter 1: Robots and their Applications.*
- Christoforou, E., & Müller, A. (2017). *Robot and Robotics: The origin and beyond.* Dept. of Electrical and Computer Enginnering, University of Cyprus, Cyprus, Institute of Robotics, JKU Johannes Keppler University, Austria.
- Cinemática y dinámica de sistemas mecatrónicos. (2019-2020). *Tema 1: Introducción a la Robótica.* Máster en Ingeniería Mecatrónica, Universidad de Jaén, Jaén.
- Dellaerty, F., Fox, D., Burgard, W., & Thrun, S. (s.f.). *Monte Carlo Localization for Mobile Robots.* Pittsburgh: Carnegie Mellon University.
- Filipenko, M., & Afanasyev, I. (2018). *Comparison of Various SLAM Systems for Mobile Robot in Indoor Environment.* Rusia: Institute of Robotics, Innopolis University.
- Fox, D., Burgard, W., Dellaert, F., & Thrun, S. (2017). *Monte Carlo Localization: Efficient Position Estimation for Mobile Robots.* Pittsburgh & Bonn: Carnegie Mellon University & University of Bonn.
- Gallart del Burgo, X. (2013). *Semantic Mapping in ROS.* The Royal Institute of Technology, Stockholm, Sweden.
- González, C. (2015). *Desarrollo de una aplicación para el guiado automático de un vehículo eléctrico.* Valencia: Universidad Politécnica de Valencia.
- Google Cloud. (2007). *Speech-to-Text Documentation*. Obtenido de https://cloud.google.com/speech-to-text?hl=es
- Grisetti, G., Stachniss, C., & Burgar, W. (2007). *GMapping.*
- Guizzo, E. (2020). *What is a Robot?* Robots: your guide to the world of robotics, IEEE.

- Kazin, A. (2014). *Robots: the computer controlled machines.*
- Kohlbrecher, S., von Stryk, O., Meyer, J., & Klingauf, U. (2011). *A Flexible and Scalable SLAM System with Full 3D Motion Estimation.* Alemania: Universidad Tecnica de Darmstadt.
- Lentin, J. (2017). *ROS Robotics Projects.* Birmingham: Packt Publishing.
- Ortega, E. (2017). *Inclusión del robot PMB-2 en la plataforma basada en ROS.* Escuela Politécnica Superior de Jaén, Departamento de ingeniería electrónica y automática, Universidad de Jaén.
- *PAL Robotics*. (2020). Obtenido de http://pal-robotics.com/es/
- Pérez, A. (2017). *Desarrollo de sistema de navegación para vehículos autónomos terrestres utilizando ROS.* Escuela Técnica Superior de Ingeniería Industrial, Universidad Politécnica de Cartagena.
- Quigley, M., Gerkey, B., & Smart, W. (2015). *Programming Robots with ROS: A practical introduction to the robot operating system.* O'Reilly Media; Edición: 1.
- Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., . . . Ng, A. (2009). *ROS: an open-source Robot Operating System.* Computer Science Department, Stanford University, Stanford, CA, Willow Garage, Menlo Park, CA y Computer Science Department, University of Southern California.
- Real Academia Española. (2014). *Diccionario de la lengua española.* 23a ed.
- Revista de Robots. (2020). *Qué es un robot humanoide. Significado y ejemplos*. Obtenido de https://revistaderobots.com/robots-y-robotica/robots-humanoides/
- Robins, M., & Somashekhar S, H. (2016). *Trajectory tracking and control of differential drive robot for predefined regular geometrical path.* Department of Mechanical Engineering, Indian Institute of Technology, India.
- Robotic Industries Association. (2010). *Robot Terms and Definitions*. Obtenido de https://www.robotics.org/product-catalog-detail.cfm/Robotic-Industries-Association/Robot-Terms-and-Definitions/productid/2953
- Robotics Knowledgebase. (Febrero de 2020). *Adaptive Monte Carlo Localization*. Obtenido de https://roboticsknowledgebase.com/wiki/state-estimation/adaptive-monte-carlo-localization/
- *ROS*. (2007-2020). Obtenido de https://www.ros.org/

- ROS Components. (2016). *Hokuyo URG-04LX-UG01 by Hokuyo Automatic*. Obtenido de https://www.roscomponents.com/es/lidar-escaner-laser/83-urg-04lx-ug01.html
- ROS Wiki. (2013). *AMCL*. Obtenido de Package Summary: http://wiki.ros.org/amcl
- ROS Wiki. (2013). *Dwa_local_planner*. Obtenido de Package Summary: http://wiki.ros.org/dwa_local_planner
- ROS Wiki. (2013). *Gmapping*. Obtenido de Package Summary: http://wiki.ros.org/gmapping
- ROS Wiki. (2013). *Hector_mapping*. Obtenido de Package Summary: http://wiki.ros.org/hector_mapping
- ROS Wiki. (2013). *Move_base*. Obtenido de Package Summary: http://wiki.ros.org/move_base
- ROS Wiki. (2013). *Navigation*. Obtenido de Package Summary: http://wiki.ros.org/navigation
- ROS Wiki. (2013). *SLAM_karto*. Obtenido de Package Summary: http://wiki.ros.org/slam_karto
- ROS Wiki. (2013). *SMACH*. Obtenido de Package Summary: http://wiki.ros.org/smach
- ROS Wiki. (2015). *Global_planner*. Obtenido de Package Summary: http://wiki.ros.org/global_planner
- *ROS: Documentation Wiki*. (2007-2020). Obtenido de http://wiki.ros.org/
- Siegwart, R., Scaramuzza, D., & Nourbakhsh, I. (2011). *Introduction to Autonomous Mobile Robot.*
- The Construct. (2017). *ROS navegation course*. Obtenido de Robot Ignate Academy: https://www.theconstructsim.com/robotigniteacademy_learnros/ros-courses-library/ros-courses-ros-navigation-in-5-days/
- Thrun, S. (2002). *Robotic Mapping: A survey.* Pittsburgh: School of Computer Science, Carnegie Mellon University.
- Vanelli, B. (2019). *Comparison and benchmarking for SLAM in mobile robots.* Blumenau: Universidade Federal de Santa Catarina.

- Wang, E. (2017). *Description of Monte Carlo Localization: Efficient Position Estimation for Mobile Robots paper*. Obtenido de Synced: AI technology & Industry review: https://syncedreview.com/2017/02/22/monte-carlo-localization-efficient-position-estimation-for-mobile-robots/
- Zamora, E. (2015). *Map-building and planning for autonomous navigation of a mobile robot.* Mexico: Center for Research and Advanced Studies of the National Polytechnic Institute.
- Zheng, K. (2016). *ROS Navigation Tuning Guide.*

ANEXO II. REPOSITORIO DEL PROYECTO

Los archivos creados en este proyecto al igual que los videos con los resultados se encuentran en el repositorio: https://github.com/lfvm0001/pmb2_UJA/tree/main

En este se encuentran los tres directorios de los paquetes principales con sus respectivos archivos. Además de un directorio con los scripts utilizados para copiar las claves locales y terminar el proceso de los nodos no utilizados en el robot. Y finalmente un directorio con los videos donde se muestra el funcionamiento del proyecto.

ANEXO II. ARCHIVOS DE SIMULACIÓN

Para realizar pruebas del robot *PMB-2* en *Rviz* y *Gazebo*, es necesario tener los archivos de simulación que carguen los nodos y tópicos equivalentes a los disponibles con el robot físico.

Para tener acceso a estos archivos es necesario crear un *workspace* único con el nombre deseado. Seguidamente es necesario descargar el archivo pmb2_public.rosinstall y copiarlo en este espacio para ejecutar el siguiente comando:

```
rosinstall src /opt/ros/kinetic pmb2_public.rosinstall
```

Para comprobar que todas las dependencias se hayan instalado correctamente:

```
sudo rosdep init
rosdep update
```

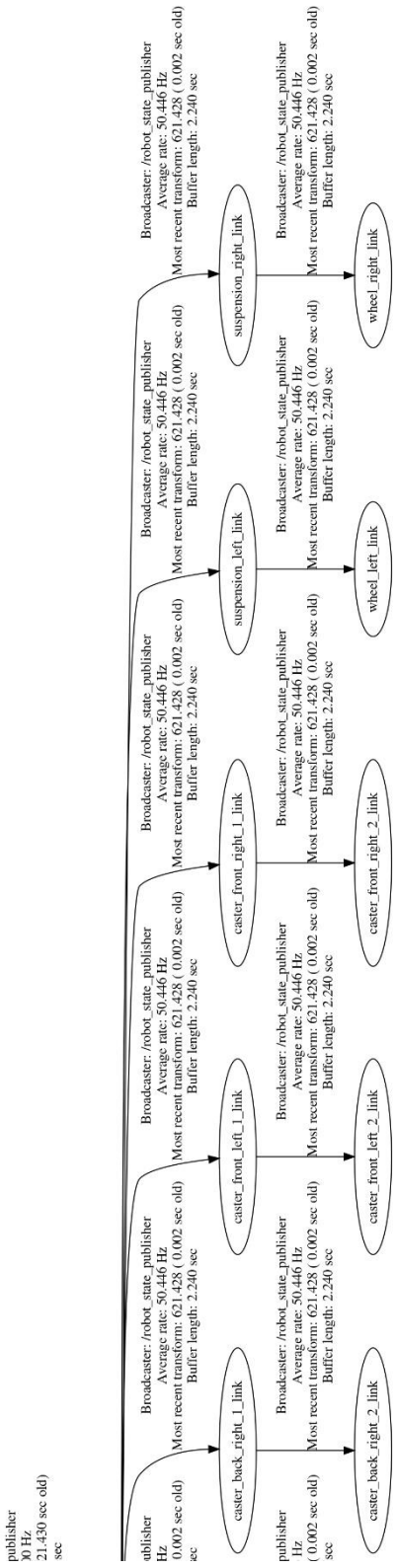
```
rosdep install --from-paths src --ignore-src --rosdistro kinetic --skip-keys
"pal_gazebo_plugins speed_limit_node sensor_to_cloud pmb2_rgbd_sensors
pal_vo_server pal_karto pal_usb_utils pal_local_planner pal_filters hokuyo_node
rrbot_launch robot_pose pal_pcl rviz_plugin_covariance pal-orbbec-openni2
slam_toolbox"
```

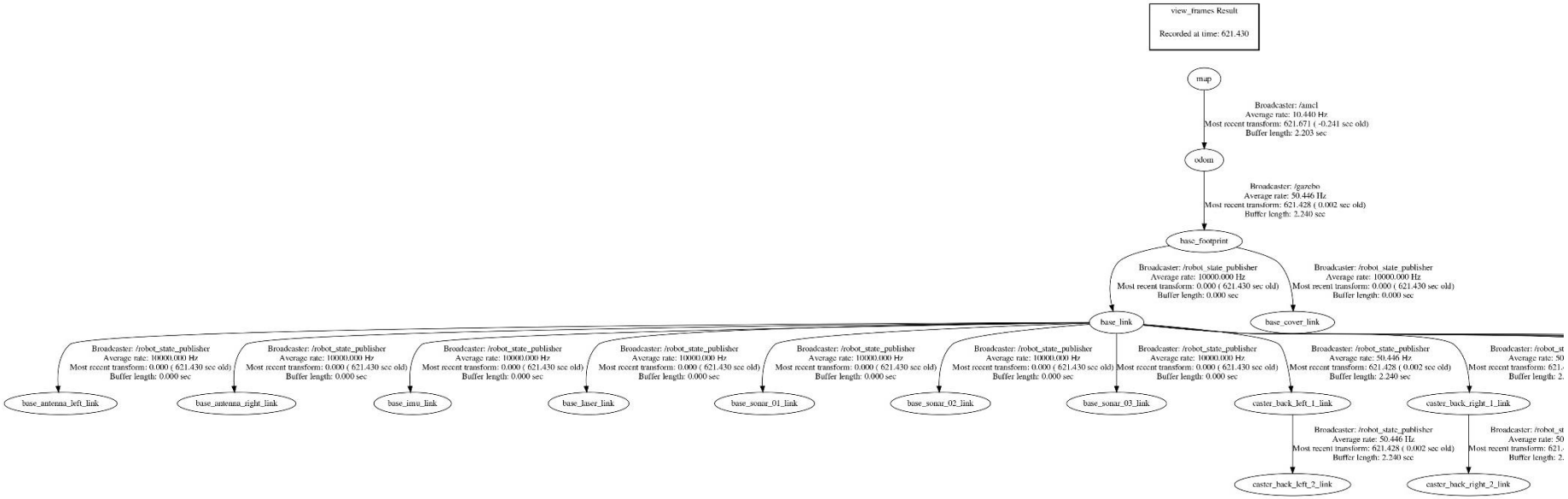
Finalmente se reconstruye el espacio de trabajo:

```
source /opt/ros/kinetic/setup.bash
catkin build
```

Para más información y tutoriales sobres las simulaciones, se puede consultar el manual de usuario del robot o consultar la página: http://wiki.ros.org/Robots/PMB-2/Tutorials

ANEXO III. ÁRBOL DE TRANSFORMADAS DEL SISTEMA




view_frames Result
Recorded at time: 621.430
map
Broadcaster: /amcl
Average rate: 10.440 Hz
Most recent transform: 621.671 (-0.241 sec old)
Buffer length: 2.203 sec
odom
Broadcaster: /gazebo
Average rate: 50.446 Hz
Most recent transform: 621.428 (0.002 sec old)
Buffer length: 2.240 sec
base_footprint
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
base_link
base_cover_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
base_antenna_left_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
base_antenna_right_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
base_imu_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
base_laser_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
base_sonar_01_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
base_sonar_02_link
Broadcaster: /robot_state_publisher
Average rate: 10000.000 Hz
Most recent transform: 0.000 (621.430 sec old)
Buffer length: 0.000 sec
base_sonar_03_link
Broadcaster: /robot_state_publisher
Average rate: 50.446 Hz
Most recent transform: 621.428 (0.002 sec old)
Buffer length: 2.240 sec
caster_back_left_1_link
Broadcaster: /robot_st
Average rate: 50
Most recent transform: 621.
Buffer length: 2.
caster_back_right_1_link
Broadcaster: /robot_state_publisher
Average rate: 50.446 Hz
Most recent transform: 621.428 (0.002 sec old)
Buffer length: 2.240 sec
caster_back_left_2_link
Broadcaster: /robot_st
Average rate: 50
Most recent transform: 621.
Buffer length: 2.
caster_back_right_2_link