UNIVERSIDAD AUTÓNOMA DE MADRID

ESCUELA POLITÉCNICA SUPERIOR

Grado en Ingeniería Informática

TRABAJO FIN DE GRADO

Desarrollo de algoritmos para la teleoperación y navegación autónoma de un mini-robot

Javier Bernal Sánchez
Tutor: Juan Jesús Roldán Gómez
Ponente: Iván Cantador Gutiérrez

Junio 2021

Desarrollo de algoritmos para la teleoperación y navegación autónoma de un mini-robot

AUTOR: Javier Bernal Sánchez
TUTOR: Juan Jesús Roldán Gómez

Departamento de Ingeniería Informática
Escuela Politécnica Superior
Universidad Autónoma de Madrid
Junio de 2021

Resumen (castellano)

La robótica es una rama muy compleja de la ingeniería que requiere de experiencia en numerosos campos. Por esta razón, hay dos circunstancias que son clave para facilitar el camino a aquellas personas que deseen dar sus primeros pasos y aprender esta ciencia que combina varias disciplinas tecnológicas: la existencia de entornos de simulación y pruebas de bajo coste y la posibilidad de reutilizar en un proyecto aquellos algoritmos y herramientas que han sido implementados por otros desarrolladores previamente.

El objetivo de este Trabajo de Fin de Grado es la integración de un mini-robot terrestre con un PC con el sistema operativo Linux y la arquitectura ROS, extendiendo así las funcionalidades del robot en un *framework open source* y generando un entorno que pueda ser utilizado en trabajos posteriores. De esta manera, se permite el desarrollo de algoritmos de teleoperación y navegación autónoma, que se ejecutan en el PC y no sufren las limitaciones de cálculo del propio mini-robot. El resultado es un entorno de pruebas intermedio entre la simulación y el despliegue en un robot de mayor coste, que facilita la aplicación de algoritmos desarrollados a propósito para este sistema o proporcionados por la comunidad robótica.

Para ello se ha utilizado el robot LEGO® MINDSTORMS® NXT, el cual recibe las órdenes del PC a través del protocolo Bluetooth y a su vez, el NXT envía los datos leídos por cada uno de sus sensores. El proyecto se ha desarrollado en el lenguaje Python, integrando el módulo NXT-Python con la arquitectura de ROS. Con objeto de probar la integración, se han programado un algoritmo de teleoperación y otro de navegación autónoma, además de otros algoritmos de prueba.

En este documento se explican el diseño y la arquitectura del sistema, los aspectos técnicos empleados en el desarrollo, las pruebas realizadas y los resultados obtenidos en cada una de ellas, así como las posibles utilidades y trabajos futuros para los que nuestro proyecto puede ser un buen punto de partida.

Palabras clave (castellano)

ROS, Sistema Operativo Robótico, LEGO, NXT, mini-robot, robot autónomo, teleoperación, Python, Bluetooth

Abstract (English)

Robotics is a very complex branch of engineering that requires experience in many areas. For this reason, there are two circumstances that are key to pave the way for those who wish to take their first steps and learn this science that combines several technological disciplines: the existence of low-cost simulation and test environments and the possibility of reusing in a project those algorithms and tools that have been implemented by other developers previously.

The objective of this Bachelor's Degree Final Project is the integration of a mini-robot with a PC with the Linux operating system and the ROS architecture, to extend the robot's functionalities in an open source framework and generating an environment that can be used in future projects. In this way, the development of autonomous teleoperation and navigation algorithms is allowed, which are launched on the PC and do not suffer from the calculation limitations of the mini-robot. The result is a test environment between simulation and deployment on more expensive robot, which facilitates the application of algorithms developed purposely for this system or provided by the robotics community.

To do this, we have used the LEGO® MINDSTORMS® NXT robot, which receives orders from the PC through the Bluetooth protocol and at the same time, the NXT sends the data read by each of its sensors. The project has been developed in the Python language, integrating the NXT-Python module with the ROS architecture. In order to test the integration, a teleoperation algorithm and an autonomous navigation algorithm have been programmed, in addition to other test algorithms.

This document explains the design and architecture of the system, the technical aspects used in the development, the tests passed and the results achieved in each one of them, as well as the possible utilities and future work for which our thesis can be a good starting point.

Keywords (English)

ROS, Robot Operating System, LEGO, NXT, mini-robot, autonomous robot, teleoperation, Python, Bluetooth

Agradecimientos

En primer lugar, quiero dar las gracias a mi tutor, Juan Jesús, por darme la oportunidad de conocer y adentrarme en este mundo tan complejo y enorme que es la robótica. Mil gracias por la ayuda y las ideas aportadas durante todo el proyecto, pero más aún por la paciencia, por comprender la difícil situación laboral y personal vivida el último año.

Quiero agradecer a todos los profesores que he tenido, tanto en la universidad como fuera de ella, por haber fomentado, en mayor o menor medida, el desarrollo de mi curiosidad. También a la Universidad Autónoma de Madrid por todos los recursos, tanto físicos como humanos, por abrirme las puertas al mundo laboral y cruzar en mi camino a todas esas personas que me han formado como profesional y como persona, realizando un trabajo que nunca se podrá valorar lo suficiente.

Gracias a mis amigos, compañeros de universidad y de residencia, por toda la ayuda y por acompañarme en este camino. Hemos compartido muchos días de biblioteca, de salas de estudio, de laboratorio (y también muchísimas risas en nuestros "descansos" de varias horas), habéis sido una motivación y un apoyo constante, estoy seguro de que sin vosotros estas líneas no existirían.

Por último, y no por ello menos importante, gracias a mi familia, me habéis dedicado todo vuestro tiempo, esfuerzo y recursos con tal de educarme y formarme lo mejor posible para afrontar la vida. Gracias por confiar en mí en todo momento, incluso cuando yo no tuve las fuerzas suficientes para hacerlo.

INDICE DE CONTENIDOS

1 Introducción ... 1
1.1 Motivación ... 1
1.2 Objetivos ... 1
1.3 Organización de la memoria ... 2
2 Estado del arte ... 3
2.1 Estudio de los distintos tipos de robots móviles ... 3
2.1.1 Clasificación de los tipos de robots ... 3
2.1.1.1 Robots Terrestres ... 3
2.1.1.2 Robots Acuáticos ... 4
2.1.1.3 Robots Aéreos ... 6
2.1.2 Enjambres de robots ... 7
2.2 LEGO® MINDSTORMS® ... 8
2.3 ROS 11
2.3.1 Conceptos básicos ... 12
2.4 Integración LEGO® NXT con ROS ... 13
2.5 Estudio de proyectos similares ... 14
3 Diseño ... 17
3.1 Recursos utilizados ... 17
3.1.1 Robot LEGO® MINDSTORMS® NXT ... 17
3.1.2 Sensores y motores ... 18
3.1.3 ROS ... 20
3.1.4 NXT-Python ... 20
3.2 Arquitectura del sistema ... 21
4 Desarrollo ... 23
4.1 Conexión y comunicación con NXT ... 23
4.2 Control por teleoperación ... 26
4.3 Procesamiento de datos de los sensores ... 28
5 Pruebas y resultados ... 29
5.1 Conexión por Bluetooth y lectura de los sensores ... 29
5.2 Latencia de comunicación ... 30
5.3 Algoritmo de control por teleoperación ... 30
5.4 Algoritmo de navegación autónoma ... 31
6 Conclusiones y trabajo futuro ... 33
6.1 Conclusiones ... 33
6.2 Trabajo futuro ... 33
Referencias ... 35
Glosario de acrónimos ... 37
Anexos ... I
A. Manual de Instalación ... I
A.1 Instalación de ROS ... I
A.2 Instalación del paquete NXT-Python ... III
B. Creación de un paquete en ROS ... V

INDICE DE FIGURAS

Figura 2-1: Robot con ruedas ... 3
Figura 2-2: Robot con patas ... 4
Figura 2-3: Robot con patas ... 4
Figura 2-4: Clasificación de los tipos de robots acuáticos [6] ... 5
Figura 2-5: Ejemplo pez robot SoFi [7] ... 6
Figura 2-6: Clasificación de los tipos de robots aéreos [6] ... 6
Figura 2-7: Enjambre de robots [9] ... 7
Figura 2-8: LEGO® MINDSTORMS® RCX ... 9
Figura 2-9: LEGO® MINDSTORMS® NXT ... 9
Figura 2-10: LEGO® MINDSTORMS® EV3 ... 10
Figura 2-11: Esquema publicador/suscriptor en ROS ... 12
Figura 2-12: Diseño final del robot EV3 ... 14
Figura 2-13: Diseño del proyecto Laser Scanner ... 14
Figura 2-14: Diseño Segway con EV3 ... 15
Figura 2-14: Diseño vehículo autónomo NXT ... 15
Figura 3-16: Robot de nuestro proyecto ... 17
Figura 3-17: Servomotor para NXT ... 18
Figura 3-18: Sensor de color ... 18
Figura 3-19: Sensor de ultrasonidos ... 19
Figura 3-20: Sensor giroscópico ... 19
Figura 3-21: Direcciones de los ejes del acelerómetro ... 19
Figura 3-22: Diagrama del proyecto ... 21
Figura 4-23: Código para la conexión ... 23
Figura 4-24: Publicadores en tópicos y creación del nodo ... 24
Figura 4-25: Código para la publicación de mensajes ... 24
Figura 4-26: Suscripción al tópico para la Teleoperación ... 25
Figura 4-27: Ejecución de movimientos ... 25
Figura 4-28: Publicador de las órdenes por teclado ... 26
Figura 4-29: Código del listener del teclado ... 26
Figura 4-30: Código al presionar teclado ... 27
Figura 4-31: Código al liberar teclado ... 27
Figura 4-32: Creación y escritura en un *bag* ... 28
Figura 5-33: Resultados conexión con NXT y lectura de sensores ... 29
Figura 5-34: Resultados latencia PC-NXT ... 30
Figura 5-35: Diagrama del sistema con rqt_graph ... 30
Figura A-36: Configuración de los repositorios ... I

INDICE DE TABLAS

Tabla 2-1: Comparativa entre RCX, NXT y EV3 10

1 Introducción

1.1 Motivación

El diseño de algoritmos para robots es una actividad compleja y que requiere de experiencia en muchos campos. Por este motivo, hay dos factores clave: la posibilidad de contar con entornos de simulación y de pruebas de bajo coste y también el poder reutilizar y adaptar algoritmos desarrollados previamente por la comunidad.

ROS *(Robot Operating System*) [1] es un *framework* de código abierto mantenido por la OSRF (*Open Source Robotics Foundation*) [2], que proporciona al usuario un conjunto de herramientas para iniciar su aplicación desde un nivel superior, esto nos permite ceñirnos al núcleo del proyecto ahorrándonos el volver a crear lo que otros desarrolladores ya han hecho previamente.

Impulsado por estos factores, se ha decidido generar un entorno que sirva para trabajos posteriores, en el que se integre un robot LEGO® MINDSTORMS® NXT con una versión actualizada de ROS y de esta forma poder reutilizar herramientas ya existentes, desarrollar nuevos algoritmos, así como realizar pruebas en simulación y en un entorno real.

1.2 Objetivos

El objetivo principal de este Trabajo de Fin de Grado consiste en realizar la conexión de un robot móvil a través del protocolo Bluetooth, extender las funcionalidades del robot consiguiendo su integración en ROS Melodic (versión actual de ROS) [3], desarrollar una serie de algoritmos en Python y que puedan ser ejecutados en el robot gracias a esta comunicación.

Para lograrlo, se divide el trabajo en los siguientes subobjetivos:

- Conexión inalámbrica por Bluetooth.
- Desarrollo de algoritmos de prueba en el PC y ejecución en el robot con la librería NXT-Python [4].
- Lectura y tratamiento de los datos de los sensores del robot desde el PC.
- Integración con la arquitectura de ROS en versiones no obsoletas y de esta forma, ampliar el soporte y las funcionalidades de Lego NXT en sistemas operativos Linux con un *framework open source*.
- Empleando la arquitectura desarrollada, implementación de un algoritmo de control del robot por teleoperación y de navegación autónoma.

Además, para una correcta simulación en tiempo real, se ha marcado como requisito esencial del trabajo que la latencia de la comunicación (petición/respuesta) no debe superar los 150ms.

1.3 Organización de la memoria

El presente documento se ha redactado siguiendo la normativa para los Trabajos de Fin de Grado. Esta se divide en los siguientes capítulos:

- Capítulo 1: Introducción al trabajo que vamos a realizar, la motivación y lo que nos empuja para su realización y los objetivos que se deben cumplir en el mismo.

- Capítulo 2: Estado del arte que enmarca el trabajo, se realiza un estudio de los tipos de robots móviles existentes, la evolución y las características de LEGO® MINDSTORMS® que es la línea de robots al que pertenece el que utilizaremos en nuestro trabajo, el *framework* ROS y trabajos similares realizados anteriormente.

- Capítulo 3: Diseño, se definen los recursos utilizados y profundizaremos un poco acerca de NXT-Python, ROS y su arquitectura, así como el diagrama y su explicación.

- Capítulo 4: Desarrollo, se detalla cada uno de los pasos seguidos en el proyecto, desde la primera toma de contacto con ROS y NXT-Python hasta la implementación de los algoritmos ejecutados.

- Capítulo 5: Integración, pruebas y resultados obtenidos. En este capítulo se muestran las evidencias de que el diseño y el desarrollo planteados funcionan correctamente.

- Capítulo 6: Conclusiones y trabajo futuro, se describen las conclusiones obtenidas tras la realización del trabajo y las líneas futuras a seguir.

2 Estado del arte

El objetivo de este capítulo es presentar un estudio sobre el estado del arte de la investigación e implementación realizadas. Este estudio ha consistido en la búsqueda de los distintos tipos de robots móviles existentes, investigación acerca de los recursos con los que trabajamos para la realización del Trabajo de Fin de Grado y también se ha profundizado en otros proyectos similares que han sido realizados previamente.

2.1 Estudio de los distintos tipos de robots móviles

Los robots móviles se clasifican principalmente en tres tipos: terrestres, acuáticos y aéreos. El diseño varía en función del tipo de entorno en el cual van a desempeñar sus tareas.

A lo largo de los años se han realizado avances en la robótica con la finalidad de mejorar los mecanismos utilizados para una mejor movilidad. Con el uso de sensores, estos robots son capaces de moverse por sí mismos, pueden ser autónomos o teleoperados, pero en ambos casos se mantiene el contacto con el operador. Los terrestres se desplazan mediante ruedas, patas u orugas y sus aplicaciones están basadas en el rastreo de obstáculos, evasión o traslado de elementos. Los aéreos (denominados drones) son artefactos no tripulados controlados remotamente, proporcionan imágenes de reconocimiento del terreno. Los robots acuáticos se especializan en la navegación dentro del agua con sensores de sonar, radar, visión y usando sistemas para poder sumergirse.

2.1.1 Clasificación de los tipos de robots

2.1.1.1 Robots Terrestres

Los robots terrestres, generalmente conocidos como vehículos terrestres no tripulados (UGV, por el inglés *Unmanned Ground Vehicle*), trabajan sobre el suelo, es decir, utilizan la gravedad como soporte. En función del tipo de terreno, estos pueden tener ruedas, orugas, patas o ser humanoides (robots con forma humana). Las aplicaciones de estos robots son muy variadas: hogar, vigilancia, servicios y militares.

Según el artículo de ScienceDirect [5], *las ruedas constituyen la solución más popular y la más ampliamente utilizada tanto en robots móviles como en vehículos autónomos*. Esto se debe principalmente a que las ruedas son más simples mecánicamente y más eficientes (en términos de energía consumida frente a distancia recorrida). Además, usando más de tres ruedas el vehículo se encuentra completamente equilibrado. Sin embargo, esta buena eficiencia se reduce cuando el terreno es menos consistente.

Figura 2-1: Robot con ruedas

Las patas al tener un único punto de contacto con el suelo se ven mucho menos afectadas en los casos de terrenos poco compactos, en cambio, las ruedas sufren una mayor fricción. El inconveniente de la traslación mediante patas es que requiere más grados de libertad y, por lo tanto, mayor complejidad mecánica que las ruedas u orugas.

Figura 2-2: Robot con patas

Las orugas son una alternativa interesante a las ruedas y las patas para terrenos poco compactos. Este sistema hace uso de pistas de deslizamiento, lo que supone un mayor contacto con el terreno, una mejor maniobrabilidad y tracción que las ruedas y una movilidad superior a la que ofrecen las patas. Además, el diseño mecánico de las orugas es menos complejo que el de las patas (sólo necesita de dos actuadores para la tracción) y el de las ruedas (no se requiere un sistema de suspensión).

Figura 2-3: Robot con patas

Teniendo en cuenta lo anterior, la decisión entre ruedas, orugas o patas se debe basar en: el tipo de misión del robot, el perfil del terreno y las propias características del vehículo (dimensiones, peso, etc.).

2.1.1.2 Robots Acuáticos

Los robots acuáticos están inspirados y desarrollados en la morfología de los animales, la cual está siendo un hallazgo muy importante para la robótica. La gran parte de las investigaciones en robots submarinos se han focalizado en la agilidad y eficiencia de los peces, facilitando el desarrollo de sistemas para tareas de inspección y mantenimiento de tuberías de petróleo, plataformas oceánicas de aceite o gas y la exploración de las profundidades marinas.

Los robots acuáticos utilizan hélices para su propulsión, el objetivo es el ahorro de energía para su recorrido y distribuirla de forma equitativa para un mejor control del robot.

Se pueden clasificar de distintas maneras: por el nivel de autonomía para realizar un trabajo específico, como se observa en la Figura 2-4.

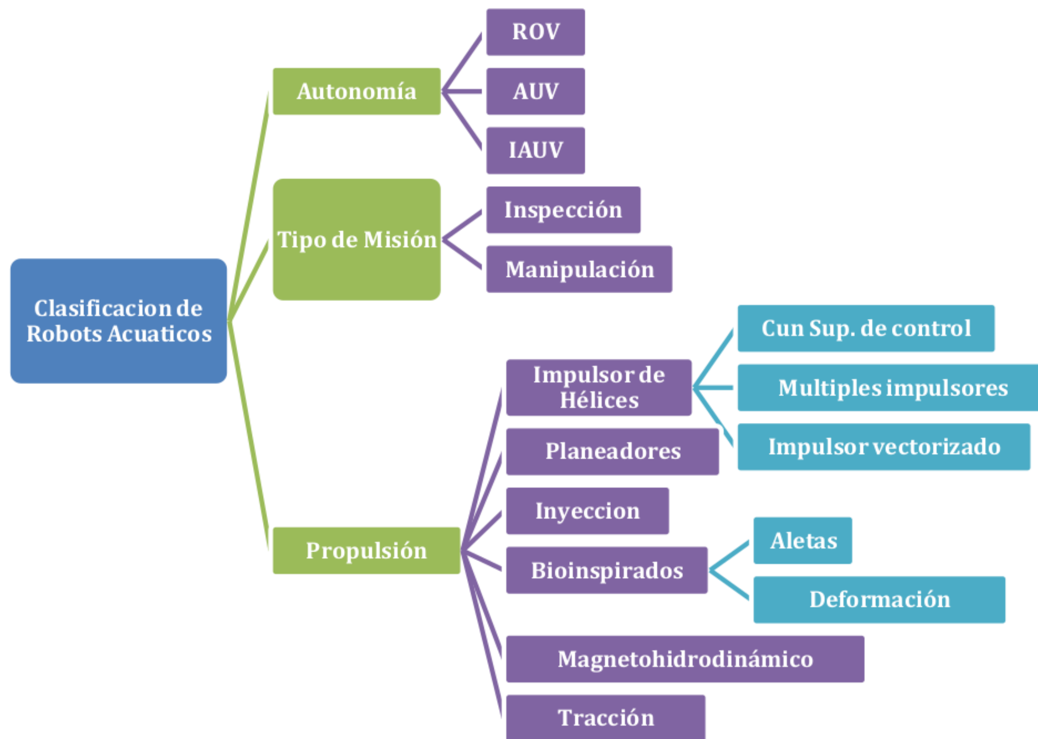

Figura 2-4: Clasificación de los tipos de robots acuáticos [6]

Para este primer caso se presenta la clasificación de los robots ya sean autónomos, teleoperados o híbridos. También pueden clasificarse según su misión a realizar, ya sea de inspección o manipulación; esta característica define el tipo de sensores que se han de utilizar y la estructura que debe poseer. Para la definición del sistema de propulsión se estudia el consumo de energía según el tipo de ambiente y los movimientos a realizar por el robot.

Los robots acuáticos han sido diseñados para realizar misiones como son:

- Inspección autónoma donde el robot navega sin la necesidad de un manipulador.
- Toma de imágenes mediante una o varias cámaras.
- Medición de la temperatura y calidad del agua mediante sensores.
- Revisión de instalaciones, ya sea en tuberías o espacios abiertos.
- Teleoperación para la apertura y cierre de válvulas, ensamble y desmontaje de componentes, recolección de muestras para estudios…

Figura 2-5: Ejemplo pez robot SoFi [7]

2.1.1.3 Robots Aéreos

Su principal desarrollo fue impulsado para aplicaciones militares, donde los UAV (vehículo aéreo no tripulado, del inglés *Unmaned Aerial Vehicle*) destacan en campos como el reconocimiento aéreo, vigilancia aérea, adquisición de objetivos, evaluación de daños, búsqueda y rescate en zonas de combate, etc. Sin embargo, paralelamente se han ideado aplicaciones menos belicistas, especialmente en el campo del reconocimiento aéreo e incluso en la exploración espacial.

Los vehículos aéreos tienen la capacidad de realizar misiones con cierta autonomía. Por lo tanto, son aeronaves no tripuladas que se despliegan según un criterio preestablecido para una mejor navegabilidad, en este caso, se encuentran los vehículos no tripulados según el tipo de despegue.

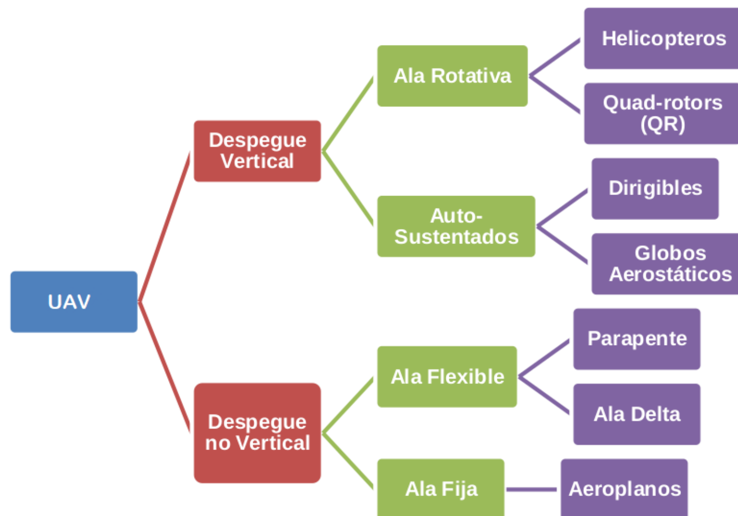

Figura 2-6: Clasificación de los tipos de robots aéreos [6]

Los UAV también destacan en otras aplicaciones como la entrega de suministros o materiales en zonas sensibles, búsqueda y rescate, turismo, ocio, etc.

2.1.2 Enjambres de robots

Los sistemas que involucran el uso de un robot individual dentro de un determinado ambiente, en según qué aplicaciones, están siendo sustituidos por sistemas que involucran una gran cantidad de robots simples y pequeños, de esta forma se consiguen minimizar costes, optimizar la ejecución de tareas, tener un sistema tolerable a fallos, flexible y fácilmente escalable. A este tipo de sistemas se les denomina enjambres de robots.

Un sistema basado en enjambres es una solución a tener en cuenta por su gran capacidad para resolver distintos tipos de problemas, según el artículo [8]: *control de tráfico, formaciones en movimiento, exploración, mapeo, búsqueda y rescate, simulación de escenarios y comportamientos biológicos, planificación de rutas, entre muchos otros.*

Figura 2-7: Enjambre de robots [9]

A pesar de las diversas definiciones sobre lo que es y representa un enjambre, existe un consenso sobre las características que debe tener un grupo de robots para ser considerado un enjambre. A continuación, se presentan dichas características según lo presentado en [8]:

- ***Simplicidad*****:** La forma en la que están construidos los robots y el hardware requerido para su funcionado debe ser sencilla, esto les permite responder de manera rápida y flexible a los cambios de ambiente que se encuentren.
- ***Autonomía*****:** El objetivo es que los robots no sean controlados de forma central ni remotamente por servidores o seres humanos, aunque pueden existir "líderes" en el enjambre que deciden el comportamiento de otros individuos.
- ***Control del sistema*****:** Puede ser *centralizado o descentralizado*.
- ***Comunicación*****:** La comunicación entre los robots es fundamental para la organización y cumplimiento de la tarea que deben realizar. Se pueden encontrar tres tipos de formas de comunicación: *el medio ambiente, de sensores y de comunicaciones.*

- ***Inspiración biológica*:** Dado el origen de los enjambres, se desea que su comportamiento esté, en cierto modo, inspirado biológicamente. Existe un debate abierto por lo que se han creado dos corrientes: *enfoque minimalista* en el cual las características físicas y de comunicación deben limitarse a un nivel mínimo y un *enfoque tecnológico* que incentiva a utilizar capacidades ofrecidas por la electrónica moderna.

Las ventajas de utilizar enjambres de robots son:

- ***Paralelismo*:** Gracias a que el tamaño de la población de un enjambre es bastante grande, los robots pueden dividirse para realizar múltiples tareas al mismo tiempo.
- ***Escalabilidad*:** Una comunicación local dentro del enjambre permite que varios individuos se puedan unir o salir de las actividades que estén realizando sin interrumpir el esquema general del enjambre.
- ***Estabilidad*:** Si una parte del enjambre deja las actividades debido a algún factor externo, permite al sistema no verse afectado por ello.
- ***Económico*:** El coste de crear y mantener robots pequeños es considerablemente más bajo si lo comparamos con un robot de gran tamaño y complejidad.
- ***Flexibilidad*:** Un mismo enjambre puede realizar diferentes tareas utilizando el mismo hardware o con pequeñas modificaciones de este, gracias a esto pueden cambiar de estrategia de acuerdo con el medio en que se encuentren para el cumplimiento del objetivo de manera más eficiente.

Para la realización del proyecto se ha decidió la utilización de un mini-robot, ya que por las prestaciones que ofrece y el bajo coste, este podría tener utilidad como miembro de un enjambre robótico en un futuro.

2.2 LEGO® MINDSTORMS®

MINDSTORMS® es una línea de robótica educativa fabricada por la empresa LEGO®. Permite construir, programar y controlar tus propios robots haciendo uso de piezas ensamblables y sensores acoplables con un dispositivo programable llamado brick o ladrillo. Cada vez es más habitual utilizar esta plataforma con fines educativos ya que permite experimentar con la tecnología de una manera creativa y sencilla.

Hasta el momento existen 3 generaciones de LEGO® MINDSTORMS®: El bloque RCX, el bloque NXT y el bloque EV3.

- **Primera generación: RCX**

La primera versión de estos bloques inteligentes, desarrollado conjuntamente por LEGO® y el Massachusetts Institute of Technology (MIT). A partir del trabajo realizado se desarrolla la primera generación que sale al mercado en 1998. El bloque RCX tiene tres versiones oficiales: 1.0, 1.5 y 2.0.

Figura 2-8: LEGO® MINDSTORMS® RCX

- **Segunda generación: NXT**

La generación con la que nosotros trabajamos. El nuevo brick salió al mercado en el año 2006. LEGO® vendió la generación NXT en dos versiones: *Retail Version* y *Education Base Set*. Además, la marca lanzó al mercado diferentes kits según las características de los programas que se deseara desarrollar. Existen tres versiones del bloque NXT: 1.0, 1.1, 2.0 y 2.1.

Figura 2-9: LEGO® MINDSTORMS® NXT

- **Tercera generación: EV3**

Es la última generación y llegó al mercado en 2013. Esta nueva generación presenta una mayor capacidad de procesamiento y sensores mejorados. También tiene una memoria mayor y una conectividad mejorada. Existen dos versiones: *Home Edition* y *Education Edition*.

Figura 2-10: LEGO® MINDSTORMS® EV3

En la Tabla 2-1 se muestra una comparativa entre los distintos modelos de LEGO® MINDSTORMS®.

	RCX	NXT	EV3
Aparición	1998	2006	2013
Procesador	Hitachi H8/300	Atmel AT91SAM7S256 (ARM7TDMI core)	TI Sitara AM1808 (ARM926EJ-S core)
Velocidad	16 MHz	48 MHz	300 MHz
Pantalla	LCD monocromo segmentado	LCD monocromo 100x64 pixel	LCD monocromo 178x128 pixel
Memoria	32 KB RAM 16 KB ROM	64 KB RAM 256 KB Flash	64 MB RAM 16 MB Flash
Memoria ampliable	NO	NO	microSD
USB (de salida)	NO	NO	SI
WIFI	NO	NO	Dongle opcional vía puerto USB
Bluetooth	NO	SI	SI
Salidas	3 Puertos (A, B, C)	3 Puertos (A, B, C)	4 Puertos (A, B, C, D)
Entradas	3 Puertos (1, 2, 3)	3 Puertos (1, 2, 3)	4 Puertos (1, 2, 3, 4)
Altavoz	SI	SI	SI
Interfaz usuario	4 botones	4 botones	6 botones retroiluminados
Alimentación	6 pilas AA	6 pilas AA o kit baterías recargables	6 pilas AA o kit de baterías recargables
Compatible *con* **Android/iOS**	NO	NO	SI

Tabla 2-1: Comparativa entre RCX, NXT y EV3

2.3 ROS

Como se ha indicado en el comienzo de la memoria, en este proyecto se ha abordado la integración de un robot en *Robot Operating System* (ROS), pero ¿qué es exactamente esta herramienta? ¿Cuáles son sus ventajas y qué puede aportar al desarrollador?

Según la siguiente referencia [10], *ROS provee los servicios estándar de un sistema operativo tales como abstracción del hardware, control de dispositivos de bajo nivel, implementación de funcionalidad de uso común, paso de mensajes entre procesos y mantenimiento de paquetes*, pero también provee herramientas y librerías para obtener, construir, escribir y ejecutar código a través de varios ordenadores.

ROS no es realmente un sistema operativo, sino un *framework* con un conjunto enorme de herramientas, las cuales proveen la funcionalidad de un sistema operativo. ROS posee más de 2000 librerías de software [11], y cada librería tiene una funcionalidad especializada. ROS está desarrollado y soportado por su propia comunidad por lo que, después de varios años de vida, ha dado lugar a una gran cantidad de paquetes reutilizables y fáciles de integrar gracias a la arquitectura del sistema.

Lo más característico de ROS es la forma en la que se ejecuta el software y su comunicación, esto permite diseñar software muy complejo sin el conocimiento real del funcionamiento del hardware. ROS permite conectar una red de procesos o nodos con un eje central, llamado *master*, que es el que conoce la existencia de todos los procesos o nodos.

Se pueden crear sistemas complejos al conectar soluciones existentes para pequeños problemas. Según el siguiente artículo [12], la arquitectura de ROS permite:

- Sustitución de componentes con interfaces similares sobre la marcha, sin la necesidad de detener el sistema para la realización de varios cambios.
- Envío de mensajes de múltiples componentes a una entrada para otro componente, permitiendo la solución paralela de varios problemas.
- Interconexión de componentes desarrollados en distintos lenguajes de programación (principalmente C++ y Python) únicamente con la implementación de los conectores apropiados al sistema de mensajes, lo cual facilita el desarrollo de software al conectar módulos existentes de varios desarrolladores.
- Crear nodos sobre una red de dispositivos sin la preocupación sobre dónde se ejecuta un código e implementar los sistemas de comunicación entre procesos (IPC) y llamadas a procedimiento remotos (RPC).

Esto permite que se puedan desarrollar fácilmente soluciones simples e iterativas. Además, ROS es multiplataforma por lo que es capaz de conectar procesos de múltiples dispositivos y permite la utilización de cualquier idioma de programación simplemente con determinar o desarrollar las clases correspondientes. Otro punto a favor es el reducido espacio que ocupa, permitiendo integrarlo con otras infraestructuras de software de robots como OpenRAVE (*Open Robotics Automation Virtual Environment*).

2.3.1 Conceptos básicos

A continuación, se exponen los conceptos sobre la arquitectura de ROS y que son utilizados a lo largo de este proyecto según la web oficial de ROS [13].

- **Paquete:** Es la principal unidad de organización en ROS. Contiene todos los procesos/nodos que se ejecutan, una librería dependiente de ROS, los archivos de configuración o cualquier elemento que sea útil, organizados de forma compacta. Un paquete es el elemento más pequeño e independiente que puede ser compilado.

- **Nodos:** Son los procesos que habilitan la comunicación y son imprescindibles para la misma. Están definidos en las librerías incluidas en ROS, como roscpp (C++) o rospy (Python).

- ***Master*:** Es el nodo central, núcleo de ROS donde se conectan todos los nodos y se intercambia la información.

- **Mensaje:** Los nodos se comunican entre sí enviando mensajes. Es una estructura comprimida de *arrays* de datos como *integer*, *floating point, boolean*, etc. Además, también se pueden incluir estructuras o *arrays*, existen muchos tipos de mensajes ya predefinidos o creados por la comunidad y que podemos reutilizar.

- **Tópico:** Los mensajes se envían a través de un sistema de publicación y suscripción. Un nodo es capaz de enviar información publicando en el tópico correspondiente y el nodo que esté interesado en esa información puede suscribirse a ese tópico para recibirla. Un nodo puede publicar en varios tópicos, como también estar suscrito a varios tópicos. Cualquier nodo puede publicar en un tópico, siempre y cuando utilice el mismo tipo de mensajes.

En la Figura 2-11 se puede ver un esquema de cómo funciona la comunicación entre nodos, siguiendo el modelo publicador/suscriptor, para ello es necesario que los nodos implicados utilicen el mismo tipo de mensaje.

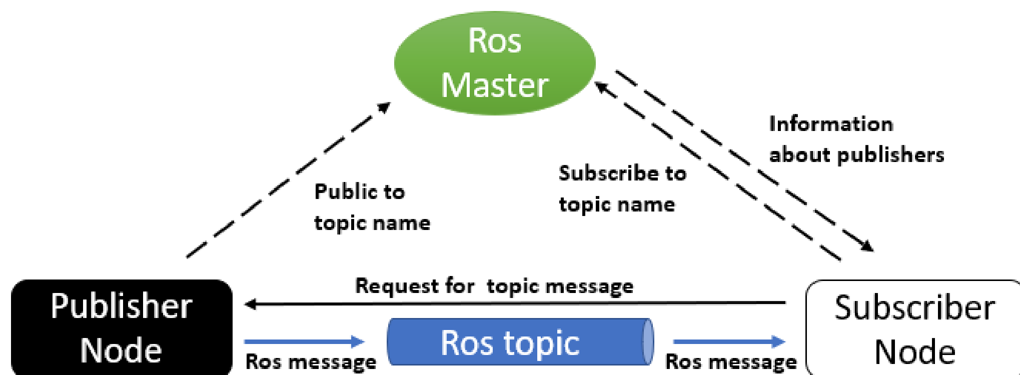

Figura 2-11: Esquema publicador/suscriptor en ROS

El compilado en ROS se realiza con catkin [14], que combina macros de CMake con scripts de Python para otorgar las funcionalidades de cualquier proceso normal de CMake. Catkin es muy similar a CMake pero aporta herramientas como la búsqueda automática de paquetes o la compilación de proyectos dependientes y múltiples al mismo tiempo.

Para compilar, se necesita la localización del código fuente, las dependencias del código y dónde se encuentran, o cuáles y dónde deben ser compilados los objetivos y dónde deben ser instalados. CMake suele utilizar un archivo llamado CMakeLists.txt para especificar toda esa información y en el caso de catkin extiende las funcionalidades de CMake para conseguir utilizar dependencias entre paquetes.

Para ello utiliza dos archivos: package.xml [15] y CMakeLists.txt [16]. El primero es el responsable de la secuencia de configuración y las dependencias de los paquetes en los espacios de trabajo. CMakeLists.txt es el encargado de preparar y ejecutar la compilación.

2.4 Integración LEGO® NXT con ROS

Los robots de LEGO® MINDSTORMS® son pequeños y versátiles, y son adecuados para la investigación. A lo largo de estos años se ha conseguido integrar el modelo LEGO® NXT con ROS, gracias al paquete NXT-ROS [13], compatible para la versión Electric en una máquina Ubuntu.

El paquete NXT-ROS es compatible con la distribución ROS Electric [3]. Esta versión de ROS fue lanzada el 30 de agosto de 2011, es una versión bastante antigua y que actualmente no dispone de soporte, esto supone un problema para la realización de nuevos proyectos con este modelo de robot, ya que no disponemos de todas las ventajas y recursos con los que cuenta la comunidad en la actualidad.

Además, NXT-ROS ofrece la posibilidad de realizar la comunicación por USB y por Bluetooth, pero esta última presenta un retardo muy elevado, lo cual limita mucho el control remoto, simulación y el tratamiento de los datos leídos por los sensores a tiempo real.

Teniendo en cuenta los problemas mencionados, si queremos trabajar con un robot NXT utilizando una distribución actual de ROS (Melodic, Lunar o Kinetic), existe la posibilidad de crear tu propio entorno de trabajo integrando la librería NXT-Python [4], como ya sabemos, ROS permite la programación en lenguajes como C++ y Python.

2.5 Estudio de proyectos similares

En este punto se lleva a cabo un estudio y análisis de proyectos similares realizados con un robot LEGO® MINDSTORMS®.

<u>Diseño de una Plataforma Interoperable ROS – LEGO® MINDSTORMS® EV3 [18]</u>

En esta tesis del año 2018, se lleva a cabo la realización de una plataforma de control Web para el modelo EV3 de LEGO®. En este caso, otro de los objetivos es ampliar el soporte de EV3 al sistema operativo Ubuntu 16.04 y ampliar su funcionalidad con ROS Kinetic [3].

La plataforma Web se desarrolla con fines educativos para la práctica de la programación con el lenguaje de programación visual Blockly.

Figura 2-12: Diseño final del robot EV3

<u>ROS como plataforma para extender las capacidades de LEGO® NXT [19]</u>

En este proyecto realizado en 2013 se integra el robot LEGO® MINDSTORMS® NXT con ROS Electric [3] utilizando el paquete NXT-ROS [17], del cual hablamos anteriormente y la versión 11.10 de Ubuntu.

Una vez conseguida la integración, se diseña un Láser Scanner, el cual tiene como objetivo escanear un objeto que se encuentra en cierta posición para que un láser lo ilumine de arriba hacia abajo y tome una foto de este.

Figura 2-13: Diseño del proyecto Laser Scanner

Desarrollo de un prototipo de robot educacional tipo Segway con control remoto [20]

El Trabajo de Fin de Grado fue llevado a cabo por unos compañeros de la Universidad Politécnica de Valencia en el curso 2017/2018. En él se utilizan los elementos de un LEGO® MINDSTORMS® EV3 para construir un robot tipo segway y la implementación del control de estabilidad con el software LabVIEW [21] (software de pago y de programación visual), no se ha utilizado ROS para el desarrollo ni Linux como sistema operativo.

También se diseñan e implementan dos algoritmos:

- Conexión por Bluetooth y control remoto desde el PC.
- Algoritmo de control que permite al robot seguir una trayectoria previamente conocida.

Figura 2-14: Diseño Segway con EV3

Sistema sensor para navegación en vehículos autónomos [22]

Se desarrolló en noviembre de 2019 y al igual que en el proyecto anterior, se ha implementado utilizando el software LabVIEW y Windows como sistema operativo, no se ha utilizado ROS. El objetivo es desarrollar un algoritmo de navegación autónoma para un NXT, la conexión se realiza tanto por USB como por Bluetooth y para su realización se aprovechan los sensores propios de LEGO®.

Figura 2-15: Diseño vehículo autónomo NXT

Tras analizar detenidamente los proyectos mencionados, en ellos se ha necesitado para su realización software como el de LabVIEW (de pago y de programación visual) o adaptarse a distribuciones obsoletas de ROS y realizar la comunicación por USB. La incompatibilidad con Linux y ROS supone una limitación importante en el año 2021 y no permite aprovechar al completo las características de este tipo de robots terrestres ni las ventajas y herramientas del *framework* y su enorme comunidad.

Por lo tanto, este Trabajo de Fin de Grado propone superar estas dificultades y desarrollar un entorno y un software que pueda ser reutilizable para trabajos venideros.

3 Diseño

En este capítulo profundizaremos acerca de los recursos utilizados en el desarrollo del proyecto (LEGO® MINDSTORMS® NXT, Sensores, ROS, la librería NXT-Python...), las decisiones tomadas para su implementación y la arquitectura del sistema.

3.1 Recursos utilizados

3.1.1 Robot LEGO® MINDSTORMS® NXT

Este será el modelo de robot que utilizaremos en nuestro proyecto (ver Figura 2-9), las características del NXT están descritas en la Tabla 2-1.

La gran cantidad de posibilidades que ofrece para el montaje, la variedad de sensores en el mercado y su reducido tamaño hacen que sea uno de los robots más populares e idóneo para aquellos que desean iniciarse en el mundo de robótica.

En la Figura 3-16, mostramos cuál es el resultado del nuestro montaje al completo, con los distintos sensores y los motores que se van a utilizar.

Figura 3-16: Robot de nuestro proyecto

3.1.2 Sensores y motores

- **Servomotores**

Nuestro montaje del robot cuenta con tres servomotores, conectados a los puertos de entrada A, B y C. Los servomotores de los puertos A y B están conectados a las orugas derecha e izquierda, respectivamente. El servo del puerto C está conectado con el sensor de distancia de la parte superior y permite que este rote sobre su propio eje.

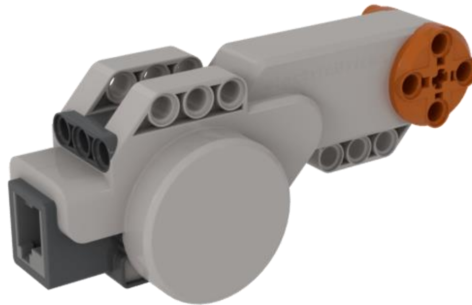

Figura 3-17: Servomotor para NXT

- **Sensor de color**

En el puerto de entrada 1 tenemos un sensor de color, el cual puede distinguir seis colores distintos (negro, azul, verde, amarillo, rojo y blanco), detectar la intensidad de la luz reflejada o ambiental o emitir una luz roja, verde o azul.

Figura 3-18: Sensor de color

- **Sensor ultrasónico**

Conectado al puerto de salida 2, este sensor permite medir la distancia hasta un obstáculo (en centímetros), detectar movimientos y ver o reconocer objetos a través de ondas de sonido. La distancia máxima a la que es capaz de medir son 2,5m.

Figura 3-19: Sensor de ultrasonidos

- **Giroscopio**

El sensor giroscópico es capaz de medir la velocidad angular a la que rota el robot, esta se mide en grados por segundo. Está conectado al puerto de entrada 3, este sensor nos permite calcular el giro del robot teniendo en cuenta el tiempo que accionamos los servomotores.

Figura 3-20: Sensor giroscópico

- **Acelerómetro**

Este sensor se encuentra conectado en el puerto 4 del robot y a simple vista es exactamente igual que el giroscopio. Mide la aceleración en los ejes X, Y, Z y la inclinación a lo largo de cada eje, se mide en el intervalo -2G a 2G. El acelerómetro permite al robot reconocer todas las direcciones (arriba, abajo, izquierda y derecha).

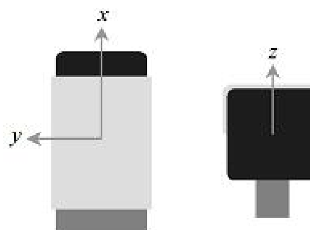

Figura 3-21: Direcciones de los ejes del acelerómetro

3.1.3 ROS

Como ya hemos mencionado anteriormente, entre los objetivos de este proyecto se encuentra la integración del robot NXT con ROS Melodic [3], versión actual y con soporte en la fecha de realización del trabajo, esta versión fue lanzada el 23 de mayo de 2018. La instalación de ROS y el desarrollo del proyecto se han realizado con una máquina Ubuntu en su versión 18.04 LTS [18] (ver instalación de ROS en el Anexo A.1).

En el apartado 2.3.1 vimos algunos conceptos básicos de ROS, esenciales para la familiarización con el *framework* y con los que vamos a trabajar. Para el manejo e implementación de estos conceptos y herramientas de ROS utilizaremos la API del módulo rospy [23]. Rospy proporciona la interfaz para la programación en Python y las funciones con las operaciones básicas de los nodos, tópicos, servicios y parámetros.

3.1.4 NXT-Python

NXT-Python [4] es un módulo/librería para el lenguaje Python que permite la comunicación con el robot NXT mediante USB o Bluetooth, es capaz de ejecutar comandos directos en el robot y la lectura de sensores.

Existen varias versiones en su repositorio, en nuestro caso se ha utilizado la v2.2.2 ya que es la última que se lanzó para Python2 (ver instalación en Anexo A.2), la versión para Python3 no se terminó de desarrollar y fue abandonada.

Antes del desarrollo del proyecto, es necesario familiarizarse con el módulo y conocer los programas/scripts y sus funciones más importantes, estas serán necesarias para la integración de ROS con el robot. A continuación, los describimos:

- **locator.py**

Realiza la búsqueda y conexión por Bluetooth con el NXT llamando a la función "*find_one_brick*", si la conexión se realiza correctamente devuelve un objeto "*Brick"* (ladrillo en inglés, término con el que se denomina al bloque de control LEGO®). También incluye las funciones para la lectura/escritura de un fichero de configuración, el cual puede utilizarse para realizar la conexión más rápidamente.

- **motor.py**

Incluye las clases y funciones para el control de cada uno de los motores del robot. Se crea un objeto de tipo "Motor" para cada uno y es necesario haber realizado la conexión previamente y conocer el puerto en el que está conectado. En la clase "*SynchronizedMotors*" disponemos de las funciones para la activación o detención de los motores.

- **generic.py**

Este script contiene las clases para la lectura de todos los sensores (excepto el acelerómetro y giroscopio). Para la creación de la clase es necesario pasar por parámetro el objeto "*Brick"* obtenido tras la conexión y el puerto de salida al que está conectado el sensor (A, B, C o D).

- **hitechnic.py**

El acelerómetro y giroscopio, aunque son originales de LEGO®, están fabricados por la marca HiTechnic. Las clases “*Gyro*” y “*Accelerometer*” contienen las funciones para la lectura de estos sensores y se encuentran dentro de este fichero. La única diferencia con respecto a los otros sensores se encuentra en que el giroscopio dispone de un método para la calibración (el offset puede variar en función del ambiente y otros factores externos) y antes de realizar la lectura del sensor es necesario realizar el calibrado.

3.2 Arquitectura del sistema

En la Figura 3-22 se muestra un diagrama con la arquitectura desarrollada para la realización de este proyecto.

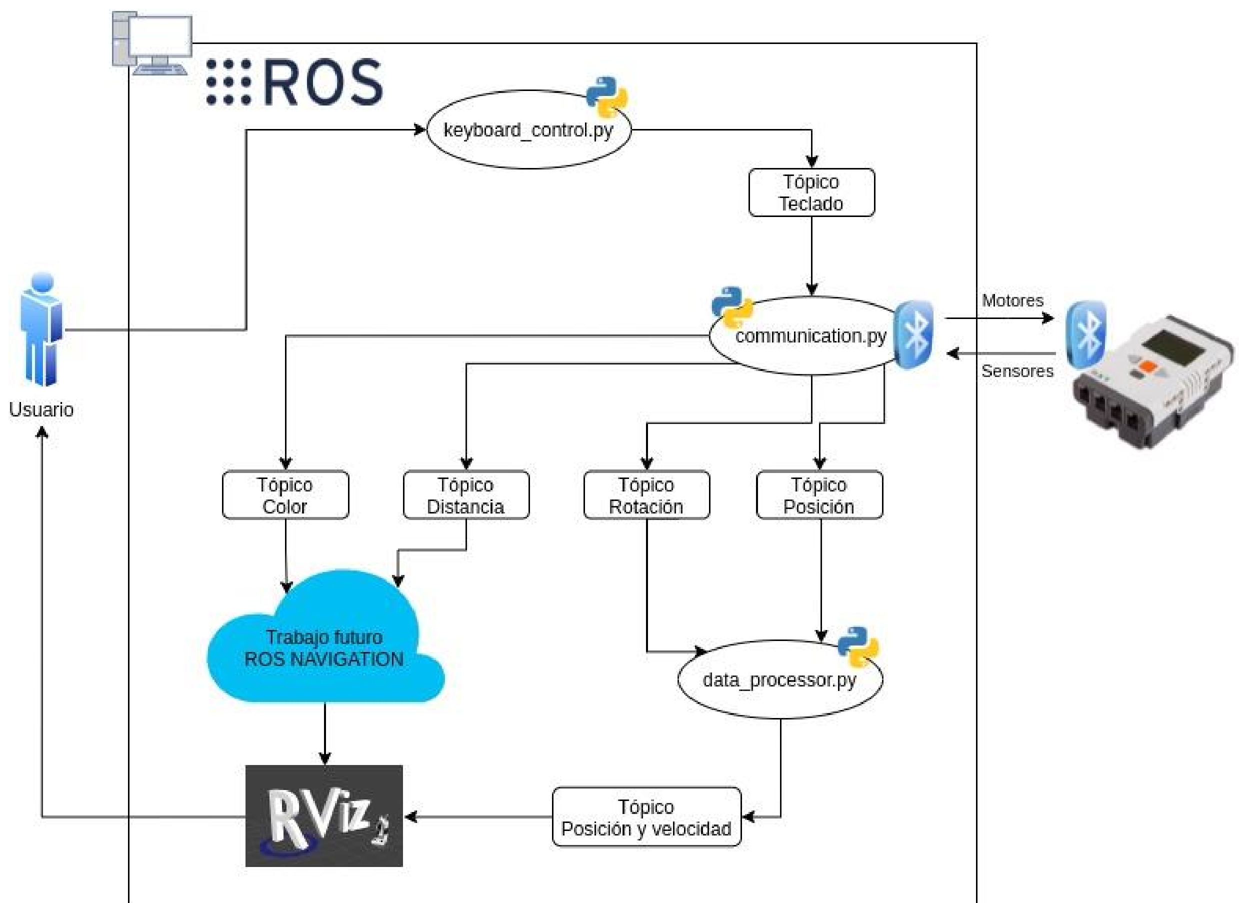

Figura 3-22: Diagrama del proyecto

En él se muestran cada uno de los componentes y cómo se enlazan cada uno de ellos. Los explicamos a continuación:

- **Usuario:** Se necesita un actor para interactuar con el teclado e introducir los movimientos del robot que son recogidos por el script “*keyboard_control.py*”.

- **Nodos:** Se muestran en el diagrama como elipses, aquellos que publican en tópicos son representados con flechas de salida y los que están suscriptos en algún nodo tienen flechas entrantes en las elipses. Los nodos han sido desarrollados utilizando rospy [23] (librería de ROS para Python) junto con NXT-Python.

- **Tópicos:** Representados con rectángulos. El tipo de los mensajes es distinto en cada tópico. En el diagrama podemos ver qué nodos publican en un tópico y los que están suscritos a él.

- **NXT:** Establece la comunicación con el nodo "*communication.py*" a través del protocolo Bluetooth. Una vez realizada la conexión, el NXT recibe las órdenes a ejecutar por los motores y este a su vez envía los datos leídos por los sensores.

- **ROS Navigation:** Se representa en el diagrama el paquete de navegación de ROS [24], es una opción a tener en cuenta para la implementación de la navegación autónoma en un futuro. Este utilizaría los datos publicados en los tópicos del sensor de distancia y de color. Lo indicamos con una nube ya que es una parte que no está implementada, pero necesario para entender la publicación de mensajes en estos tópicos.

4 Desarrollo

Una vez analizado el diseño y la arquitectura del proyecto, en este capítulo se va a explicar cómo se ha realizado el desarrollo de los distintos componentes del sistema. En este caso, lo vamos a dividir en subsecciones, cada una dedicada a un nodo de la aplicación.

Para ello necesitamos haber instalado y configurado ROS en nuestro PC (Anexo A.1) y la librería NXT-Python (Anexo A.2), además de haber creado un paquete para nuestro proyecto (Anexo B).

Todo el código del proyecto se encuentra subido en un repositorio de GitHub [25] en este apartado únicamente se mostrarán las partes más relevantes del mismo. Los nodos explicados a continuación, están representados en la Figura 3-22.

4.1 Conexión y comunicación con NXT

El script *"communication.py"* contiene la creación de un nodo para conexión y comunicación con el robot, así como la publicación de los datos de los sensores en cada uno de los tópicos. El primer paso a realizar es la conexión por Bluetooth, como ya hemos mencionado, para ello se utiliza la librería NXT-Python.

```
# Python and NXT
import nxt.locator
from nxt.sensor import *
from nxt.motor import *
# ============================================================================

# Connection
brick = nxt.locator.find_one_brick(debug=True, method=nxt.Method(usb=False, bluetooth=True))
m_left = Motor(brick, PORT_B)
m_right = Motor(brick, PORT_A)

# Synchronized motors
both = nxt.SynchronizedMotors(m_left, m_right, 0)
rightboth = nxt.SynchronizedMotors(m_left, m_right, 100)
leftboth = nxt.SynchronizedMotors(m_right, m_left, 100)

# Sensors
color = Color20(brick, PORT_1)
u = Ultrasonic(brick, PORT_2)
gyro = HTGyro(brick, PORT_3)
acc = HTAccelerometer(brick, PORT_4)
```

Figura 4-23: Código para la conexión

En el parámetro *"method"* de la función *"find_one_brick"* indicamos el método de comunicación. Si se ha realizado correctamente, esta función devuelve un objeto *"Brick"*, el cual utilizaremos para interactuar con los motores y sensores. En la Figura 4-23 se muestra la conexión y sincronización de los motores, así como con cada uno de los sensores del robot. El único requisito previo para la conexión por Bluetooth es haber vinculado el NXT previamente con nuestro PC, como cualquier otro dispositivo (auriculares, móvil, etc.).

```python
# Publish the data read by the sensor on each topic
def publishments():
    pub_color = rospy.Publisher('color_topic', Color, queue_size=10)
    pub_distance = rospy.Publisher('distance_topic', Distance, queue_size=10)
    pub_gyro = rospy.Publisher('gyro_topic', Rotation, queue_size=10)
    pub_acc = rospy.Publisher('acc_topic', Acceleration, queue_size=10)

    rospy.init_node('communication', anonymous=True)
    rate = rospy.Rate(10)  # 10hz
```

Figura 4-24: Publicadores en tópicos y creación del nodo

Para la publicación de los datos leídos por los sensores, cada uno de ellos escribe en un tópico con un tipo de mensaje distinto y que posteriormente, serán recibidos por los nodos suscritos a cada uno de estos tópicos. En la Figura 4-24 se muestra el código para la publicación en los tópicos utilizando rospy, el tipo de mensaje y la inicialización del nodo.

```python
# Loop until stop execution
while not rospy.is_shutdown():
    msg_color = Color()
    msg_color.header.stamp = rospy.Time.now()
    msg_color.header.frame_id = 'Color_Sensor'
    msg_color.color = color.get_color()

    msg_dist = Distance()
    msg_dist.header.stamp = rospy.Time.now()
    msg_dist.header.frame_id = 'Ultrasonic_Sensor'
    msg_dist.cms = u.get_distance()

    msg_gyro = Rotation()
    msg_gyro.header.stamp = rospy.Time.now()
    msg_gyro.header.frame_id = 'Gyro_Sensor'
    msg_gyro.degrees_sec = gyro.get_rotation_speed()

    msg_acc = Acceleration()
    acceleration = acc.get_acceleration()
    msg_acc.header.stamp = rospy.Time.now()
    msg_acc.header.frame_id = 'Accel_Sensor'
    msg_acc.x = acceleration.x
    msg_acc.y = acceleration.y
    msg_acc.z = acceleration.z

    pub_color.publish(msg_color)
    pub_distance.publish(msg_dist)
    pub_gyro.publish(msg_gyro)
    pub_acc.publish(msg_acc)

    rate.sleep()
```

Figura 4-25: Código para la publicación de mensajes

En la Figura 4-25 se muestra el bucle en el que se publican los datos de los sensores en cada uno de los tópicos. Estos valores proceden del robot y se ponen a disposición de cualquier nodo de ROS. La frecuencia de publicación en cada tópico es fija y está configurada a 10Hz (100ms). La ejecución no termina hasta que no se finaliza el nodo.

```
# Subscribe to the topic and receive keyboard commands
rospy.Subscriber('movement_topic', Move, movement)
```

Figura 4-26: Suscripción al tópico para la Teleoperación

```
# Callback function for NXT move
def movement(data):
    # Go forward
    if data.dir == 'w':
        both.run(100)

    # Turn left
    elif data.dir == 'a':
        leftboth.run()

    # Go back
    elif data.dir == 's':
        both.run(-100)

    # Turn right
    elif data.dir == 'd':
        rightboth.run()

    # Stop movement
    elif data.dir == 'STOP':
        rightboth.brake()
        leftboth.brake()
        both.brake()

    # Incorrect key pressed
    else:
        rospy.loginfo('Key not allowed')
        rightboth.brake()
        leftboth.brake()
        both.brake()
```

Figura 4-27: Ejecución de movimientos

Por último, este nodo en el caso del control por teleoperación, también está suscrito al tópico en el que se publican las instrucciones por teclado (Figura 4-26). En función del contenido del mensaje, se ejecuta una acción u otra en el robot (Figura 4-27), como avanzar, rotar a izquierda o derecha, retroceder o detenerse.

4.2 Control por teleoperación

Esta parte del desarrollo se encuentra en el script "*keyboard_control.py*". En él se implementa un nodo que publica en un tópico los mensajes con los movimientos a realizar por el robot, estos son recibidos e interpretados por el nodo de comunicación.

```
# Declare script as publisher
pub_movement = rospy.Publisher('movement_topic', Move, queue_size=10)
```

Figura 4-28: Publicador de las órdenes por teclado

```
# Node creation and control for pressed and released key
def remote_control():
    # Create node
    rospy.init_node('keyboard_control', anonymous=True)

    # Loop until stop execution
    while not rospy.is_shutdown():
        # Listen actions in keyboard
        rospy.loginfo('Listening commands')
        with keyboard.Listener(
                on_press=on_press,
                on_release=on_release) as listener:
            try:
                listener.join()
            except AttributeError:
                tcflush(sys.stdin, TCIFLUSH)
                print 'Special Keys is not allowed'

        rospy.loginfo('Aborted!')
```

Figura 4-29: Código del listener del teclado

En la Figura 4-29 mostramos la inicialización del nodo, se ha utilizado la librería Pynput [26] para implementar un *listener,* el cual captura las acciones del teclado. La lógica del evento de pulsar o levantar una tecla se implementa en los métodos "*on_press*" (Figura 4-30) y "*on_release*" (Figura 4-31), utilizamos la tecla "W" para avanzar, "A" para girar a la izquierda, "D" girar a la derecha y "S" para retroceder. Esto se puede modificar y agregar nuevas funcionalidades, como aumentar o reducir la velocidad del robot.

```python
# Function that publish in topic when press compatible keys (w, a ,s, d)
def on_press(key):
    global previous_key

    try:
        if previous_key != key.char:
            if key.char == 'w' or key.char == 'a' or key.char == 's' or key.char == 'd':
                previous_key = key.char

                # Send movement
                msg_move.dir = key.char
                msg_move.header.stamp = rospy.Time.now()
                pub_movement.publish(msg_move)

    except AttributeError:
        print('special key {0} pressed'.format(
            key))
        return True
```

Figura 4-30: Código al presionar teclado

```python
# Send stop command when the key is released
def on_release(key):
    global previous_key

    # Reset global parameter
    previous_key = ''

    # Send stop all motors
    msg_move.header.stamp = rospy.Time.now()
    msg_move.dir = 'STOP'
    pub_movement.publish(msg_move)

    # Stop execution with Esc, Space or 'q'
    if key == keyboard.Key.esc or key == keyboard.Key.space or key.char == 'q':
        # Stop listener
        tcflush(sys.stdin, TCIFLUSH)
        return False
```

Figura 4-31: Código al liberar teclado

4.3 Procesamiento de datos de los sensores

El script "*data_processor.py*" se desarrolla con el objetivo de tratar los datos leídos por los sensores, adaptándolos a los necesarios para ser visualizados en RViz [27]. RViz es una herramienta de visualización en 3D para aplicaciones de ROS, que permite al operador conocer el estado del robot y su entorno a partir de la información enviada por este.

Al no conseguir la visualización del robot con este paquete de ROS, se ha desarrollado un nodo que almacena los datos leídos por cada uno de los sensores en un fichero *bag* [28]. Esta herramienta permite guardar la información de los tópicos de un robot para reproducirlas en otro momento, puede servir para guardar movimientos y acciones del robot y esta información utilizarla en experimentos posteriores.

En la Figura 4-32 se muestra el código para la creación y escritura de un *bag* con el paquete *rosbag*.

```
bag = rosbag.Bag('sensor_data.bag', 'w')

def write_rotation(data):
    bag.write('Rotation_Data', data)

def write_position(data):
    bag.write('Acceleration_Data', data)

def write_distance(data):
    bag.write('Distance_Data', data)

def write_color(data):
    bag.write('Color_Data', data)

def data_processor():
    try:
        # Node initialization
        rospy.init_node('data_processor', anonymous=True)

        # Subscribe to Gyro and Acceleration data (position and rotation)
        rospy.Subscriber('gyro_topic', Rotation, write_rotation)
        rospy.Subscriber('acc_topic', Acceleration, write_position)
        rospy.Subscriber('distance_topic', Distance, write_rotation)
        rospy.Subscriber('color_topic', Color, write_position)

        # spin() simply keeps python from exiting until this node is stopped
        rospy.spin()
    finally:
        bag.close()
```

Figura 4-32: Creación y escritura en un *bag*

5 Pruebas y resultados

Una vez explicados el diseño y el desarrollo del sistema, en este capítulo se llevan a cabo las pruebas y la comprobación de que los resultados son los esperados y se cumplen los objetivos que se habían planteado para el trabajo en el apartado 1.2, son los siguientes:

- Conexión inalámbrica por Bluetooth y lectura de los sensores desde el PC.
- Latencia de comunicación inferior a 150ms.
- Desarrollo de un algoritmo de control por teleoperación empleando la arquitectura desarrollada.
- Implementación de un algoritmo de navegación autónoma.

A continuación, se especifican cada una de las pruebas realizadas y los resultados obtenidos. Todas ellas se encuentran subidas en el repositorio de GitHub [25].

5.1 Conexión por Bluetooth y lectura de los sensores

En esta prueba únicamente se verifica que es posible realizar la conexión con el NXT de forma inalámbrica, enviar instrucciones y leer los valores de cada uno de los sensores.

```
#####################################
#   CONNECTIVITY AND SENSORS TEST   #
#####################################
Connecting with the NXT...
Connected!
-----------------------------------

Color:                          6
Ultrasonic:                     46
Gyro without calibrate:         578
...calibrating...
Gyro calibrated:                0
Accelerometer X:                -448
Accelerometer Y:                32
Accelerometer Z:                64
------------------------------------
```

Figura 5-33: Resultados conexión con NXT y lectura de sensores

En la Figura 5-33 observamos que la conexión se ha realizado y que obtenemos los valores de cada uno de los sensores del robot.

5.2 Latencia de comunicación

El objetivo de esta prueba es comprobar que, tras integrar el robot con ROS, la latencia (petición/respuesta) entre el PC y el NXT es asumible y prácticamente inapreciable por cualquier persona. Una latencia elevada imposibilitaría una correcta simulación en tiempo real. Basándonos en los valores obtenidos por muchos de los auriculares Bluetooth, un retardo superior a 150ms es apreciable por cualquier persona e imposibilita la sincronización del audio y vídeo, por lo que se elige este valor como objetivo.

```
########################################
#                                      #
#             LATENCY TEST             #
#         Integration NXT-ROS          #
########################################
Connecting with the NXT...
Connected!
---------------------------------------

Color latency: 79.4833707809 ms
Ultrasonic latency: 116.28674984 ms
Gyro latency: 38.0599308014 ms
Acc latency: 113.899230957 ms
```

Figura 5-34: Resultados latencia PC-NXT

Los resultados de la Figura 5-34 muestran el retardo medio obtenido por cada sensor, se han tomado un total 100 medidas y calculado el tiempo transcurrido entre la petición y respuesta de cada una. Por lo tanto, se ha superado el objetivo con un margen bastante amplio y logrado una comunicación sin retardo entre los componentes del sistema.

5.3 Algoritmo de control por teleoperación

Utilizando la arquitectura y cada uno de los recursos explicados en este trabajo, se ha desarrollado un algoritmo de teleoperación con el teclado de nuestro ordenador. De forma inalámbrica podemos navegar con el robot NXT utilizando la arquitectura de ROS, todos los datos de los sensores son leídos y publicados en los tópicos a tiempo real.

En la Figura 5-35 mostramos un grafo del sistema gracias a la herramienta de ROS rgt_graph [29].

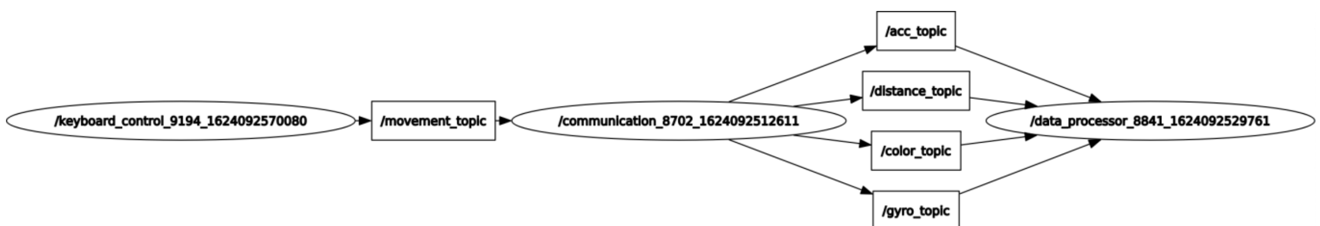

Figura 5-35: Diagrama del sistema con rqt_graph

El resultado de esta prueba se encuentra en una carpeta compartida de Drive, adjuntamos la URL de esta en la siguiente referencia [30].

5.4 Algoritmo de navegación autónoma

Otro de los objetivos del proyecto era conseguir la navegación autónoma del robot con la herramienta de visualización en 3D RViz. Este objetivo no se ha superado, principalmente por dos problemas:

- No hemos logrado la visualización del NXT en la herramienta, sólo se ha conseguido con el paquete NXT-ROS [17] y una conexión por USB, pero no con nuestra arquitectura.

- La incapacidad de interpretar los valores obtenidos por el acelerómetro para conseguir la posición del robot, la librería NXT-Python nos ha facilitado la lectura de los sensores, excepto para este caso, a pesar de haber investigado y leído toda la documentación posible. Muy probablemente hubiese sido más fácil hacer un desarrollo, leer los valores brutos y realizar el tratamiento de estos.

En cualquier caso, se ha desarrollado un algoritmo de navegación autónoma en ROS sin esta herramienta. Utilizando el sensor de ultrasonidos, el giroscopio (para calcular los ángulos de giro) y con la ayuda de dos sensores de contacto, el robot es capaz de navegar esquivando cada uno de los obstáculos de una habitación sin la intervención de ningún agente.

Al igual que en la prueba anterior, el resultado y demostración de esta prueba se encuentra en la siguiente referencia [30], en la cual podemos ver al robot desplazándose por un recinto con distintos obstáculos y realizando los movimientos en función de los datos leídos por los sensores en el PC.

6 Conclusiones y trabajo futuro

6.1 Conclusiones

El objetivo propuesto inicialmente en este trabajo era lograr la conexión de un mini-robot terrestre a través del protocolo Bluetooth, extender sus funcionalidades gracias a la integración con ROS (*framework open source)* en versiones no obsoletas y desarrollar algoritmos con esta arquitectura, lo cual no era posible antes de la realización de este proyecto. También se ha conseguido y superado con creces el objetivo de la comunicación con una latencia inferior a 150ms, hemos sido capaces de trabajar con cada uno de los sensores del robot, realizar el tratamiento estos datos desde el PC a tiempo real y desarrollar un algoritmo de teleoperación empleando la arquitectura.

En relación con la tecnología y la parte técnica, el desarrollo de este TFG ha provocado resultados muy positivos, puesto que se ha implementado un entorno de simulación y de pruebas de bajo coste y se han extendido las capacidades de este modelo de robots gracias a su integración con ROS. Este trabajo también ha servido de punto de partida para la realización del Trabajo de Fin de Grado "Entrenamiento de un mini-robot para realizar tareas mediante aprendizaje automático" [31].

Por último, como estudiante este proyecto ha conseguido adentrarme por primera vez en el mundo de la robótica, conocer sus entresijos, las dificultades que supone trabajar en una rama de la ingeniería tan inmensa y conocer la arquitectura de uno de los *frameworks* más utilizados hoy en día.

6.2 Trabajo futuro

Tras la finalización del trabajo, se ha conseguido la integración con la arquitectura propuesta y una versión funcional para un entorno de simulación. A continuación, se listan una serie de posibles mejoras identificadas durante las pruebas o bien por el autor a lo largo del proyecto:

- Se proporciona un entorno de pruebas de bajo coste para probar cualquier algoritmo desarrollado por la comunidad robótica. Esta herramienta puede servir a investigadores en robótica móvil, sistemas multi-robot o enjambres.

- Modo autónomo más conseguido, utilizando alguna herramienta de visualización en 3D (RViz o similar).

- Desarrollo para el tratamiento de los datos leídos por el acelerómetro, leer los valores brutos y procesarlos de forma que sus medidas cumplan con cualquiera de las hojas de especificaciones.

- Integración del entorno con el paquete de navegación de ROS (ROS Navigation), con el objetivo de que el robot sea capaz de desplazarse de un punto a otro siguiendo una ruta preestablecida.

Referencias

[1] ROS. [En línea]. https://www.ros.org/
[2] Open Robotics. [En línea]. https://www.openrobotics.org/
[3] List of Distributions, Wiki ROS. [En línea]. http://wiki.ros.org/Distributions#List_of_Distributions
[4] Librería NXT-Python. GitHub. [En línea]. https://github.com/Eelviny/nxt-python
[5] R. González, F. Rodríguez, J.L. Guzmán. “Robots móviles con orugas: Historia, modelado, localización y control”. July 10, 2014. Science Direct. [En línea]. https://www.sciencedirect.com/science/article/pii/S1697791214000788
[6] D.Y. Forero Quintero, M.A. Meza Calderón. “Diseño y construcción de un robot acuático”. February 25, 2015. Universidad Piloto de Colombia. [En línea]. http://repository.unipiloto.edu.co/handle/20.500.12277/1063
[7] Pez robot SoFi. ElPaís. [En línea]. https://elpais.com/elpais/2018/03/26/ciencia/1522074731_827249.html
[8] Rebeca Solís-Ortega. “Enjambres de robots y sus aplicaciones en la exploración y comunicación”. August 16, 2019. Revistas TEC (Tecnológico de Costa Rica). [En línea]. https://revistas.tec.ac.cr/index.php/memorias/article/view/4530
[9] Luke Dormehl. “Algorithm lets swarms of robots work together to create shapes without colliding”. March 10, 2020. Digital Trends. [En línea]. https://www.digitaltrends.com/cool-tech/northwestern-swarm-robotics-throw-shapes/
[10] Robot Operating System. Wikipedia. [En línea]. https://es.wikipedia.org/wiki/Robot_Operating_System
[11] Wiki ROS. [En línea]. http://wiki.ros.org/es
[12] A. Ademovic, “An introduction to robot operating system: The ultimate robot application framework.” Toptal. [En línea]. https://www.toptal.com/robotics/introduction-to-robot-operating-system
[13] ROS, Concepts. [En línea]. http://wiki.ros.org/ROS/Concepts
[14] Catkin, Wiki ROS. [En línea]. http://wiki.ros.org/catkin
[15] Package.xml, Wiki ROS. [En línea]. http://wiki.ros.org/catkin/package.xml
[16] CMakeLists.txt, Wiki ROS. [En línea]. http://wiki.ros.org/catkin/CMakeLists.txt
[17] NXT-ROS. Wiki ROS. [En línea]. http://wiki.ros.org/nxt
[18] Yessica Rosas Cuevas. “Diseño de una Plataforma Interoperable ROS – LEGO® MINDSTORMS® EV3”. Universidad Nacional de San Agustín. [En línea]. http://repositorio.unsa.edu.pe/handle/UNSA/8283
[19] M. F. Utreras Abad, D. C. Decimavilla Alarcón. “ROS como plataforma para extender las capacidades de LEGO NXT”. Escuela Superior Politécnica del Litoral. [En línea]. http://www.dspace.espol.edu.ec/xmlui/handle/123456789/25358
[20] J. R. Vizcaíno Espejo. “Desarrollo de un prototipo de robot educacional tipo Segway con control remoto”. Universidad Politécnica de Valencia. [En línea]. https://riunet.upv.es/handle/10251/112325
[21] NI LabVIEW. [En línea]. https://www.ni.com/es-es/shop/labview.html
[22] X. Viñolo Arcos. “Sistema sensor para navegación en vehículos autónomos”. Universidad Politécnica de Catalunya. [En línea]. https://upcommons.upc.edu/handle/2099.1/9377
[23] Rospy. Wiki ROS. [En línea]. http://wiki.ros.org/rospy
[24] ROS Navigation. Wiki ROS. [En línea]. http://wiki.ros.org/navigation

[25] Repositorio TFG. GitHub. [En línea]. https://github.com/JBernal145/nxt-integration
[26] Pynput. PyPI Packages. [En línea]. https://pypi.org/project/pynput/
[27] RViz. Wiki ROS. [En línea]. http://wiki.ros.org/rviz
[28] rosbag. Wiki ROS [En línea]. http://wiki.ros.org/rosbag
[29] rqt_graph. Wiki ROS [En línea]. http://wiki.ros.org/rqt_graph
[30] Resultado de las pruebas del TFG. Google Drive. [En línea]. https://drive.google.com/drive/folders/1hbCdaqZVrzwWGtOULrUOvEArggxSCL3U?usp=sharing
[31] Detalle de TFG. EPS UAM. [En línea]. https://tfg.eps.uam.es/tfgs/detalleTFG?tfg_codigo=1920_1825_COTI
[32] Python. Web Oficial. [En línea]. http://www.python.org
[33] Instalación PyBluez. GitHub. [En línea]. https://github.com/pybluez/pybluez/blob/0.23/docs/install.rst
[34] Creación de un workspace con catkin. Wiki ROS. [En línea]. http://wiki.ros.org/catkin/Tutorials/create_a_workspace

Glosario de acrónimos

ROS	Robot Operating System
TFG	Trabajo de Fin de Grado
OSRF	Open Source Robotics Foundation
UGV	Unmanned Ground Vehicle
UAV	Unmaned Aerial Vehicle
ROV	Remotely Operated Vehicle
AUV	Autonomous Underwater Vehicle
IAUV	Intervention Autonomous Underwater Vehicle
MIT	Massachusetts Institute of Technology
IPC	Inter-Process Communication
RPC	Remote Procedure Call
OpenRAVE	Open Robotics Automation Virtual Environment
USB	Universal Serial Bus
LTS	Long Term Support
API	Application Programming Interface

Anexos

A. Manual de Instalación

A.1 Instalación de ROS

En este manual se explica la instalación de ROS Melodic en un PC con Ubuntu en su versión 18.04 LTS. Para versiones anteriores o posteriores, verificar en la Wiki de ROS qué distribución [3] es la compatible con tu sistema operativo.

1. Configurar los repositorios de Ubuntu

Para la instalación necesitamos configurar los repositorios de Ubuntu para permitir aquellos que son "restricted", "universe" y "multiverse". Para ello accedemos a "Software y Actualizaciones y habilitamos las tres opciones:

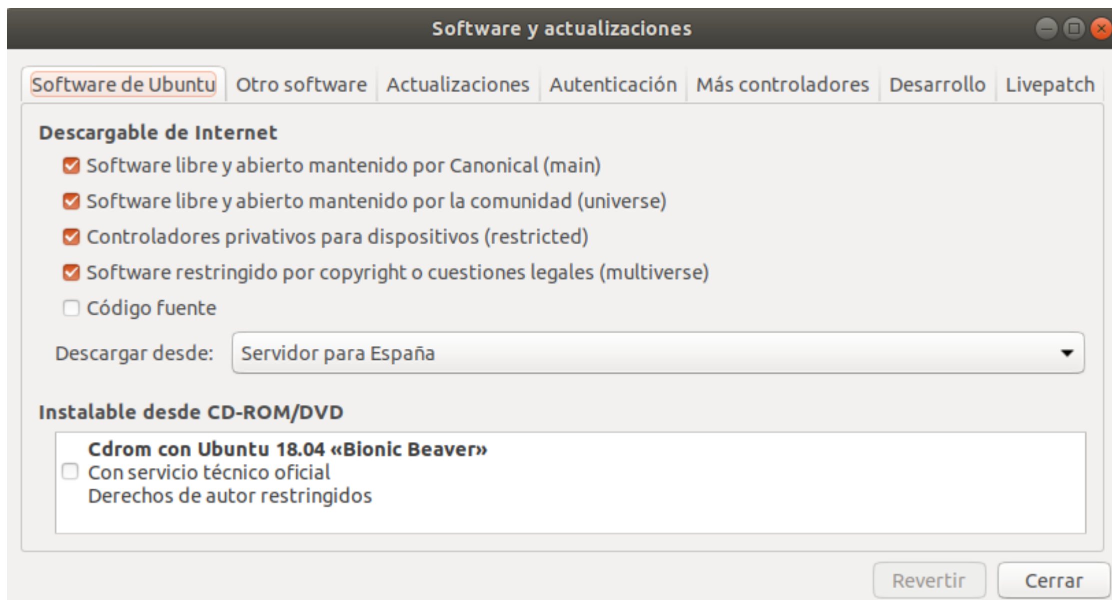

Figura A-36: Configuración de los repositorios

2. Configurar el fichero sources.list

Necesitamos configurar nuestro PC para aceptar los paquetes de software procedentes de "packages.ros.org", ejecutamos el siguiente comando en la terminal:

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

3. Configuramos nuestras claves

Para ello ejecutamos en la terminal:

```
$ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc
| sudo apt-key add -
```

4. Instalación

Primero debemos asegurarnos de que nuestros índices de los paquetes están actualizados, simplemente ejecutamos:

```
$ sudo apt update
```

Existen muchas y diferentes librerías y herramientas en ROS. En función de lo que deseemos instalar, ejecutamos un comando u otro, los describimos a continuación:

- Instalación completa (recomendada), contiene ROS, rqt, rviz, librerías robot-generic, simuladores 2D/3D:

  ```
  $ sudo apt install ros-melodic-desktop-full
  ```

- Versión de escritorio, contiene ROS, rqt, rviz y las líbrerías robot-generic:

  ```
  $ sudo apt install ros-melodic-desktop
  ```

- ROS-Base, contiene ROS, compilación y bibliotecas de comunicación, no incluye herramientas de interfaz gráfica:

  ```
  $ sudo apt install ros-melodic-ros-base
  ```

- Paquete individual, también es posible instalar un paquete específico de ROS, para ello simplemente:

  ```
  $ sudo apt install ros-melodic-PACKAGE
  ```

  Para encontrar los paquetes disponibles, podemos consultarlos con el comando:

  ```
  $ apt search ros-melodic
  ```

5. Configuración del entorno

Por comodidad, es recomendable que las variables de entorno de ROS se agreguen automáticamente a la sesión de *bash* cada vez que se lanza una nueva terminal. Para ello simplemente ejecutamos:

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

$ source ~/.bashrc
```

Finalizados estos cinco pasos, ya tendríamos completada la instalación, si surgiese cualquier problema podemos consultar en la Wiki de ROS [11].

A.2 Instalación del paquete NXT-Python

Para la instalación de esta librería es necesario cumplir los siguientes requisitos:

- Tener instalada una versión de Python (2.6 o superior, pero no 3.x) [32].
- Para la comunicación por Bluetooth en Linux es necesario haber instalado previamente el módulo PyBluez [33].

Hecho esto, descargamos la versión v2.2.2 de NXT-Python [4], todas las *releases* se encuentran en la pestaña "*Tags*". Seleccionamos la versión y descargamos todo el código fuente en nuestro PC, se nos descargará con el formato .zip.

1. Descomprimimos el fichero .zip que hemos descargado
2. En la raíz del paquete, ejecutamos:

```
$ sudo python setup.py install
```

Una vez instalado, en la carpeta descargada tenemos otra subcarpeta llamada "*examples*", en ella podemos encontrar distintos ejemplos con códigos de prueba para nuestro NXT.

B. Creación de un paquete en ROS

Para la creación de un paquete utilizaremos catkin, será necesario haber creado previamente un *workspace* [34].

1. **Nos situamos en la carpeta "/src" del *workspace.***

```
$ cd ~/catkin_ws/src
```

2. **Ahora con el comando *catkin_create_pkg* indicamos el nombre del nuevo paquete y las dependencias de este.**

Ejemplo:

```
$ catkin_create_pkg <package_name> [depend1] [depend2] [depend3]
```

Para nuestro caso:

```
$ catkin_create_pkg nxt_integration std_msgs rospy
```

3. **En el siguiente paso, tenemos que hacer un *build* del paquete dentro del *workspace*:**

```
$ cd ~/catkin_ws
```

```
$ catkin_make
```

4. **Para añadir el workspace a nuestro entorno de ROS, necesitamos ejecutar el fichero de configuración generado:**

```
$ . ~/catkin_ws/devel/setup.bash
```

5. **Por último, personalización del paquete, para ello tenemos el fichero "*package.xml*".**

   - Actualizamos la etiqueta "<description>" con una descripción breve del paquete.
   - La etiqueta "<manteiner> se completa con el contacto de la persona que ha desarrollado el paquete.
   - Se necesita la etiqueta "<license>", en la siguiente referencia podemos leer más acerca de ella en la siguiente referencia [REFERENCIA].
   - Las etiquetas de dependencias describen cada de las dependencias del paquete. Las dependencias se dividen en "<build_depend>", "<buildtool_depend>", "<exec_depend>", "<test_depend>".

- **Adjuntamos un ejemplo del fichero *package.xml*.**

```
<?xml version="1.0"?>
<package format="2">
  <name>nxt_integration</name>
  <version>1.0.0</version>
  <description>LEGO NXT integration with ROS and navigation algorithms
</description>

  <maintainer email="jbernal145@gmail.com">Javier Bernal</maintainer>
  <license>BSD</license>
  <url type="website">http://wiki.ros.org/nxt_integration</url>
  <author email="jbernal145@gmail.com">Jane Doe</author>

  <buildtool_depend>catkin</buildtool_depend>

  <build_depend>rospy</build_depend>
  <build_depend>std_msgs</build_depend>
  <build_depend>message_generation</build_depend>

  <build_export_depend>rospy</build_export_depend>
  <build_export_depend>std_msgs</build_export_depend>

  <exec_depend>rospy</exec_depend>
  <exec_depend>std_msgs</exec_depend>
  <exec_depend>message_runtime</exec_depend>


  <!-- The export tag contains other, unspecified, tags -->
  <export>
    <!-- Other tools can request additional information be placed here --
>

  </export>
</package>
```