Universidad Zaragoza
1542

Trabajo Fin de Grado

Robot Móvil Controlado por ROS

Mobile Robot Powered by ROS

Autor

David Pérez Román

Director

Javier Esteban Escaño

Escuela Universitaria Politécnica La Almunia
2017

Escuela Universitaria
Politécnica - La Almunia
Centro adscrito
Universidad Zaragoza

ESCUELA UNIVERSITARIA POLITÉCNICA
DE LA ALMUNIA DE DOÑA GODINA (ZARAGOZA)

MEMORIA

Robot Móvil Controlado por ROS

Mobile Robot Powered by ROS

424.20.39

Autor: David Pérez Román

Director: Javier Esteban Escaño

Fecha: 22/06/2021

INDICE DE CONTENIDO

INDICE DE ILUSTRACIONES

INDICE DE TABLAS

INDICE DE ECUACIONES

1. RESUMEN

En este proyecto se ha realizado un acercamiento así como un análisis de la herramienta ROS, usada en el desarrollo de software para robots móviles, pasando por la historia de la herramienta, su estado actual en la industria, su uso, implementación y, finalmente, posibles oportunidades futuras para la herramienta. No obstante, pese a que el proyecto está centrado en ROS, también se exploran temas relacionados con el robot móvil a controlar, tales como sus componentes electrónicos, diseño del chasis, autonomía, sensores y, tanto diseños eléctrico como electrónico.

Para realizar el control del robot mediante ROS se ha intentado emplear un ESP32, ROS2 y Micro-ROS. Donde el esp32 actuaría como agente en el sistema de ROS gracias a la integración de Micro-ROS en el microprocesador, consiguiendo que el esp32 pueda ser el único dispositivo encargado del control del robot, pero debido a varios problemas se acabó descartando la idea y se terminó optando por un robot controlado por ROS en una Raspberry Pi.

Por lo que respecta al diseño del robot móvil, se han empleado elementos comerciales en la medida de lo posible, mientras que los elementos específicos se han diseñado empleando el programa de diseño 3D Autodesk Inventor para las piezas de la estructura y KiCAD para el plano electrónico y diseño de la PCB.

1.1. PALABRAS CLAVE

- ROS
- Robot móvil
- Arduino
- Electrónica
- Diseño

2. ABSTRACT

In this project, an approach as well as an analysis of the ROS tool, used in the development of software for mobile robots, has been carried out, going through the history of the tool, its current state in the industry, its use, implementation and, finally, possible future opportunities for the tool. However, despite the fact that the project is focused on ROS, ideas related to the mobile robot to be controlled are also explored, such as its electronic components, chassis design, autonomy, sensors, and both electrical and electronic designs.

To control the robot using ROS, an attempt has been made to use an ESP32, ROS2 and Micro-ROS. Where the esp32 would act as an agent in the ROS system thanks to the integration of Micro-ROS in the microprocessor, achieving that the esp32 could be the only device in charge of controlling the robot, but due to several problems the idea was discarded and it was decided to develop a ROS controlled robot on a Raspberry Pi.

Regarding the design of the mobile robot, commercial elements have been used where possible, while specific elements have been designed using the 3D design program Autodesk Inventor for the parts of the structure and KiCAD for the electronic drawing and PCB layout.

3. INTRODUCCIÓN

3.1. MOTIVACIÓN

La idea para este TFG surgió cuando en una asignatura del 4º curso de la carrera se nos propuso la realización de un tooling para un AGV. Mientras estaba realizando el estudio del arte me di cuenta de que en muchos de los proyectos se usaba ROS para el control del robot, entonces decidí investigar sobre el asunto y acabé descubriendo de qué trataba ROS y pensé que sería un buen tema de estudio para el TFG.

Una vez descubrí ROS pensé ¿Por qué usar ROS? Y, tras investigar un poco, vi que los robots móviles se han convertido, con el paso del tiempo, en un recurso cada vez más empleado en empresas de todo tipo y, con estos robots, han aparecido nuevos desafíos para los equipos de ingenieros responsables de su funcionamiento. Es en este momento cuando aparece ROS, un marco de trabajo o "Framework" flexible orientado a la escritura de software para robots, con el objetivo de simplificar la tarea de crear un comportamiento complejo y robusto para un robot.

Con esta información pensé ¿Puede una persona sin experiencia con ROS, hacer un proyecto basado en esta herramienta? La respuesta a esa pregunta es este mismo proyecto.

3.2. OBJETIVO

Mi objetivo principal en este proyecto es realizar con éxito la programación del robot empleando ROS y así poder familiarizarme con su uso y aplicaciones. Sin embargo, de forma secundaría pretendo demostrar si un individuo, en este caso yo, sin ninguna experiencia previa con ROS, puede realizar un proyecto y conseguir que funcione de forma óptima. Con esto me gustaría presentar un referente para que otros individuos principiantes también intenten usar ROS y, si con suerte se presenta la ocasión, que la propia universidad se plantee incluir ROS en alguna asignatura, o como un curso optativo para los estudiantes puedan familiarizarse con una herramienta que, posiblemente, les resulte de gran ayuda en su futuro profesional.

En lo referente al robot, el objetivo es realizar un diseño que cumpla con unas determinadas características tanto en el apartado mecánico como en el eléctrico/electrónico.

4. ESTADO DEL ARTE

4.1. ¿QUÉ ES ROS?

ROS, o por sus siglas Robot Operating System, es un marco de trabajo o "Framework" flexible usado para la creación de software empleado en robots. También puede ser considerado como una colección de herramientas, librerías y convenciones cuyo objetivo es simplificar la tarea de crear un comportamiento robótico complejo y robusto en una amplia variedad de plataformas robóticas (Open Robotics, 2013).

En este punto podemos preguntarnos: ¿Por qué la necesidad de aprender a usar ROS si podemos crear el robusto y complejo comportamiento antes mencionado sin emplear ROS? La respuesta es simple:

Porque crear un software robótico de propósito general verdaderamente robusto y complejo es difícil. Desde la perspectiva del robot, los problemas que parecen triviales para los humanos a menudo pueden variar enormemente entre instancias de tareas y entornos. Hacer frente a estas variaciones es tan difícil que ningún individuo, laboratorio o institución puede esperar hacerlo por sí solo.

Como resultado, ROS se creó para fomentar el desarrollo colaborativo de software de robótica. Por ejemplo, un laboratorio podría tener expertos en mapeo de entornos interiores y podría contribuir con un sistema de clase mundial para producir mapas. Otro grupo podría tener expertos en el uso de mapas para navegar, y otro grupo podría haber descubierto un enfoque de visión por ordenador que funciona bien para reconocer objetos pequeños en desorden. ROS fue diseñado específicamente para que grupos como estos colaboren y se basen en el trabajo de los demás. (Open Robotics, 2013)

Ilustración 1 Esquema del ecosistema de ROS (Open Robotics, 2013)

Un sistema ROS se compone de varios nodos independientes, cada uno de los cuales se comunica con los demás nodos mediante un modelo de comunicación de publicación/suscripción.

Hay que tener en cuenta que los nodos en ROS no tienen por qué estar en el mismo sistema (varios ordenadores) o incluso en la misma arquitectura. Se podría tener un Arduino publicando mensajes, un ordenador portátil suscrito a ellos y un teléfono Android controlando unos motores. Esto hace que ROS sea realmente flexible y adaptable a las necesidades del usuario. ROS también es de código abierto, mantenido por muchas personas (Dattalo, 2018).

4.2. HISTORIA DE ROS

ROS es un gran proyecto con muchos antepasados y contribuyentes. Muchas personas de la comunidad de investigación en robótica sintieron la necesidad de un marco de colaboración abierto, y se han creado muchos proyectos con este objetivo. (Open Robotics, 2013)

A principios de 2007, las primeras piezas de lo que eventualmente se convertiría en ROS estaban comenzando a juntarse en la Universidad de Stanford. Eric Berger y Keenan Wyrobek, estudiantes de doctorado que trabajaban en el laboratorio de robótica de Kenneth Sailsbury en Stanford, dirigían el Personal Robotics Program o Programa Personal de Robótica. Mientras trabajaban en robots para realizar tareas de manipulación en entornos humanos, los dos estudiantes notaron que muchos de sus compañeros se veían reprimidos por la naturaleza diversa de la robótica. En un intento por remediar esta situación, los dos estudiantes se propusieron crear un sistema de base que proporcionaría un punto de partida para que otros en el mundo académico pudieran basarse.

Ilustración 2 Logo del programa personal de robótica de la universidad de Stanford (Stanford, 2010)

Antes de que las ideas se fusionaran por completo para convertirse en ROS, se crearon prototipos de varios marcos de software de robótica en proyectos de investigación en Stanford, incluido el Robot de Inteligencia Artificial STanford (STAIR) y el Programa Personal de Robótica (PR).

En noviembre de 2007 Willow Garage (laboratorio de investigación en robótica y tecnología dedicada al desarrollo de hardware y software de código abierto para aplicaciones de robótica personal) comienza su proyecto de Robótica Personal y ROS se convierte en una entidad formal. Grupos de más de veinte instituciones hicieron

Ilustración 3 Logo de Willow garaje (WGarage, 2014)

contribuciones a ROS, tanto en el software central como en el creciente número de paquetes los cuales trabajaron con ROS para formar un ecosistema de software mayor. El hecho de que personas ajenas a Willow Garage contribuyesen a ROS (en particular desde el proyecto STAIR de Stanford) significó que ROS era una plataforma de multi-robot desde el principio.

La primera distribución oficial de ROS: ROS Box Turtle, fue lanzada el 2 de marzo de 2010, marcando la primera vez que ROS se distribuyó oficialmente con un conjunto de paquetes versionados para uso público. Estos desarrollos hicieron posible que apareciera el primer dron controlado por ROS, el primer automóvil autónomo controlado con ROS, y la adaptación de ROS para Lego Mindstorms.

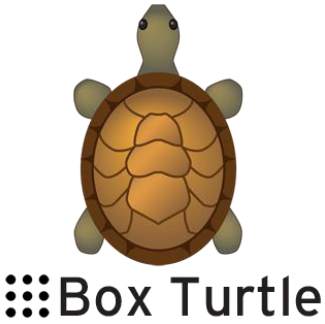


Ilustración 4 Logo ROS Box Turtle (Open Robotics, 2013)

2011 fue un año excepcional para ROS con el lanzamiento de ROS Answers el 15 de febrero, un foro de preguntas y respuestas para usuarios de ROS y la presentación del exitoso kit Turtlebot el 18 de abril. Willow Garage comenzó en abril de 2012 con la creación de Open Source Robotics Foundation (OSRF) (Koening, 2012). La OSRF recibió inmediatamente un contrato de software por parte de DARPA. Más tarde ese año, se llevó a cabo la primera ROSCon, con más de 200 participantes (kwc, 2012), en St. Paul, Minnesota. Además, se publicó el primer libro sobre ROS, ROS By Example, y Rethink Robotics anunció el Baxter, el primer robot comercial en ejecutar ROS. Poco después de cumplir su quinto aniversario en noviembre, ROS comenzó a funcionar en todos los continentes el 3 de diciembre de 2012.

Ilustración 5 Logo de la OSRF (Open Robotics, 2013)

En febrero de 2013, la responsabilidad del desarrollo central y el mantenimiento de ROS es transferido de Willow Garage a la aún joven Open Source Robotics Foundation. En este punto, ROS había lanzado siete versiones principales (hasta ROS Groovy) y tenía usuarios en todo el mundo.

En los años transcurridos desde que OSRF se hace cargo del desarrollo primario de ROS, se ha lanzado una nueva versión cada año, mientras que el interés en ROS sigue creciendo. Las ROSCons se han realizado todos los años desde 2012, ubicadas conjuntamente con ICRA (International Conference on Robotics and Automation) o IROS (International Conference on Intelligent Robots and Systems), dos conferencias emblemáticas de robótica. Se han organizado reuniones de desarrolladores de ROS en una variedad de países, se han publicado varios libros de ROS y se han iniciado muchos programas educativos. El 1 de septiembre de 2014, la NASA anunció el primer robot que usaría ROS en el espacio: Robotnaut 2, en la Estación Espacial Internacional.

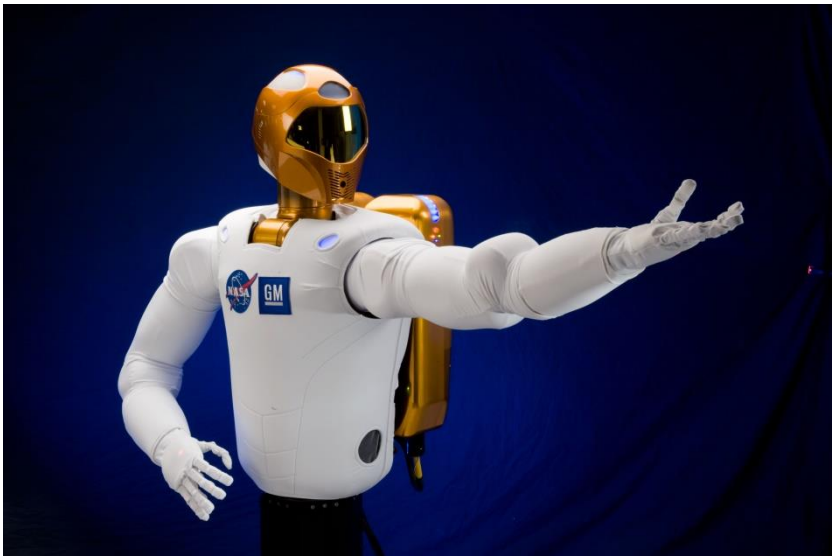


Ilustración 6 Robonaut 2 (Open Robotics, 2013)

Los gigantes tecnológicos Amazon y Microsoft comenzaron a interesarse en ROS durante este tiempo, con Microsoft transfiriendo (porteando) el núcleo ROS a Windows en septiembre de 2018, seguido de Amazon Web Services lanzando RoboMaker en noviembre. (Open Robotics, 2013)

4.3. ROS 2

ROS 2 se define como un middleware basado en un mecanismo de publicación/suscripción anónimo que permite el paso de mensajes entre diferentes procesos ROS.

Desde que ROS fue concebido en 2007, muchas cosas han cambiado en la comunidad de robótica y ROS. El objetivo del proyecto ROS 2 es adaptarse a estos cambios, aprovechando los puntos buenos de ROS 1 y mejorando los que no lo son. (ROS2, 2019)

El entorno ROS fue desarrollado en Willow Garage para el robot PR2 de Willow Garage. El PR2 es un robot humanoide que puede navegar de forma autónoma en un entorno conocido. En ese momento sabían que el PR2 no sería el único robot en el mercado y en lugar de desarrollar un programa personalizado que funcionara solo para PR2, querían implementar un software escalable y estándar para mejorar o modificar el robot en el futuro. Entonces, trabajaron en un middleware, ROS, definiendo niveles de abstracción para ser útiles en otros robots. (Mazzari, 2019)

Se guiaron por las características del PR2:

- Un solo robot
- Sin requisitos de tiempo real
- Excelente conectividad de red
- Multiplataforma (que funciona tan bien en bases móviles como en humanoides)

Hoy vemos que ROS se usa no solo en robots como el PR2 y similares, sino también en robots con ruedas de todos los tamaños, humanoides con piernas, brazos industriales, vehículos terrestres para exteriores y vehículos aéreos. Al mismo tiempo, ROS se adopta en múltiples dominios, más allá de la comunidad de investigación académica que era su enfoque inicial. Cada vez hay más robots fabricados a base de ROS disponibles en el mercado y las agencias gubernamentales están invirtiendo en ROS para sus propios sistemas de campo.

Ilustración 7 Personal Robot 2: PR2 (Bernier, 2013)

Con todos estos nuevos usos, ROS se está llevando a sus límites. Si bien es verdad que se está manteniendo bien, la comunidad cree que se puede desarrollar una plataforma mejor basada en los nuevos requisitos de los robots:

- Equipos de varios robots
- Pequeñas plataformas integradas
- Sistemas en tiempo real

- Redes no ideales
- Entornos de producción

Otra razón para implementar ROS 2 es aprovechar la oportunidad de mejorar las API de cara al usuario. Se diseñarán algunas API nuevas, incorporando lo mejor de la experiencia de la comunidad con las API de primera generación.

Como regla general, los cambios descritos anteriormente podrían integrarse en el código central de ROS existente. Esta opción, considerada y dada la naturaleza intrusiva de los cambios, presenta demasiado riesgo asociado con cambiar el sistema de ROS actual en el que confían tantas personas. ROS debería permanecer del mismo modo en el que es hoy. Así que ROS2 se construye como un conjunto paralelo de paquetes que se pueden instalar e interoperar con ROS.

Cada robot y componente compatible con ROS será compatible con ROS2 una vez que los controladores, paquetes y SDK estén disponibles para las implementaciones de ROS2. (Gerkey, 2007)

4.4. MICRO-ROS

Micro-ROS coloca ROS 2 en micro controladores, convirtiéndolos en participantes de primera clase del entorno ROS 2.

Los principales cambios en comparación con ROS 2 "estándar" es que micro-ROS utiliza un sistema operativo en tiempo real (RTOS) en lugar de Linux, y DDS para entornos extremadamente restringidos de recursos (DDS-XRCE) en lugar de DDS clásico.

Ilustración 8 Logo de Micro-ROS (Micro-ROS, 2020)

Básicamente, se sigue la arquitectura ROS 2 y se hace uso de su capacidad de conexión de middleware para usar DDS-XRCE, que es adecuado para micro controladores. Además, se utiliza un RTOS basado en POSIX (NuttX) en lugar de Linux. También se agregan abstracciones de RTOS, para facilitar la migración a otros RTOS. (FIWARE, 2020)

Micro-ROS apunta a familias de micro controladores de 32 bits de rango medio y alto rendimiento. Por el momento, todos los puertos oficiales se basan en la serie STM32 de ST y en el ESP32 de Espressif. El primero cuenta con procesadores ARM Cortex-M con muchos periféricos como GPIO, comunicación o coprocesadores, y el

segundo son chips altamente integrados con interruptores de antena incorporados, RF balun, amplificador de potencia, amplificador de recepción de bajo ruido, filtros y módulos de gestión de potencia.

Por defecto, micro-ROS usa NuttX RTOS, pero también tiene puertos para FreeRTOS y Zephyr. Estos RTOS tienen una gran variedad de MCU y placas de desarrollo compatibles.

Entre las muchas placas que se pueden usar potencialmente debido a la variedad de RTOS compatibles, por el momento oficialmente solo se apoyan las cuatro placas enumeradas a continuación:

- Olimex LTD STM32-E407
- STM32L4 Discovery kit IoT
- Crazyflie 2.1 Drone
- ESP32-DevKitC

Estas son las plataformas de hardware que se utilizan para probar y desarrollar micro-ROS, con las que también se presentan accesorios a los que se refieren frecuentemente, como placas complementarias y sondas JTAG.

Para facilitar su uso, micro-ROS proporciona un ejemplo listo para usar para algunas placas de desarrollo. Estos ejemplos listos para usar tienen como objetivo mostrar las capacidades de micro-ROS y también son un punto de partida para desarrollar aplicaciones integradas de ROS 2. (Lange, 2018)

4.5. ANTECEDENTES

4.5.1. Nox - A House Wandering Robot

En este proyecto se desarrolla un robot autónomo capaz de navegar por espacios cerrados mientras genera un mapa empleando la tecnología SLAM (*Simultaneous localization and mapping*) y ROS. Robin Baran, autor del proyecto, emplea distintos elementos para la elaboración del robot, donde los que constituyen la base del proyecto son: (Baran, 2018)

Ilustración 9 NOX (Baran, 2018)

- Raspberry Pi 3 Model B
- Arduino Mega 2560
- Adafruit Motor/Stepper/Servo Shield para Arduino v2 Kit - v2.3
- Sensor Kinect de Microsoft
- Motores de 12V DC con Encoders
- Convertidor automático DC-DC Boost Buck step-up step-down ajustable: Módulo XL6009
- Convertidor DC-DC Buck 10A
- Batería LIPO de 11.1V

En este proyecto la Raspberry tiene instalado ROS en su interior y juega el papel del cerebro del robot. Recibiendo información del sensor Kinect y generando un mapa empleando SLAM, mediante una de las numerosas librerías de ROS. Posteriormente la Raspberry transmite la información al Arduino Mega, el cual comanda los motores que ponen en movimiento al robot. Además, la Raspberry está conectada a un ordenador mediante Wi-Fi, el cual controla el robot. El ordenador puede recibir el mapa del robot y la posición estimada, y puede enviar un punto de destino que el robot intentará alcanzar.

Nox funciona con una batería de iones de litio de 11,1 V y funciona con dos motores. El panel frontal se puede quitar para cambiar la batería. Un orificio ranurado y un tornillo lo sujetan y permiten colocar baterías de diferentes longitudes.

Los motores son dos motores de 12 V CC (107 rpm). Según el autor estos motores están bien para el proyecto, pero en realidad no es necesario que el robot

vaya tan rápido y hubiese sido mejor intercambiar parte de la velocidad por encoders más precisos.

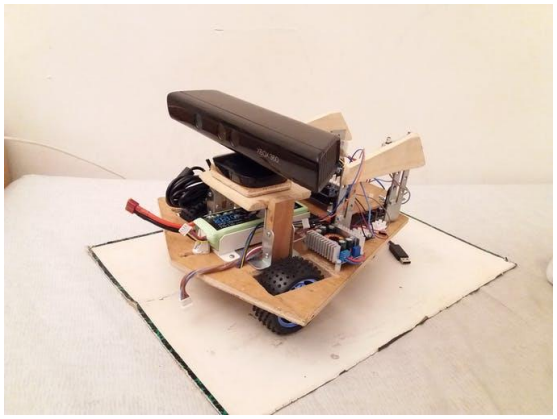


Ilustración 10 Estructura interna de NOX (Baran, 2018)

Sobre el diseño, la principal limitación es tener algo que se integre bien con Kinect, ya que se está construyendo el robot a su alrededor. De modo que el autor tomó inspiración en el aspecto triangular de muchos objetos de estilo moderno.

Las luces laterales sirven para indicar los estados del robot. Cuando el Arduino no está conectado al ROS Master (lo que indica que el programa del robot aún no se ha iniciado), las luces parpadean tres veces seguidas muy rápidamente. Al iniciar el movimiento, las luces parpadean de forma distinta dependiendo la velocidad de parpadeo de la velocidad del robot.

El robot es de conducción diferencial, por lo que los motores se colocan en el mismo eje. La base está hecha de madera con dos ruedas locas para apoyo. El resto de la estructura está hecha principalmente de soportes de madera y metal.

En el interior del robot, el controlador principal es una Raspberry Pi 3 B con Ubuntu y ROS. Se puede acceder a la Raspberry desde un ordenador externo a través de WiFi para dar órdenes al robot. El programa ROS lleva a cabo cálculos de odometría, planificación de navegación y cartografía utilizando los datos obtenidos por el Kinect. La Raspberry Pi envía el comando de velocidad a un Arduino que controla los dos motores con un PID a través de un Motor Shield de Adafruit. Éste lee el valor del encoder, calcula la velocidad de cada motor y envía el valor a la Raspberry para el cálculo de la odometría. El Arduino y la Raspberry Pi están conectados por USB y el programa del Arduino actúa como un nodo de ROS.

La placa de Arduino empleada en un Arduino Mega 2560. En un inicio se planeó usar un Arduino UNO básico, pero éste no tenía suficientes pines de interrupción para leer los valores del encoder. Además, la velocidad y la precisión eran limitadas. Posteriormente se intentó usar un Arduino Leonardo, pero el factor limitante era la memoria. Finalmente se escogió a un Arduino Mega 2560.

Se utilizan dos convertidores de DC para convertir el voltaje. El motor está funcionando directamente con el voltaje de la batería (más o menos 11.1V). La parte de control (que incluye Raspberry Pi, Arduino y encoders) funciona con 5V convertidos

de la batería. El Kinect necesita un voltaje estabilizado de 12V y un convertidor también lo proporciona desde la batería.

Por lo que respecta al mapeado se emplea SLAM, donde el algoritmo utilizado es gmapping. Como ya existe una buena implementación en ROS es esta la que se usa.

Con respecto a la planificación de la ruta, se probaron diferentes planificadores locales disponibles en ROS y se decidió emplear el planificador local de DWA, ya que da el mejor resultado para los robots de accionamiento diferencial. (Baran, 2012)

4.5.2. My personal robotic companion

Proyecto muy similar al visto anteriormente. Pues este proyecto es una de las inspiraciones del autor anterior. En este caso tenemos un robot que, de igual forma que el anterior, usa Kinect como sensor principal para la adquisición de datos del entorno para generar un mapa y navegar empleando SLAM, con la diferencia de que en este proyecto también se emplea un Smartphone con sistema Android, el cual aporta datos para así ayudar con la odometría. Además, se emplea un ordenador portátil como Master de ROS el cual va directamente sobre el robot mientras este se desplaza. Por lo que respecta a los materiales empleados, se ven similitudes con el proyecto anterior:

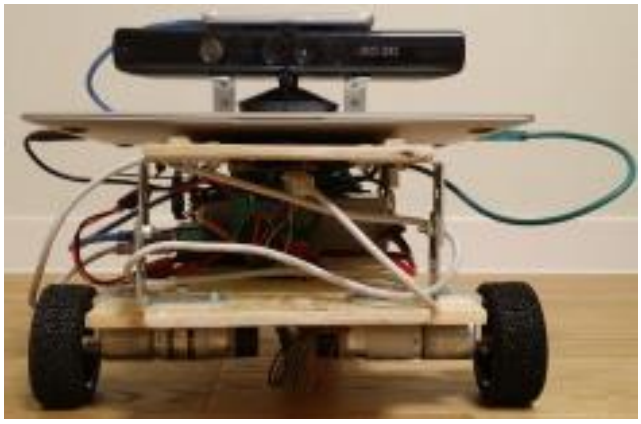


Ilustración 11 Prototipo de My Personal Robotic Companion (Jik Cha, 2015)

- Sensor Kinect de Microsoft
- Macbook Air 11 con Lubuntu 14.04 y ROS Indigo
- Galaxy S6 edge con Android
- Arduino Mega 2560
- Adafruit Motor Shield v2
- Batería LIPO de 11.1V
- Convertidor DC-DC step-up step-down
- Madera, tornillos, etc. Para el chasis del robot.

En este proyecto, el autor, Sung Jik Cha, no aporta tanta información sobre la elaboración del robot. Aun así, aporta información acerca de las librerías de ROS empleadas para implementar SLAM en el proyecto.

Para el mapeado se emplea el paquete de ROS gmapping, el cual utiliza localización y mapeo simultáneo (SLAM) para producir un mapa 2D a partir de datos de escaneo láser. Modificando los parámetros "linearUpdate", "angularUpdate" y "particles" se puede obtener un mapa razonablemente preciso.

Para la navegación se emplearon los paquetes amcl, base_local_planner y costmap_2d del conjunto de navegación de ROS.

Además de la información anterior el autor aporta un diagrama de bloques donde se puede apreciar cada elemento involucrado en el movimiento del robot: (Jik Cha, 2015)

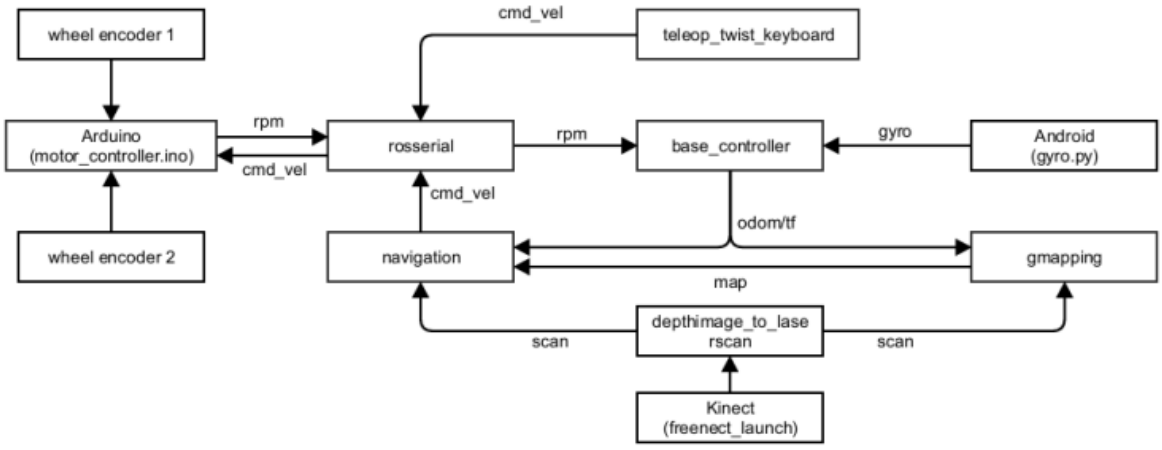


Ilustración 12 Diagrama de bloques (Jik Cha, 2015)

4.5.3. Shaky - A Wall Following and Obstacle Avoidance Robot

En este proyecto vemos un robot basado en Raspberry Pi, también conocido como Shaky, que puede detectar obstáculos y navegar alrededor de ellos. Está destinado a seguir las paredes. Utiliza tres sensores de distancia ultrasónicos y dos sensores de distancia infrarrojos. Un Arduino Uno gestiona la lectura del sensor, mientras que ROS maneja el sistema de control.

A diferencia de los proyectos antes vistos, se ha optado por emplear sensores menos avanzados que el Kinect de Microsoft para la adquisición de datos del entorno.

No obstante, el resto de materiales empleados sí muestra una similitud con el resto de proyectos:

- Raspberry Pi 3B
- Motores 6V DC con encoder: DG01D-A130
- DC & Stepper Motor Driver HAT Kit para Raspberry Pi
- Arduino UNO
- Sensor de ultrasonidos: HC-SR04
- Sensor de infrarrojos: GP2Y0A41SK0F
- Batería de polímero de Litio 2.4A a 5V DC

En este antecedente se usa un Duckiebot, un robot producido por Duckietown, una organización que produce robots con el fin de ser empleados de forma didáctica, de modo que el chasis es el original y se busca conseguir una adaptación para los sensores empleados.

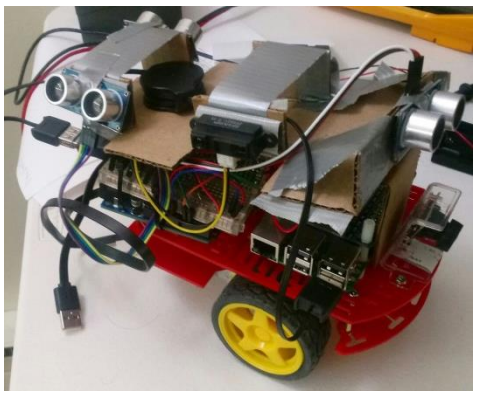


Ilustración 13 Prototipo de Shaky (Gurny, 2012)

En este caso no se emplea SLAM para la generación de un mapa, puesto que no se genera ningún tipo de mapa. El robot realiza comprobaciones con todos sus sensores para delimitar si hay un obstáculo en su camino, cuando los sensores detecten un obstáculo, el robot maniobrará para evadirlo. (Gurny, 2012)

4.5.4. TurtleBot3

TurtleBot es un robot de estándar de la plataforma ROS. Turtle se deriva del robot Turtle, que fue impulsado por el lenguaje educativo de programación informática Logo en 1967. Además, el nodo turtlesim, que aparece por primera vez en los tutoriales básicos de ROS, es un programa que imita el sistema de comando del programa turtle de Logo. TurtleBot, que se originó a partir de Turtle de Logo, está diseñado para enseñar fácilmente a las personas que son nuevas en ROS a través de TurtleBot, así como para enseñar el lenguaje de programación de ordenadores usando Logo.

Ilustración 14 Primer modelo de Turtlebot (TurtleBot, 2020)

Hay 3 versiones de la serie TurtleBot. TurtleBot1 que fue desarrollado por Tully (Gerente de Plataforma en Open Robotics) y Melonee (CEO de Fetch Robotics) de Willow Garage basándose en el robot de investigación de iRobot basado en Roomba, Create, para la implementación de ROS. Fue desarrollado en 2010 y ha estado a la venta desde 2011. En 2012, TurtleBot2 fue desarrollado por Yujin Robot basado en el robot de investigación, iClebo Kobuki. En 2017, TurtleBot3 se desarrolló con características para complementar las funciones deficientes de sus predecesores y las demandas de los usuarios. El TurtleBot3 adopta el actuador inteligente ROBOTIS DYNAMIXEL para la conducción.

Ilustración 15 Segundo modelo de Turtlebot y sus variantes (TurtleBot, 2020)

TurtleBot3 es un robot móvil pequeño, asequible, programable y basado en ROS para uso en educación, investigación, hobbies y creación de prototipos de productos. El objetivo de TurtleBot3 es reducir drásticamente el tamaño de la plataforma y bajar el precio sin tener que sacrificar su funcionalidad y calidad, mientras que al mismo tiempo ofrece capacidad de expansión. El TurtleBot3 se puede personalizar de varias formas dependiendo de cómo reconstruya las partes mecánicas y use partes opcionales como la computadora y el sensor. Además, TurtleBot3 ha evolucionado con un SBC (Single-Board Computer) rentable y de tamaño pequeño que es adecuado para un sistema integrado robusto, sensor de distancia de 360 grados y tecnología de impresión 3D.

La tecnología central de TurtleBot3 es SLAM, navegación y manipulación, lo que lo hace adecuado para robots de servicio doméstico. El TurtleBot puede ejecutar algoritmos SLAM (localización y mapeo simultáneos) para construir un mapa y puede conducir por una habitación. Además, se puede controlar de forma remota desde un ordenador portátil, un joypad o un Smartphone con Android. Además, el TurtleBot3 se puede utilizar como un manipulador móvil capaz de manipular un objeto adjuntando un manipulador como OpenMANIPULATOR. El OpenMANIPULATOR tiene la ventaja de ser compatible con TurtleBot3 Waffle y Waffle Pi. A través de esta compatibilidad se puede compensar la falta de libertad y se puede tener una mayor completitud como robot de servicio con el SLAM y las capacidades de navegación que tiene el TurtleBot3.

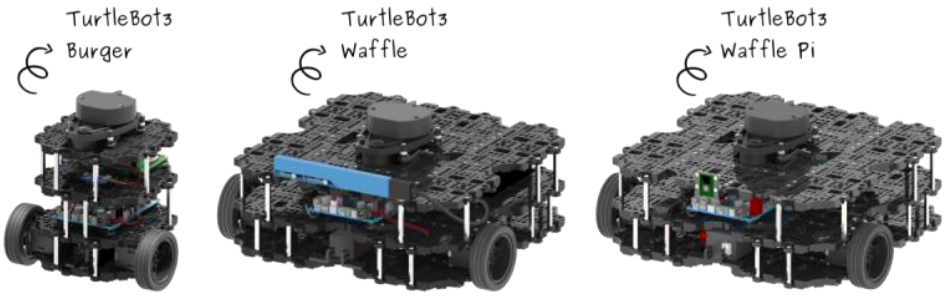


Ilustración 16 Tercer modelo de Turtlebot y sus variantes (Turtlebot3, 2018)

El turtlebot3 emplea distintos componentes en función del modelo a construir. En la siguiente lista se pueden ver los componentes principales de cada modelo: (TurtleBot3, 2018)

SBC

- Raspberry Pi 3 Modelo B (TurtleBot3 Burger, Waffle Pi)
- Raspberry Pi 3 Modelo B + (TurtleBot3 Burger, Waffle Pi)
- Intel® Joule ™ 570x (TurtleBot3 Waffle)

Sensores

- Sensor de distancia láser 360º LDS-01 (TurtleBot3 Burger, Waffle, Waffle Pi)
- Intel® Realsense ™ R200 (TurtleBot3 Waffle)
- Módulo de cámara Raspberry Pi v2.1 (TurtleBot3 Waffle Pi)

Placa integrada

- OpenCR1.0 (TurtleBot3 Burger, Waffle, Waffle Pi)

Actuadores

- XL430 (TurtleBot3 Burger)
- XM430 (TurtleBot3 Waffle, Waffle Pi)

5. MARCO TEÓRICO

5.1. ROS

5.1.1. Conceptos

ROS funciona empleando un modelo de publicación/suscripción donde diferentes nodos pueden comunicarse entre ellos. Para poder explicar con claridad el funcionamiento de este sistema es necesario tener claros una serie de conceptos los cuales juegan un papel de gran importancia en dicho sistema.

ROS tiene tres niveles de conceptos: el nivel de sistema de archivos, el nivel de gráfico de computación y el nivel de la comunidad. Además de los tres niveles de conceptos, ROS también define dos tipos de *nombres*. Nombres de recursos de paquetes y Nombres de recursos de gráficos. (Aaron, 2014)

5.1.1.1. Nivel de Sistema de Archivos

Los conceptos de nivel de sistema de archivos cubren principalmente los recursos ROS que se encuentran en el disco. (Aaron, 2014)

5.1.1.1.1. Paquetes

Los paquetes son la unidad principal para organizar el software en ROS. Un paquete puede contener procesos en tiempo de ejecución de ROS (nodos), una biblioteca dependiente de ROS, conjuntos de datos, archivos de configuración o cualquier otra cosa que se organice de manera útil en conjunto. Los paquetes son el elemento de construcción y lanzamiento más atómico en ROS. Lo que significa que lo más granular que puede crear y lanzar es un paquete. (Saito, 2019)

5.1.1.1.2. Metapaquetes

Los metapaquetes son paquetes especializados que solo sirven para representar un grupo de otros paquetes relacionados. Por lo general, los metapaquetes se utilizan como un marcador de posición compatible con versiones anteriores de ROS. (Weaver, 2014)

5.1.1.1.3. Manifiestos de paquetes

Los manifiestos proporcionan metadatos sobre un paquete, incluido su nombre, versión, descripción, información de licencia, dependencias y otra metainformación como paquetes exportados. (Belanger, 2019)

5.1.1.1.4. Repositorios

Una colección de paquetes que comparten un sistema VCS común. Los paquetes que comparten un VCS comparten la misma versión y se pueden lanzar juntos usando la herramienta de automatización de liberación de catkin, bloom. Los repositorios también pueden contener un solo paquete. (Aaron, 2014)

5.1.1.1.5. Tipos de mensajes (msg)

Las descripciones de los mensajes definen las estructuras de datos para los mensajes enviados en ROS. (Aaron, 2014)

ROS utiliza un lenguaje de descripción de mensajes simplificado para describir los valores de datos (también conocidos como mensajes) que publican los nodos de ROS. Esta descripción facilita que las herramientas de ROS generen automáticamente el código fuente para el tipo de mensaje en varios idiomas de destino.

Hay dos partes en un archivo .msg: campos y constantes. Los campos son los datos que se envían dentro del mensaje. Las constantes definen valores útiles que se pueden usar para interpretar esos campos. (Hendrix, 2019)

Ejemplo:

int32 x

int32 y

5.1.1.1.6. Tipos de servicios (srv)

Las descripciones de servicios definen las estructuras de datos de solicitud y respuesta para los servicios en ROS. (Aaron, 2014)

ROS utiliza un lenguaje de descripción de servicios simplificado ("srv") para describir los tipos de servicios de ROS. Esto se basa directamente en el formato ROS msg para permitir la comunicación de solicitud/respuesta entre nodos.

Un archivo de tipo .srv consta de una solicitud y un tipo de mensaje de respuesta, separados por '---'. Dos archivos .msg cualquiera concatenados junto con '---' son una descripción del servicio legal. (Saito, 2017)

Ejemplo:

string str

---

string str

5.1.1.2. Nivel de gráfico de computación (Computation Graph Level)

El gráfico de computación *(Computation Graph)* es la red P2P (peer-to-peer) de procesos de ROS que procesan datos juntos. Los conceptos básicos de gráficos de cálculo de ROS son nodos, master, servidor de parámetros, mensajes, servicios, temas y bolsas, donde todos ellos proporcionan datos al gráfico de diferentes maneras. (Aaron, 2014)

5.1.1.2.1. Nodos

Los nodos son procesos que realizan cálculos. ROS está diseñado para ser modular a una escala fina; un sistema de control de un robot generalmente comprende muchos nodos. Por ejemplo, un nodo controla un telémetro láser, un nodo controla los motores de las ruedas, un nodo realiza la localización, un nodo realiza la planificación de la ruta, un nodo proporciona una vista gráfica del sistema, y así sucesivamente. Un nodo de ROS se escribe con el uso de una biblioteca del cliente de ROS, como roscpp o rospy. (Oladepo, 2018)

5.1.1.2.2. Master

El ROS Master proporciona servicios de nomenclatura y registro al resto de los nodos del sistema ROS. Sin el Master, los nodos no podrían encontrarse, intercambiar mensajes o invocar servicios. (Aaron, 2014)

Además, realiza un seguimiento de los publicadores y suscriptores de temas y servicios. La función del Master es permitir que los nodos ROS individuales se ubiquen entre sí. Una vez que estos nodos se han ubicado entre sí, se comunican entre sí de igual a igual.

El Master también proporciona el servidor de parámetros.

El Master se ejecuta normalmente con el comando "roscore", que carga el ROS Master junto con otros componentes esenciales. (Wu, 2018)

5.1.1.2.3. Servidor de parámetros

El servidor de parámetros permite que los datos se almacenen por clave en una ubicación central. Actualmente forma parte del Master. (Aaron, 2014)

Un servidor de parámetros es un diccionario multivariable compartido al que se puede acceder a través de las APIs de red. Los nodos utilizan este servidor para almacenar y recuperar parámetros durante ejecución. Está destinado a ser visible globalmente para que las herramientas puedan inspeccionar fácilmente el estado de configuración del sistema y modificarlo si es necesario. (Miller, 2018)

5.1.1.2.4. Mensajes

Los nodos se comunican entre sí pasando mensajes. Un mensaje es simplemente una estructura de datos, que comprende campos escritos. Se admiten los tipos primitivos estándar (entero, punto flotante, booleano, etc.), al igual que las matrices de tipos primitivos. Los mensajes pueden incluir estructuras y matrices anidadas arbitrariamente (al igual que las estructuras de C). (Kurzaj, 2016)

5.1.1.2.5. Tópicos

Los mensajes se enrutan a través de un sistema de transporte con semántica de publicación/suscripción. Un nodo envía un mensaje publicándolo en un tópico determinado. El tópico es un nombre que se utiliza para identificar el contenido del mensaje. Un nodo que esté interesado en cierto tipo de datos se suscribirá al tema correspondiente. Puede haber múltiples publicadores y suscriptores concurrentes para un solo tópico, y un solo nodo puede publicar y/o suscribirse a múltiples tópicos. En general, los publicadores y suscriptores no conocen la existencia de los demás. La idea es desvincular la producción de información de su consumo. Lógicamente, uno puede pensar en un tópico como un bus de mensajes fuertemente tipado. Cada bus tiene un nombre, y cualquiera puede conectarse al bus para enviar o recibir mensajes siempre que sean del tipo correcto. (Foote, 2019)

5.1.1.2.6. Servicios

El modelo de publicación/suscripción es un paradigma de comunicación muy flexible, pero su transporte unidireccional de muchos a muchos no es apropiado para las interacciones de solicitud/respuesta, que a menudo se requieren en un sistema distribuido. La solicitud/respuesta se realiza a través de servicios, que están definidos por un par de estructuras de mensajes: una para la solicitud y otra para la respuesta. Un nodo proveedor ofrece un servicio con un nombre y un cliente utiliza el servicio enviando el mensaje de solicitud y esperando la respuesta. Las bibliotecas del cliente de ROS generalmente presentan esta interacción al programador como si fuera una llamada a un procedimiento remoto. (Koubaa, 2019)

5.1.1.2.7. Bolsas

Las bolsas son un formato para guardar y reproducir datos de mensajes de ROS. Las bolsas son un mecanismo importante para almacenar datos, como los datos de los sensores, que pueden ser difíciles de recopilar, pero son necesarios para desarrollar y probar algoritmos. (Staples, 2020)

5.1.1.3. Comunicación entre los nodos y el master

El ROS Master actúa como un servicio de nombres en el gráfico de computación de ROS. Almacena información de registro de temas y servicios para los nodos de ROS. Los nodos se comunican con el Master para reportar su información de registro. A medida que estos nodos se comunican con el master, pueden recibir información sobre otros nodos registrados y realizar conexiones según corresponda. El master también hará devoluciones de llamada a estos nodos cuando cambie esta información de registro, lo que permite a los nodos crear conexiones de forma dinámica a medida que se ejecutan nuevos nodos. (Aaron, 2014)

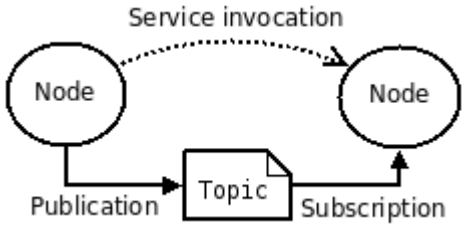


Ilustración 17 Esquema simple de comunicación de ROS (Sin master) (Aaron, 2014)

Los nodos se conectan directamente a otros nodos; el master solo proporciona información de búsqueda, al igual que un servidor DNS. Los nodos que se suscriben a un tema solicitarán conexiones de los nodos que publican ese tema y establecerán esa conexión a través de un protocolo de conexión acordado. El protocolo más común utilizado en ROS se llama TCPROS, que utiliza sockets TCP/IP estándar.

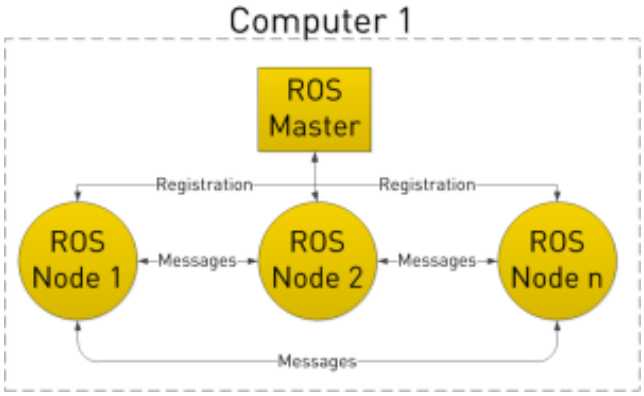


Ilustración 18 Esquema simple de comunicación de ROS (Con master) (Baranov, 2014)

Esta arquitectura permite una operación desacoplada, donde los nombres son el medio principal por el cual se pueden construir sistemas más grandes y complejos. Los nombres tienen un papel muy importante en ROS: los nodos, temas, servicios y parámetros tienen todos nombres. Cada biblioteca del cliente de ROS admite la

reasignación de nombres en la línea de comandos, lo que significa que un programa compilado se puede reconfigurar en tiempo de ejecución para operar en una topología de gráfico de computación diferente. (Baranov, 2014)

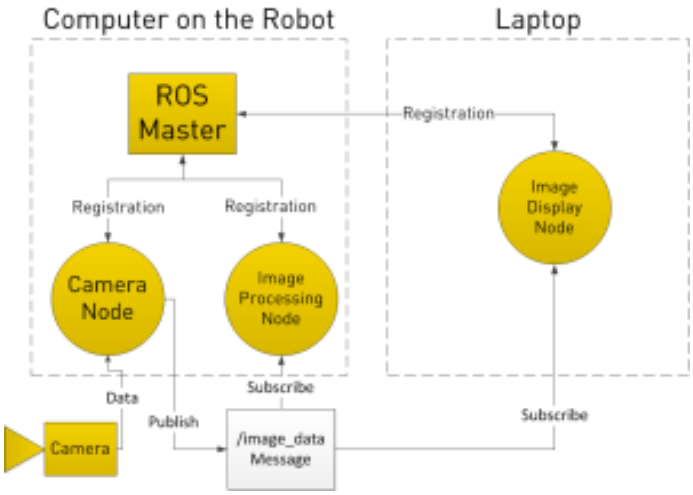


Ilustración 19 Ejemplo de grafico de computación de ROS (Baranov, 2014)

5.1.1.4. Nivel de comunidad

Los conceptos del nivel de comunidad de ROS son recursos de ROS que permiten a comunidades separadas intercambiar software y conocimiento. (Aaron, 2014)

5.1.1.4.1. Distribuciones

Las distribuciones de ROS son colecciones de pilas versionadas que pueden ser instaladas. Las distribuciones juegan un papel similar a las distribuciones de Linux: facilitan la instalación de una colección de software y también mantienen versiones consistentes en un conjunto de software. (Staples, 2020)

5.1.1.4.2. Repositorios

ROS se basa en una red federada de repositorios de código, donde diferentes instituciones pueden desarrollar y lanzar sus propios componentes de software para robots. En lugar de tener un lugar real para todos los paquetes ROS, se anima a los usuarios y desarrolladores de todo el mundo a alojar sus propios repositorios de paquetes de ROS. Cada repositorio puede ser administrado y licenciado según lo desee el respectivo mantenedor, y el mantenedor retiene la propiedad directa y el control sobre el código. (Foote, 2014)

5.1.1.4.3. Wiki de ROS

La Wiki de la comunidad de ROS es el foro principal para documentar información sobre ROS. Cualquiera puede registrarse para obtener una cuenta y

contribuir con su propia documentación, proporcionar correcciones o actualizaciones, escribir tutoriales y más. (Aaron, 2014)

5.1.1.5. Nombres

5.1.1.5.1. Nombres de recursos de gráficos

Los nombres de recursos de gráficos proporcionan una estructura de nomenclatura jerárquica que se utiliza para todos los recursos en un gráfico de computación de ROS, como nodos, parámetros, temas y servicios. Estos nombres son muy poderosos en ROS y fundamentales para la forma en que se componen los sistemas más grandes y complicados en ROS, por lo que es fundamental comprender cómo funcionan estos nombres y cómo pueden ser manipulados.

Los nombres de recursos de gráficos son un mecanismo importante en ROS para proporcionar encapsulación. Cada recurso se define dentro de un espacio de nombres, que puede compartir con muchos otros recursos. En general, los recursos pueden crear recursos dentro de su espacio de nombres y pueden acceder a recursos dentro o por encima de su propio espacio de nombres. Se pueden realizar conexiones entre recursos en distintos espacios de nombres, pero esto generalmente se hace mediante el código de integración por encima de ambos espacios de nombres. Esta encapsulación aísla diferentes partes del sistema de la captura accidental del recurso con nombre incorrecto o del secuestro global de nombres.

Los nombres se resuelven relativamente, por lo que los recursos no necesitan saber en qué espacio de nombres se encuentran. Esto simplifica la programación, ya que los nodos que trabajan juntos se pueden escribir como si todos estuvieran en el espacio de nombres de nivel superior. Cuando estos nodos se integran en un sistema más grande, se pueden colocar en un espacio de nombres que define su colección de código. (Aaron, 2014)

Nodo	Relativo (por defecto)	Global	Privado
/node1	bar -> /bar	/bar -> /bar	~bar -> /node1/bar
/wg/node2	bar -> /wg/bar	/bar -> /bar	~bar -> /wg/node2/bar
/wg/node3	foo/bar -> /wg/foo/bar	/foo/bar -> /foo/bar	~foo/bar -> /wg/node3/foo/bar

Tabla 1 Ejemplos de resolución de nombres

5.1.1.5.2. Nombres de recursos de paquetes

Los nombres de recursos de paquetes se utilizan en ROS con conceptos de nivel de sistema de archivos para simplificar el proceso de referencia a archivos y tipos de datos en el disco. Los nombres de recursos del paquete son muy simples: son solo el nombre del paquete en el que se encuentra el recurso más el nombre del recurso. Por ejemplo, el nombre "std_msgs/String" se refiere al tipo de mensaje "String" en el paquete "std_msgs".

Los nombres de recursos de paquetes son muy similares a las rutas de archivo, excepto que son mucho más cortos. Esto se debe a la capacidad de ROS para localizar paquetes en el disco y realizar suposiciones adicionales sobre su contenido. Por ejemplo, las descripciones de los mensajes siempre se almacenan en el subdirectorio msg y tienen la extensión .msg, por lo que std_msgs/String es la abreviatura de ruta/a/std_msgs/msg/String.msg. De manera similar, el tipo de nodo foo/bar es equivalente a buscar un archivo llamado bar en el paquete foo con permisos ejecutables.

Los nombres de recursos de paquetes tienen reglas de nomenclatura estrictas, ya que a menudo se usan en código generado automáticamente. Por esta razón, un paquete de ROS no puede tener caracteres especiales que no sean un guion bajo y deben comenzar con un carácter alfabético. Un nombre válido tiene las siguientes características: (Aaron, 2014)

- El primer carácter es un carácter alfabético ([a-z | A-Z])
- Los caracteres siguientes pueden ser alfanuméricos ([0-9 | a-z | A-Z]), guiones bajos (_) o una barra diagonal (/)
- Hay como máximo una barra diagonal ('/').

5.2. ROS 2

ROS 2 funciona de manera similar a ROS 1. Tal y como se ha mencionado en el estado del arte, ROS 2 intenta mejorar algunas de las características débiles de ROS, mientras que mantiene las que son buenas. Debido a esto, gran parte de los conceptos antes mencionados se mantienen sin cambios, donde los nodos, mensajes y tópicos no se ven afectados.

La principal diferencia entre ROS y ROS 2 es la ausencia del Master en el último. ROS2 utiliza un DDS (Servicio de distribución de datos) para publicar y suscribirse en lugar de un controlador de mensajes personalizado.

ROS 2 admite múltiples implementaciones DDS/RTPS. Hay muchos factores que puede considerar al elegir una implementación de middleware: consideraciones logísticas como la licencia o consideraciones técnicas como la disponibilidad de la plataforma o la capacidad de cálculo. Los proveedores pueden proporcionar más de una implementación de DDS o RTPS dirigida a satisfacer diferentes necesidades.

Para utilizar una implementación DDS/RTPS con ROS 2, es necesario crear un paquete de "ROS Middleware Interface" (también conocido como interfaz rmw o simplemente "rmw") que implemente la interfaz abstracta del middleware ROS utilizando la API y las herramientas de implementación DDS o RTPS. (ROS2, 2018)

Una de las características principales de ROS2 es la gestión del ciclo de vida de los nodos. Esta característica permite un mayor control sobre el estado del sistema ROS. De hecho, permite que roslaunch se asegure de que todos los componentes se hayan instanciado correctamente antes de permitir que cualquier componente comience a ejecutar su comportamiento. También permite que los nodos se reinicien o reemplacen en línea. (ROS2, 2016)

Para conectar dos nodos en ROS 2 se introduce el concepto de "discovery":

El descubrimiento de nodos ocurre automáticamente a través del middleware subyacente de ROS 2. Se puede resumir de la siguiente manera:

- Cuando se inicia un nodo, anuncia su presencia a otros nodos en la red con el mismo dominio ROS (configurado con la variable de entorno ROS_DOMAIN_ID). Los nodos responden a este anuncio con información sobre ellos mismos para que se puedan realizar las conexiones adecuadas y los nodos puedan comunicarse.

- Los nodos anuncian periódicamente su presencia para que se puedan establecer conexiones con entidades recién encontradas, incluso después del período de descubrimiento inicial.
- Los nodos se anuncian a otros nodos cuando se desconectan.

Los nodos solo establecerán conexiones con otros nodos si tienen una configuración de calidad de servicio compatible. (ROS2, 2016)

5.2.1. Configuración de calidad de servicio

ROS 2 ofrece una amplia variedad de políticas de calidad de servicio (QoS) que le permiten ajustar la comunicación entre nodos. Con el conjunto correcto de políticas de calidad de servicio, ROS 2 puede ser tan confiable como TCP o tan eficaz como UDP, con muchos estados posibles en el medio. A diferencia de ROS 1, que principalmente solo admitía TCP, ROS 2 se beneficia de la flexibilidad del transporte DDS subyacente en entornos con redes inalámbricas con pérdidas donde una política de "mejor esfuerzo" sería más adecuada, o en sistemas informáticos en tiempo real donde la calidad adecuada del perfil de servicio es necesaria para cumplir con los plazos.

Un conjunto de "políticas" de QoS se combinan para formar un "perfil" de QoS. Dada la complejidad de elegir las políticas de QoS correctas para un escenario dado, ROS 2 proporciona un conjunto de perfiles de QoS predefinidos para casos de uso comunes (por ejemplo, datos de sensores). Al mismo tiempo, los desarrolladores tienen la flexibilidad de controlar políticas específicas de los perfiles de QoS.

Se pueden especificar perfiles de QoS para publicadores, suscripciones, servidores de servicios y clientes. Se puede aplicar un perfil de QoS de forma independiente a cada instancia de las entidades antes mencionadas, pero si se utilizan perfiles diferentes, es posible que no se conecten. (ROS2, 2020)

5.2.1.1. Políticas de calidad de servicio

Para cada una de las políticas que no es una duración, también existe la opción de "sistema predeterminado", que utiliza el valor predeterminado del middleware subyacente. Para cada una de las políticas que es una duración, también existe una opción "predeterminada" que significa que la duración no está especificada, que el middleware subyacente normalmente interpretará como una duración infinitamente larga. (ROS2, 2020)

El perfil de QoS base actualmente incluye configuraciones para las siguientes políticas:

5.2.1.1.1. Historial

- Mantener último: solo almacena hasta N muestras, configurable a través de la opción de profundidad de cola.
- Conservar todo: almacena todas las muestras, sujeto a los límites de recursos configurados del middleware subyacente. (ROS2, 2020)

5.2.1.1.2. Profundidad

- Tamaño de la cola: solo se respeta si la política de "historial" se estableció en "mantener el último". (ROS2, 2020)

5.2.1.1.3. Fiabilidad

- Mejor esfuerzo: intenta entregar muestras, pero puede perderlas si la red no es sólida.
- Confiable: garantía de que las muestras se entregan, puede volver a intentarlo varias veces. (ROS2, 2020)

5.2.1.1.4. Durabilidad

- Local transitorio: el editor se hace responsable de conservar las muestras para las suscripciones de "unión tardía".
- Volátil: no se intenta conservar muestras. (ROS2, 2020)

5.2.1.1.5. Fecha límite

- Duración: la cantidad de tiempo máxima esperada entre los mensajes posteriores que se publican en un tema (ROS2, 2020)

5.2.1.1.6. Esperanza de vida

- Duración: la cantidad máxima de tiempo entre la publicación y la recepción de un mensaje sin que el mensaje se considere obsoleto o caducado (los mensajes caducados se eliminan silenciosamente y, de forma efectiva, nunca se reciben). (ROS2, 2020)

5.2.1.1.7. Vitalidad

- Automático: el sistema considerará que todos los publicadores del nodo están activos durante otro "período de arrendamiento" cuando alguno de sus editores haya publicado un mensaje.
- Manual por tema: el sistema considerará que el publicador está vivo durante otro "período de arrendamiento" si afirma manualmente que todavía está vivo (mediante una llamada a la API del publicador). (ROS2, 2020)

5.2.1.1.8. Duración del arrendamiento

- Duración: el período máximo de tiempo que un publicador tiene para indicar que está vivo antes de que el sistema considere que ha perdido vitalidad (perder vitalidad podría ser una indicación de un fallo). (ROS2, 2020)

5.2.1.2. Perfiles de calidad de servicio

Los perfiles permiten a los desarrolladores concentrarse en sus aplicaciones sin preocuparse por cada configuración de QoS posible. Un perfil de QoS define un conjunto de políticas que se espera que funcionen bien juntas para un caso de uso particular. (ROS2, 2020)

5.2.1.2.1. Configuración de QoS predeterminada para publicadores y suscripciones

Para facilitar la transición de ROS 1 a ROS 2, es deseable ejercer un comportamiento de red similar. De forma predeterminada, los publicadores y las suscripciones en ROS 2 tienen "mantener el último" para el historial con un tamaño de cola de 10, "confiable" para la fiabilidad, "volátil" para la durabilidad y "predeterminado del sistema" para la vitalidad. La fecha límite, la vida útil y la duración del arrendamiento también están configuradas como "predeterminados". (ROS2, 2020)

5.2.1.2.2. Servicios

Al igual que los publicadores y las suscripciones, los servicios son fiables. Es especialmente importante que los servicios utilicen una durabilidad volátil, ya que, de lo contrario, los servidores de servicio que se reinician pueden recibir solicitudes

desactualizadas. Si bien el cliente está protegido de recibir múltiples respuestas, el servidor no está protegido de los efectos secundarios de recibir solicitudes desactualizadas. (ROS2, 2020)

5.2.1.2.3. Datos del sensor

En el caso de los datos de los sensores, en la mayoría de los casos es más importante recibir las lecturas de manera oportuna, en lugar de asegurarse de que lleguen todas. Es decir, los desarrolladores quieren las últimas muestras tan pronto como se capturan, a costa de tal vez perder algunas. Por esa razón, el perfil de datos del sensor utiliza mejor esfuerzo y un tamaño de cola más pequeño. (ROS2, 2020)

5.2.1.2.4. Parámetros

Los parámetros en ROS 2 se basan en servicios y, como tales, tienen un perfil similar. La diferencia es que los parámetros utilizan una profundidad de cola mucho mayor para que las solicitudes no se pierdan cuando, por ejemplo, el cliente de parámetros no puede llegar al servidor de servicio de parámetros. (ROS2, 2020)

5.2.1.2.5. Sistema por defecto

Esto utiliza los valores predeterminados de la implementación de RMW para todas las políticas. Las diferentes implementaciones de RMW pueden tener diferentes valores predeterminados. (ROS2, 2020)

5.2.2. Interfaces de ROS 2

Las aplicaciones de ROS normalmente se comunican a través de interfaces de uno de tres tipos: mensajes, servicios y acciones. ROS 2 utiliza un lenguaje de descripción simplificado, el lenguaje de definición de interfaz (IDL), para describir estas interfaces. Esta descripción facilita que las herramientas de ROS generen automáticamente código fuente para el tipo de interfaz en varios idiomas de destino. (ROS2, 2019)

Tanto los mensajes como los servicios mantienen el mismo formato que en ROS. Por otro lado, las acciones son introducidas en ROS2 y se definen como:

- Acción: los archivos .action describen acciones. Se componen de tres partes: una meta, un resultado y una realimentación. Cada parte es una declaración de mensaje en sí misma. (ROS2, 2019)

El IDL de ROS 2 está estrechamente relacionado con el IDL de ROS 1. La mayoría de los archivos .msg y .srv de ROS 1 existentes deberían poder usarse tal cual con ROS 2. Además del conjunto de características existente de ROS 1, el IDL de ROS 2 introduce algunas características nuevas:

- **Matrices limitadas:** mientras que el IDL de ROS 1 permite matrices ilimitadas (p. ej., int32 [] foo) y matrices de tamaño fijo (p. ej., int32 [5] bar), el IDL de ROS 2 permite además matrices acotadas (p. ej., int32 [<= 5] bat). Hay casos de uso en los que es importante poder colocar un límite superior en el tamaño de una matriz sin comprometerse a usar siempre tanto espacio.
- **Cadenas limitadas:** Mientras que el IDL de ROS 1 permite strings ilimitadas (por ejemplo, string foo), el IDL de ROS 2 permite además strings limitadas (por ejemplo, string <= 5 bar).
- **Valores predeterminados:** mientras que el IDL de ROS 1 permite campos constantes (por ejemplo, int32 X = 123), el IDL de ROS 2 permite además especificar valores predeterminados (por ejemplo, int32 X 123). (ROS2, 2019)

5.2.3. Librerías cliente de ROS 2

Las librerías cliente son las API que permiten a los usuarios implementar su código en ROS. Son lo que los usuarios utilizan para acceder a conceptos ROS como nodos, temas, servicios, etc. Las librerías cliente vienen en una variedad de lenguajes de programación para que los usuarios puedan escribir código ROS en el lenguaje que mejor se adapte a su aplicación. En ROS 1 estas librerías son los paquetes, presentes en el nivel de sistema de archivos.

En ROS 1, todas las librerías cliente se desarrollan desde cero. Esto permite que la librería cliente de ROS 1 "Python" se implemente exclusivamente en lenguaje Python, por ejemplo, lo que brinda beneficios como no necesitar compilar código. Sin embargo, las convenciones de nomenclatura y los comportamientos no siempre son consistentes entre las librerías cliente o paquetes, correcciones de errores deben realizarse en varios lugares y hay muchas funciones que solo se han implementado en un paquete. (ROS2, 2016)

5.2.4. Parámetros en ROS 2

Los parámetros de ROS están asociados con los nodos. Los parámetros se utilizan para configurar nodos externamente en tiempo de ejecución (y durante el

tiempo de ejecución). La vida útil de un parámetro está vinculada a la vida útil del nodo (aunque el nodo podría implementar algún tipo de persistencia para recargar valores después del reinicio).

Los parámetros se tratan por nombre de nodo, espacio de nombre de nodo, nombre de parámetro y espacio de nombre de parámetro. Proporcionar un espacio de nombre de parámetro es opcional.

Cada parámetro consta de una clave y un valor, donde la clave es una cadena y el valor podría ser uno de los siguientes tipos: bool, int64, float64, string, byte[], bool[], int64[], float64[] o string[]. (ROS2, 2020)

YAML se usa para escribir archivos de parámetros tanto en ROS 1 como en ROS 2. La principal diferencia en ROS 2 es que los nombres de nodo deben usarse para direccionar los parámetros. Además del nombre de nodo completo, se usa la clave "ros__parameters" para señalar el inicio de los parámetros del nodo. (ROS2, 2020)

5.3. MICRO-ROS

Tal y como se ha mencionado en el estado del arte, ROS 2 y Micro-ROS son bastante similares, donde los principales cambios en comparación con ROS 2 "estándar" es que micro-ROS utiliza un sistema operativo en tiempo real (RTOS) en lugar de Linux, y DDS para entornos extremadamente restringidos de recursos (DDS-XRCE) en lugar de DDS clásico. Lo cual se resume en la siguiente imagen:

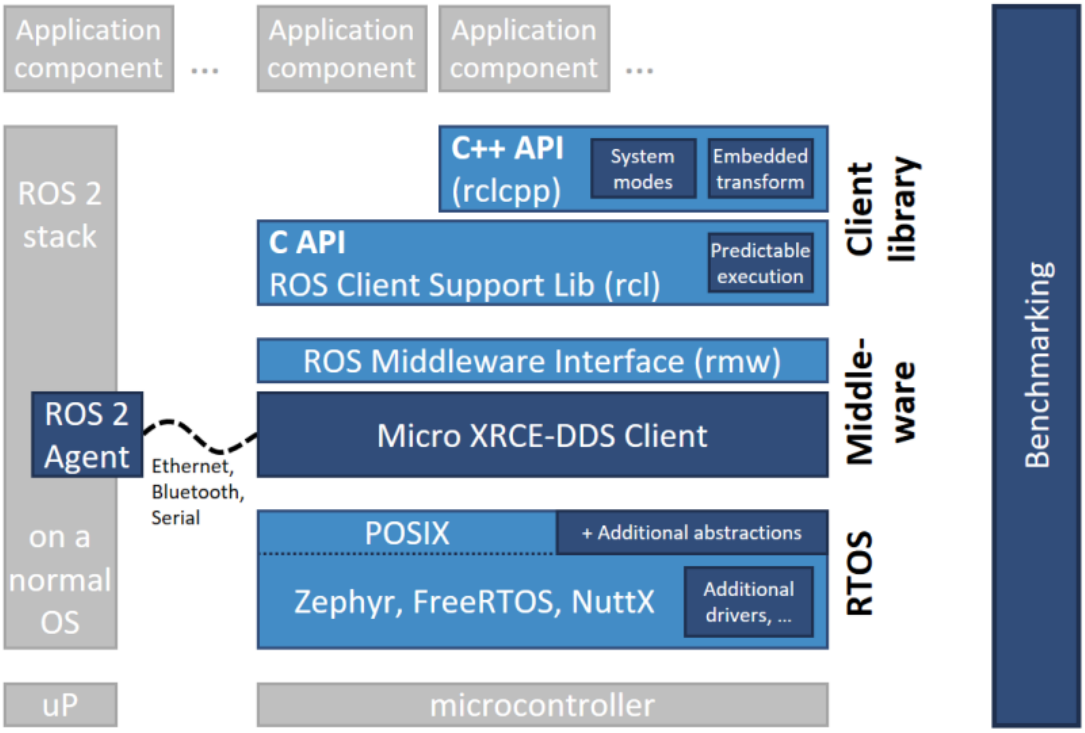


Ilustración 20 Esquema resumen de Micro-ROS (Micro-ROS, 2020)

Donde los componentes de color azul oscuro están desarrollados específicamente para Micro-ROS. Los componentes de color azul claro se toman de la pila ROS 2 estándar. (Lange, 2018)

Micro-ROS utiliza el DDS estándar de recursos optimizados para entornos con restricciones extremas de recursos (DDS-XRCE), implementado por el Micro-XRCE-DDS de eProsima.

Este DDS admite una gran variedad de protocolos de transporte y enlace de datos. Sin embargo, el soporte depende del RTOS subyacente, como se especifica en la siguiente tabla: (Lange, 2020)

	Serial UART	Serial USB	UDP por Ethernet	UDP por IEEE 802.11	6LoWPAN	Bluetooth
FreeRTOS	✓	WIP	✓			✓
NuttX	✓	✓	✓	✓	✓	
Zephyr	✓	✓	En progreso	En progreso		

Tabla 2 Protocolos con soporte en función del RTOS subyacente

5.3.1. Micro XRCE-DDS

El Micro XRCE-DDS de eProsima es un protocolo de código abierto que implementa el estándar OMG DDS (Object Management Group Data Distribution Service) para entornos extremadamente restringidos (DDS-XRCE, e**X**tremely **R**esource **C**onstrained **E**nvironment). El objetivo del protocolo DDS-XRCE es proporcionar acceso al espacio global de datos (Global-Data-Space) de DDS desde dispositivos con recursos limitados. Esto se logra gracias a una arquitectura cliente-servidor, donde los dispositivos de bajos recursos, llamados clientes XRCE o XRCE Clients, están conectados a un servidor, llamado agente XRCE o XRCE Agent, que actúa en nombre de sus clientes en el espacio global de datos de DDS. (Lange, 2020) (Lange, 2019)

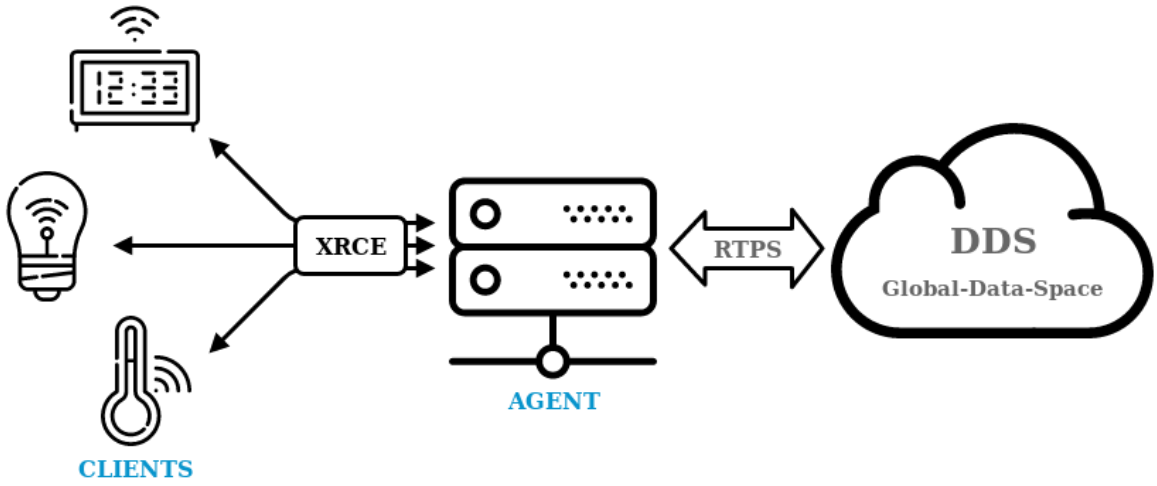


Ilustración 21 Esquema resumen del Micro XRCE-DDS (Lange, 2020)

El Micro XRCE-DDS está compuesto por dos elementos principales:

- Agente Micro XRCE-DDS: una aplicación C++11 lista para usar que implementa la funcionalidad del Agente XRCE.

- Cliente Micro XRCE-DDS: una librería C99 que implementa la funcionalidad del lado del cliente XRCE.

Este protocolo especifica cómo interactúan los clientes XRCE con el agente XRCE. Básicamente, hay dos formas de interacción entre ellos:

- Los clientes XRCE pueden manejar entidades proxy en el agente. Estas entidades proxy son las que actúan en la red DDS. Esto permite que los Clientes operen como un sistema sin estado, ya que el estado reside en el Agente. Esto, entre otras cosas, permitirá que los ciclos de suspensión y los dispositivos con recursos limitados se comuniquen con DDS.
- Los Clientes XRCE pueden publicar/suscribirse a/desde cualquier tópico DDS como si fueran participantes habituales de DDS. Esto hace que la separación entre dispositivos con recursos limitados y redes DDS sea difusa, para que puedan participar como entidades reales en el mundo DDS.

En pocas palabras, DDS-XRCE proporciona al cliente XRCE un patrón de mensajería de publicación/suscripción simple con el que comunicarse con las redes DDS existentes. (Lange, 2020)

Aparte de estos, el Micro XRCE-DDS utiliza otros dos componentes:

- Micro CDR: Un motor de deserialización utilizado en la librería cliente.
- Micro XRCE-DDS Gen: Una herramienta generadora de código utilizada para generar la función de deserialización de Micro CDR y ejemplos de aplicaciones de cliente a partir de fuentes IDL. (Lange, 2019)

5.3.2. Librería cliente

La librería cliente proporciona la API micro-ROS para el código de usuario, es decir, para los nodos micro-ROS a nivel de aplicación. El objetivo general es proporcionar todos los conceptos importantes de ROS 2 relevantes en una implementación adecuada para microcontroladores. Siempre que sea posible, la compatibilidad de API con ROS 2 se logrará para facilitar la portabilidad.

Si bien C sigue siendo el lenguaje de programación dominante para los microcontroladores, existe una clara tendencia hacia el uso de lenguajes de nivel superior, en particular C++. Esta tendencia también está impulsada por microcontroladores modernos que cuentan con varios cientos de kilobytes o incluso unos pocos megabytes de RAM. Es por esto que Micro-ROS ofrece soporte para dos API: (Lange, 2019)

- Una API de C basada en la librería cliente de soporte ROS 2 (rcl), enriquecida con paquetes modulares para la gestión de la ejecución, diagnósticos, parámetros, etc.
- Una API de C++ basada en ROS 2 rclcpp, que al principio requiere analizar la aptitud de rclcpp para su uso en microcontroladores, en particular con respecto al consumo de memoria y CPU, así como a la gestión dinámica de la memoria.

5.4. SLAM

Gran parte de los robots móviles emplean algoritmos para generar la odometría. En el caso de los robots que usan ROS, muchos deciden emplear paquetes, por ejemplo el paquete gmapping, que usen SLAM para generar dicha odometría, como ya se ha visto en los antecedentes donde todos, excepto uno, emplean SLAM. Hay que remarcar que este algoritmo es de los métodos más complejos y exactos de localización para robots móviles.

El algoritmo SLAM *(Simultaneous Localization And Mapping)* es el proceso de creación de un mapa de un entorno desconocido mediante la recopilación de datos procedentes de sensores extrospectivos ubicados en un robot en movimiento. En el SLAM se emplean varios robots para detectar el entorno mediante sus distintos sensores para así crear el mapa de forma probabilística. Debido a las vibraciones del robot y al ruido perteneciente a las mediciones, la certeza del mapeado no es absoluta. (Mathworks, 2020)

Podemos distinguir entre varias tipologías de mapas en SLAM, de entre las cuales hay tres más conocidas:

- Mapeado basado en puntos de referencia *(landmark-based map)*
- Mapeado en función de ocupación de cuadrículas *(occupancy-grid map)*
- Mapeado topológico *(topological map)*

En el caso de ROS, existen dos paquetes muy populares que emplean SLAM: Gmapping y Hector_SLAM. Donde ambos realizan una mapeado en función de ocupación de cuadriculas. (Gerkey, 2017) (Kohlbrecher y Meyer, 2014)

En el mapeado por ocupación de cuadrículas se representa el entorno en una cuadrícula discreta compuesta de celdas. A cada celda se le asigna un valor que representa la probabilidad de ocupación donde el estado x_i de la celda i_n en la cuadrícula puede estar ocupado o no.

Ecuación 1

$$x_i = \{Ocupada, Desocupada\}, \qquad \forall i = 1,\cdots,n$$

Para realizar la navegación el algoritmo calcula la probabilidad de ocupación $P_i(x_i|Z^k)$ de cada celda dada una secuencia de observaciones $Z^k = \{z1, ..., zk\}$. Dado un modelo de sensor $p(z_k|x_i)$, el cual es la forma en la que puede ser descrita la observación de un sensor, se obtiene:

Ecuación 2

$$P_i(x_i|Z^k) = \frac{p(z_k|x_i)P_i(x_i|Z^{k-1})}{\sum_{x_i \in X} p(z_k|x_i)P_i(x_i|Z^{k-1})}$$

La cuadrícula se inicializa configurando todas las celdas para que tengan una probabilidad de 0.5 de estar ocupadas y desocupadas. Esta distribución uniforme inicial para todos los estados de cada celda corresponde a la distribución menos informativa, o a la más aleatoria. Este tipo de mapeado es excelente para la navegación en entornos pequeños y para evitar obstáculos. (Makarenko, Williams, Grocholsky, Hugh y Durrant-Whyte, 2002)

5.5. LÁSERES LIDAR

Los sistemas se sensorización empleados en robots móviles suelen estar basados en tecnologías ToF (*Time of Flight*) o Tiempo de Vuelo, la cual consiste en medir el tiempo que tarda un objeto, partícula u onda (ya sea acústica, electromagnética, etc.) en recorrer una distancia a través de un medio.

En el caso de los lidar esta medición se realiza mediante una serie de pulsos láser, de modo que el dispositivo calcule el tiempo Δt desde que se dispara el haz hasta que este retorna, obteniendo así la distancia a la que se encuentra el objetivo. (Liu, Sun, Fan y Jia, 2018)

Una vez obtenido Δt se puede calcular la distancia empleando la ecuación, donde c es la velocidad de la luz y R la distancia al objetivo.

Ecuación 3

$$R = \frac{1}{2} \cdot c \cdot \Delta t$$

La distancia máxima a la que este tipo de método puede medir depende de la frecuencia de emisión de pulsos de luz que posea el sistema, de modo que a bajas frecuencias *f* se obtiene una distancia mayor de detección al aumentar el período *T* entre pulsos, mientras que con frecuencias elevadas se reduce la esta distancia.

Ecuación 4

$$R_{max} = \frac{1}{2} \cdot c \cdot T = \frac{1}{2} \cdot \frac{c}{f}$$

Empleando la misma lógica que para la distancia máxima, podemos saber la distancia mínima a la que podremos detectar obstáculos. En este caso, teóricamente, la distancia mínima no debería tener límite, ya que el periodo es siempre mayor a Δt. Pero en un caso real esto no se aplica, debido a que las señales tienen una respuesta no lineal haciendo que se superpongan si el periodo se hace demasiado pequeño. (Piatek, 2017)

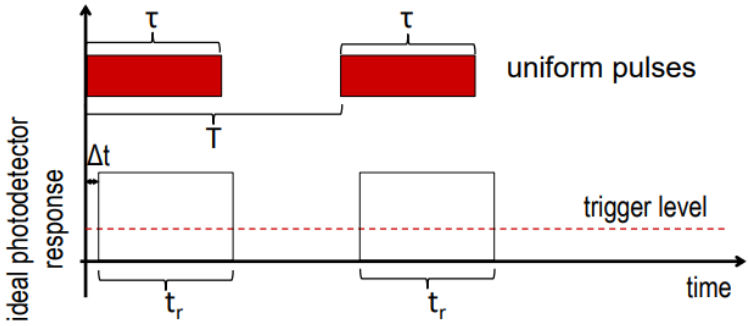


Ilustración 22 Grafica de respuesta ideal del fotodetector (distancia mínima sin límite) (Piatek, 2017)

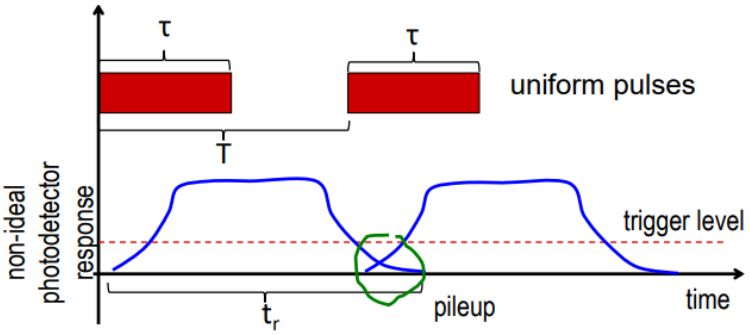


Ilustración 23 Grafica de respuesta real del fotodetector (superposición de las señales) (Piatek, 2017)

5.6. MOTORES CON ESCOBILLAS

Casi todos los movimientos mecánicos que vemos a día de hoy se realizan mediante un motor eléctrico. Un motor eléctrico toma energía eléctrica y la transforma en energía mecánica. Los motores eléctricos vienen en varias clasificaciones y tamaños. Más recientemente, los motores de CC sin escobillas o *Brushless* se han vuelto cada vez más populares, pero para muchas aplicaciones los motores de CC con escobillas siguen siendo la elección correcta. Los motores con escobillas son de menor coste y más fáciles de manejar, por lo que siguen siendo una opción popular. (Birks, 2020)

Cuando la energía eléctrica se aplica a un conductor que se coloca perpendicular a la dirección del campo magnético, el resultado de la interacción entre la corriente eléctrica que fluye a través del conductor y el campo magnético es una fuerza. Esta fuerza empuja al conductor en la dirección perpendicular tanto a la corriente como al campo magnético, por lo tanto, la fuerza es de naturaleza mecánica. El valor de dicha fuerza se puede calcular si se conocen la densidad del campo magnético B, la longitud del conductor L y la corriente que fluye en el conductor I.

Ecuación 5

$$F = B \cdot I \cdot L$$

La dirección del movimiento del conductor se puede determinar con la ayuda de la regla de la mano izquierda o regla de Fleming, siendo esta regla aplicable a todos los motores eléctricos. Cuando un conductor que lleva corriente se coloca en un campo magnético, una fuerza actúa sobre el conductor que es perpendicular tanto a las direcciones del campo magnético como a la corriente. (Anusha, 2015)

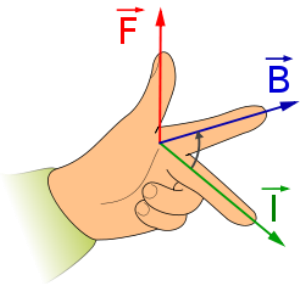


Ilustración 24
Regla de Fleming
(PNGEGG, 2020)

Todos los motores (con algunas excepciones únicas, como los motores piezoeléctricos) dependen de la interacción entre el campo electromagnético en el cuerpo fijo (el estator), el campo electromagnético en la armadura giratoria (el rotor) y cómo son estos campos controlados y cambiados para inducir atracción/repulsión magnética y, por lo tanto, movimiento.

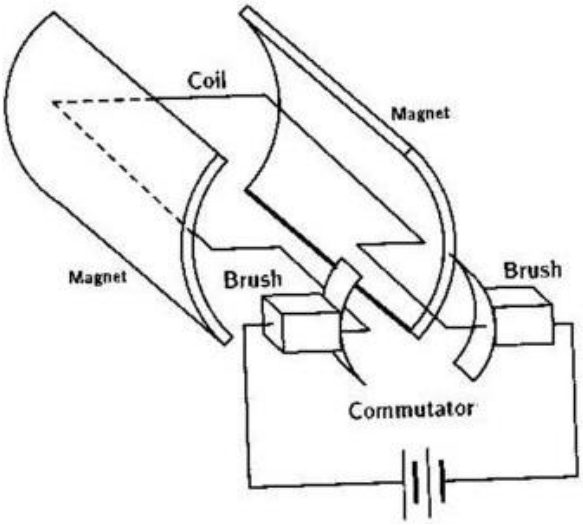


Ilustración 25 Esquema de la estructura interna de un motor con escobillas (Anusha, 2015)

El motor con escobillas básico es completamente mecánico y no requiere electrónica. El control, en la medida de lo posible, es simple: conectando la fuente de CC, el motor se enciende automáticamente y luego gira; existe un problema potencial de arranque debido a una "zona muerta" de ángulo de contacto pequeño, pero que puede resolverse con algunos ajustes de diseño mecánico. Para invertir la dirección de rotación, simplemente se invierten los cables de alimentación; para ajustar la velocidad (dentro de un rango limitado), se aumenta o disminuya el voltaje. En general, la velocidad (revoluciones por minuto, RPM) de un motor de corriente continua on escobillas es proporcional a la fuerza electromotriz (FEM) en su bobina de campo (FEM es el voltaje que se le aplica menos el voltaje perdido debido a su resistencia), mientras que el par es proporcional a la corriente que se consume.

Para generar el campo giratorio necesario para forzar la rotación del inducido sin necesidad de un controlador externo, un par de escobillas ubicadas a 180° en la carcasa del motor dirigen la corriente aplicada a las bobinas del inducido a través de dos contactos del conmutador (placas); cada una de las placas cubre casi 180° del inducido. A medida que el motor gira, la trayectoria del flujo de corriente entre la fuente, las escobillas, las placas del conmutador y las dos bobinas del motor (devanados) cambia automáticamente cada media vuelta. El campo magnético de la armadura interactúa con el campo magnético del rotor, que puede producirse mediante bobinas fijas en el cuerpo del motor o mediante imanes permanentes. A medida que gira el rotor, su campo magnético se invierte y, por lo tanto, el motor sigue girando, porque el campo desarrollado por las bobinas está cambiando y es atraído/rechazado con referencia al campo generado por las bobinas externas o imanes permanentes. (Schweber, 2017)

5.7. ENCODER

Los encoders son componentes que se agregan a un motor de corriente continua para convertir el movimiento mecánico en pulsos digitales que pueden ser interpretados por la electrónica de control integrada. En concreto, se trata de un transductor rotativo, que mediante una señal eléctrica sirve para indicar la posición angular de un eje, velocidad y aceleración del rotor de un motor. El objetivo principal de los distintos tipos de encoders es transformar la información de un formato a otro para estandarización, ajuste de velocidad o control de seguridad.

Los motores de CC tienen un control complejo de posición y velocidad, su comportamiento no es lineal y depende en gran medida de la carga soportada; es por eso que necesitan un encoder (que puede estar integrado o no) para determinar y asegurar una posición correcta del eje. (Dynapar, 2007)

En cuanto a su composición, un encoder está compuesto por un disco que está conectado a un eje giratorio. El disco puede estar fabricado en vidrio o plástico, y se codifica combinando áreas transparentes y opacas que bloquean el paso de la luz.

A continuación, en lo que respecta a su funcionamiento, cuando el eje gira, la fuente de luz infrarroja emite una luz que es interpretada por un sensor óptico (o fototransistor), que a su vez genera los pulsos digitales en función de si la luz pasa por el disco o es bloqueado por las regiones opacas. Esto da como resultado una secuencia de información que permite controlar aspectos como la dirección del movimiento, el radio de giro y, en ciertos casos, la velocidad.

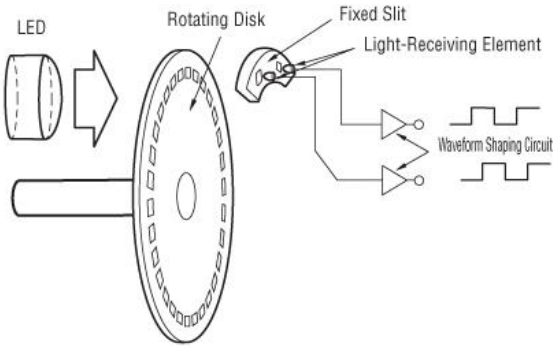


Ilustración 26 Esquema de funcionamiento de un encoder óptico (Arrow, 2017)

Algunas de sus aplicaciones más extendidas son la robótica, los pequeños electrodomésticos o determinadas aplicaciones industriales que requieren una medida angular. (WhitePaper, 2016)

Existen diferentes tipos de encoders según su diseño y funcionalidad.

5.7.1. Encoder óptico

Es el tipo de encoder más extendido y está compuesto por una fuente de luz, un disco giratorio y un detector de luz.

El disco está montado sobre un eje giratorio y tiene regiones opacas y transparentes en su cara. La luz emitida por la fuente es recibida por el fotodetector o interrumpida por el patrón de la región opaca, lo que genera señales de pulso que son leídas por un controlador que incluye un microprocesador para determinar el ángulo exacto del eje. (Arrow, 2017)

5.7.2. Encoder lineal

Es un dispositivo o sensor que tiene una escala graduada para determinar su posición. Los sensores en el encoder leen la escala y luego convierten su posición codificada en una señal digital que puede ser interpretada por un controlador de movimiento electrónico.

Los codificadores lineales pueden ser absolutos o incrementales, y existen diferentes tipos de codificadores lineales según la tecnología utilizada en su mecanismo.

Este tipo de codificador se utiliza en metrología, sistemas de movimiento y para controlar instrumentos de alta precisión en la fabricación de herramientas. (CLR, 2017)

5.7.3. Encoder absoluto

Se basa en la información proveída para determinar la posición absoluta en secuencia. Un encoder absoluto ofrece un codigo único para cada posición.

Se dividen en dos grupos: los encoders de un solo giro y los encoders absolutos de giro múltiple y su tamaño es pequeño para permitir una integración más simple. (Celeramotion, 2018)

5.7.4. Encoder incremental

Como su nombre lo indica, es un encoder que determina el ángulo de posición por medio de realizar cuentas incrementales.

Esto quiere decir que el encoder incremental provee una posición estratégica desde donde siempre comenzará la cuenta. La posición actual del encoder es

incremental cuando es comparada con la última posición registrada por el sensor. (Ingmecafenix, 2017)

5.8. PUENTE EN H

Un puente H es un circuito electrónico simple que nos permite aplicar voltaje a la carga en cualquier dirección. Se utiliza comúnmente en aplicaciones robóticas para controlar motores de CC. Al usar un puente en H, podemos hacer funcionar el motor de CC en sentido horario o anti horario. Este circuito también se utiliza para producir formas de onda alternas en inversores. (Dejan, 2017)

Los elementos de conmutación (S1...S4) suelen ser transistores bipolares o FET, en algunas aplicaciones de alta tensión IGBT. También existen soluciones integradas, pero si los elementos de conmutación están integrados con sus circuitos de control o no, no es relevante en su mayor parte para su funcionamiento.

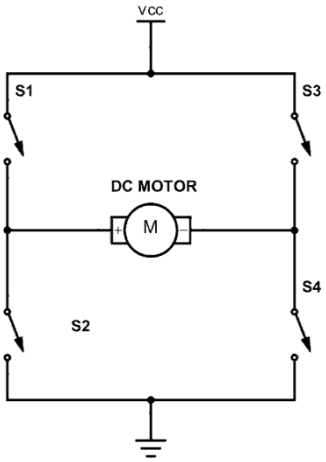


Ilustración 27 Circuito eléctrico de un puente en H (Øyvind, 2018)

El extremo superior del puente está conectado a una fuente de alimentación (batería, por ejemplo) y el extremo inferior está conectado a tierra. En general, los cuatro elementos de conmutación se pueden encender y apagar de forma independiente, aunque existen algunas restricciones obvias. (Tantos, 2011)

Un efecto secundario de cómo funciona un motor es que el motor también generará energía eléctrica. Cuando se desactivan los transistores para detener el funcionamiento del motor, esta energía debe liberarse de alguna manera. Si se agregan diodos en la dirección inversa para los transistores, se da una ruta para que la corriente tome esta energía. Sin ellos, corre el riesgo de que el voltaje aumente y dañe los transistores. (George, 2018)

Si se pretende un funcionamiento a una velocidad inferior a la máxima, los interruptores se controlan mediante PWM. Es una señal periódica, de frecuencia constante. El contenido de información, que se utiliza para cambiar los parámetros operativos del puente, es la relación entre el tiempo de conexión y el tiempo de inactividad. Los distintos modos de conducción difieren en cómo se configuran los interruptores durante el tiempo de encendido y el tiempo de apagado. (Øyvind, 2018)

5.9. CONTROL PWM

La modulación de ancho de pulso *Pulse Width Modulation* (PWM) es un término empleado para describir un tipo de señal digital. La modulación de ancho de pulso se utiliza en una variedad de aplicaciones, incluyendo circuitos de control sofisticados. Podemos lograr una variedad de resultados en ambas aplicaciones porque la modulación de ancho de pulso nos permite variar cuánto tiempo la señal es alta de forma analógica. Si bien la señal solo puede ser alta (generalmente 5V) o baja (tierra) en cualquier momento, podemos cambiar la proporción de tiempo en que la señal es alta en comparación con cuando es baja en un intervalo de tiempo constante. (Developer.android, 2019)

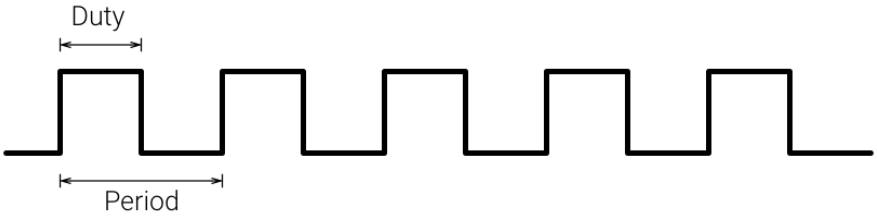


Ilustración 28 Señal PWM (Devloper.android, 2019)

El uso de la modulación por ancho de pulso para controlar un motor pequeño tiene la ventaja de que la pérdida de potencia en el transistor de conmutación es pequeña porque el transistor está completamente "encendido" o completamente "apagado". Como resultado, el transistor de conmutación tiene una disipación de potencia muy reducida, lo que le da un tipo de control lineal que da como resultado una mejor estabilidad de velocidad. Además, la amplitud del voltaje del motor permanece constante, por lo que el motor siempre está a plena potencia. El resultado es que el motor se puede girar mucho más lentamente sin que se ahogue. (AspenCore, 2014)

5.9.1. Ciclo de trabajo

El término ciclo de trabajo describe la proporción de tiempo "encendido" con respecto al intervalo regular o "período" de tiempo; un ciclo de trabajo bajo corresponde a baja potencia, porque la alimentación está apagada la mayor parte del tiempo. El ciclo de trabajo se expresa en porcentaje, siendo el 100% completamente encendido. Cuando una señal digital está encendida la mitad del tiempo y apagada la otra mitad del tiempo, la señal digital tiene un ciclo de trabajo del 50% y se asemeja a una onda "cuadrada". Cuando una señal digital pasa más tiempo en el estado encendido que en el estado apagado, tiene un ciclo de trabajo superior al 50%.

Cuando una señal digital pasa más tiempo en el estado apagado que en el estado encendido, tiene un ciclo de trabajo inferior al 50%. (Jordandee, 2013)

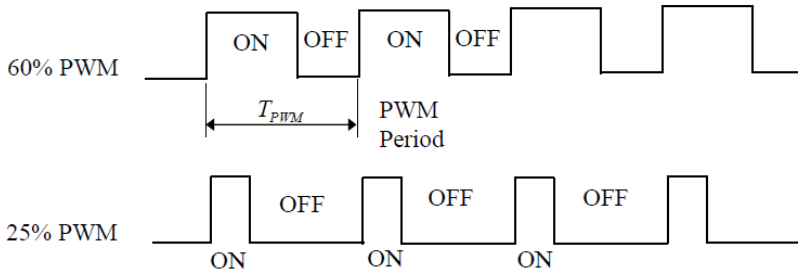


Ilustración 29 Ejemplo de diferentes valores de ciclo de trabajo (Jordandee, 2013)

5.10. BATERÍA

La batería del robot es un elemento de vital importancia, puesto que una elección errónea del tipo de batería o del dimensionado de esta puede repercutir en gran medida al funcionamiento del robot. Existen diversos tipos de batería, donde cada uno se ajusta más a una aplicación que el resto. Basándonos en la información obtenida gracias a los antecedentes, se decide estudiar el funcionamiento de las baterías de polímero de litio (LiPo).

Las baterías de polímero de litio son baterías recargables de tecnología de iones de litio. Este tipo de baterías son las más populares en el mundo de los vehículos RC. A diferencia de las celdas cilíndricas y prismáticas, las LiPos vienen en un paquete o bolsa suave, lo que las hace más livianas pero también menos rígidas. Cada tipo de química de la batería, ya sea polímero de litio, iones de litio, hidruro metálico de níquel u otros, tiene características específicas que definen su funcionamiento eléctrico, tamaño, peso y otras propiedades. (Virtual Amrita Laboratories, 2015)

Ilustración 30 Batería LiPo (Energy, 2019)

En comparación con sus antiguas contrapartes de NiCd/NiMH, las baterías LiPo tienen tres cosas principales a su favor que las convierten en la opción de batería perfecta para vehículos RC:

- Las baterías LiPo son livianas y se pueden fabricar en casi cualquier forma y tamaño.

- Las baterías LiPo tienen grandes capacidades, lo que significa que tienen mucha energía en un paquete pequeño.
- Las baterías LiPo tienen altas tasas de descarga para alimentar los sistemas eléctricos más exigentes.

Este tipo de baterías también presentan algunas desventajas en cuanto a su uso en vehículos RC:

- Problemas de seguridad: debido al electrolito volátil utilizado en las LiPo, pueden incendiarse o explotar.
- Las baterías LiPo requieren un cuidado único y adecuado si van a durar más tiempo que cualquier otra tecnología de batería. La carga, descarga y almacenamiento afectan la vida útil. Si se hace mal, una batería LiPo es inservible en tan solo un error. (Salt, 2020)

Las baterías de iones de litio se clasifican en baterías de iones de litio líquidos y baterías de iones de litio de polímero o baterías de iones de litio de plástico, de acuerdo con los materiales electrolíticos utilizados en las baterías de iones de litio. Los materiales positivos y negativos utilizados en la batería de polímero de iones de litio son los mismos que los iones de litio líquido.

La batería de polímero de litio utiliza una aleación de rutenio como electrodo positivo y utiliza un material conductor de polímero, poliacetileno, polianilina o poliparanfenileno como electrodo negativo y un disolvente orgánico como electrolito. La energía específica de la batería de polianilina de litio puede alcanzar los 350W.h/kg, pero la potencia específica es de solo 50-60W/kg, la temperatura de uso es de -40-70 grados y la vida útil es de aproximadamente 330 veces. El electrolito de la batería de polímero de litio es un polímero sólido flexible, y la hoja de rutenio de metal está sellada en la batería y puede llegar a funcionar normalmente a una temperatura de hasta 180 °C.

Dado que el polímero reemplaza el electrolito líquido con un electrolito sólido, la batería de polímero de iones de litio tiene las ventajas de ser más delgada, de modo que la batería puede estar hecha de una película de compuesto plástico-aluminio. (Energy, 2019)

Es de vital importancia emplear el método adecuado para cargar una batería LiPo, ya que si se usa un método no adecuado se corre el riesgo de destruir la batería. Las baterías LiPo utilizan el concepto de corriente constante/voltaje constante (CC/CV *ConstantCurrent/ConstantVoltage*) para cargar. Básicamente, el cargador mantendrá la corriente, o tasa de carga, constante hasta que la batería alcance su voltaje máximo

(4.2v por celda en un paquete de baterías). Entonces mantendrá ese voltaje, mientras reduce la corriente.

Por otro lado, las baterías de NiMH y NiCd se cargan mejor con un método de carga por pulsos. Cargar una batería LiPo de esta manera puede tener efectos dañinos, por lo que es importante tener un cargador compatible con LiPo.

Del mismo modo que se debe tener cuidado durante el proceso de carga de la batería, es importante que ese cuidado se mantenga durante el proceso de descarga. Como la duración de la batería de polímero de litio es más larga y proporcionan mayor potencia, existe una amenaza constante de incendio y explosión si las baterías no se utilizan correctamente. Una mayor resistencia interna implica una mayor cantidad de calor generado. Cuando se genera calor, se libera en forma de oxígeno, que es un gas combustible que conduce al fuego. (DNK, 2019)

El estado de carga y descarga de la batería se pueden apreciar en las curvas características de la misma:

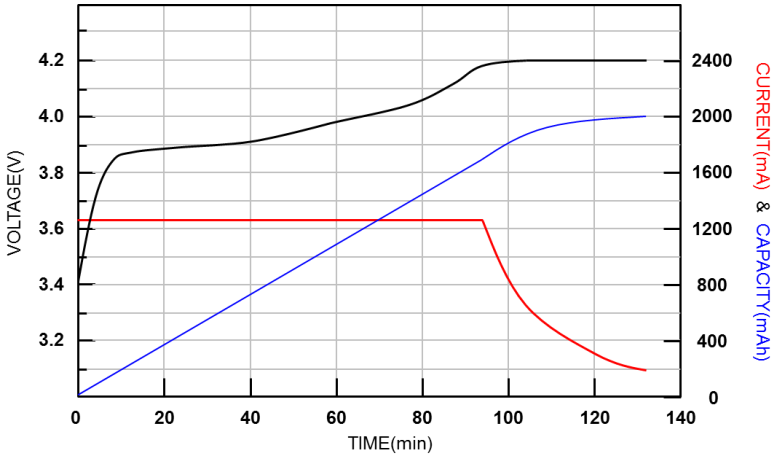


Ilustración 31 Curva de carga de una batería Lipo (DNK, 2019)

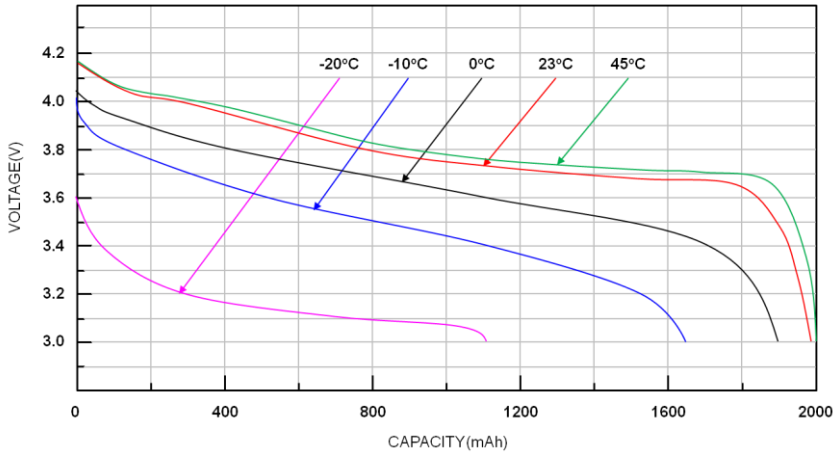


Ilustración 32 Curva de descarga de una batería LiPo (DNK, 2019)

Durante el proceso de selección de la batería se deben tener en cuenta una serie de consideraciones como las que se han detallado arriba. Asimismo, es también necesario prestar atención a las características de la batería para que esta desempeñe su función de forma correcta. Las características se ven reflejadas en la nomenclatura empleada para clasificar las baterías, conociendo esta nomenclatura y su significado se puede realizar la selección de la betería de forma óptima para el proyecto en el que se vaya a emplear la batería seleccionada.

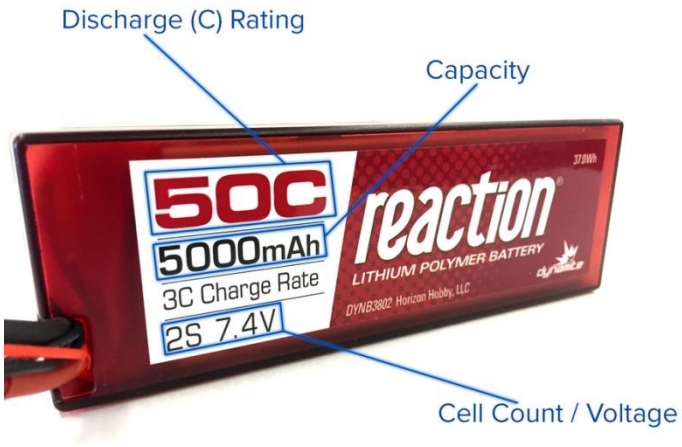


Ilustración 33 Nomenclatura de una batería LiPo (Salt, 2020)

5.10.1. Clasificación por número de celdas S

A diferencia de las celdas de NiCd o NiMH convencionales que tienen un voltaje de 1,2 voltios por celda, las celdas de una batería LiPo tienen una clasificación de 3,7 voltios por celda y 4,2 voltios cuando están completamente cargadas. El beneficio es que se pueden usar menos celdas para formar un paquete de baterías. El numero S hace referencia al número de celdas que posee la batería y viene acompañado por el número de tensión nominal de la batería. Además, los números pueden ir acompañados de una P la cual indicaría dos paquetes en serie de dos celdas conectados en paralelo para duplicar la capacidad. Algunos ejemplos del empleo de la nomenclatura S serian: (Mobus, 2016)

- Batería de 3,7 voltios = 1 celda x 3,7 voltios (1S)
- Batería de 7,4 voltios = 2 celdas x 3,7 voltios (2S)
- Batería de 11,1 voltios = 3 celdas x 3,7 voltios (3S)

5.10.2. Capacidad indicada en mAh

La capacidad indica cuánta energía puede contener la batería y se indica en miliamperios hora (mAh). Esta es una forma de mostrar cuánta carga o descarga

(medida en miliamperios) se puede poner en la batería durante 1 hora, momento en el que la batería se descargará por completo. (Erle Robotics S.L., 2014)

En un principio se puede pensar que si se quiere una mayor duración de la batería hay que escoger siempre una con mayor capacidad, pero en algunos casos esto puede suponer un problema, ya que cuanta más capacidad tiene la batería, mayor es su tamaño y peso. Esto puede acarrear un problema, debido a que el robot puede no disponer de suficiente espacio para la batería, o de la suficiente fuerza para desplazarse con una batería muy pesada.

5.10.3. Tasa de descarga, número C

El voltaje y la capacidad tienen un impacto directo en ciertos aspectos del vehículo, ya sea la velocidad o el tiempo de funcionamiento. Esto los hace fáciles de entender. La tasa de descarga es un poco más difícil de entender, y esto ha llevado a que sea el aspecto más exagerado e incomprendido de las baterías LiPo.

La tasa de descarga es el valor que permite determinar la cantidad de amperios que la batería puede generar continuamente sin sufrir daños; por lo tanto, también es una medida de la tasa a la que la batería se puede descargar de manera segura, es decir, sin dañar la batería. Una batería con un índice de descarga de 10C se descargaría a una velocidad 10 veces mayor que la capacidad de la betería, una batería de 15C = 15 veces más, una batería de 20C = 20 veces más, y así sucesivamente. (Schneider, 2012)

Usando una batería de 1000 mAh como ejemplo; si tiene una clasificación de descarga de 20C, significaría que se podría obtener una carga máxima sostenida de hasta 20,000 miliamperios o 20 Amperios de esa batería. Desde un punto de vista de tiempo puramente teórico, esto equivale a 333 mAh de extracción por minuto, por lo que la batería de 1000 mAh se agotaría por completo en aproximadamente 3 minutos si se expone a la tasa de descarga máxima nominal de 20C todo el tiempo.

5.11. CINEMÁTICA DIFERENCIAL

La cinemática diferencial es un tipo sistema motriz empleado en vehículos, el cual otorga al mismo la capacidad de rotar sobre sí mismo. Esto puede llegar a ser muy útil en determinados espacios ya que la cinemática diferencial permite un giro en menos espacio que, por ejemplo, un modelo cinemático Ackermann (Modelo similar al empleado en los automóviles convencionales). (Piñera-García, Amigó-Vega, Concepción-Álvarez, Casimiro-Martínez, Fernández-Rodríguez, y Casanovas-Perer, 2013)

Normalmente este tipo de sistemas se realizan con un sistema de tracción formado por una rueda loca, que simplemente es una rueda que irá guiada por el resto del sistema de tracción y dos ruedas de tracción las cuales llevan acoplados sus respectivos motores CC independientes.

La rotación y traslación en este tipo de sistemas las otorgan el hecho de que las ruedas sean independientes, es decir que cada una puede girar a una velocidad o incluso girar una con la otra parada. Para determinar el giro, disponemos de las siguientes ecuaciones:

Ecuación 6

$$\dot{x} = v(t) \cdot \cos(\theta(t))$$

Ecuación 7

$$\dot{y} = v(t) \cdot \sin(\theta(t))$$

Ecuación 8

$$\dot{\theta}(t) = \omega(t)$$

Donde *x* e *y* son las velocidades en cada uno de los ejes en el plano cartesiano del robot y *w(t)* es la velocidad angular del giro en función del tiempo.

Una vez obtenido todo lo anterior podemos determinar la posición y orientación del robot integrando las velocidades anteriores en un periodo de tiempo Δt.

Ecuación 9

$$x(t) = x(t_0) + \int_{\Delta t} v(t) \cdot \cos\big(\theta(t)\big)\, dt$$

Ecuación 10

$$y(t) = y(t_0) + \int_{\Delta t} v(t) \cdot \sin\big(\theta(t)\big)\, dt$$

Ecuación 11

$$\theta(t) = \theta(t_0) + \int_{\Delta t} v(t)dt$$

Donde Δt es el período de desplazamiento, y en caso de que tendiese a 0, las integrales anteriores se podrían reemplazar por desplazamientos diferenciales Δx, Δy, $\Delta\theta$. (Valencia, Johnny, Montoya y Hernando, 2009)

5.12. ODOMETRÍA

La mayoría de los problemas de la robótica fundamentalmente se reducen a las preguntas: ¿dónde estoy? ¿Cómo se puede ir de un lugar a otro sin tener una idea de la posición actual? Se podría proceder ingenuamente, dando tumbos, esperando que el objetivo aparezca dentro del alcance de los sensores del robot, pero un mejor plan es navegar. Un método básico de navegación, utilizado por prácticamente todos los robots, es la odometría, que utiliza el conocimiento del movimiento de las ruedas para estimar el movimiento del vehículo. (Ttuadvancedrobotics, 2010)

La odometría es el uso de recopilar datos de actuadores y encoders para determinar la posición y la velocidad de un robot desde su posición inicial. La odometría es una forma de navegación por estima o *Dead Reckoning* en inglés. Cada nueva posición se basa en la posición anterior. De modo que se acumulan los errores que se produzcan cada vez que se calcula una nueva posición. Cuanto más lejos tenga que viajar el robot, más errores acumulará, ya que cada nueva posición se basa en la anterior. A partir de la distancia recorrida d = vt, se puede calcular la nueva posición del robot. En una dimensión, el cálculo es trivial, pero se vuelve un poco más complejo cuando el movimiento implica giros. (Ben-Ari y Mondada, 2018)

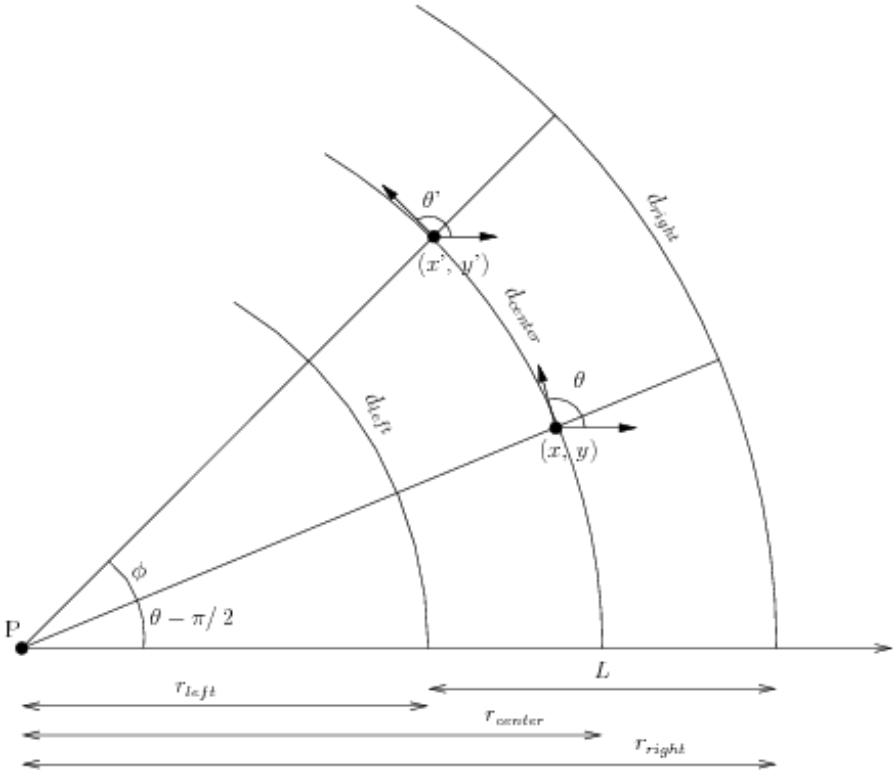


Ilustración 34 Geometría detallada de la odometría de un robot al girar (Edwin, 2004)

Si en un intervalo de tiempo corto, Δt, las velocidades de las dos ruedas son relativamente constantes, entonces la velocidad de avance del robot, V_x, y la velocidad de rotación, ω, también se pueden considerar constantes y podemos actualizar la posición global desde $P_i = (x, y, \theta)$ a $P_{i+1} = (x', y', \theta')$. Las nuevas posiciones x' e y' vienen dadas por:

Ecuación 12

$$x' = x + r_{center}(-sin\theta + sin\Phi \cdot cos\theta + sin\theta \cdot cos\Phi)$$

Ecuación 13

$$y' = y + r_{center}(cos\theta - cos\Phi \cdot cos\theta + sin\theta \cdot sin\Phi)$$

Para simplificar significativamente las ecuaciones, se puede hacer una aproximación. Si Φ es pequeño, como suele ser el caso para pasos de tiempo pequeños, podemos aproximar $sin(\Phi) = \Phi$ y $cos(\Phi) = 1$. Con estas consideraciones obtenemos:

Ecuación 14

$$x' = x + r_{center}(-sin\theta + \Phi\, cos\theta + sin\theta)$$

Ecuación 15

$$x' = x + r_{center}\, \Phi\, cos\theta$$

Ecuación 16

$$x' = x + d_{center}\, cos\theta$$

Para el eje x, mientras que para el eje y obtenemos:

Ecuación 17

$$y' = y + r_{center}(cos\theta - cos\theta + \Phi\, sin\theta)$$

Ecuación 18

$$y' = y + r_{center}\, \Phi\, sin\theta$$

Ecuación 19

$$y' = y + d_{center}\, sin\theta$$

El cambio en la dirección de orientación del robot, Φ, es la diferencia de las distancias recorridas por las ruedas divididas por el radio de rotación, L, que es la distancia entre las ruedas. Las ecuaciones de la odometría para (x', y', θ') son: (Edwin, 2004)

Ecuación 20

$$d_{center} = \frac{d_{left} + d_{right}}{2}$$

Ecuación 21

$$\Phi = \frac{d_{right} - d_{left}}{L}$$

Ecuación 22

$$P_{i+1} = \begin{bmatrix} X_i \\ Y_i \\ \theta_i \end{bmatrix} + \begin{bmatrix} d_{center}\, cos\theta_i \\ d_{center}\, sin\theta_i \\ \Phi \end{bmatrix}$$

6. DESARROLLO

Durante el desarrollo del proyecto se han realizado numerosas modificaciones al mismo. Este apartado se dividirá en diversos sub apartados en los que se explicará el procedimiento seguido en el transcurso de los mismos.

6.1. ESP32 CON ROS2 Y MICRO-ROS

Inicialmente se propuso que en el proyecto se emplease un esp32, el cual sería el núcleo del robot. Para conseguir esto era necesario que se implementase ROS dentro del mismo esp32, lo cual no es posible puesto que ROS solo puede ser instalado en ordenadores, ya sean ordenadores de sobremesa, portátiles o SBCs (Single Board Computer, como por ejemplo una Raspberry Pi), comúnmente si se desea emplear arduino en un proyecto con ROS, se usa el paquete de ROSSerial, el cual permite realizar una comunicación serial entre el Arduino y el ordenador en el que está corriendo ROS. Así que era necesario encontrar una alternativa, que en este caso sería ROS2 y Micro-ROS que, aunque no introduzcan todo el sistema de ROS directamente en el esp32, sí que se aproximan más a introducir el microprocesador/microcontrolador como elemento activo dentro del sistema de ROS.

ROS2 como tal no era posible implementarlo puesto que es, fundamentalmente, igual que ROS. En este caso, lo que se haría sería emplear ROS2 y el paquete ROS2Arduino, el cual es parecido al ROSSerial, pero a diferencia del último este emplea un agente Micro-XRCE-DDS, haciendo una implementación distinta a la de ROSSerial. Sabiendo esto, se decidió emplear Micro-ROS, el cual emplea también el agente Micro-XRCE-DDS, pero este está directamente implementado en el sistema y además, Micro-ROS trae más características que lo hacen más orientado al trabajo con microcontroladores y microprocesadores.

Para poder emplear Micro-ROS, y cualquier otra versión de ROS o ROS2, es necesario emplear un ordenador con el sistema operativo Ubuntu, si bien es verdad que también es posible usar ROS en Windows, los mismos desarrolladores recomiendan emplear Ubuntu puesto que es donde aseguran el correcto funcionamiento de ROS. Además, es necesario instalar ROS2 puesto que Micro-ROS depende de la arquitectura de ROS2 para funcionar. De este modo se necesita hacer una instalación de ROS2 en Ubuntu y una vez instalado se ha de instalar Micro-ROS.

Una vez realizada la instalación de Micro-ROS y después de haber completado los tutoriales presentes en la misma página de Micro-ROS se decidió empezar a hacer

pruebas con el esp32. Pero, tras muchos intentos, no se consiguió realizar la conexión entre el esp32 y el agente Micro-XRCE-DDS.

Tras intentar solventar durante varios días sin cambio alguno, se decide abandonar la idea de emplear el esp32 con ROS2/Micro-ROS. Siendo la causa principal la falta de información disponible al respecto, pues ROS2 es un proyecto joven y Micro-ROS es aún más joven, siendo la única información disponible acerca de Micro-ROS en esp32, durante el proceso de desarrollo de este proyecto, la noticia de los desarrolladores de Micro-ROS informando del porteo de Micro-ROS a esp32 el día 27 de Agosto de 2020.

6.2. RASPBERRY PI CON ROS

Tras el fracaso con Micro-ROS y el esp32, se decidió optar por un proyecto más convencional, en el que se emplearía una Raspberry Pi como núcleo o "cerebro" del robot.

Del mismo modo que con ROS2 es necesario realizar una instalación de ROS en un sistema operativo Ubuntu. En este proyecto se decidió emplear la versión de ROS Kinetic Kame, que pese a no ser la versión más reciente de ROS a la fecha de inicio del proyecto, era la recomendada puesto que la versión más nueva, ROS Melodic Morenia, aún tenía algunos paquetes sin actualizar y podía presentar problemas en cuanto a la funcionalidad del software. Para instalar ROS Kinetic es necesaria la versión de Ubuntu Xenial 16.04. Para realizar esta instalación se puede cambiar el sistema operativo de un ordenador. En este caso, como la instalación se llevaría a cabo en una Raspberry Pi, los desarrolladores de ROS recomiendan emplear Ubuntu Mate 16.04. Desgraciadamente no se disponía de una Raspberry Pi en la que hacer la instalación, por lo que se decidió emplear una máquina virtual por ser una opción más sencilla. La página de ROS cuenta con tutoriales detallados para la instalación del mismo, de modo que no se va a hacer hincapié al proceso de instalación del software durante el desarrollo, a excepción de casos excepcionales en los que deban ser destacadas algunas consideraciones durante la instalación.

Una vez instalado ROS, se decidió seguir los números tutoriales disponibles en la página de ROS para así familiarizarse con el sistema. De entre esos tutoriales es importante destacar algunos de ellos puesto que resultarán especialmente útiles más adelante en el proyecto.

Una vez seguidos los tutoriales se puede dar paso al desarrollo del robot.

6.2.1. Desarrollo del robot

Para poder empezar con el desarrollo del robot es necesario dejar previamente definidas las características que debe tener el robot. En este proyecto se busca:

- Que el robot sea capaz de navegar de forma autónoma en un entorno desconocido.
- La generación de un mapa del entorno.
- Que el robot sea capaz de girar sobre sí mismo (Cinemática diferencial)

Sabiendo las características del robot se puede dar paso a la selección de los paquetes de ROS los cuales van a determinar el comportamiento del robot. La opción más sencilla es buscar un paquete que le otorgue al robot una de las características que buscamos y a partir de ahí escoger el resto de paquetes en función del ya seleccionado. Por ejemplo, deseamos que el robot sea capaz de navegar en un entorno desconocido de forma autónoma, para ello existen diferentes métodos los cuales nos permitirían conseguir dicha característica, en nuestro caso también queremos que se genere un mapa por lo que se decide emplear SLAM para cumplir así con el mapeo. En ROS existen diversos paquetes que implementan algoritmos de SLAM, pero de entre ellos hay dos que son los más recomendados por la comunidad: gmapping y Hector_SLAM. De estos dos, el que mejor se ajusta al proyecto es gmapping puesto que este está pensado para un uso más general, mientras que Hector_SLAM está orientado a usos específicos en los que ciertos aspectos del robot sean incompatibles con gmapping. De este modo, al seleccionar el primer paquete podemos ver que dependencias tiene este paquete para así decidir el resto de elementos del robot.

6.2.1.1. gmapping

En la página del paquete gmapping podemos ver la información general del paquete: cómo funciona, que nodos activa en el sistema de ROS, requerimientos de hardware, etc. De la información que podemos obtener en la página del paquete, deberemos distintos aspectos, pero sobretodo en los requerimientos del paquete, los cuales pueden ser de ROS o de algún otro elemento de software o hardware. En el caso de gmapping, tenemos los siguientes requisitos:

A nivel de hardware necesitamos un robot móvil que proporcione datos de odometría y esté equipado con un telémetro láser fijo y montado horizontalmente. Pese a que se especifique "telémetro láser", se pueden emplear diversos tipos de

sensores, en muchos casos se emplea un sensor Kinect de Microsoft. En este caso se decidió emplear un láser Lidar.

A nivel de sistema de ROS, el paquete tiene un nodo llamado *slam_gmapping*, este nodo se suscribe y publica en ciertos tópicos, en concreto, se suscribe a dos tópicos: el tópico tf y el tópico scan. De estos dos, el tópico tf tiene una gran importancia en prácticamente todos los proyectos de ROS. Del mismo modo, el tópico scan es de gran importancia puesto que es en este tópico en el que se transmiten los datos del o de los sensores que tenga el robot, en concreto en el caso de gmapping en el tópico scan lo que hace es leer un mensaje del tipo sensor_msgs/LaserScan donde van los datos recopilados por el lidar.

Por otro lado, el nodo publica en tres tópicos: map_metadata, map y ~entropy. De estos tres, el importante es el tópico map, en el que se transmite un mensaje del tipo nav_msgs/OccupancyGrid. En este mensaje está el mapa generado por el nodo y que luego será leído para realizar la navegación.

Asimismo, el paquete cuenta con una serie de parámetros los cuales nos permitirán configurar el funcionamiento del paquete. Hay una gran variedad de parámetros, pero en varios de estos se puede dejar el valor por defecto, mientras que en otros es necesario realizar diversas pruebas para obtener el valor ajustado.

Una vez vistos los parámetros, vemos que el paquete necesita una serie de transformadas, estas son las que proporciona el paquete tf y lo que hacen es, según la descripción del paquete: *tf es un paquete que permite al usuario realizar un seguimiento de múltiples marcos de coordenadas a lo largo del tiempo. tf mantiene la relación entre los marcos de coordenadas en una estructura de árbol almacenada en el tiempo y permite al usuario transformar puntos, vectores, etc. entre dos marcos de coordenadas en cualquier momento deseado.*

Este paquete es, probablemente, unos de los más importantes de ROS, y el paquete gmapping nos especifica que necesita dos transformadas para operar:

- <the frame attached to incoming scans> → base_link
- base_link → odom

Además de las requeridas, el paquete también proporciona una transformadas:

- map → odom

Hay varios métodos para obtener estas transformadas, en función de cómo de desarrolle el robot será más sencillo emplear unos métodos u otros. Como en este proyecto se planea emplear un modelo URDF se decide emplear los paquetes

robot_state_publisher y joint_state_publisher para obtener las transformadas puesto que ambos son compatibles con URDF, el cual se explicará más adelante.

6.2.1.2. Rplidar

Como se decide emplear un láser lidar para la obtención de datos del entorno, es necesario poder convertir los datos que manda el lidar a un mensaje del tipo sensor_msgs/LaserScan y que así pueda ser leído por el resto de elementos del sistema. Para ello se pueden emplear diversos paquetes, como para este proyecto se busca un modelo low cost, se decide buscar un modelo de lidar que no sea muy costoso. De todos los modelos encontrados se decide emplear un RPLidar A1M8 y para convertir los datos se emplea el paquete de ROS rplidar, el cual está orientado a ser empleado con esta marca de láseres.

Ilustración 35 RPLidar A1M8 (Slamtec, 2019)

El paquete emplea un nodo llamado rplidarNode el cual publica el mensaje LaserScan en el tópico scan. Para ello lee el puerto USB en el que está conectado el lidar y transforma los datos y posteriormente los publica. Lo único que es necesario configurar es el puerto USB en el que el nodo leerá los datos.

6.2.1.3. Estructura

Una vez escogido el sensor, es necesario empezar a plantearse que estructura se va a emplear para el proyecto. Por un lado se podría plantear una estructura 100% original y realizar un diseño completo lo cual conseguiría que se adaptase mejor al proyecto, pero por otro lado eso supondría una mayor carga de trabajo, así como un mayor coste final debido a las piezas personalizadas. Por esas dos razones se decide emplear una estructura comercial. Tras una búsqueda de las posibles estructuras disponibles en el mercado, se decide emplear una estructura CR0010 - 4WD Robot Kit.

Ilustración 36 Kit CR0010 - 4WD (ViewTek, 2018)

Este kit es uno de los más comunes y empleados en proyectos sencillos en los que no es necesario desarrollar una estructura

personalizada. Pese a que este kit se ajusta bien al proyecto, es necesario realizar algunas substituciones de sus elementos para así mejorar el funcionamiento del mismo.

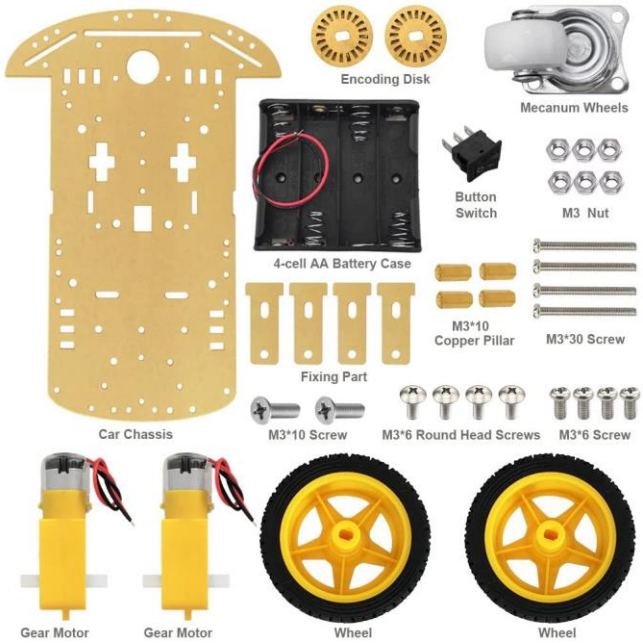


Ilustración 37 Componentes del kit (ViewTek, 2018)

Primero, los motores han de ser substituidos. Los que trae el propio kit son, según diferentes usuarios que ya han adquirido el kit previamente y lo han empleado, poco potentes si se quiere poner algún elemento encima de la base del robot que haga que este sea algo pesado, llegando a decir en alguna ocasiones que, a no ser que se pretenda usar el kit en vacío, se deberían cambiar los motores.

Asimismo, es posible que sea necesario añadir algún elemento a la estructura para ampliar el espacio disponible para poner los elementos sobre el robot.

Por otro lado, también es recomendable cambiar la rueda loca que va en la zona trasera de la base, puesto que el diseño de la misma puede dar problemas en un futuro al tratarse de un robot con cinemática diferencial. La rueda podría bloquearse y hacer que el robot tenga dificultades para girar.

Una vez decididos los cambios a realizar en los elementos del kit, se puede empezar a buscar los reemplazos. Para los motores se busca un modelo que cuente con encoder puesto que será necesario en un futuro para obtener feedback de la velocidad del robot. Además se valora que el motor cuente con la rueda y con el soporte para ser unido a la base del robot.

Después de valorar diversos modelos, se decide emplear un DC Gear Motor Encoder Motor with Mounting Bracket and Wheel. Se trata de un modelo genérico que cuenta con un encoder y se puede elegir entre 3 versiones del motor:

- 12V 320 rpm
- 12V 107 rpm
- 6V 160 rpm

Se decide emplear la versión de 12V 107 rpm puesto que en la versión de 320 rpm se pierde resolución del encoder y además no se prevé que el robot necesite que los motores giren tan rápido. Además, la documentación que el fabricante facilita en este modelo es mucho más detallada que la que presenta el resto, esto ha contribuido en gran medida a la elección de este motor.

Ilustración 38 DC Gear Motor Encoder Motor with Mounting Bracket and Wheel (CHIHAI, 2017)

Por lo que respecta a la rueda trasera, una buena alternativa sería emplear unidades de transferencia de bolas o bolas transportadoras. Este tipo de unidades son idóneas para el movimiento preciso y suave de cargas. Estas ruedas se pueden encontrar en distintos sitios web como por ejemplo Amazon o Ebay, pero normalmente en estos sitios no se tiene una buena documentación del elemento (medidas, material, etc.) Debido a esto se decide buscar una empresa dedicada a la fabricación de este tipo de elementos y se decide emplear un LF25 de la empresa Omnitrack.

La unidad LF25 esta categorizada como una aplicación de baja carga en el catálogo de Omnitrack, pero esta puede llegar a cargar con hasta 55 kg por lo que la hace idónea para el proyecto puesto que el peso total del robot no se espera que llegue a los 2 kg.

Ilustración 39 Unidad LF25 (Omnitrack, 2019)

Como hemos decidido emplear la unidad LF25, la base no queda nivelada puesto que la altura de la bola transportadora no es la misma que al de la rueda loca que originalmente venía incluida en el kit. Como estamos empleando un lidar, es imperativo que la base esté lo más nivelada posible, para que la veracidad de los datos obtenidos sea la máxima.

Con esto en mente, debemos conseguir que la base quede a nivel, para ello debemos diseñar un elemento que separe la unidad LF25 de la base una cierta

distancia. Para obtener esta distancia debemos calcular cuanta distancia hay entre la base y el suelo, para luego restar la altura de la unidad de transferencia.

Para saber a qué altura esta la base del robot debemos buscar las especificaciones de los motores, puesto que en función de estos podremos calcular la altura.

Como vemos en la ilustración 40, las ruedas tienen 80 mm de diámetro. Con esta medida podríamos considerar que la distancia al suelo es la mitad, 40 mm que es el radio de las ruedas. Pero en el caso de estos motores no podemos hacer eso por una razón. Hay una pieza que sirve para anclar el motor a la base del robot, la cual no da algo más de distancia.

Ilustración 40 Diámetro y anchura de las ruedas (CHIHAI, 2017)

Para obtener la distancia de forma correcta se deben tener en cuenta el factor antes mencionado, para ello debemos conocer las medidas de la pieza de anclaje.

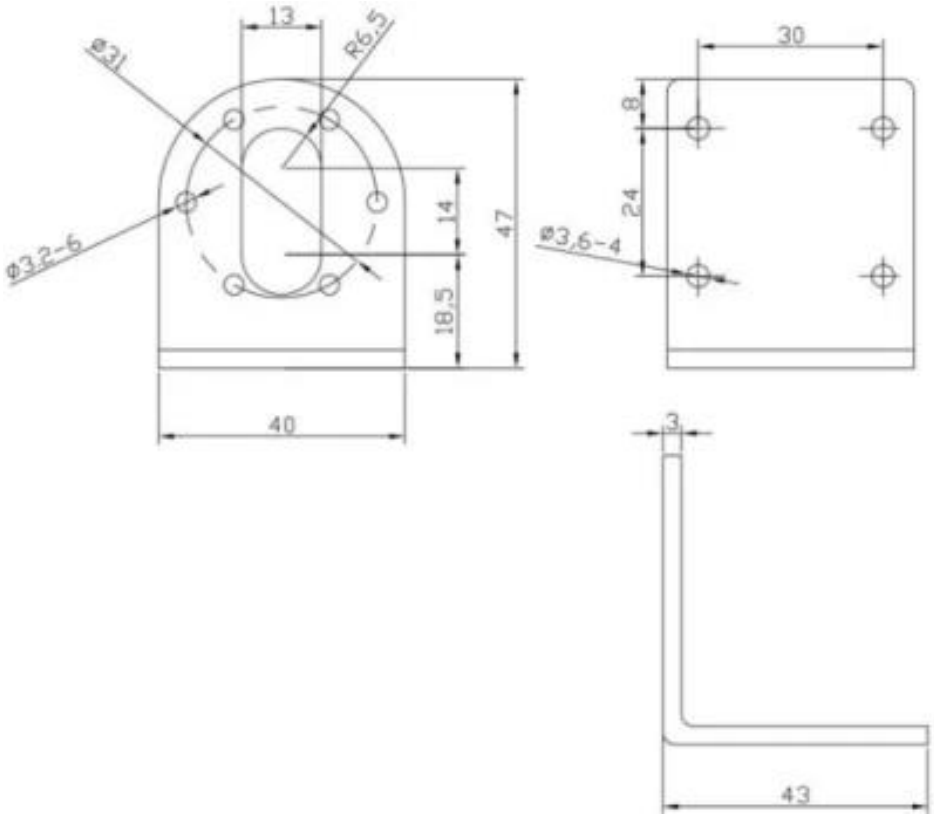


Ilustración 41 Plano de la pieza de anclaje de las ruedas del robot (CHIHAI, 2017)

De las medidas que podemos observar en el plano de la pieza, las que debemos tener en cuenta son las que nos das las distancias desde el extremo de la pieza a los dos focos del agujero con forma de elipse, puesto que el eje del motor, y por lo tanto el centro de la rueda, se sitúa en el foco superior, por lo tanto, si sumamos las dos cotas obtenemos la distancia de la base al centro de la rueda.

Ecuación 23

$$14\ mm + 18.5\ mm = 32.5\ mm$$

Sabiendo esta distancia podemos sumar el radio de la rueda para obtener la distancia total de la base al suelo.

Ecuación 24

$$32.5\ mm + 40\ mm = 72.5\ mm$$

Lo único que debemos hacer en este punto es restar la altura de la unidad LF25 a los 72.5 mm que acabamos de calcular para obtener la distancia que debemos cubrir con la pieza que diseñemos.

Para obtener la altura de la unidad LF25 solo tenemos que ir a la página web del fabricante, donde están todas las medidas.

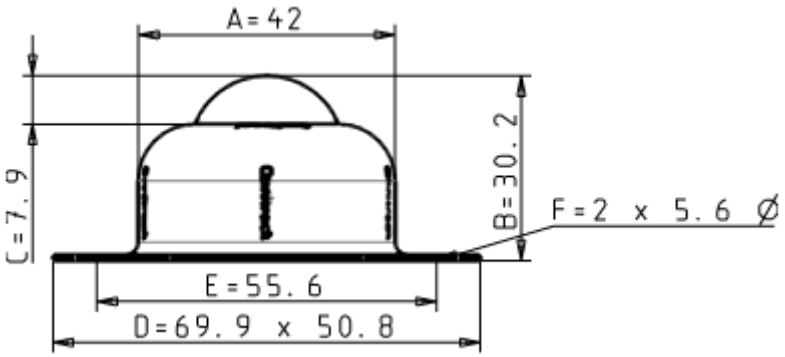


Ilustración 42 Plano de la unidad LF25 (Omnitrack 2019)

La cota que debemos emplear es la de 30.2 mm, puesto que es la que nos da la medida de un extremo a otro de la unidad LF25.

Ecuación 25

$$72.5\ mm - 30.2\ mm = 42\ mm$$

Antes de poder diseñar la pieza debemos tener en cuenta dos valores más, presentes en la ilustración 42. Primero, la distancia entre los agujeros de la base de la unidad LF25, 55.6 mm. Segundo, el diámetro de estos mismos agujeros, 5.6 mm.

Debemos tener en cuenta estos valores puesto que hemos de poder fijar la pieza que creemos a la unidad de transferencia.

Con estas dimensiones podemos dar paso al diseño de la pieza.

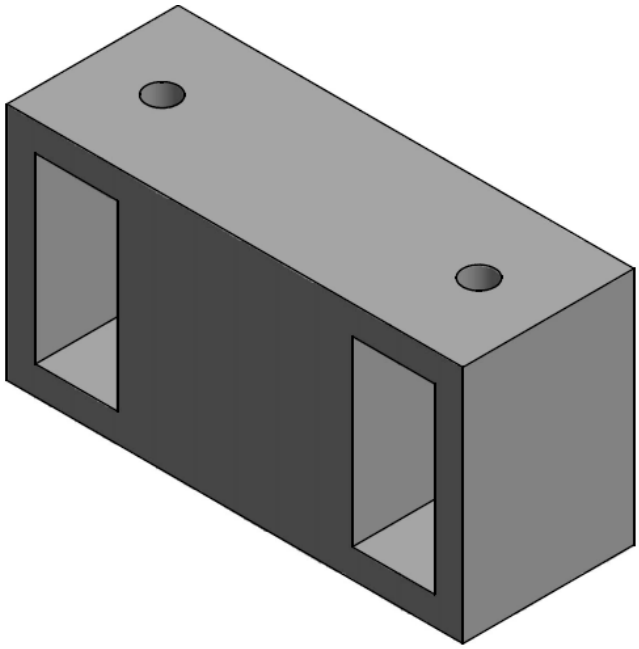


Ilustración 43 Pieza niveladora (Elaboración propia)

Como se ve en la imagen, se decide diseñar una pieza sencilla que cumpla con su función. La única función que debe cumplir la pieza es la de separar lo suficiente la unidad LF25 de la base para que la última este nivelada y el lidar funcione bien, con eso en mente se podría emplear una pieza maciza ya que también cumpliría con la función, pero en este caso se ha decidido vaciar algo de material para hacer la pieza más ligera pero manteniendo la rigidez necesaria. Para unir la pieza a la base se deberán hacer dos agujeros nuevos en la parte trasera de modo que la pieza niveladora quede bien alineada. Para unir los tres elementos se emplearán dos tornillos STM32 M5x50 – H3 y para fijarlos dos tuercas M5.

Como se ha mencionado antes, es necesario añadir un elemento extra para ampliar el espacio disponible para poner los elementos sobre el robot. En este caso se decide optar por la funcionalidad, por encima de la complejidad o estética de la pieza. De este modo se decide crear una segunda base la cual irá sobre la propia base del kit. Para crear esta base se emplea un tablero contrachapado de 3 mm de espesor, cortado a unas medidas de 190x120 mm. Para unirlo a la base se emplean pilares hexagonales huecos de cobre de métrica M3 de doble paso, con una longitud de 30 mm.

6.2.1.4. Creación del paquete de ROS

Para poder empezar a trabajar en el sistema de ROS debemos crear un paquete el cual tendrá en su interior los nodos, archivos .launch, archivos de configuración, etc. Para crear el paquete podemos emplear el comando *catkin_create_pkg* en una ventana de terminal de Ubuntu. Con este comando podemos crear un paquete con el nombre que queramos y con las dependencias que este necesite. Pese a que las dependencias puedan ser establecidas en el momento de creación del paquete, más adelante se podrán añadir o quitar más modificando el archivo package.xml.

Sabiendo esto, podemos emplear el comando únicamente dejando las dependencias básicas. Estas dependencias básicas se resumen en 2 paquetes: ROSCpp y ROSPy. Estos paquetes nos permitirán desarrollar los nodos que necesitemos en nuestro proyecto, para crear estos nodos deberemos escoger el lenguaje de programación que emplearemos. Aquí es donde entran estos dos paquetes, los cuales adaptan dos lenguajes C++ y Python respectivamente. Con esto nos quedaría el siguiente comando:

Catkin_create_pkg robot_tfg roscpp rospy

Una vez creado el paquete podemos dar paso a la creación de nodos, servicios y mensajes que queramos emplear dentro del paquete. De momento no podemos crear mucho puesto que todavía no tenemos claros cuantos paquetes emplearemos en el proyecto ni si alguno nos condicionará cunando queramos crear los distintos elementos de nuestro paquete. Por este motivo es mejor seguir determinando los paquetes que se van a emplear antes de dar paso a la programación.

6.2.1.5. Navegación

Ahora que ya se ha escogido una estructura, un sensor con el que escanear el entorno y se está generando un mapa con los datos obtenidos. Es momento de empezar a plantearse qué hacer con ese mapa. En un principio se ha especificado que una de las características que el robot debía tener era la capacidad de navegar por un entorno desconocido de forma autónoma. Para conseguir esto, hay un paquete extensamente usado y recomendado por toda la comunidad de ROS: El navigation Stack.

El navigation Stack, más que un paquete, se trata de un metapaquete. Este es bastante simple a nivel conceptual: Toma información de los sensores y odometría y emite comandos de velocidad para enviarlos a un robot móvil. Sin embargo, el uso del Navigation Stack en un robot no genérico es un poco más complicado. Como requisito previo para el uso del navigation stack, el robot debe emplear ROS, tener un árbol de transformadas tf y publicar los datos del sensor utilizando los tipos de mensajes ROS correctos (LaserScan o PointCloud, en función de que sensor se emplee). Además, el navigation stack debe configurarse para que funcione correctamente.

Asimismo, hay una serie de requisitos necesarios que debemos cumplir a nivel de hardware si queremos emplear el metapaquete:

- Está diseñado para robots con cinemática diferencial y holonómicos únicamente. Se asume que la base móvil se controla enviando los comandos de velocidad deseados en forma de: velocidad x, velocidad y, velocidad theta.
- Requiere un láser plano montado en algún lugar de la base móvil. Este láser se utiliza para la construcción y localización de mapas.
- El navigation stack se desarrolló en un robot cuadrado, por lo que su rendimiento será mejor en robots que sean casi cuadrados o circulares. Funciona en robots de formas y tamaños arbitrarios, pero puede tener dificultades con robots rectangulares grandes en espacios estrechos como puertas.

Los requisitos no son difíciles de cumplir, ya que los tres han sido cumplidos en el apartado anterior. El primero es más un tema del planteamiento del robot y el segundo lo cumplimos al emplear un láser lidar. Por lo que respecta al tercero, si bien es verdad que el robot planteado no es del todo cuadrado, se puede aproximar su huella a un cuadrado y no se prevé que suponga ningún problema en cuanto al funcionamiento del navigation stack.

A nivel de software, lo más importante es, como cabría esperase, el correcto funcionamiento de los distintos elementos de ROS. Donde es sumamente importante tener muy claro el funcionamiento del paquete tf y de cómo funciona ROS a nivel general de lectura y transmisión de mensajes.

Dentro del paquete podemos encontrar un diagrama con los elementos que deberán estar presentes en el sistema si queremos que el navigation stack funcione correctamente:

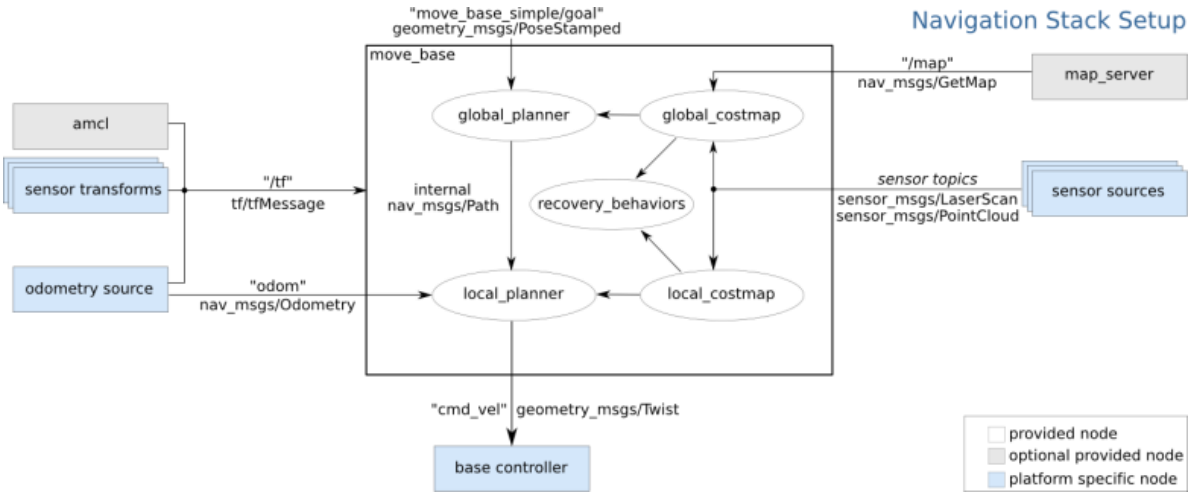


Ilustración 44 Diagrama de funcionamiento del navigation stack (Gill, 2018)

En el diagrama se aparecen tres tipos de elementos según un código de colores:

- Nodos proporcionados por el navigation stack, en blanco
- Nodos proporcionados opcionales (elementos opcionales del metapaquete, nodo encargado de generar un mapa, etc.), en gris
- Nodos específicos de la plataforma o robot (son los nodos que cada usuario debe crear o tener en su robot para que este pueda emplear el navigation stack), en azul

De los tres tipos de elementos, debemos centrarnos en los azules, que son los que no proporciona el navigation stack y los que definirán el comportamiento del robot. Por el contrario, los blancos podemos casi ignorarlos puesto que son los que el sistema nos está proporcionando y podemos considerarlos como una caja negra, únicamente teniendo que configurar los distintos parámetros que definen su funcionamiento. En el caso de los grises, no les prestaremos mucha atención puesto que el map_server ya lo tenemos gracias al paquete gmapping, y el amcl no es necesario emplearlo si no queremos.

En el diagrama también vemos cómo interactúan los nodos entre ellos, estando indicado el tópico y el mensaje o servicio que emplean para comunicarse. De este

modo podemos ver que mensajes deben ser publicados en qué tópicos y también cuales debemos leer en los nodos que creemos.

De los elementos azules en el diagrama, nos vamos a centrar en el base controller y el odometry source, puesto que el sensor sources ya lo tenemos en el sistema gracias al paquete rplidar y el sensor transforms también gracias a los paquetes que se mencionaron antes y se explicarán más adelante: robot_state_publisher y joint_state_publisher.

Como hemos creado anteriormente el paquete del proyecto con las dependencias de ROSCpp y ROSPy, no es necesario que escojamos el lenguaje de programación de los nodos todavía, pero sería una buena decisión empezar a considerar el lenguaje para así evitar futuros problemas de compatibilidades. Tanto el proceso de desarrollo de los nodos, como las consideraciones durante el mismo se explicarán en un futuro apartado. Por lo que respecta al diagrama, debemos fijarnos en los tópicos en los que publican/se suscriben para poder delimitar las características de los nodos.

El base controller se trata del cerebro, por así decirlo, del robot. Este nodo será el encargado de comunicar el sistema de ROS con el exterior empleando el tópico cmd_vel para recibir instrucciones de dirección del sistema. Por otro lado, el nodo de odometry source será el encargado de publicar la odometria computada a partir de los datos obtenidos de los sensores pertinentes (en nuestro caso, de los encoders de los motores), en el sistema para que se pueda realizar un seguimiento a tiempo real de la posición del robot en el espacio.

Para familiarizarse con el metapaquete hay una lista diversa y extensa de tutoriales que permiten ir entendiendo conceptos de forma gradual. Además, es posible crear el modelo funcional del paquete (nodos, parámetros, configuración, etc) siguiendo los tutoriales por lo que, pese a no entrar en detalle en el procedimiento de alguno de estos tutoriales, se hará un seguimiento general de los mismos puesto que gracias a estos es posible obtener el resultado final deseado del navigation stack.

De todos los tutoriales disponibles para familiarizarse con el paquete, el que mejor sigue el proceso para que el paquete sea funcional es el Setup and Configuration of the Navigation Stack on a Robot, y es el que se va a seguir en este apartado para explicar la configuración del paquete.

Primero, es necesario cumplir con los requisitos para poder empezar la configuración del paquete. Estos requerimientos son los que se han visto anteriormente en el diagrama y, pese a ser requerimientos, no es necesaria su presencia para configurar los parámetros. Obviamente, serán necesarios una vez se quiera poner en marcha la navegación del robot.

El primer paso del tutorial es crear un paquete donde se almacenará toda la configuración y los archivos .launch para el navigation stack. Este paquete dependerá de cualquier paquete utilizado para cumplir con los requisitos previamente mencionados. Para ello basta con usar el comando *catkin_create_pck* con el nombre del paquete y con las dependencias del mismo. Este paso se puede omitir si se planea emplear un paquete general para el robot, de este modo podemos reducir el número de paquetes y de nodos en el sistema. Por este motivo, omitiremos el paso de creación del paquete (puesto que ya hemos creado el paquete general del robot anteriormente) y nos centraremos en la creación de los archivos .launch los cuales nos permitirán ejecutar todos los nodos al mismo tiempo con sus respectivos parámetros.

Para crear los archivos .launch no necesitamos nada en particular, por supuesto, los parámetros que introduciremos deberán ser posteriormente probados y calibrados para que el robot funcione de forma correcta. Aun así, en la página del paquete hay una serie de plantillas que podemos emplear para poner los parámetros por defecto. En los anexos se pueden encontrar el código empleado para el archivo .launch empleado en el proyecto.

Una vez creado el .launch es momento de configurar los elementos dentro del navigation stack, para ello deberemos usar archivos con extensión .yaml lo cuales crearemos dentro de un nuevo directorio dentro del paquete robot_tfg llamado "cfg". Estos archivos son los que contendrán toda la configuración del paquete y los que deberán ser modificados en función de cómo se desee que se comporte el robot. Con estos parámetros configuraremos los 3 elementos principales del navigation stack:

- local_costmap
- global_costmap
- local_planner

Donde los costmaps son los mapas que el paquete emplea para saber dónde está el robot, siendo el global el que emplea para la posición general y el local el que emplea para el posicionamiento de obstáculos cercanos al robot, y el local planner es el encargado de computar los comandos de velocidad que se le dan al robot. Del mismo modo que con el archivo .launch, el paquete ofrece una serie de plantillas con parámetros por defecto las cuales se van a emplear para la creación de los archivos puesto que hay tantos parámetros y son tan diversos que explicarlos todos supondría un gasto enorme de tiempo. Además, la explicación de los parámetros y que hace cada uno se puede encontrar en la documentación de los respectivos paquetes: costmap_2d y base_local_planner.

Una vez creados los archivos de configuración el navigation stack estará listo para ser usado en cuanto estén todos los requisitos presentes en el sistema y se ejecute el .launch.

6.2.1.6. Modelo URDF

Como se ha mencionado repetidas veces, es necesario que haya publicado en el sistema un árbol de transformadas, y también se ha mencionado que para publicar ese árbol se emplearían dos paquetes: el robot_state_publisher y el joint_state_publisher. Para que estos dos paquetes funcionen de forma correcta se deben emplear junto con otro paquete, URDF.

El paquete urdf nos permite crear un modelo del robot con el que posteriormente se publicarán las transformadas, este paquete contiene un parser de C++ para el formato de descripción de robot unificado (*Unified Robot Description Format*) o URDF, que es un formato en XML para representar un modelo de robot. Del mismo modo que con el resto de paquetes, existen una gran variedad de tutoriales para comprender el funcionamiento del paquete. En este caso no se va a seguir ninguno en el desarrollo pero si que se va a explicar en detalle, con algunos ejemplos, el funcionamiento de este paquete puesto que es uno de los pilares principales del sistema.

Para empezar es necesario crear un nuevo paquete el cual llamaremos como el paquete principal seguido de "_description", este nuevo paquete tendrá las dependencias de urdf, robot_state_publisher y joint_state_publisher. De este modo tenemos el paquete robot_tfg_description, este paquete no tendrá ningún nodo puesto que lo único que nos importa es el modelo urdf, el cal se encontrará dentro de un directorio llamado urdf.

Para ayudarnos durante el proceso de creado del modelo, emplearemos RViz, un visualizador nativo de ROS que nos permitirá ver el modelo para así tener una idea de lo que estamos haciendo. Para poder ver el modelo modificaremos un archivo .launch de los tutoriales de urdf en el que cambiaremos la ruta del modelo para que lea el nuestro. Para lanzar el .launch emplearemos el siguiente comando en una ventana de la consola de comandos:

roslaunch robot_tfg_description robot_tfg_rviz.launch model:='$(find robot_tfg_description)/urdf/modelo.urdf

Una vez empezado el modelo podremos ir usando ese comando cada vez que guardemos el archivo para comprobar si lo que estamos desarrollando lo estamos haciendo de forma correcta.

Para empezar, podemos crear un modelo con geometrías básicas. Para ello, debemos crear los enlaces o "links" y las uniones o "joints".

```
<link name="base_link">
  <visual>
    <geometry>
      <box size="0.21 0.15 0.003"/>
    </geometry>
    <material name="blue"/>
  </visual>
</link>

<link name="right_front_wheel">
  <visual>
    <geometry>
      <cylinder length="0.035" radius="0.04"/>
    </geometry>
    <origin rpy="-1.57075 0 0" xyz="0 0 0"/>
    <material name="green"/>
  </visual>
</link>
```

Ilustración 45 Links de la base y las ruedas (Elaboración propia)

En la imagen podemos ver el código para crear dos elementos en el modelo: la base y una de las ruedas. Para poder orientarnos debemos saber que, para ROS, el robot se mueve hacia adelante en el eje X y que, por lo tanto, deberemos tener en cuenta la orientación del robot en ese aspecto.

Teniendo clara la orientación podemos crear la base del robot, la cual será el elemento que dejaremos fijo y que actuará como "padre" del resto (recordemos que tf funciona empleando una estructura de tipo árbol, por lo tanto un elemento podrá tener vario hijos, pero jamás más de un padre). Para ello crearemos una geometría simple con las dimensiones generales de la base del robot, además también podemos crear cuatro cilindros que harán de ruedas también con las dimensiones aproximadas a las que se van a emplear.

Una vez creados los elementos podemos dar paso a las joints. Estas son las que serán responsables de la unión de los dos elementos y son las que realmente importan al final, tanto que el link únicamente nos importa que esté ahí. Ni su geometría ni su material serán relevantes a no ser que se quieran añadir cálculos de inercia al modelo, aunque no es necesario. De todos modos, es útil desarrollar una representación visual fidedigna puesto que nos ayudara a visualizar el robot y a entender su funcionamiento.

```
<joint name="base_to_right_front_wheel" type="continuous">
  <parent link="base_link"/>
  <child link="right_front_wheel"/>
  <axis xyz="0 1 0"/>
  <origin xyz="0.105 -0.0925 0"/>
</joint>

<joint name="base_to_left_front_wheel" type="continuous">
  <parent link="base_link"/>
  <child link="left_front_wheel"/>
  <axis xyz="0 1 0"/>
  <origin xyz="0.105 0.0925 0"/>
</joint>

<joint name="base_to_right_rear_wheel" type="continuous">
  <parent link="base_link"/>
  <child link="right_rear_wheel"/>
  <axis xyz="0 1 0"/>
  <origin xyz="-0.105 -0.0925 0"/>
</joint>

<joint name="base_to_left_rear_wheel" type="continuous">
  <parent link="base_link"/>
  <child link="left_rear_wheel"/>
  <axis xyz="0 1 0"/>
  <origin xyz="-0.105 0.0925 0"/>
</joint>
```

Ilustración 46 Joints de la base y las ruedas (Elaboración propia)

Algo a tener en cuenta durante la creación de las joints es que la posición en las que las decidamos poner será la que mas adelante se publicará en el sistema, es decir, la joint es la que marca la posición general del elemento, mientras que la posición del elemento es relativa a la joint (esto es útil cuando se quiere que la joint no esté en el centro del elemento, como por ejemplo el láser lidar). Para crear la joint debemos especificar el padre y el hijo, asi como el tipo de joint en función del elemento que queramos unir.

Una vez hemos creado el diseño básico podemos guardar el archivo y emplear el comando mencionado anteriormente para poder visualizar en RViz lo que hemos creado.

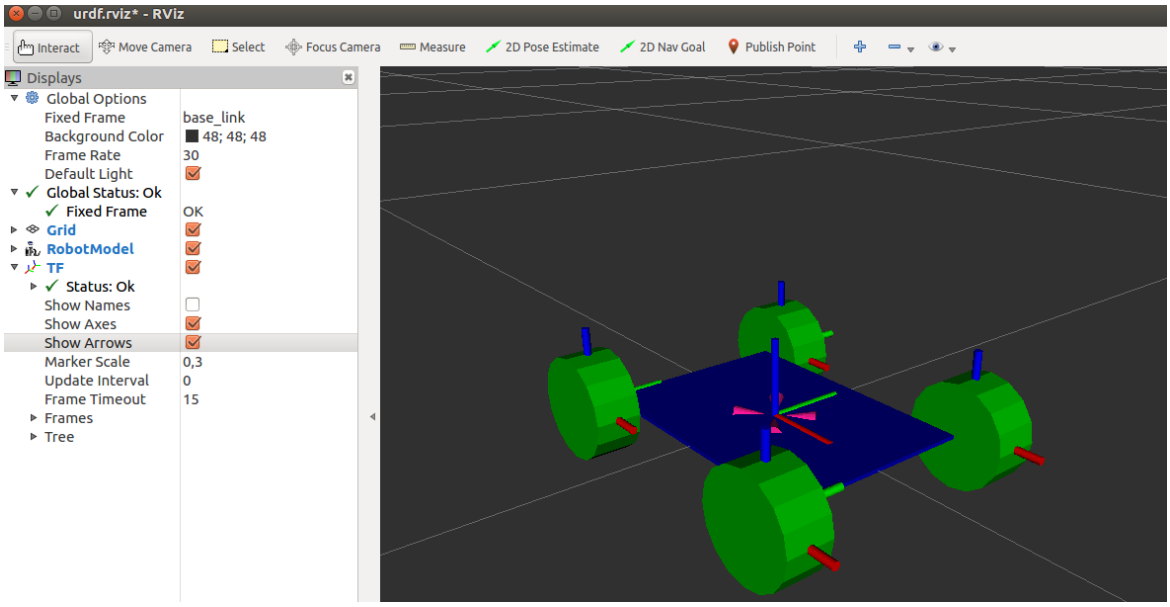


Ilustración 47 Visualización del modelo básico en RViz (Elaboración propia)

Como se ve en la imagen, nos queda un modelo algo crudo, pero que funcionará si las joints están en el sitio correcto. Además, podemos apreciar como los ejes de coordenadas de cada elemento están centrados en el elemento puesto que no hemos modificado donde están respecto a las joints (excepto las ruedas que hemos tenido que rotar 90º o 1.5705 radianes para que esté en la orientación correcta)

Asimismo, se pueden apreciar un poco unas flechas que van de las ruedas al centro de la base, esas son las relaciones que están creando las joints entre los elementos.

Para que los elementos aparezcan de diferente color se pueden crear materiales, los cuales no son más que colores que se les asignaran a los links. Para crearlos debemos introducir los valores del color en RGB y si queremos transparencia deberemos introducir un canal alfa también.

```
<material name="blue">
 <color rgba="0 0 0.8 1"/>
</material>
<material name="green">
 <color rgba="0 0.8 0 1"/>
</material>
<material name="red">
 <color rgba="0.8 0 0 1"/>
</material>
<material name="orange">
 <color rgba="0.9 0.6 0.01 1"/>
</material>
```

Ilustración 48 Código para la creación de materiales (Elaboración propia)

Como se ha dicho, el modelo queda algo crudo y simple. Por suerte podemos arreglar este problema empleando meshes. Los meshes no son más que archivos de elementos en 3d que nos permitirán añadir geometrías complejas al modelo. El paquete URDF acepta varias extensiones de archivo para los meshes, pero nos centraremos en 2 de ellas: .dae y .stl. Para obtener los meshes podemos emplear distintos programas de edición 3d como Inventor, Blender, FreeCAD, etc.

En este proyecto se han empleado distintos programas para obtener los meshes. Para la base se empleó Blender, un software gratuito de diseño 3D. No es un CAD/CAE, pues se centra en un diseño desde un punto de vista artístico, pero para crear meshes cumple bien su función. Para las ruedas delanteras se empleó FreeCAD, un software CAD gratuito. Para el resto de elementos se empleó Inventor.

Para añadir un mesh al modelo solo debemos crear un directorio con los meshes en su interior y luego substituir la geometría del link por un enlace al mesh.

```
<link name="base_link">
  <visual>
    <geometry>
      <mesh filename="package://robot_tfg_description/meshes/base.dae"/>
    </geometry>
    <material name="blue"/>
  </visual>
</link>

<link name="right_front_wheel">
  <visual>
    <geometry>
      <mesh filename="package://robot_tfg_description/meshes/rueda2.dae"/>
    </geometry>
    <origin rpy="-1.57075 0 0" xyz="0 -0.016 0"/>
    <material name="green"/>
  </visual>
</link>
```

Ilustración 49 Links con los meshes añadidos (Elaboración propia)

Hay que tener en cuenta que es posible que se deban hacer correcciones en la orientación de las piezas puesto que puede ser que al pasar del programa de edición 3d a ROS se giren los ejes.

Una vez añadidos los meshes podemos ver que tal queda el modelo.

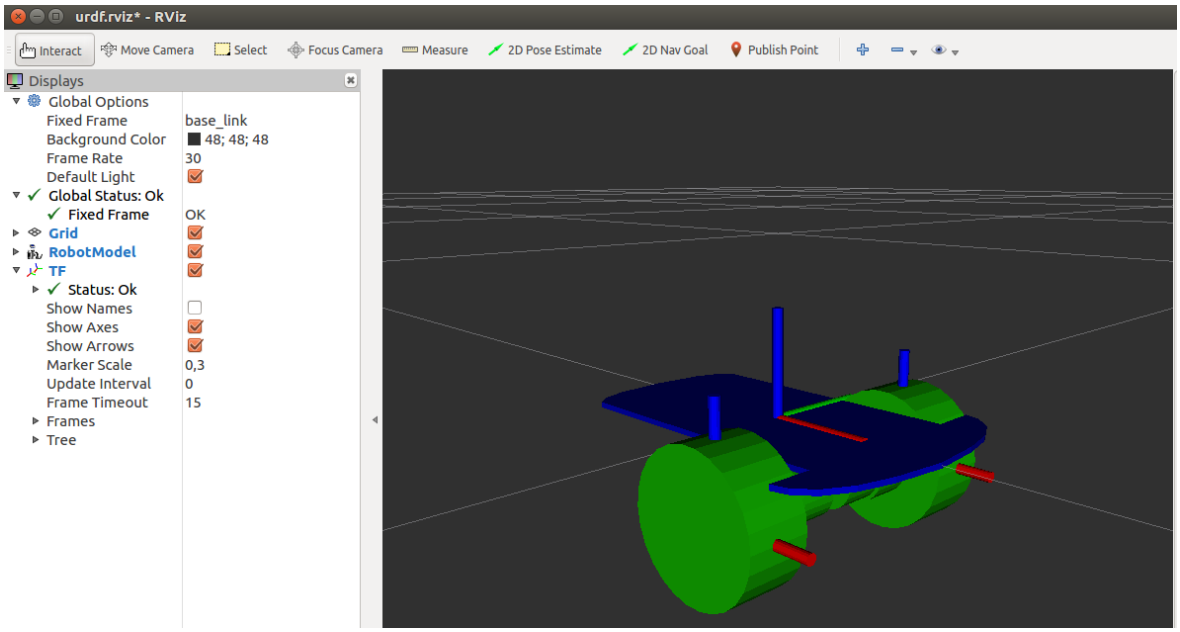


Ilustración 50 Modelo con meshes de la base y las ruedas (Elaboración propia)

Como se aprecia en la imagen, el modelo va cogiendo una mejor forma y va siendo más parecido a algo que podríamos encontrarnos en la realidad.

Una vez llegados a este punto no queda mucho más que hacer con el modelo más allá de repetir los pasos ya seguidos para añadir el resto de elementos al modelo: la rueda trasera, la segunda base, el lidar, la batería y la Raspberry Pi.

Hay que destacar que el modelo URDF se ha ido desarrollando a lo largo de todo el proyecto y que, pese a que se muestre a continuación el producto final, pasó bastante tiempo desde que se inició el modelo hasta que se fueron escogiendo todos los elementos del robot y pudieron ser añadidos al modelo.

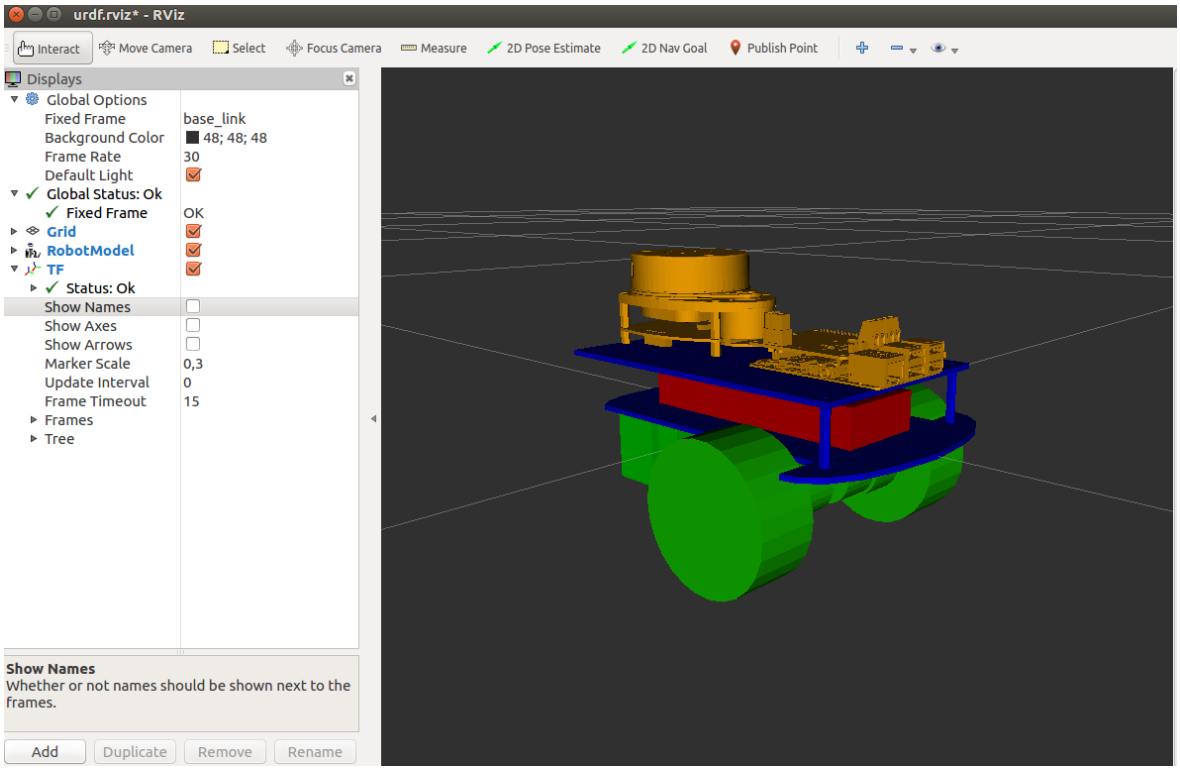


Ilustración 51 Modelo final con todos los meshes 1 (Elaboración propia)

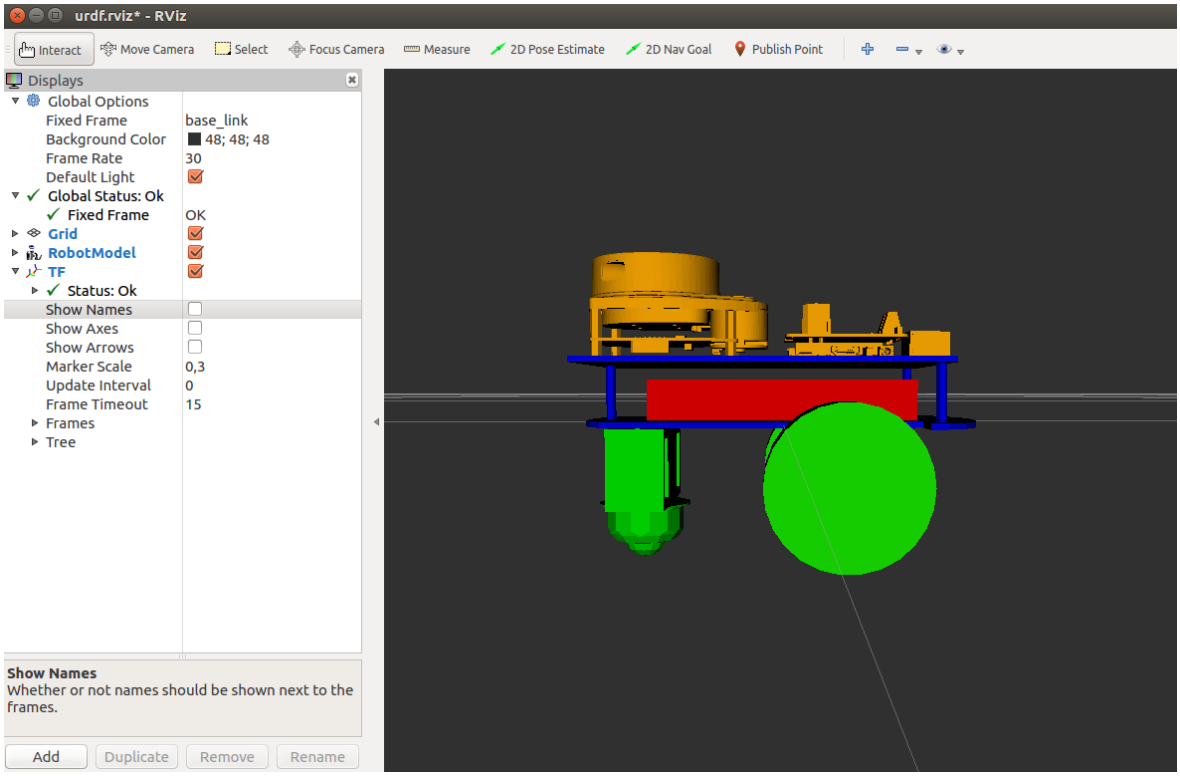


Ilustración 52 Modelo final con todos los meshes 2 (Elaboración propia)

Como podemos ver ahora sí que se ve algo mejor el modelo y podemos decir que es una representación fidedigna de lo que veríamos en la realidad. En las imágenes vemos cuatro tipos de elementos:

- Verdes: Elementos de motores (Las ruedas del robot)
- Azules: Elementos estructurales (Las bases sobre las que se asientan los elementos del robot)
- Rojos: Elementos de alimentación (La batería)
- Naranjas:
- Elementos electrónicos (El lidar y la Raspberry con su HAT)

6.2.1.7. Diseño del HAT de la Raspberry

Para que el robot funcione es necesaria una electrónica que nos permita controlar los motores que hemos escogido anteriormente y recibir información de los encoders que tienen los mismos.

Si nos guiamos por los antecedentes, podríamos emplear un Arduino para el control de los motores y la lectura de los encoders. Pero en este proyecto se ha decidido aprovechar al máximo el hecho de que estemos empleando una Raspberry y se ha decidido emplear los pines GPIO de la Raspberry para el control de los elementos externos.

Como estamos empleando una Raspberry como cerebro del proyecto, antes de diseñar una placa que contenga toda la electrónica y que esté de forma individual, podemos intentar conseguir un mejor integración de la placa en el proyecto.

La mejor opción en este caso sería desarrollar un HAT para la Raspberry y que este aloje toda la electrónica necesaria. De este modo tendremos un elemento que podremos acoplar a la Raspberry, y que ocupará poco espacio.

Para diseñar el HAT emplearemos el software de diseño electrónico KiCAD. Este software es gratuito y se adapta perfectamente a las necesidades del proyecto. Para poder diseñar el HAT tenemos varias opciones, primero podríamos buscar las especificaciones estándar de los HATs para así poder diseñar un elemento que cumpla con las medidas estándar. Por otro lado, podríamos buscar alguna plantilla y basarnos en esta para diseñar. Pero, gracias a que estemos empleando KiCAD, en el mismo programa tenemos tres plantillas para diseñar el HAT:

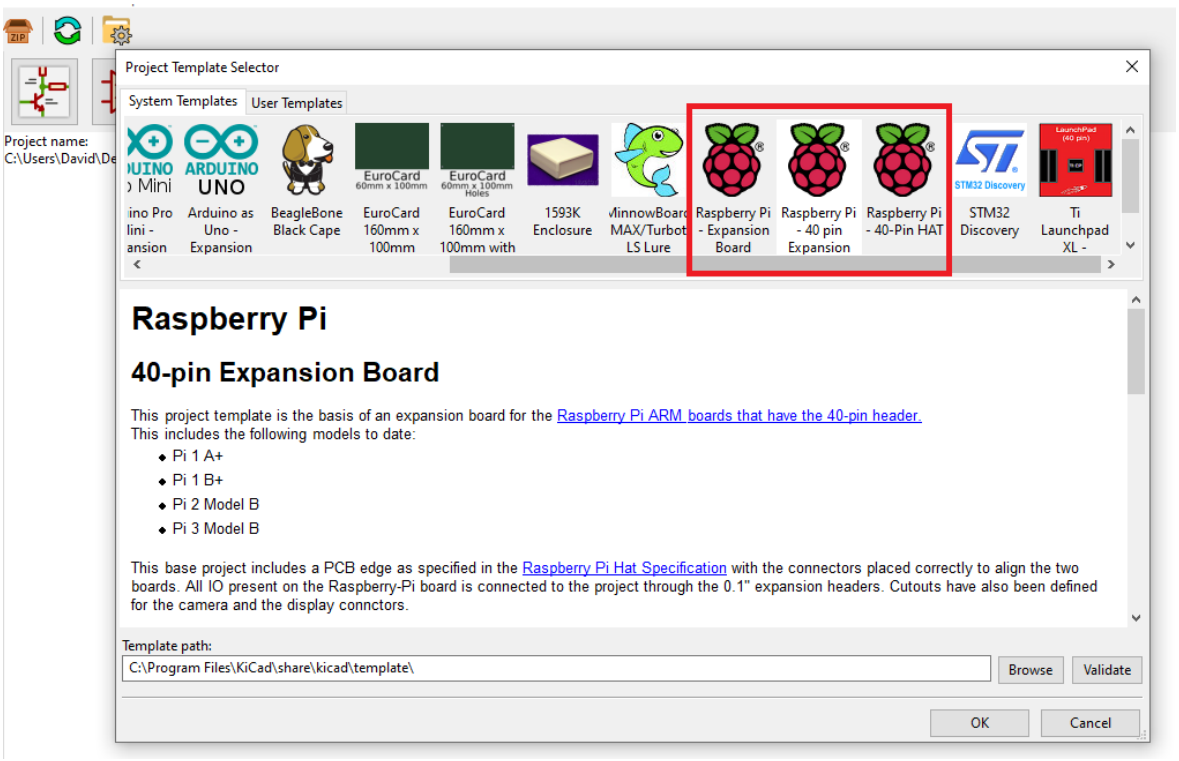


Ilustración 53 Plantillas para el HAT en KiCAD (Elaboración propia)

De la tres plantillas de las que disponemos, decidimos elegir la "Raspberry Pi – 40 Pin Expansion Board". No hay mucha diferencia respecto de las demás, pero se ha optado por esta porque nos da algo más de libertad en cuanto al diseño de la placa a nivel estructural (la placa como tal).

Ahora que ya tenemos la plantilla para la placa, debemos escoger qué vamos a poner encima de la misma. Para poder escoger los componentes que emplearemos debemos ver que componentes necesitamos en primer lugar.

Empleamos dos motores de corriente continua, si queremos que puedan girar en ambos sentidos, podemos emplear un puente en H para así conmutar sus sentidos de giro. Además, deberemos alimentar los motores desde el HAT, por lo que tendremos que añadir los conectores necesarios. Por otro lado, tenemos el lidar, el cual emplea una conexión tipo UART, pero en este caso el lidar trae una placa con una interfaz UART/USB con la que podemos conectarlo directamente a la Raspberry sin preocuparnos por el HAT. El otro tipo de información que leeremos es la de los encoders de los motores, para ello deberemos añadir los pertinentes conectores para así poder leer la información.

Para escoger el puente en H que emplearemos podemos considerar varios factores: eficiencia, tamaño, consumo, precio, etc. En este proyecto se ha decidido

optar por un acercamiento distinto al de buscar modelos hasta dar con uno que se adapte. En varios antecedentes se emplea un arduino con un shield de Adafruit, de modo que buscar las especificaciones del shield puede ser una buena opción para buscar componentes, puesto que podemos darles un voto de confianza a dichos componentes ya que son empleados por empresas de renombre.

Observando las especificaciones del shield de Adafruit, vemos que para la conmutación de los motores emplean un puente en H, específicamente un TBN6612FNG, el cual se trata de un controlador integrado para motores de DC con un transistor de salida en estructura LD MOS. El puente dispone de dos señales de entrada y se puede escoger entre cuatro modos para los motores: CW (clockwise o sentido horario), CCW (counterclockwise, o sentido anti horario), short brake y modo stop.

Si seguimos fijándonos en el shield, vemos que también emplean un driver para generar señales PWM y así controlar los puentes en H. El integrado se trata de un PCA9685. En nuestro caso, como estamos intentando aprovechar las capacidades de la Raspberry, decidimos no emplear este integrado puesto que podemos mandar señales PWM con los GPIO de la Raspberry.

Llegados a este punto, vemos que el único elemento de control que debemos añadir al HAT es el TBN6612FNG. Sabiendo esto podemos diseñar el resto de la placa para que el puente en H funcione de forma correcta.

Asimismo, debemos considerar en este punto como vamos a alimentar todos los componentes eléctricos del robot, pero más que el qué, debemos pensar en el cómo, es decir, se puede prever que se empleará una batería para alimentar todos los elementos, pero debemos decidir si alimentaremos la Raspberry con la batería y a partir de la misma el resto de elementos o emplearemos otro método. De este modo, se decide alimentar el HAT con la batería y desde este alimentar el resto de componentes. Para ello debemos añadir un conector específico al HAT, que en este caso se trata de un conector XT60. Para conectar los motores y los encoders al HAT emplearemos conectores genéricos puesto que no es necesario ningún conector en específico.

Para alimentar el robo se decide emplear una batería LiPo, en concreto una Turnigy 5000mAh 3S 20C LiPo Paquete con XT-60, este tipo de baterías son muy empleadas en vehículos RC y son altamente confiables. Por lo que respecta a la autonomía no hay problema puesto que el robot no está pensado para funcionar durante largos periodos de tiempo.

Como estamos alimentando el HAT directamente con la batería, se decide alimentar el resto de elementos desde el HAT. Para ello solo hay que alimentar los componentes que lleve el HAT y la Raspbery, puesto que el lidar va conectado a la Raspberry y no al HAT. Para poder alimentar una Raspberry desde el HAT es necesario alimentarla desde los pines de 5V y masa.

Con los conectores escogidos nos podemos centrar en el elemento principal de la placa, el TBN612FNG. Para saber los valores a los que funciona el integrado, debemos consultar el datasheet del mismo. En este vemos que a nivel de alimentación debemos tener en cuenta dos cosas. Primero, el chip se alimenta a una tensión de 5V. Segundo, el chip dispone de tres pines para la alimentación de los motores los cuales van de 2.5 a 13.5V. En nuestro caso deberemos conectar esos pines a 12V puesto que es a la tensión a la que funcionan los motores.

Como estamos conectado la batería al HAT, disponemos de una tensión de 11.1V, por lo que podemos alimentar los motores a esta tensión, pero para obtener los 5V deberemos emplear una fuente buck para pasar de los 11.1V de la batería a los 5V necesarios. Para agilizar el trabajo recurrimos al Webench Power Designer de Texas instruments, el cual nos da diseños de fuentes en función de unos parámetros que le introduzcamos.

Empleando el Webench obtenemos una gran variedad de diseños que se ada`tan a nuestro proyecto, de entre estos diseños decidimos escoger el siguiente.

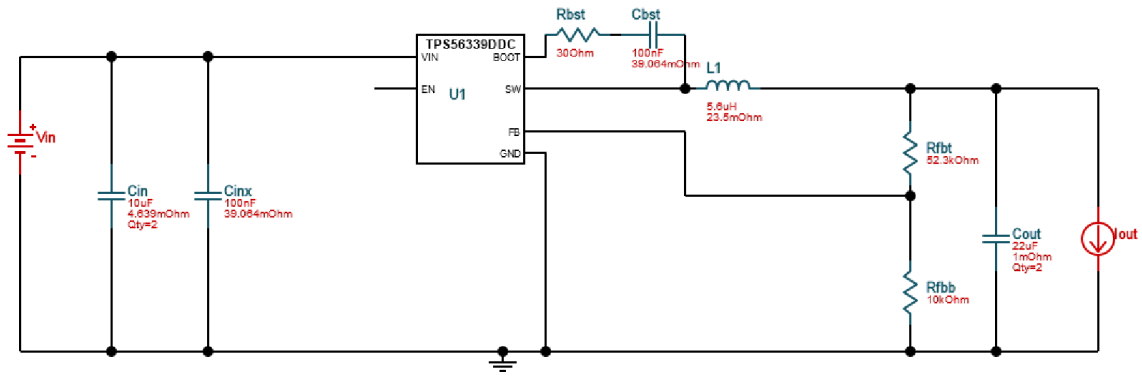


Ilustración 54 Diseño fuente de tensión 12V a 5V (Instruments, 2021)

En la imagen podemos ver que el corazón de la fuente es el TPS56339DDC, un convertidor síncrono buck de 3 A con un rango de tensiones de entrada de 4.5 V a 24V. En el diseño vemos que se ha dejado libre el pin del enable, pero como puede dar fallos si se deja en ese estado, se decide alimentarlo con una resistencia limitadora para que no nos ocasione problemas en un futuro.

Teniendo la fase de alimentación de la placa podemos dar paso al diseño del HAT. Como filosofía de diseño se intenta que los elementos no estén esparcidos por

toda la placa de forma que s en un futuro es necesario añadir algún elemento extra este pueda tener el espacio suficiente para que no interfiera con el resto de elementos.

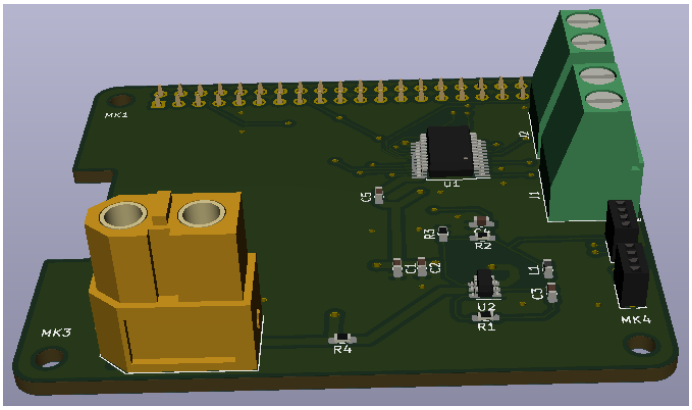


Ilustración 55 Vista isométrica del HAT (Elaboración propia)

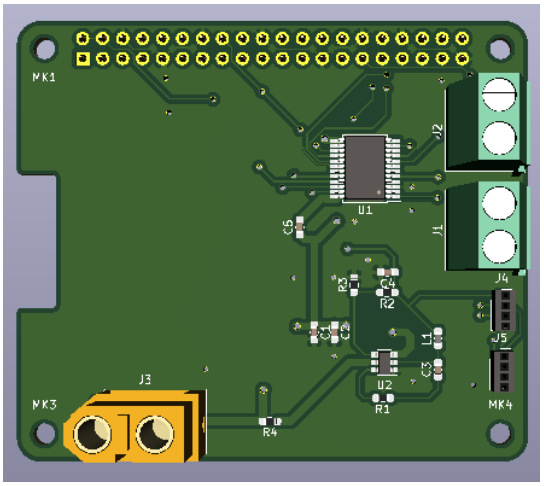


Ilustración 56 Vista superior del HAT (Elaboración propia)

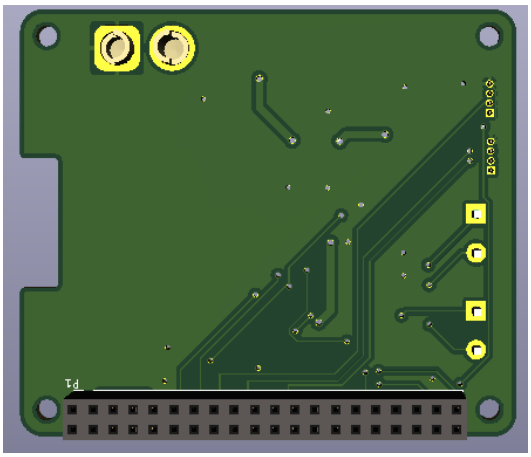


Ilustración 57 Vista inferior el HAT (Elaboración propia)

Como se ve en las imágenes, ha quedado bastante espacio vacío en la placa, el cual nos permitirá añadir más elementos en un futuro en caso de que fuese necesario. Además, se puede apreciar que se ha conseguido el objetivo de centralizar los elementos de modo que estén cerca unos de los otros.

6.2.1.8. Programación

Para realizar la programación deberemos escoger el lenguaje que emplearemos. Como se ha mencionado anteriormente, existen dos paquetes que implementan dos lenguajes de programación en ROS: ROSCpp y ROSPy, que implementan C++ y Python respectivamente. Se puede elegir el lenguaje que se va a emplear en función de distintos criterios, pero el de más peso suele ser la experiencia que se tenga con el lenguaje en sí. En este proyecto se decidió emplear C++, por ser el lenguaje con el que estaba más familiarizado, pero hay otros factores que podrían forzarnos a cambiar el lenguaje. De estos factores, el más importante es la Raspberry pi. Para poder controlar los pines GPIO de la Raspberry necesitamos emplear una librería. Existen diversas librerías que nos permiten controlar los pines tanto para C++ como para Python.

Por la parte de C++, existen diversas librerías: WiringPi, PigPIO, LIO, etc. De entre todas las opciones nos decidimos por PigPIO puesto que el resto presentan problemas de algún modo u otro, mientras que PigPIO parece ser la más estable de todas.

Por la parte de Python, hay una librería que destaca por encima del resto: RPi.GPIO. Por lo que, en caso de programar en Python se decidiría emplear esta librería.

Hay diversos elementos que debemos crear para que el paquete que hemos creado al inicio, robot_tfg, funcione correctamente. De estos elementos hay dos que son los más importantes, los nodos. Exceptuando estos, lo otro que debemos crear son archivos .launch y cambiar la configuración de generación del paquete en el CMakeLists. Tarea que hemos realizado antes con los .launch y por lo que respecta al CMakeLists, es fácil hacer los cambios para que funcione todo de forma correcta.

Se deciden crear dos nodos: base_controller y low_level_control. Con estos dos nodos publicaremos odometría en el sistema y controlaremos los elementos conectados a la Raspberry. Es importante destacar que sería posible hacer un único nodo que lo hiciese todo, pero separando los dos podemos facilitar cualquier futura modificación ya que en este caso un nodo exclusivamente controla los elementos conectados a la Raspberry y por lo tanto no se ha de suscribir a los tópicos que, por ejemplo, se ha de suscribir el otro nodo.

El base_controller se subscribe a un tópico llamado speed, que es publicado por el nodo de low_level_control. El mensaje es de tipo Vector3Stamped, que se compone de un mensaje tipo Vector3 y una marca de tiempo, y dentro van las velocidades lineales de los motores, así como un intervalo de tiempo. Este mensaje no está pensado para ser empleado de este modo, la finalidad del mensaje es representar un vector en el espacio con sus componentes xyz, pero como el mensaje se compone de 3 float64, es muy útil para pasar la información. Al final del día, el mensaje como tal no interpreta la información que contiene, solo se preocupa de que los tipos de dato que se le pasan sean los correctos. De modo que si un 0.25 representa una velocidad en m/s, o una coordenada x en el espacio; es irrelevante. Emplear el mensaje así no trae ninguna consecuencia negativa, de hecho es empleado de este modo en varios proyectos, porque al emplearlo se evitan tener que crear un mensaje nuevo para transmitir datos. Con las velocidades se genera una odometría empleando ecuaciones básicas y esta se publica en el sistema en un mensaje de tipo geometry_msgs/Odometry.

El low_level_control se suscribe al tópico cmd_vel, el cual le viene del Navigation Stack, y es donde vienen las instrucciones para el robot. Dentro de cmd_vel viene un mensaje del tipo geometry_msgs/Twist, el cual está compuesto por dos mensajes Vector3: Uno que representa la velocidad lineal, y otro que representa la velocidad angular. El nodo es, dicho de forma muy básica, una imitación de un arduino, de hecho en varios comentarios se ha puesto el equivalente en arduino para facilitar la comprensión del nodo. Más allá de esto, el nodo es bastante sencillo. Se extrae la velocidad de cmd_vel y se conmutan los motores en función de la velocidad deseada (en función de si se está leyendo un valor negativo o uno positivo), se leen los encoders y se publica en el tópico speed la información de las velocidades actuales de los motores en m/s.

Una vez creados los dos nodos es necesario modificar el archivo CMakeLists.txt. Este archivo es el responsable de que nuestro paquete sea creado correctamente cuando invoquemos el comando catkin_make en la terminal. Por suerte los cambios en este archivo no son complicados, puesto que lo único que debemos hacer es añadir lo siguiente al final del archivo.

```
add_executable(base_controller src/base_controller.cpp)
target_link_libraries(base_controller ${catkin_LIBRARIES})

add_executable(low_level_control src/low_level_control.cpp)
target_link_libraries(low_level_control ${catkin_LIBRARIES} -lpigpio)
```

Hay que remarcar que el –lpigpio es muy importante añadirlo, porque si no se añade ROS no sabrá que librería buscar para ese nodo y al hacer el catkin_make para compilar el paquete nos dará errores porque no reconocerá los elementos presentes en el código que pertenezcan a la librería en cuestión. Debido a esto se estuvo a punto de cambiar el lenguaje de programación que se iba a emplear en el proyecto porque la librería de control de los pines GPIO de la Raspberry para C++ no funcionaba, mientras que la librería para Python sí que funcionaba sin añadir nada al CMakeLists.

Una vez creados los nodos, y modificado el CMakeLists, podemos crear el último elemento necesario para el paquete, el .launch que lanzará todos los nodos necesarios para que funcione todo. Aunque lo estemos poniendo como necesario, en realidad no lo es. Pero es muy útil puesto que de este modo podemos ejecutar un único archivo, mientras que, por el contrario deberíamos ejecutar cada nodo por separado.

En el archivo que se ha creado se llama robot_tfg.launch y es responsable de lanzar los nodos de rplidar, gmapping, navigation stack, base_contoller, low_level_control, robot_state_publisher y joint_state_publisher. Con los dos últimos antes mencionados, además llama al paquete robot_tfg_description creado en el apartado URDF, el cual emplea el modelo creado para publicar el árbol de tf.

7. CONCLUSIONES

Desde el inicio del proyecto el objetivo principal ha sido familiarizarse con el entorno de ROS y explorar la posibilidad de desarrollar un proyecto que se adapte a la filosofía de trabajo y a las diversas características propias de ROS.

Para ello se dedicó una importante cantidad de tiempo al inicio del proyecto a familiarizarse con ROS, lo cual se compuso de horas y horas de lectura de la wiki oficial de ROS, visualización de videos explicativos de la plataforma e investigación de proyectos que emplean ROS.

Gracias al tiempo dedicado se pudo desarrollar satisfactoriamente un paquete de ROS que es posible emplear con los elementos seleccionados y que, con los ajustes necesarios, podría funcionar sin ningún problema.

El proyecto ha ido cambiando en gran medida según avanzaba el desarrollo, en un principio se planteó la idea de integrar ROS en el microprocesador esp32, de este modo se podría emplear el IDE de arduino para programar los nodos del proyecto. Desgraciadamente esta opción no pudo llevarse a cabo por lo que se tuvieron que buscar alternativas.

Posteriormente, se decidió que se emplearía una Raspberry puesto que era el elemento que gran parte de los proyectos más parecidos a este empleaban. Aun así, tras este punto el proyecto también vio algunos cambios.

En un inicio de planteaba la posibilidad de utilizar un chasis de un coche RC y adaptarlo para que al añadirle un sensor de posicionamiento, como por ejemplo el lidar, Kinect, etc. El robot pudiese navegar de forma autónoma, pero esta decisión traía consigo problemas que añadían una complejidad al proyecto la cual no era deseada. Para empezar, si se empleaba el chasis de un coche RC nos limitábamos a un modelo de giro tipo Ackerman, lo cual complicaba la implementación del navigation stack, puesto que está orientado al modelo de cinemática diferencial. Por otro lado, era necesario realizar una adaptación de los elementos presentes en el chasis. Para empezar, se necesitaba poder administrar la energía del robot, como el chasis ya portaba una batería sería necesario poder conectar de algún modo este al resto del robot, además era necesario añadir algún elemento que aportase feedback de la posición del robot para así poder generar una odometría. Este último problema resultaba ser especialmente complejo puesto que ROS es algo especial en cuanto a la odometría, siendo el mínimo básico una odometría creada a partir de velocidades obtenidas a partir de encoders. Se planteó la posibilidad de añadir IMUs, pero tras

consultar la comunidad de ROS se decidió desestimar la idea, puesto que era sabido que las IMUs no eran fuentes confiables de datos para generar la odometría. También se presentó la opción de añadir encoders a las ruedas del chasis, pero también se descartó la idea debido al limitado espacio para poder añadir los elementos. Tras todos estos problemas se decidió optar por un modelo de robot con cinemática diferencial, puesto que no presentaba tantos problemas como los que estaba presentando el modelo en ese momento.

Tras cambiar la estructura del robot el proyecto avanzó sin ningún percance y pudo ser completado de forma satisfactoria.

Referente a los objetivos planteados en el inicio del proyecto, se han completado en gran medida. Como se ha mencionado antes, el objetivo principal era la familiarización con ROS, lo cual se ha conseguido. Por otro lado, si bien es verdad que en el robot final no se ha incluido arduino, se intentó realizar una implementación y, pese a que no se consiguiera en un final, se planté la idea y se intentó en la medida de lo posible.

Algunos aspectos que se podrían mejorar en el proyecto en un futuro sería una mejor implementación de los distintos elementos del navigation stack, puesto que ran parte de los parámetros se calibran a base de hacer pruebas y mediciones empíricas, las cuales no ha sido posibles en este proyecto debido a la limitación en cuanto a la obtención del material necesario.

8. BIBLIOGRAFÍA

Open Robotics. (2013). *About ROS* Recuperado de https://www.ros.org/about-ros/

Stanford. (2010). *Personal Robotics Program* Recuperado de http://personalrobotics.stanford.edu/

WGarage. (2014). *Willow Garage* Recuperado de http://www.willowgarage.com/

Bernier, C. (2013). *Collaborative Robot Series : PR2 from Willow Garage* Recuperado de https://blog.robotiq.com/bid/65419/Collaborative-Robot-Series-PR2-from-Willow-Garage

Dattalo, A. (2018). *Introduction* Recuperado de http://wiki.ros.org/ROS/Introduction

Open Robotics (2013). *ROS History* Recuperado de https://www.ros.org/history/

Koening, N. (2012). *OSRF Launch* Recuperado de https://www.osrfoundation.org/osrf-launch/index.html

kwc (2012). *Thanks for a great ROSCon 2012!* Recuperado de https://www.ros.org/news/2012/05/thanks-for-a-great-roscon-2012.html

ROS2 (2019). *ROS 2 Documentation* Recuperado de https://index.ros.org/doc/ros2/

Mazzari, V. (2019). *ROS vs ROS2* Recuperado de https://blog.generationrobots.com/en/ros-vs-ros2/

Gerkey, B. (2007). *Why ROS 2?* Recuperado de https://design.ros2.org/articles/why_ros2.html

FIWARE (2020). *GETTING STARTED WITH MICRO-ROS: CORE AND ADVANCED TUTORIALS* Recuperado de https://www.fiware.org/2020/06/16/getting-started-with-micro-ros-core-and-advanced-tutorials/

Micro-ROS (2020). *Micro-ROS* Recuperado de https://micro.ros.org/

Lange, R. (2018). *Features and Architecture* Recuperado de https://micro-ros.github.io//docs/overview/features/

Baran, R. (2018). *Nox_robot* Recuperado de https://github.com/RBinsonB/Nox_robot#readme

Baran, R. (2018). *Nox - A House Wandering Robot (ROS)* Recuperado de https://create.arduino.cc/projecthub/robinb/nox-a-house-wandering-robot-ros-652315

Jik Cha, S. (2015). *My Personal Robotic Companion* Recuperado de https://sungjik.wordpress.com/2015/09/28/my_personal_robotic_companion/

Gurny, S. (2012). *Shaky - A Wall Following and Obstacle Avoidance Robot* Recuperado de https://hackaday.io/project/28646-wall-following-and-obstacle-avoidance

TurtleBot3 (2018). *TurtleBot3* Recuperado de https://emanual.robotis.com/docs/en/platform/turtlebot3/overview/

TurtleBot (2020). *TurtleBot* Recuperado de http://wiki.ros.org/Robots/TurtleBot

Aaron, M. (2014). *Concepts* Recuperado de http://wiki.ros.org/ROS/Concepts

Saito, I. (2019). *Packages* Recuperado de http://wiki.ros.org/Packages

Weaver, J. (2014). *Metapackages* Recuperado de http://wiki.ros.org/Metapackages

Belanger, M. (2019). *Package.xml* Recuperado de http://wiki.ros.org/catkin/package.xml

Hendrix, A. (2019). *msg* Recuperado de http://wiki.ros.org/msg

Saito, I. (2017). *srv* Recuperado de http://wiki.ros.org/srv

Oladepo, H. (2018). *Nodes* Recuperado de http://wiki.ros.org/Nodes

Yanqing Wu (2018). *Master* Recuperado de http://wiki.ros.org/Master

Miller, B. (2018). *Parameter Server* Recuperado de http://wiki.ros.org/Parameter%20Server

Kurzaj, D. (2016). *Messages* Recuperado de http://wiki.ros.org/Messages

Foote, T. (2019). *Topics* Recuperado de http://wiki.ros.org/Topics

Koubaa, A. (2019). *Services* Recuperado de http://wiki.ros.org/Services

Staples, G. (2020). *Bags* Recuperado de http://wiki.ros.org/Bags

Baranov, I. (2014). *ROS 101: INTRO TO THE ROBOT OPERATING SYSTEM* Recuperado de https://clearpathrobotics.com/blog/2014/01/how-to-guide-ros-101/

Staples, G. (2020). *Distributions* Recuperado de http://wiki.ros.org/Distributions

Foote, T. (2014). *Repositories* Recuperado de http://wiki.ros.org/Repositories

ROS2 (2018). *About Different Middleware Vendors* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-Different-Middleware-Vendors/

ROS2 (2016). *Concepts* Recuperado de https://index.ros.org/doc/ros2/Concepts /

ROS2 (2016). *Discovery* Recuperado de https://index.ros.org/doc/ros2/Concepts/#discovery

ROS2 (2020). *About Quality of Service Settings: Overview* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/#overview

ROS2 (2020). *About Quality of Service Settings: QoS Policies* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/#qos-policies

ROS2 (2020). *About Quality of Service Settings: QoS Profiles* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-Quality-of-Service-Settings/#qos-profiles

ROS2 (2019). *About ROS Interfaces* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-ROS-Interfaces/

ROS2 (2019). *About ROS Interfaces: Background* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-ROS-Interfaces/#background

ROS2 (2019). *About ROS Interfaces: New Features in ROS 2 Interfaces* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-ROS-Interfaces/#new-features-in-ros-2-interfaces

ROS2 (2016). *About ROS 2 Client Libraries* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-ROS-2-Client-Libraries/

ROS2 (2020). *About ROS 2 Parameters* Recuperado de https://index.ros.org/doc/ros2/Concepts/About-ROS-2-Parameters/

ROS2 (2020). *Migrating YAML parameter files from ROS 1 to ROS 2* Recuperado de https://index.ros.org/doc/ros2/Tutorials/Parameters-YAML-files-migration-guide/#yaml-ros1-ros2

Micro-ROS (2020). *Features and Architecture* Recuperado de https://micro.ros.org//docs/overview/features/

Lange, R. (2020). *Transports and Data Links* Recuperado de https://micro-ros.github.io/docs/overview/transports/

Lange, R. (2019). *Micro XRCE-DDS* Recuperado de https://micro-ros.github.io/docs/concepts/middleware/Micro_XRCE-DDS/

Lange, R. (2020). *DDS-XRCE, MQTT & IoT* Recuperado de https://micro-ros.github.io/docs/concepts/middleware/IoT/

Lange, R. (2019). *Introduction to Client Library* Recuperado de https://micro-ros.github.io//docs/concepts/client_library/

Mathworks (2020). *What Is SLAM? 3 things you need to know* Recuperado de https://www.mathworks.com/discovery/slam.html

Gerkey, B. (2017). *Gmapping* Recuperado de http://wiki.ros.org/gmapping

Kohlbrecher, S. y Meyer, J. (2014). *Hector SLAM* Recuperado de http://wiki.ros.org/hector_slam

Makarenko, A., Williams, S., Grocholsky, B., Hugh F. y Durrant-Whyte (2002) *Information Based Adaptive Robotic Exploration*

Liu, J., Sun, Q., Fan, Z. y Jia, Y. (2018) *TOF Lidar Development in Autonomous Vehicle*

Piatek, S. (2017) *LiDAR and Other Techniques: Measuring Distance with Light for Automotive Industry*

Birks, D. (2020). *Brushed DC Motors and How to Drive Them* Recuperado de https://www.diodes.com/design/support/technical-articles/driving-brushed-dc-motors/

Anusha P. (2015). *DC Motor* Recuperado de https://www.electronicshub.org/dc-motor/

PNGEGG (2020). *Regla de la mano izquierda de Fleming* Recuperado de https://www.pngegg.com/es/png-nghxr

Schweber, B. (2017). *Don't Ignore the Humble Brushed DC Motor* Recuperado de https://www.mouser.es/applications/dont-ignore-the-brushed-dc-motor/

Dynapar (2007). *Motor Encoder Overview* Recuperado de https://www.dynapar.com/technology/encoder_basics/motor_encoders/

WhitePaper (2016). *The Basics of How an Encoder Works* Recuperado de https://www.encoder.com/wp2011-basics-how-an-encoder-works

Arrow (2017). *Optical Encoders* Recuperado de https://www.arrow.com/en/categories/electromechanical-encoders/optical-encoders

CLR (2017). *Types of encoders and their applications in relation to motors* Recuperado de https://clr.es/blog/en/types-of-encoders-and-their-applications-in-relation-to-motors/

Celeramotion (2018). *What is an Absolute Encoder?* Recuperado de https://www.celeramotion.com/zettlex/absolute-encoder/

Ingmecafenix (2017). *Encoder ¿cómo funciona? y sus tipos* Recuperado de https://www.ingmecafenix.com/automatizacion/encoder/

Dejan (2017). *Arduino DC Motor Control Tutorial – L298N | PWM | H-Bridge* Recuperado de https://howtomechatronics.com/tutorials/arduino/arduino-dc-motor-control-tutorial-l298n-pwm-h-bridge/

Tantos, A. (2011). *H-Bridges – the Basics* Recuperado de https://www.modularcircuits.com/blog/articles/h-bridge-secrets/h-bridges-the-basics/

George, L. (2018). *DC Motor Driving using H Bridge* Recuperado de https://electrosome.com/dc-motor-driving-using-h-bridge/

Øyvind, N. (2018). *What Is an H-Bridge?* Recuperado de https://www.build-electronic-circuits.com/h-bridge/

Developer.android (2019). *PWM* Recuperado de https://developer.android.com/things/sdk/pio/pwm

AspenCore (2014). *Pulse Width Modulation* Recuperado de https://www.electronics-tutorials.ws/blog/pulse-width-modulation.html

Jordandee (2013). *Pulse Width Modulation* Recuperado de https://learn.sparkfun.com/tutorials/pulse-width-modulation/all

Virtual Amrita Laboratories (2015). *Li-Po Battery* Recuperado de https://vlab.amrita.edu/?sub=77&brch=270&sim=1631&cnt=1

Salt, J. (2020). *RC LiPo Batteries How to get the most life & fun out of them!* Recuperado de https://www.rchelicopterfun.com/lipo-batteries.html

Energy, B. (2019). *The working principle and charging method of polymer lithium battery* Recuperado de http://www.benzoenergy.com/blog/post/the-working-principle-and-charging-method-of-polymer-lithium-battery.html

DNK (2019). *Lithium Polymer Battery Complete Guide* Recuperado de https://www.dnkpower.com/lithium-polymer-battery-guide/

Mobus (2016). *¿Qué es una batería LiPo?* Recuperado de https://mobus.es/blog/que-es-una-bateria-lipo/

Erle Robotics S.L (2014). *Erle-Copter Gitbook, 3.1.1 LiPo Batteries* Recuperado de https://erlerobotics.gitbooks.io/erle-robotics-erle-copter/content/en/safety/lipo.html

Schneider, B. (2012). *A Guide to Understanding LiPo Batteries* Recuperado de https://rogershobbycenter.com/lipoguide

Piñera-García, J., Amigó-Vega, J., Concepción-Álvarez, J., Casimiro-Martínez, R., Fernández-Rodríguez, A. y Casanovas-Perer, M. (2013). *Diseño de un robot móvil con modelo cinemático Ackermann*

Valencia, V., Johnny, A., Montoya, O. y Hernando, L. (2009). *Modelo cinemático de un robot móvil tipo diferencial y navegación a partir de la estimación odométrica*

Ttuadvancedrobotics (2010). *A Guide to Understanding LiPo* Recuperado de: *http://ttuadvancedrobotics.wikidot.com/odometry*

Ben-Ari, M. y Mondada, F. (2018). *Elements of Robotics, Chapter 5: Robotic Motion and Odometry*

Edwin O. (2004). *A Primer on Odometry and Motor Control*

Texas Instruments. (2021). *Webench power designer* Recuperado de https://webench.ti.com/power-designer/switching-regulator/select

Slamtec. (2019). *RPLidar A1M8 - Kit de Desarrollo de Escáner Láser de 360 Grados* Recuperado de https://www.robotshop.com/es/es/rplidar-a1m8-kit-desarrollo-escaner-laser-360-grados.html

ViewTek. (2018). *CR0010 - 4WD Robot Kit de chasis para Coche Inteligente de 4 Ruedas con Encoder de Velocidad para DIY - Arduino Smart Car Chassis Kit - Tacómetros de 2 velocidades - Fast Speed - DYI KIT para Arduino* Recuperado de https://www.amazon.es/ViewTek-CR0010-Smart-Chassis-Arduino/dp/B07797PHMY/ref=sr_1_17?__mk_es_ES=%C3%85M%C3%85%C5%BD%C3%95%C3%91&dchild=1&keywords=diy+robot+car&qid=1606909883&sr=8-17

CHIHAI. (2017). *12V 320rpm/12V 107rpm/6V 160rpm DC Gear Motor Encoder Motor with Mounting Bracket and Wheel - 12V 107RPM* Recuperado de https://www.banggood.com/12V-320rpm12V-107rpm6V-160rpm-DC-Gear-Motor-Encoder-Motor-with-Mounting-Bracket-and-Wheel-p-1059966.html?rmmds=detail-left-hotproducts&cur_warehouse=CN&ID=522022

Omnitrack. (2019). *LF25 – Standard Materials* Recuperado de https://www.omnitrack.com/es/producto/lf25-standard-materials/

Gill, J. (2018). *Setup and Configuration of the Navigation Stack on a Robot* Recuperado de http://wiki.ros.org/navigation/Tutorials/RobotSetup

Relación de documentos

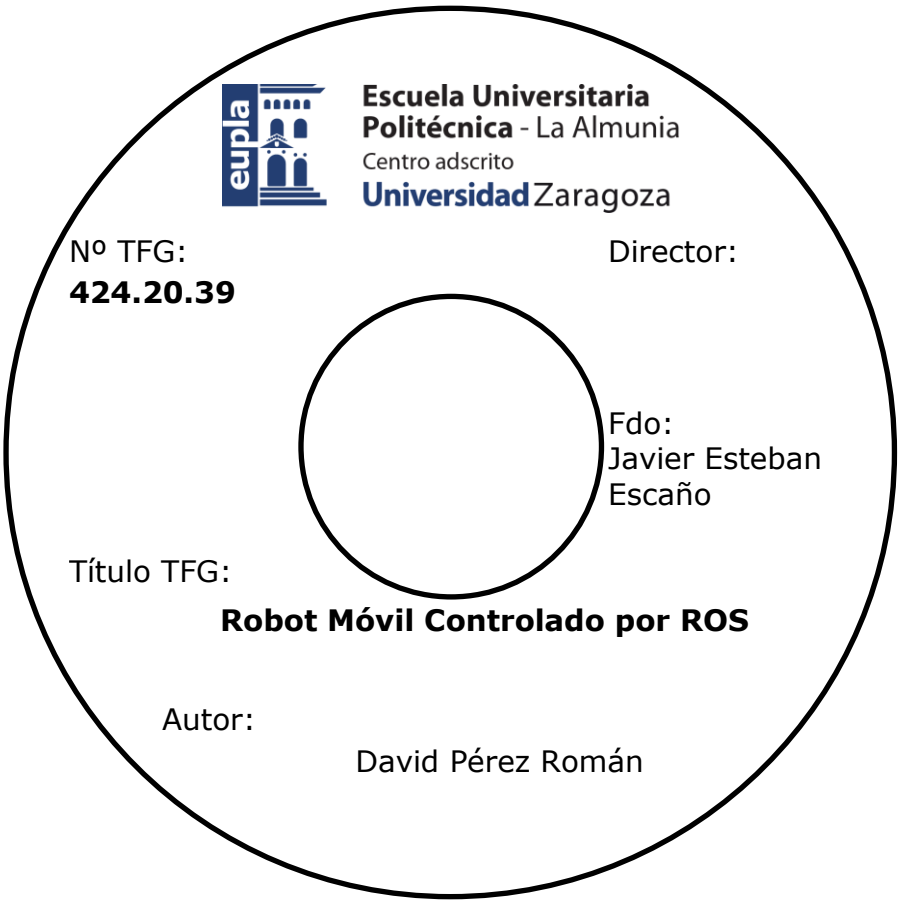
(X) Memoria ..... 95 páginas

(_) Anexos .....157 páginas

La Almunia, a dd de mm de 2021

Firmado: David Pérez Román.

Etiquetas para CD/DVD



Escuela Universitaria
Politécnica - La Almunia
Centro adscrito
Universidad Zaragoza

ESCUELA UNIVERSITARIA POLITÉCNICA
DE LA ALMUNIA DE DOÑA GODINA (ZARAGOZA)

Robot Móvil Controlado por ROS

Mobile Robot Powered by ROS

424.20.39

Autor: David Pérez Román

Director: Javier Esteban Escaño

Fecha: 22/06/2021