

Universidad de Alcalá  
Escuela Politécnica Superior

Ingeniería de Computadores



ESCUELA POLITECNICA  
**Autor:** Jaime Mas Santillán  
**Tutor/es:** Roberto Javier López Sastre

2021

UNIVERSIDAD DE ALCALÁ  
Escuela Politécnica Superior  
Ingeniería de Computadores



Trabajo Fin de Grado  
**INTEGRACIÓN DE MODELOS DE  
INTELIGENCIA ARTIFICIAL Y SISTEMA  
OPERATIVO ROS EN PLATAFORMA  
ROBÓTICA ASISTENCIAL**

Autor: Jaime Mas Santillán

Director: Roberto Javier López Sastre.

**TRIBUNAL:**

*Presidente: D. Javier Acevedo Rodríguez*

*Vocal 1º: Dña. Pilar Martín Martín*

*Vocal 2º: D. Roberto Javier López Sastre*



# Integración de modelos de inteligencia artificial y sistema operativo ROS en plataforma robótica asistencial.

Jaime Mas Santillán.

15 de septiembre de 2021



*"The most important step a man can take. It's not the first one, is it? It's the next one.  
Always the next step, Dalinar"*



# Agradecimientos

Quiero agradecer, en especial, a mi tutor de trabajo de fin de grado, gracias Roberto por haberme enseñado tanto durante este año. También a Javier Acevedo, mi segundo tutor, ya que sin su ayuda este trabajo de fin de grado no habría sido posible. Quiero agradecer a mis padres todo el apoyo que me han dado durante este último año porque, aun en las malas, siempre están ahí para echarme un cable. Y por último quiero agradecer a mis amigas Lucía, Irene, Ana, Valentina, Leonor y Omi por haberme apoyado en todo momento durante estos largos años de carrera.



# Índice general

|   |             |
|---|-------------|
| <b>Agradecimientos</b>  | <b>v</b>    |
| <b>Resumen</b>  | <b>xI</b>   |
| <b>Abstract</b>   | <b>xIII</b> |
| <b>Resumen Extendido</b>  | <b>xV</b>   |
| <b>1. Introducción</b>  | <b>1</b>    |
| 1.1. Objetivos y campos de aplicación . . . . .                             | 2           |
| 1.2. Metodología de trabajo . . . . .                                       | 2           |
| <b>2. Estado del Arte</b>   | <b>5</b>    |
| 2.1. ¿Qué es la robótica? . . . . .   | 5           |
| 2.2. Tecnologías usadas en el desarrollo de plataformas robóticas . . . . . | 6           |
| 2.2.1. Lenguajes de programación . . . . .                                  | 6           |
| 2.2.2. Plataformas de procesado de bajo coste . . . . .                     | 8           |
| 2.3. La robótica asistencial . . . . .                                      | 10          |
| <b>3. Introducción a ROS</b>  | <b>15</b>   |
| 3.1. Orígenes . . . . .   | 16          |
| 3.2. Funcionamiento . . . . .   | 16          |
| 3.2.1. Objetivos de ROS . . . . .   | 17          |
| 3.2.2. Sistema de archivos . . . . .  | 17          |
| 3.2.3. Nodos . . . . .  | 18          |
| 3.2.4. Topics . . . . .   | 19          |
| 3.2.5. Servicios y parámetros . . . . .                                     | 22          |
| 3.2.5.1. Servicios . . . . .  | 22          |
| 3.2.5.2. Parámetros . . . . .   | 23          |
| 3.2.6. Roslaunch . . . . .  | 24          |
| 3.3. Paquetes que se han utilizado . . . . .                                | 25          |
| 3.3.1. Navigation Stack . . . . .   | 25          |
| 3.3.2. RVIZ . . . . .   | 27          |

|  |           |
|--|-----------|
| 3.4. ROS2 . . . . .  | 28        |
| <b>4. Plataforma LOLA</b>  | <b>29</b> |
| 4.1. Elementos físicos de la plataforma . . . . .                            | 30        |
| 4.2. Arduino . . . . .   | 33        |
| 4.2.1. Objetivos . . . . .   | 33        |
| 4.2.2. Explicación del funcionamiento . . . . .                              | 34        |
| 4.2.3. Instalación y pruebas . . . . .                                       | 36        |
| 4.3. ROS . . . . .   | 40        |
| 4.3.1. Objetivos . . . . .   | 40        |
| 4.3.2. Instalar ROS en la Jetson TX2 . . . . .                               | 41        |
| 4.3.3. Instalar ROS en cualquier computador . . . . .                        | 43        |
| 4.3.4. Creación espacio de trabajo Catkin . . . . .                          | 44        |
| 4.4. Primeros pasos con ROS y Python . . . . .                               | 45        |
| 4.4.1. Módulo Navigation Stack . . . . .                                     | 46        |
| 4.4.1.1. key_publisher . . . . .   | 47        |
| 4.4.1.2. keys_to_twist . . . . .   | 48        |
| 4.4.1.3. speed_translator . . . . .  | 49        |
| 4.4.1.4. odometry . . . . .  | 51        |
| 4.4.1.5. Directorios y Archivos . . . . .                                    | 54        |
| 4.4.1.6. Lanzamiento de la navegación mediante teclado . . . . .             | 55        |
| 4.5. Navegación autónoma . . . . .   | 57        |
| 4.5.0.1. hwinterface . . . . .   | 57        |
| 4.6. Reconocimiento de acciones online . . . . .                             | 66        |
| 4.6.1. Configuración para ejecutar nodos ROS en múltiples máquinas . . . . . | 66        |
| <b>5. Resultados</b>   | <b>79</b> |
| <b>6. Conclusiones</b>   | <b>85</b> |
| <b>Bibliografía</b>  | <b>87</b> |

# Lista de figuras

|       |   |       |
|-------|---|-------|
| 1.    | Plataforma Robótica Asistencial LOLA. . . . .   | XVII  |
| 2.    | Estructura de la plataforma KATE. . . . .   | XVIII |
| 3.    | Adaptación de la arquitectura KATE a la plataforma LOLA. . . . .                            | XVIII |
| 1.1.  | Robot Asistencial Robear [40]. . . . .  | 1     |
| 1.2.  | Parte frontal y trasera de la plataforma robótica LOLA. . . . .                             | 3     |
| 2.1.  | Placa NVIDIA Jetson TX2 [21]. . . . .   | 8     |
| 2.2.  | Placa Arduino A000066 [8]. . . . .  | 9     |
| 2.3.  | Placa Raspberry Pi 3 [10]. . . . .  | 9     |
| 2.4.  | TEO Humanoid Robot, Robotics Lab UC3M [27]. . . . .   | 11    |
| 2.5.  | Maggie, Robotics Lab UC3M [9]. . . . .  | 12    |
| 2.6.  | Nuka y Takanori Shibata [18]. . . . .   | 13    |
| 2.7.  | Robot Copito de la compañía CARTIF [25]. . . . .  | 14    |
| 3.1.  | Robot Operating System [41]. . . . .  | 15    |
| 3.2.  | Ejemplo de comunicación entre dos nodos [20]. . . . .                                       | 20    |
| 3.3.  | Funcionamiento de un topic [13]. . . . .  | 21    |
| 3.4.  | Funcionamiento del servidor de parámetros [12]. . . . .                                     | 23    |
| 3.5.  | Vista de alto nivel del nodo move base y su interacción con otros componentes [35]. . . . . | 26    |
| 3.6.  | RVIZ en funcionamiento [11]. . . . .  | 27    |
| 4.1.  | Sensores de la plataforma LOLA. . . . .   | 30    |
| 4.2.  | Comunicación entre Jetson y motor mediante Arduino. . . . .                                 | 34    |
| 4.3.  | Página de descarga del IDE de Arduino. . . . .  | 37    |
| 4.4.  | IDE de Arduino. . . . .   | 37    |
| 4.5.  | Abrir un Serial Monitor en el IDE de Arduino. . . . .                                       | 38    |
| 4.6.  | Abrir un proyecto de Arduino en el IDE de Arduino. . . . .                                  | 39    |
| 4.7.  | Instalación del paquete mcp2515 de Arduino en el IDE. . . . .                               | 40    |
| 4.8.  | Conexión de ROS con el resto de sensores. . . . .   | 41    |
| 4.9.  | Arquitectura del primer programa desarrollado con ROS y Python. . . . .                     | 46    |
| 4.10. | Estructura de directorios y archivos de la teleoperación mediante teclado. . . . .          | 54    |

|   |    |
|---|----|
| 4.11. Estructura de la plataforma KATE. . . . .                         | 58 |
| 4.12. Adaptación de la arquitectura KATE a la plataforma LOLA. . . . .  | 58 |
| 5.1. Despliegue de RVIZ para iniciar la navegación. . . . .             | 80 |
| 5.2. Plataforma LOLA navegando. . . . .                                 | 82 |
| 5.3. Sistema de detección de acciones online en funcionamiento. . . . . | 83 |

# Resumen

En este Trabajo Fin de Grado se describe como se ha desarrollado una novedosa plataforma robótica asistencial de bajo coste, con capacidades de percepción basadas en inteligencia artificial, capaz de navegar de manera autónoma utilizando Robot Operating System (ROS). La plataforma es un robot de ruedas diferencial, equipado con dos motores y encoders, que se controlan con una placa Arduino. También incluye una placa de procesamiento Jetson Xavier en la que implementar todos los procesos de IA y la arquitectura ROS. Como resultado se ha conseguido obtener una plataforma totalmente funcional, capaz de reconocer acciones online y de navegar de forma autónoma por entornos cuyo mapa ha sido precargado.

**Palabras clave:** ROS, RVIZ, robots asistenciales, Arduino, navegación.



# Abstract

This final degree project describes how we developed a novel low-cost assistive robotic platform, with AI based perception capabilities, able to navigate autonomously using Robot Operating System (ROS). The platform is a differential wheeled robot, equipped with two motors and encoders, which are controlled with an Arduino board. It also includes a Jetson Xavier processing board on which to deploy all AI processes, and the ROS architecture. As a result of the work, we have a fully functional platform, able to recognize actions online, and to navigate autonomously through environments whose map has been preloaded.

**Keywords:** ROS, RVIZ, assistive robotics, Arduino, navigation.



# Resumen Extendido

Este trabajo de fin de grado tiene como objetivo principal la implementación de un sistema de navegación autónoma en una plataforma robótica asistencial de bajo coste. Durante el desarrollo del trabajo se añadieron ciertas funcionalidades extras, como es el caso de la implementación de un sistema de detección de acciones que funciona de manera online.

El cerebro principal que va a manejar toda la plataforma robótica es el sistema operativo ROS. Esta herramienta no es un sistema operativo tal como conocemos hoy en día, pero nos aporta ciertas herramientas y frameworks que simulan las funcionalidades que conocemos de un sistema operativo. ROS está pensado para desarrollar aplicaciones robóticas de tal manera que las tareas de integración e implementación sean mucho más sencillas.

ROS nace en el año 2007 como parte de un proceso de investigación desarrollado en la universidad de *Standford* con los doctores Eric Berger y Keenan Wyrobek. El motivo de desarrollar esta herramienta se debe a la búsqueda del acercamiento de la robótica a personas del ámbito de la ingeniería que veían este campo como algo lejano y abstracto.

Con el tiempo el equipo de desarrollo de ROS fue creciendo hasta abandonar la universidad de *Standford* y empezar a trabajar con un nuevo equipo de desarrolladores en Willow Garage, un laboratorio de investigación robótica enfocado en el desarrollo de software libre.

ROS persigue una serie de objetivos tales como ser liviano, ser independiente del lenguaje de programación que use, tener un amplio número de bibliotecas a las que recurrir y ser un sistema escalado. El funcionamiento de ROS se puede resumir entendiendo como funcionan sus dos estructuras principales, los nodos y los topics. Los nodos son procesos que realizan una tarea concreta. Cada uno de estos nodos funcionan de manera individual, por lo que existe una independencia total entre todos los nodos que forman una estructura ROS. La comunicación entre estos nodos se lleva a cabo mediante el uso de topics. Un topic no es más que un punto de encuentro en el que dos o más nodos intercambian información mediante mensajes. Para llevar a cabo este intercambio, el topic le da dos opciones a cada nodo: la opción de publicar cierta información en él, para que otro nodo pueda recogerla; o la de suscribirse a él y que recoja la información que otro nodo, previa-

mente, ha depositado. A parte de estos dos términos existen otras herramientas que ROS nos proporciona y que se detallan en profundidad durante el desarrollo de esta memoria.

Nos podemos encontrar con una gran variedad de paquetes ROS, pero en este trabajo de fin de grado hemos profundizado en dos muy concretos, Navigation Stack y RVIZ. El Navigation Stack es un conjunto de paquetes que permiten a una plataforma robótica desplazarse por un entorno real, con obstáculos estáticos y dinámicos, sin chocarse y llegando a su destino siguiendo una ruta óptima en el menor tiempo posible. Este paquete es utilizado en prácticamente todas las plataformas robóticas que requieran de un sistema de navegación, ya que es sencillo de implementar y muy potente.

Los únicos requerimientos que este paquete necesita para poder funcionar son:

- Una plataforma que tenga ruedas de accionamiento diferencial y holonómicos.
- Que la base de móvil de la plataforma se controle enviando comandos de velocidades en formato: velocidad x, velocidad y, velocidad theta.
- Disponer de un láser plano montado sobre la base móvil de la plataforma.
- Es recomendable que la plataforma sea cuadrada, ya que el Navigation Stack fue concretamente desarrollado para este tipo de plataforma.

El segundo paquete que hemos utilizado es RVIZ. Éste es un paquete de visualización 3D que nos permite visualizar información por pantalla utilizando una gran cantidad de topics. RVIZ funciona como indicador de coordenadas de desplazamiento, de manera que cuando se lance el sistema de navegación, se desplegará una ventana de RVIZ en la que visualizaremos el mapa 2D del entorno por el que se va a desplazar la plataforma. Mediante una serie de opciones que nos da esta herramienta, vamos a poder señalar en el propio mapa el punto al que queremos que se desplace el robot.

Nuestra plataforma robótica LOLA, desarrollada por el grupo de investigación GRAM de la Universidad de Alcalá, es un robot diferencial con ruedas equipado con dos motores y sus correspondientes encoders, figura 1. Estos están controlados por una placa de Arduino Mega la cual se va a encargar de enviar los comandos de velocidad a cada una de las ruedas. Para poder llevar a cabo el sistema de navegación es necesario incluir una serie de sensores que proporcione la suficiente información del entorno como para permitir la navegación, estos son: un LIDAR y una cámara frontal.

Todos estos elementos van a funcionar en una única estructura con el objetivo de permitir la navegación autónoma de la plataforma. Para ello la placa de Arduino será la encargada de enviar la información de los encoders a la Jetson TX2 y ésta se encargará de procesar todos esos datos para, mediante el uso de ROS, generar una serie de comandos de velocidades que enviará a la placa de Arduino y, ésta a su vez, los enviará a los motores de las ruedas.



Figura 1: Plataforma Robótica Asistencial LOLA.

Todo el procesamiento de los datos de los encoders se realiza mediante la ayuda de los datos captados por el LIDAR. Este elemento es de vital importancia dentro de nuestro sistema, ya que es el encargado de situar a la plataforma en el mapa 2D y de detectar posibles obstáculos frente a los que la plataforma debe reaccionar.

Para poder poner en marcha la plataforma es necesario llevar a cabo una serie de instalaciones y configuraciones para poder instalar todo el sistema. Para ello esta memoria sirve a modo de manual, de manera que si se siguen todos y cada uno de los puntos descritos durante estas páginas se conseguirá llevar a cabo la correcta instalación del sistema y su puesta en marcha.

Para ello se deberán seguir una serie de pasos relativos a la puesta en marcha de ambas placas del sistema, tanto de la Arduino Mega, como de la NVIDIA Jetson TX2. También habrá que proceder a la instalación del sistema operativo ROS para poder empezar con el desarrollo de la arquitectura.

El siguiente paso a dar una vez que la instalación y configuración de las placas se haya llevado a cabo correctamente, y ROS esté corriendo en nuestra plataforma, es el de empezar a programar una pequeña estructura de nodos ROS que permita a nuestra plataforma navegar mediante el uso de teclas del teclado. Si seguimos este desarrollo vamos a poder entender como funciona ROS y como Python, mediante la librería Rospy, nos va a facilitar de gran manera la creación de nodos con unas pocas líneas de código.

Este pequeña estructura lleva un estudio detrás, habiendo tenido que decidir cuales son los nodos que conforman a este programa y como esta pensado la publicación de mensajes y comandos para que así, en futuras secciones cuando veamos el sistema de navegación autónoma, ya sepamos como funciona y como se realiza la comunicación entre nodos.

Entendida esta pequeña estructura de nodos pasaremos a ver la implementación de la navegación autónoma en nuestra plataforma. Debido a que es un sistema heredado hemos tenido que llevar a cabo un proceso de adaptación en el que la estructura previa que ha

sufrido ciertas modificaciones al tener ciertos componentes distintos a los nuestros, como es el caso del CAN USB que en nuestra plataforma no existe. Para ello se muestran un par de diagramas en los que se aprecia el cambio de arquitectura y el punto en el que nosotros nos conectaremos, figuras 2 y 3.

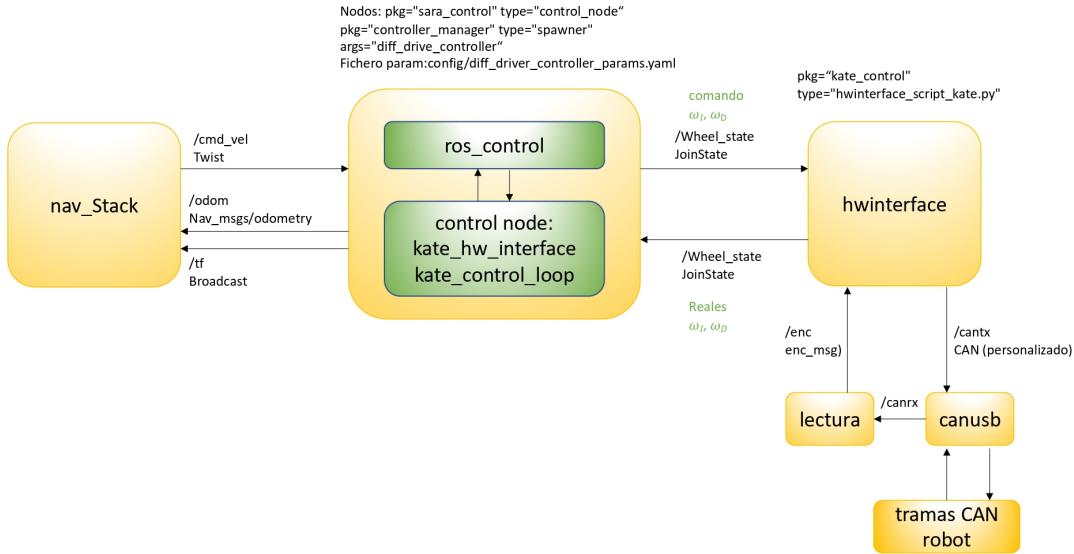


Figura 2: Estructura de la plataforma KATE.

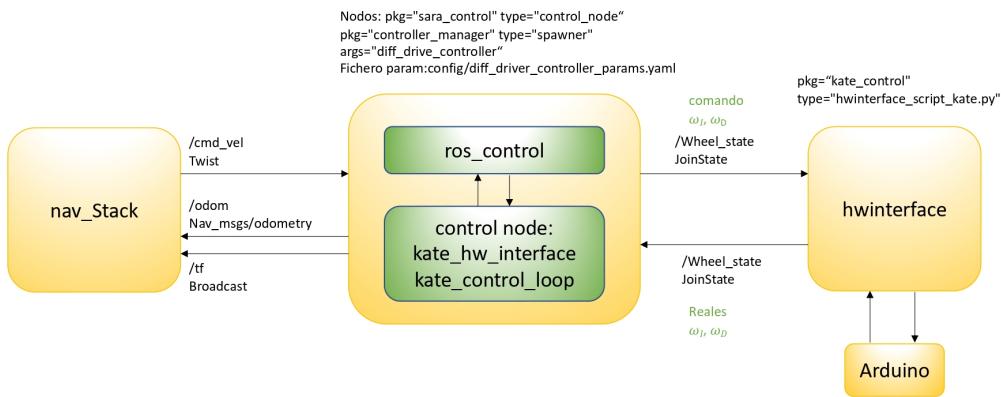


Figura 3: Adaptación de la arquitectura KATE a la plataforma LOLA.

El eje principal de esta estructura es el hwinterface. Este es el elemento central de la interconexión entre ambas arquitecturas. Este script hecho en Python se va a encargar de establecer la comunicación con la placa de Arduino Mega y de enviar todos los datos recibidos de los encoders al sistema de navegación autónoma. A parte también va a traducir los datos de las velocidades recibidos por el sistema a nuestro propio sistema de velocidades y, una vez procesados estos datos, los enviará a la placa de Arduino para que esta pueda indicarles a los motores que velocidades deben adaptar.

El segundo de los objetivos de este trabajo de fin de grado era el de implementar en la plataforma un sistema de detección de acciones. Durante el desarrollo del trabajo nos dimos cuenta de que implementar este sistema en una GPU como la NVIDIA Jetson TX2 no era viable, ya fuese por falta de potencia o por problemas de versiones entre Python y Ubuntu, es por ello que se decidió desarrollar un nodo ROS compartido que publicase las imágenes que iba captando la cámara de la plataforma LOLA y las fuese publicando en un topic online. De esta manera otro nodo ROS ejecutado en una GPU externa más potente es capaz de recoger estas imágenes, procesarlas con el detector de acciones y publicar en otro topic online cual es la acción que se está llevando a cabo por el usuario del robot asistencial.

La última parte del desarrollo de este trabajo ha sido analizar si se habían cumplido los objetivos del mismo. Para ello se llevaron a cabo diversas pruebas de navegación en las que pusimos a la plataforma a navegar por una de las plantas de la Escuela Politécnica de Alcalá de Henares. Excepto ciertas rutas que eran muy complejas para la plataforma, debido a su tamaño, ésta conseguía llegar al punto indicado con la orientación indicada. También pusimos a prueba la respuesta a diferentes obstáculos, ya fuesen estos objetos estáticos o personas que se movían por el entorno de la navegación y, en ningún caso, llegó a colisionar con ninguno de ellos.

En cuanto a la detección de acciones online también se consiguió llevar a cabo la implementación con resultados favorables, siendo capaces de enviar las imágenes a una GPU externa y esta analizarlas enviando de vuelta una cadena con el nombre de la acción que se estaba desarrollando.



# Capítulo 1

## Introducción

Tanto la inteligencia artificial, como la robótica, se ven a día de hoy como dos de los campos más revolucionarios y prometedores de la industria. Es el desarrollo de plataformas de ayuda asistencial para personas con diversidad funcional, el que permitirá acercar estas tecnologías, en su conjunto, a un grupo de personas cuya calidad de vida puede mejorar considerablemente. Un ejemplo de este tipo de plataformas asistenciales es Robear, figura 1.1 .



Figura 1.1: Robot Asistencial Robear [40].

Este trabajo de fin de grado, tiene como objetivo principal el desarrollo e implementación de modelos de inteligencia artificial, desarrollados en Python, junto con el sistema operativo ROS en una plataforma robótica cuyo componente principal sobre el que vamos a integrar el sistema operativo ROS es una máquina con Ubuntu 18.04 .

El resultado final que se espera obtener una vez implementado todo el desarrollo en la máquina con Ubuntu es, una plataforma robótica totalmente funcional capaz de navegar de manera autónoma por un entorno en el que se ha proporcionado un mapa en dos

dimensiones y que a su vez, sea capaz de reconocer acciones de manera online mediante la utilización de modelos de inteligencia artificial desarrollados en Python y nodos ROS remotos.

La plataforma debe de poder adaptarse a obstáculos dinámicos y a posibles variaciones del entorno que no estaban previstas en el mapa proporcionado previamente, ya sean nuevos obstáculos fijos o personas que son detectadas durante la navegación. Para ello se valdrá de la combinación de ROS, RVIZ, y los modelos de inteligencia artificial desarrollados en Python.

## 1.1. Objetivos y campos de aplicación

El objetivo principal de este trabajo es la integración completa de diversos modelos de inteligencia artificial en una arquitectura gestionada por el sistema operativo ROS, todo ello montado sobre una máquina con Ubuntu 18.04. De esta manera se espera obtener un sistema asistencial completo y funcional, al que añadirle módulos para realizar nuevas tareas sea mucho más sencillo gracias a la integración de ROS.

En concreto, pretendemos que el objetivo del TFG sea el de implantar ROS en la plataforma que aparece en la figura 1.2 , de modo que todas las tareas de navegación y percepción mediante inteligencia artificial estén integradas en el sistema operativo ROS.

Este sistema operativo estará montado sobre una placa NVIDIA Jetson TX2, la cual trabajará codo con codo junto a una placa de Arduino Mega y diversos sensores.

Los campos de aplicación a los que este trabajo de fin de grado se puede aplicar son:

- Mejora del desarrollo motriz de personas con diversidad funcional.
- Personalización de tareas gracias a la detección de acciones.
- Implementación de la navegación autónoma en otro tipo de plataformas robóticas.
- Ayuda al aprendizaje de personas con diversidad funcional.

## 1.2. Metodología de trabajo

Para llevar a cabo el desarrollo completo de la plataforma se va a seguir un flujo de trabajo en el que se empezará por la parte más mecánica de la plataforma y se escalará poco a poco hasta llegar a la implementación de los modelos de inteligencia artificial y las consiguientes pruebas para comprobar que la navegación y la detección de acciones funcionan correctamente.

- En primer lugar será necesario preparar la plataforma para poder trabajar sobre ella. Para ello se debe instalar en la Jetson el sistema operativo Ubuntu 18.04 y ROS Melodic (versión correspondiente a la 18.04 de Ubuntu). También es necesario tener el IDE de Arduino instalado para poder llevar a cabo la carga de software en la placa.
- Una vez realizada la configuración nos vamos a encontrar con la placa de Arduino y la comunicación de esta con los motores del robot. Para conseguir la correcta comunicación de estas dos partes se ha desarrollado un paquete de Arduino con sus correspondientes comandos capaz de interactuar con los encoders del motor. De esta manera es posible que ambas partes lleven a cabo una comunicación continua durante todo el proceso de navegación.
- Resuelta la parte del entendimiento entre motor y placa, pasaremos a ver en el capítulo 4, sección 4.3, como se ha desarrollado la estructura de nodos ROS que van a otorgar al sistema la capacidad de compartir y utilizar toda la información recibida por los diferentes sensores de la plataforma, ya sean los encoders del motor o el LIDAR.

Esta estructura de nodos se empieza a desarrollar siguiendo una base común, ya que hay determinados nodos proporcionados por el propio ROS como es el caso del Move Base, que se encuentra dentro del paquete Navigation Stack. A partir de esta base iremos escalando y creando nuestra propia estructura que completaremos con un nodo remoto para poder llevar a cabo la detección de acciones online.



Figura 1.2: Parte frontal y trasera de la plataforma robótica LOLA.

Este proyecto de fin de carrera se enmarca dentro del grupo de investigación *GRAM*. Este es un grupo de investigación perteneciente al Departamento de Teoría de la Señal y Comunicaciones cuya investigación se centra en el estudio de la visión por computador y la inteligencia artificial.

Este grupo de investigación colabora codo con codo en el programa *Padrino Tecnológico*. El programa tiene como objetivo el desarrollo de plataformas asistenciales que posteriormente son donadas a centros de atención a la discapacidad. Es por ello que este trabajo de fin de grado tiene como objetivo final servir de ayuda en el desarrollo de este tipo de plataformas para, posteriormente, ser enviadas a diferentes centros con el fin de mejorar la calidad de vida de personas con diversidad funcional.

La estructura de este trabajo de fin de grado se compone de cinco capítulos:

- **Estado del Arte, capítulo 2:** hablaremos sobre la situación en la que se encuentra a día de hoy la robótica asistencial, describiendo así el estado del arte de nuestro proyecto y describiendo qué puede aportar nuestra plataforma en comparación con las ya disponibles.
- **Introducción a ROS, capítulo 3:** en este capítulo comentaremos qué es ROS, cómo funciona y qué posibilidades ofrece al mundo de la robótica asistencial. Será un capítulo más educativo en el que hablaremos de las virtudes de este software y de como su uso ha facilitado enormemente el desarrollo de la plataforma LOLA.
- **Plataforma LOLA, capítulo 4:** pasaremos a describir la plataforma LOLA y todos sus componentes, todo ello agrupado en un tercer capítulo que formará el núcleo de este trabajo de fin de grado. Éste será el capítulo de apoyo sobre el que se tendrá que basar cualquier persona a la hora de continuar con el desarrollo de la plataforma.
- **Resultados, capítulo 5:** antes de acabar tendremos un quinto capítulo dedicado al análisis de resultados. Nos pararemos a analizar las pruebas de navegación que han sido realizadas al finalizar el desarrollo de la plataforma y como en un futuro se podría mejorar la navegación para reducir el tiempo de trayecto. También comentaremos qué resultados hemos obtenido al implementar el nodo externo de ROS que lleva a cabo el análisis de imágenes para la detección de acciones online.
- **Conclusiones, capítulo 6:** dedicado a las conclusiones del trabajo de fin de grado, en el que comentaremos los conocimientos que hemos adquirido gracias al desarrollo de la plataforma LOLA y lo que nos ha aportado colaborar en el desarrollo de la plataforma estando esta enmarcada en un grupo de investigación de la misma.

# Capítulo 2

## Estado del Arte

### 2.1. ¿Qué es la robótica?

A día de hoy la robótica se presenta como uno de los campos más prometedores de la industria. Gracias a los avances en inteligencia artificial, electrónica y mecánica hoy en día es posible crear plataformas robóticas inteligentes capaces de llevar a cabo acciones que antes podrían parecer impensables.

Es la combinación de estas ramas tecnológicas que han conseguido llevar la robótica a un nuevo plano en el que después de largos años de investigación y desarrollo empieza a dar sus frutos. Pero hay que comprender que estos son solo los primeros pasos que se dan sobre un largo camino que pondrá a la robótica en el punto de mira del progreso tecnológico.

Existen múltiples definiciones de lo que es y debe hacer un robot, pero no existe un consenso que dictamine si una plataforma es considerada como tal. Son muchas las definiciones que se han dado de la palabra desde 1920 cuando Karel Čapek inventó el término robot, palabra que deriva del checo “robota” cuyo significado es “trabajo”.

Una posible definición de la palabra a tener en cuenta puede ser “una máquina que se asemeja a un ser humano y es capaz de reproducir automáticamente ciertos movimientos y funciones humanas” [7]. Esta definición hace referencia a la imitación de las capacidades humanas, pero esto no tiene que ser siempre así, ya que podemos encontrarnos plataformas que se alejan totalmente de lo que es el aspecto humano o cuyas capacidades difieren de las nuestras.

A parte de las definiciones de lo que es un robot, también se han lanzado muchas ideas de qué leyes debe obedecer una plataforma robótica. Esto queda definido por el escritor americano Issac Asimov en “La Fundación”, una serie de libros de ciencia ficción en los que fue capaz de definir las tres leyes de la robótica:

- Un robot no hará daño a un ser humano o, por inacción, permitirá que un ser humano sufra daño.

- Un robot debe cumplir las órdenes dadas por los seres humanos, a excepción de aquellas que entrasen en conflicto con la primera ley.
- Un robot debe proteger su propia existencia en la medida en que esta protección no entre en conflicto con la primera o con la segunda ley.

Estas leyes a día de hoy se quedan cortas viendo el avance de la robótica, pero es muy interesante ver la visión, no tan desencaminada, que tenían algunos genios como Asimov sobre lo que la robótica podría llegar a ser en un futuro no tan lejano.

## 2.2. Tecnologías usadas en el desarrollo de plataformas robóticas

A la hora de desarrollar una plataforma robótica hay que tener en cuenta un gran número de tecnologías, ya que cualquier robot moderno está formado por una parte mecánica, una electrónica y otra computacional. Estas tres partes no son independientes una de otra, si no que en conjunto forman la plataforma en sí. En esta sección hablaremos en concreto de los lenguajes de programación más utilizados en la industria a la hora de desarrollar el cerebro del robot, comentaremos la importancia del sistema operativo ROS y cómo éste ha influido en la industria de la robótica, y por último comentaremos qué tipos de placas de bajo coste hay en el mercado con las que se pueden llevar a cabo grandes desarrollos.

### 2.2.1. Lenguajes de programación

Hoy en día podemos encontrarnos con una gran variedad de lenguajes de programación y frameworks en los que llevar a cabo cualquier proyecto. En concreto hay tres lenguajes que llevan liderando la industria desde hace varios años pero que tienen grandes diferencias entre sí:

- **C/C++:** estos dos lenguajes de programación son, sin lugar a duda, dos de los lenguajes más usados a día de hoy en la industria y con más relevancia a lo largo de la historia de la computación [2]. En el caso del desarrollo de plataformas robóticas ambos lenguajes son prácticamente indispensables durante el desarrollo, esto se debe a varios motivos:
  - Muchas librerías están desarrolladas para estos dos lenguajes y sería extremadamente complicado portearlas a otros lenguajes.
  - C y C++ permiten programar a un nivel mucho más bajo capaz de estar en contacto directo con el hardware de la plataforma, pudiendo así tener un control casi total sobre todos los elementos hardware del sistema.

## 2.2. TECNOLOGÍAS USADAS EN EL DESARROLLO DE PLATAFORMAS ROBÓTICAS

- Dependiendo de la plataforma que se este desarrollando se puede requerir que el rendimiento en tiempo real sea alto y ambos lenguajes son buenas opciones a la hora de desarrollar cualquier sistema en tiempo real.
- **Python:** recientemente este lenguaje de programación ha cogido gran popularidad no solo en el mundo de la robótica si no también en el de la computación, llegando a ser el tercer lenguaje más popular del 2021 [24]. Como es de esperar, en la robótica Python no se queda atrás y son varios los motivos que hacen de este lenguaje una apuesta segura [2]:
  - Comparado con otros lenguajes, Python tiene una sintaxis mucho más sencilla y clara. Esto hace que sea mucho más accesible y fácil de llegar a un punto de alto desarrollo.
  - Hay una gran cantidad de librerías disponibles para el desarrollo de plataformas robóticas y es una parte muy importante de cualquier desarrollo con el sistema operativo ROS.
  - Existen una gran cantidad de plataformas electrónicas de bajo coste que soportan Python, es el caso de la placa Raspberry Pi.
- **Java:** si hablamos de computación es imposible no mencionar la gran importancia que ha tenido, y sigue teniendo a día de hoy, el lenguaje de programación Java [2]. Es cierto que poco a poco Java está perdiendo su liderazgo y está dando a paso a tecnologías nuevas más eficientes, pero aun así este sigue siendo un lenguaje a tener en cuenta a la hora de desarrollar una plataforma robótica por varias razones:
  - Java posee una gran cantidad de herramientas en relación con el desarrollo de aplicaciones de inteligencia artificial y aprendizaje automático.
  - Al ser un lenguaje con tanto recorrido tiene una comunidad muy grande que probablemente ya tenga soluciones para los posibles problemas que podemos encontrarnos durante el desarrollo.
  - La máquina virtual de Java nos permite ejecutar instrucciones en tiempo real en cualquier máquina.

Cualquiera de estos lenguajes es una gran opción a la hora de desarrollar un robot. Para este trabajo de fin de grado desde un principio se acordó utilizar Python.

En esta memoria dedicaremos un capítulo entero a ROS, ya que la importancia que tiene actualmente en la robótica moderna es tal, que es necesario entrar en profundidad en cómo funciona este sistema operativo dedicado íntegramente al desarrollo de plataformas robóticas.

### 2.2.2. Plataformas de procesado de bajo coste

Hoy en día es muy fácil encontrar plataformas de desarrollo de bajo coste que permitan a cualquier persona, con conocimientos básicos en el campo de la computación y de la programación, iniciar sus propios proyectos.

Algunas de las más utilizadas hoy en día (incluso en proyectos importantes) tienen un coste inferior a los 400 dolares. Este es el caso de la serie Nvidia Jetson [4]. Estas son una serie de placas de bajo coste con gran potencia para poder desarrollar cualquier aplicación relacionada con inteligencia artificial. El concepto principal sobre el que se basan estas placas es conseguir crear “computadoras pequeñas y poderosas que permitan la ejecución de múltiples redes neuronales en paralelo” [30].

Ahora mismo existen diferentes versiones, yendo desde la Jetson Nano, hasta la Jetson AGX Xavier Series. Dependiendo de las necesidades técnicas que requiera nuestro proyecto podemos optar por cualquier placa dentro de este rango.

En concreto vamos a hablar un poco de la Jetson TX2, figura 2.1, ya que ha sido la placa con la que hemos trabajado. Esta tiene un tamaño muy reducido pero alberga una gran potencia.

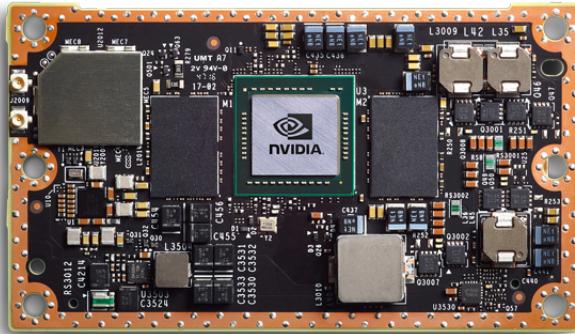


Figura 2.1: Placa NVIDIA Jetson TX2 [21].

Durante el tiempo que trabajamos con ella en el grupo de investigación fuimos capaces de llevar a cabo la instalación de ROS y la ejecución de diversos nodos que capacitaban al robot de movimiento. También probamos, con la ayuda de ROS, a lanzar el software de reconocimiento de acciones obteniendo resultados esperanzadores.

Existen otras placas de bajo coste en el mercado. En concreto nosotros hemos trabajado con Arduino y Raspberry.

## 2.2. TECNOLOGÍAS USADAS EN EL DESARROLLO DE PLATAFORMAS ROBÓTICAS9

- **Arduino:** esta es una placa electrónica de código abierto. El concepto clave de Arduino, a parte de su bajo coste, es que es de hardware y software libre dotando a este proyecto de una gran comunidad [32]. Existen muchos tipos de placas Arduino, pero todas ellas comparten la misma filosofía de funcionamiento. Una de las grandes ventajas que tiene es que funciona sobre cualquier plataforma, ya sea Linux, Windows o Mac OS. Nosotros en concreto utilizamos la placa Arduino A000066, figura 2.2.



Figura 2.2: Placa Arduino A000066 [8].

- **Raspberry Pi:** esta otra placa de bajo coste tiene es muy similar a la placa de Arduino descrita anteriormente. La mentalidad con la que fue creada es la de crear una plataforma educativa capaz de acercar la computación a personas con pocos conocimientos en el campo y siendo económicamente accesible. Aunque pueda parecer un pequeño ordenador sin grandes capacidades este tipo de placas, figura 2.3, nos abren un abanico de posibilidades, desde emular videojuegos retro, hasta tener un ordenador completamente funcional corriendo Linux [31].

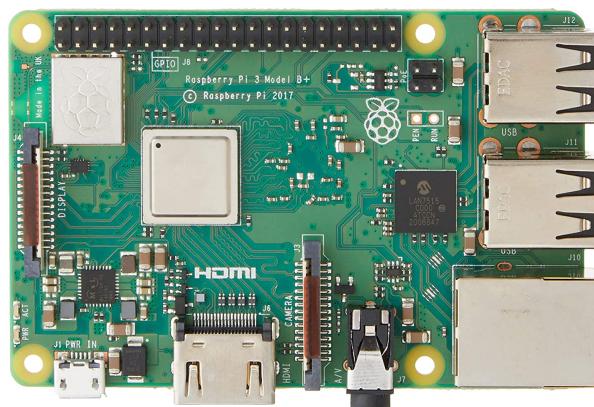


Figura 2.3: Placa Raspberry Pi 3 [10].

## 2.3. La robótica asistencial

Tradicionalmente donde más importancia ha tenido la robótica en los últimos años ha sido en el sector industrial, pero con el avance de ésta a lo largo de los años, las posibilidades de aplicación que se nos ofrecen son muy amplias. En concreto, la robótica asistencial es un campo poco explorado y con un gran potencial [29].

Una plataforma asistencial o robot asistencial tiene como objetivo llevar a cabo una tarea física o cognitiva que una persona con diversidad funcional no puede completar del modo en que lo haría una persona sin diversidad funcional.

Cada plataforma se desarrolla pensando en un aspecto concreto de una discapacidad concreta, intentando así que el robot sea capaz de ayudar a completar la actividad sobre la que se está trabajando.

Hoy en día, el problema de la asistencialidad está más presente que nunca (en parte debido al envejecimiento de la población), es por ello que esta rama de la robótica se encuentra en su punto más álgido de innovación y desarrollo. Existen diversos tipos de robots según las necesidades de la persona: educativos, médicos, terapéuticos... Aunque la mayoría de estos robots necesitan de determinadas habilidades comunes como pueden ser la visión artificial (necesaria para reconocer a la persona que está disponiendo de sus servicios), o la inteligencia artificial (necesaria para poder desarrollar tareas complejas relacionadas con la navegación, por ejemplo). Estas dos suelen ser un punto de referencia para cualquier robot asistencial, ya que se complementan mutuamente[39].

El grupo de investigación Robotics Lab de la universidad Carlos III de Madrid es un claro ejemplo de hacia qué dirección debe avanzar la robótica asistencial moderna. Entre sus proyectos se encuentran *TEO (Task Environment Operator; Operador en el Entorno de las Tareas)* y *MAGGIE*.

También mencionaré en este apartado al investigador *Takanori Shibata*, ya que no podía faltar las referencias a la robótica japonesa siendo ésta uno de los exponentes más importantes y relevantes a día de hoy, y comentaré el importante trabajo que desarrollan ciertas entidades y empresas en España, como puede ser el caso de *Hisparob* y *CARTIF*.

El robot humanoide TEO, figura 2.4, de 1,60m de altura y 60kg de peso dotado de un potente sistema sensorial que incluye un cabezal inteligente con tres cámaras, una unidad de modulo inercial y cuatro sensores de fuerza [26], es uno de los mayores exponentes en cuanto a robótica asistencial que podemos encontrar en España. Esta plataforma nace hace diez años con la intención de llegar a ser un robot asistencial capaz de comunicarse mediante lengua de signos, cosa que a día de hoy ya es capaz de hacer.

Para que esta comunicación por gestos sea lo más orgánica y clara posible se prueban

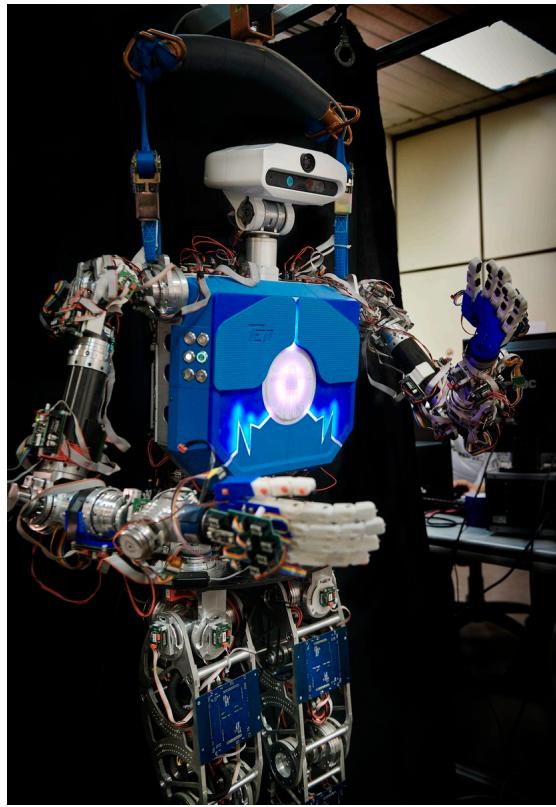


Figura 2.4: TEO Humanoid Robot, Robotics Lab UC3M [27].

varios tipos de redes neuronales hasta que se consigue el mejor resultado posible.

Tras sentar a personas sordas con TEO se llegó a la conclusión de que en el 80 por ciento de los casos la interacción había sido satisfactoria. El objetivo más próximo que tiene este grupo de investigación es la mejora de los gestos y la inclusión de expresiones más complejas capaces de llevar la comunicación aun más lejos [19].

Otra plataforma robótica asistencial desarrollada por el grupo de investigación Robotics Lab es el robot Maggie, figura 3.1. Ésta, a diferencia de TEO, es una plataforma social desarrollada para interactuar directamente con niños, su altura y diseño le dan un aspecto más infantil y amigable.

Entre las funciones de este robot se encuentran la habilidad de reconocer la voz de las personas y acciones, mantener comunicación verbal, moverse por entornos abiertos evitando obstáculos y saber cuando alguien está tocándole dotando a la plataforma de un comportamiento más humano. También posee un panel táctil para darle órdenes, ya que es una plataforma pensada para que gente que no tenga conocimientos de robótica pueda usarla fácilmente [3].

El doctor Takanori Shibata es uno de los investigadores más relevantes en el campo



Figura 2.5: Maggie, Robotics Lab UC3M [9].

de la robótica asistencial a día de hoy. Entre sus reconocimientos más destacables se encuentran el premio al robot más terapéutico del mundo por el Guinness World Records o varios reconocimientos internacionales por su labor en el campo de la robótica terapéutica.

Nuka, figura 2.6, surge después de que Shibata dedicase un largo tiempo al estudio del comportamiento de los bebés foca y se diese cuenta de que un robot que imitase la apariencia y comportamiento de estos mamíferos podría ayudar a ciertas personas dándoles afecto que, en otras circunstancias, podrían encontrar en un animal de verdad.

El robot tiene un peso de 2,5kg, posee un sensor que controla la postura en la que se encuentra, puede reconocer voces, tiene sensores en los bigotes, y responde por un nombre propio. Entre otras cosas, si le acaricias, reconoce que es algo bueno.

Después de probar el robot con distintos perfiles de personas se llegó a la conclusión de que Nuka era capaz de mejorar la calidad de vida de las personas en muchos sentidos, reduce la ansiedad, la depresión, la soledad y mejora comunicación y la sociabilidad de las personas [17].

Actualmente Shibata se encuentra trabajando en el desarrollo de nuevas técnicas y plataformas asistenciales capaces de resolver problemas que presenta la asistencialidad moderna, como es el caso de la salud mental de los astronautas.



Figura 2.6: Nuka y Takanori Shibata [18].

A día de hoy en España cabe destacar el gran trabajo que están haciendo diversas instituciones y empresas, es el caso de Hisparob y CARTIF.

Hisparob es una plataforma tecnológica española de robótica. Esta sirve como punto de encuentro para distintos agentes involucrados en el desarrollo, promoción e integración de tecnologías robóticas [1].

Durante estos últimos años en los que la asistencialidad a mayores se ha vuelto un reto, Hisparob ha organizado diversos eventos y webinars con esta temática. Siendo el último el pasado mes de julio de este mismo año, concentrando en un mismo espacio instituciones tan relevantes en el campo de la robótica asistencial como puede ser el caso de CARTIF o la Universidad Carlos III de Madrid.

CARTIF por su parte es una institución dedicada a la investigación de carácter privado cuyo objetivo principal es proveer de soluciones, sistemas y productos a los desafíos con los que se encuentra actualmente la industria.

Uno de estos desafíos es la atención que necesitan nuestros mayores y los pocos recursos humanos que tenemos para suplir esta carencia. Es por ello que uno de los objetivos de CARTIF es el desarrollo de nuevas técnicas y soluciones que ayuden a mejorar este gran problema al que nos enfrentamos.

Este es el caso de COPITO, figura 2.7. Esta plataforma asistencial pensada íntegramente para su interacción con personas de la tercera edad se encuentra funcionando actualmente en diversas residencias de la comunidad de Castilla y León. Esta plataforma tiene como objetivo el acompañamiento de los ancianos, contribuyendo así al desarrollo de sus capacidades físicas y cognitivas. Sus funciones más destacables son la total adaptación al medio físico en el que se encuentra siendo capaz de navegar autónomamente por escenarios con obstáculos dinámicos, la habilidad de poder reconocer voces y responder a preguntas básicas y seguir ciertas rutinas de manera que les sirvan de recordatorio a

las personas de la tercera edad (como puede ser el caso de ejercicios físicos básicos o la ingesta de medicamentos) [23].



Figura 2.7: Robot Copito de la compañía CARTIF [25].

Estos son solo algunos ejemplos de todos los proyectos que se encuentran a día de hoy en desarrollo. La conclusión que podemos sacar, viendo la cantidad de recursos que se están invirtiendo en robótica asistencial, es que es una rama de la robótica en auge que a lo largo de estos años sufrirá un crecimiento exponencial debido a la necesidad de buscar soluciones en el mundo de la asistencialidad humana.

# Capítulo 3

## Introducción a ROS

Desde que empezamos a plantear con qué tecnologías se iba a desarrollar este trabajo de fin de grado la palabra *ROS* estuvo muy presente. Estas siglas hacen referencia al término *Robot Operating System* que, aunque no es realmente un sistema operativo tal cual conocemos, provee de diversas herramientas y frameworks que simulan las funcionalidades que conocemos de un sistema operativo corriente, como puede ser el caso de la abstracción del hardware. En general, ROS nos va a reducir la complejidad de ciertas tareas de integración e implementación. [43]



Figura 3.1: Robot Operating System [41].

El número de bibliotecas y frameworks que ROS nos ofrece asciende a más de 2000 por lo que también hemos realizado un proceso de selección en el hemos elegido los paquetes que mejor se adaptaban a las funcionalidades que queríamos implementar. Un ejemplo es *RVIZ*. Éste es un paquete de visualizado, diseñado para trabajar conjuntamente con ROS. A lo largo de este capítulo hablaremos más en detalle de éste y otros paquetes que han sido implementados.

En este capítulo vamos a hablar sobre los orígenes de este proyecto para poder entender cual es la filosofía que sigue. Después pasaremos a comentar el funcionamiento de ROS, desde qué es un nodo, hasta como se comunican entre sí, y por último analizaremos los paquetes que hemos usado para llevar a cabo el sistema de navegación de la plataforma.

### 3.1. Orígenes

La historia comienza con Eric Berger y Keenan Wyrobek, por aquel entonces estudiantes de doctorado en la Universidad de *Stanford*. Durante el desarrollo de su tesis doctoral se dieron cuenta de que la mayor parte del tiempo lo dedicaban a reescribir código que ya había sido escrito por otras personas pero que no era del todo compatible con la plataforma que ellos tenían. Es por ello que ROS nace en el año 2007 como consecuencia de la desestructuración tan grande que había en el desarrollo de software para robótica [47].

ROS fue una solución para ahorrarse todo ese tiempo perdido y acercar la robótica a personas del ámbito de la ingeniería que veían este campo como algo lejano y de difícil acceso, es por ello que ROS es un punto de partida dentro de la robótica actual.

En todo momento ROS fue diseñado siguiendo unas premisas claras, como que debía ser de código abierto. Este punto fue de vital importancia para el desarrollo del sistema, ya que en todo momento se enfocaron en que los usuarios que utilizasen ROS dispusiesen de un núcleo al que acceder libremente para adaptar su desarrollo. Esto ha dado lugar a una gran comunidad que ha ido poco a poco mejorando y añadiendo funcionalidades a ROS a través de librerías.

Viendo las posibilidades que tenían de desarrollar un proyecto así empezaron a buscar personas que les acompañasen en la construcción de lo que a día de hoy conocemos como ROS. El punto de inflexión es la llegada de Scott Hassann, quien les proporcionó la financiación necesaria para contratar un equipo de desarrolladores que les ayudasen en la construcción de este proyecto. Así fue como abandonaron la universidad de *Standford* y empezaron a trabajar con todo un nuevo equipo en Willow Garage, laboratorio de investigación robótica enfocado en el desarrollo de software libre con aplicaciones directas en la robótica. A día de hoy OSRF es siendo la compañía encargada del desarrollo y evolución de ROS.

Gracias al trabajo de este equipo y a la filosofía de ROS este se ha convertido en un estándar dentro del mundo de la robótica que sigue en permanente evolución adaptándose a las necesidades de la industria.

### 3.2. Funcionamiento

En este apartado dentro de la introducción al sistema operativo ROS hablaremos sobre su funcionamiento general, de modo que una persona sin conocimientos previos sobre el sistema pueda entender cómo se ha desarrollado este trabajo de fin de grado y, en un futuro, lleve a cabo nuevas implementaciones.

En concreto hablaremos acerca de los principales objetivos que diferencian a ROS sobre el resto de sistemas que le hacen competencia. Continuaremos hablando sobre el sistema de archivos interno que maneja, dando una perspectiva general para así entender cómo se encuentra estructurado. Pasaremos a comentar la base sobre la que funciona cualquier proyecto en ROS: los nodos y los topics. Y por último, pasaremos a comentar que son los servicios y parámetros y cómo podemos, mediante el roslaunch, lanzar una estructura de nodos completa.

### 3.2.1. Objetivos de ROS

Antes de empezar a definir los objetivos que ROS espera cumplir, hay que dejar claro que éste es un sistema distribuido. Esto significa que la comunicación entre sus componentes, en este caso nodos, se lleva a cabo mediante el intercambio de mensajes. Esto permite que se puedan diseñar pequeños módulos independientes unos de otros, facilitando así la posibilidad de reutilizarlos en otros sistemas. Al punto al que queremos llegar es que la estructura de ROS es de esta manera debido a que está pensado para que la reusabilidad del código sea un elemento clave de su estructura.

A parte de este objetivo central de conseguir que ROS sea un punto de encuentro para desarrolladores del mundo de la robótica, existen otra serie de objetivos que intenta cumplir [44]:

- **Liviano:** uno de los objetivos más importantes de ROS es conseguir que éste sea un sistema liviano, de manera que cualquier sistema desarrollado en ROS puede ser utilizado por otro hardware o software.
- **Bibliotecas:** ROS sigue un modelo de desarrollo basado en el uso de bibliotecas agnósticas con interfaces funcionales limpias.
- **Independencia del lenguaje:** a la hora de implementar ROS en un sistema se pueden elegir varios lenguajes para hacerlo. Los más comunes (y con más garantías) son Python, C++ y Lisp, aunque se disponen de librerías en Java y Lua.
- **Escalado:** ROS está pensado para funcionar eficientemente en sistemas con grandes procesos de desarrollo y rutinas de ejecución.

### 3.2.2. Sistema de archivos

En esta sección vamos a ver cómo funciona el sistema de archivos de ROS y cuáles son los conceptos clave que hay que tener en cuenta a la hora de navegar por este sistema [?].

Existen dos conceptos básicos sobre los que se basa todo este sistema de archivos:

- **Paquetes:** un paquete es la unidad básica sobre la que se basa el sistema de archivos de ROS. Cada paquete puede albergar scripts, ejecutables, bibliotecas, ...
- **Manifiestos:** un manifiesto es una descripción de un paquete. Estos son archivos xml que sirven para indicar las dependencias que hay entre paquetes y para capturar información útil relativa al paquete, como puede ser la licencia de éste o la versión.

Durante el desarrollo de nuestra aplicación ROS distribuiremos el código a lo largo de muchos paquetes, de manera que iremos creando la estructura de paquetes que nosotros queramos. Como podemos llegar a tener una estructura muy extensa y encontrar un determinado paquete mediante comandos de la terminal de Linux puede ser un poco tedioso, ROS nos proporciona la herramienta **rospack** que nos permite obtener información de manera rápida sobre paquetes. Para utilizarla simplemente debemos utilizar el comando “rospack find [nombre del paquete]”.

Como es de esperar, también existe una herramienta para cambiar de paquetes. **rosbach**. Funciona de manera similar a **rospack**, simplemente debemos utilizar el comando “roscd [nombre del paquete]”. Gracias a estas dos herramientas se nos facilita mucho la navegación entre paquetes.

Más adelante veremos cómo se crea un paquete ROS y como se puede ir generando toda esta estructura de paquetes y manifiestos.

### 3.2.3. Nodos

Un nodo no es más que un proceso que realiza una función concreta. Cada uno de estos nodos se compila de forma individual, de manera que existe una independencia total del resto de nodos (salvo el master). La comunicación entre nodos se lleva a cabo mediante el uso de mensajes y topics [\[42\]](#).

El objetivo de haber desarrollado un diseño modular en el que cada nodo se encarga de una tarea concreta para luego comunicarse con el resto y llegar a cumplir un objetivo común, se debe a la filosofía que ROS quiere cumplir. Es así como se consigue que la reutilización del código sea posible, de manera que puedes coger un nodo aislado e implantarlo en otro sistema completamente diferente.

El uso de nodos en ROS proporciona varios beneficios al sistema en general. Existe una tolerancia a fallos adicional ya que las caídas se aíslan en nodos individuales. La complejidad del código se reduce en comparación con los sistemas monolíticos. Los detalles de implementación también están bien ocultos, ya que los nodos exponen una API mínima al resto de la estructura y las implementaciones alternativas, incluso en otros lenguajes de programación, pueden sustituirse fácilmente. Como se puede apreciar, la implementación

de nodos beneficia enormemente al sistema en términos generales. A la hora de hablar de la comunicación entre nodos hay dos términos que son de vital importancia para como funciona esta:

- **Publisher:** el publisher es el medio que utiliza un nodo para enviar información a un topic.
- **Subscriber:** el subscriber es el medio que utiliza un nodo para recibir información de un topic.

De esta manera, un nodo puede elegir o bien publicar cierta información a un topic concreto (veremos a continuación qué es un topic) o, de lo contrario, recibir cierta información de un topic. De este modo se consigue que cada nodo dentro de una estructura de nodos ROS, pueda enviar y recibir información manteniendo la independencia de ejecución que tanto caracteriza a ROS.

A la hora de identificar cada nodo, ROS crea un nombre independiente para cada nodo en ejecución. Hay ciertos nodos que ROS tiene de base con un nombre ya asignado, pero nosotros podemos darle a cada nodo que creemos un nombre concreto según la función que desarrolle éste. A la hora de crearlos, ROS nos proporciona una librería para poder programar las acciones a desarrollar por un nodo. Esta es la librería cliente de ROS y está disponible en varios lenguajes, aunque nosotros la hemos utilizado en Python [38].

Uno de los nodos más utilizados en cualquier sistema que implemente ROS es el **move base** [34]. Este nodo dado por la propia arquitectura de ROS consigue que, dado un punto en el mundo real, la plataforma sea capaz de alcanzarlo. Para que esto funcione se necesitan de más nodos y de una arquitectura más extensa, pero **move base** es un estándar en ROS que es implementado en cualquier sistema robótico que navegue por un entorno real.

Como ya hemos comentado antes, cada nodo ROS es independiente y se comunican entre si gracias el envío y recepción de mensajes mediante topics, pero aún así existe un nodo maestro que controla el resto de nodos. El **ROS Master** proporciona servicios de nomenclatura y registro a cada uno de los nodos de nuestra estructura. El papel que desempeña este nodo en un sistema ROS es el de permitir que los nodos se ubiquen, de manera que una vez ubicados estos empezarán a comunicarse entre sí. Es necesario que el nodo maestro este en ejecución durante el desarrollo del proceso, para ello simplemente hay que utilizar el comando **roscore**.

### 3.2.4. Topics

En la sección anterior cuando hablábamos de nodos ha salido la palabra **topics**. Éste término lo hemos utilizado a la hora de hablar sobre la comunicación entre nodos. Un topic no es más que un punto de encuentro en el que dos o más nodos intercambian información. Para llevar a cabo este intercambio el topic le da a cada nodo dos opciones:

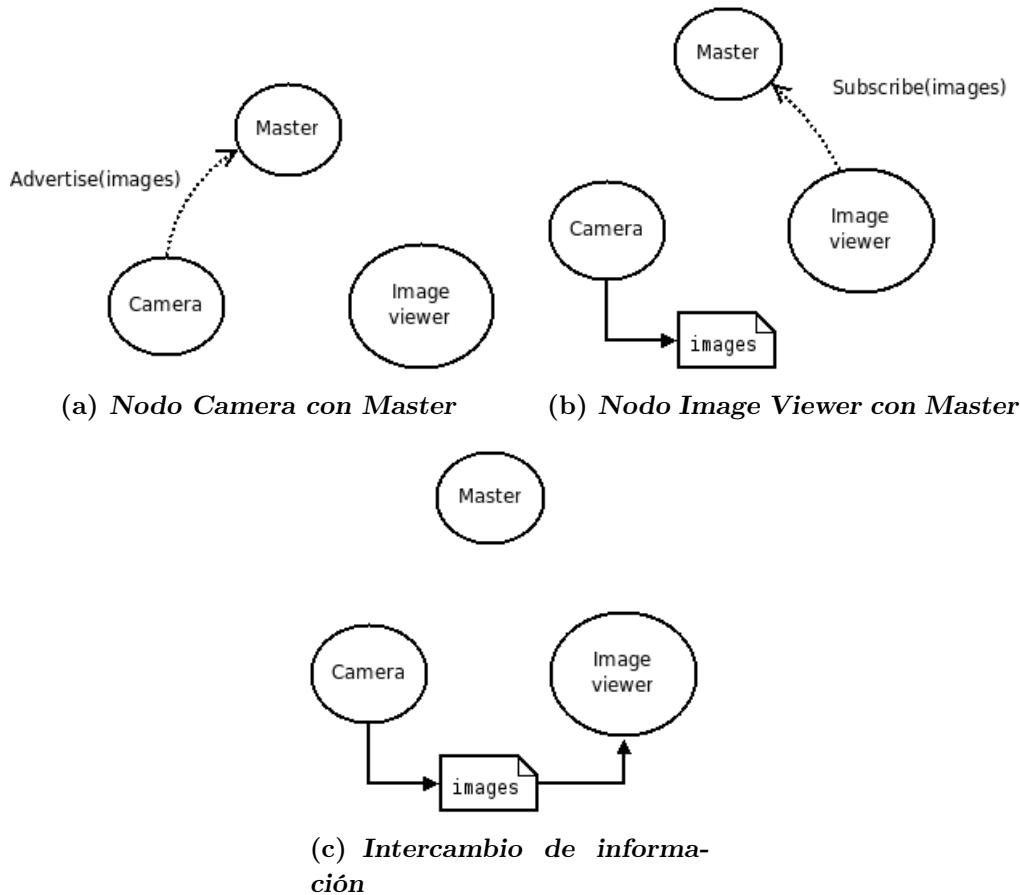


Figura 3.2: Ejemplo de comunicación entre dos nodos [20].

- **Publicar:** el nodo publica cierta información en el topic para que otro nodo pueda recogerla.
- **Suscribirse:** el nodo se suscribe al topic y es capaz de recoger la información que otro nodo ha depositado en el con anterioridad.

En general, los nodos no son conscientes de con quien se están comunicando, si no que ellos depositan y recogen la información sin saber quien la va a usar o quien la ha dejado. De esta manera se consigue una comunicación en tiempo real unidireccional.

Cada topic esta determinado por el tipo de mensaje ROS que publica, de este modo los nodos solo pueden recibir mensajes con un tipo coincidente. Aquí el nodo maestro no se encarga de comprobar si los tipos coinciden, de manera que si durante la comunicación entran en juego tipos no coincidentes esta simplemente no se llevará a cabo [33].

La comunicación que se establece entre nodos se lleva a cabo mediante el protocolo de transporte TCP/IP basado en UDP. Este se conoce normalmente como **TCPROS** y transmite los mensajes a través de conexiones TCP/IP persistentes. El transporte basado en UDP, que se conoce como **UDPROS**, solo es compatible con roscpp (biblioteca C++).

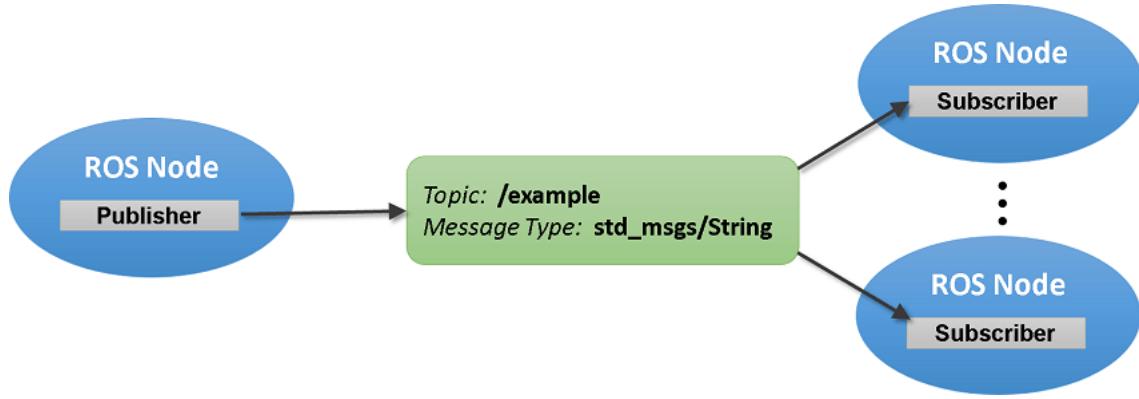


Figura 3.3: Funcionamiento de un topic [13].

de ROS). Este es un transporte con pérdidas y de baja latencia, por lo que es más adecuado para tareas como la teleoperación.

Los nodos ROS negocian el protocolo de transporte deseado en tiempo de ejecución, de manera que si un nodo prefiere el protocolo de transporte UDPROS pero el otro no lo admite, pueden recurrir al protocolo de transporte TCPROS. Este modelo dinámico permite al sistema agregar nuevos transportes con el tiempo a medida que surgen casos de uso [33].

Gracias a la librería cliente de ROS podemos crear nuestros propios topics según las necesidades que tenga nuestro sistema, pero ROS nos da de base algunos topics que son de gran utilidad y de uso casi obligatorio en cualquier implementación de este sistema operativo. Un ejemplo es el topic **cmd vel**, cuyos mensajes son del tipo Twist. Este tipo de mensajes están formados por un componente lineal ( $x, y, z$ ) correspondiente a las velocidades y otro angular ( $x, y, z$ ) correspondiente a la tasa angular.

ROS nos proporciona una herramienta para trabajar de una manera más cómoda con los topics, esta se conoce como **rostopic**. Esta nos ayuda a obtener información sobre los topics que están siendo ejecutados. Existen diferentes opciones para este comando, aquí simplemente vamos a comentar brevemente los que más hemos usado durante el desarrollo de este trabajo de fin de grado [46]:

- **rostopic list**: mediante el uso de este comando podemos visualizar una lista de todos los topics que en ese mismo momento se encuentran manejando información en nuestro sistema.
- **rostopic echo [topic]**: este comando nos muestra la información que está siendo publicada por un topic determinado. Este comando es de gran utilidad, ya que en tiempo de ejecución nos permite ver el transito de información de nuestro proyecto.
- **rostopic type [type]**: como es de esperar, este comando nos devuelve el tipo del mensaje que un topic determinado maneja. Gracias a este comando podemos ver si

en algún momento del desarrollo de nuestra arquitectura hemos puesto dos nodos en contacto con distinto tipo de mensajes.

- **rostopic pub [topic] [tipo del mensaje] [mensaje]**: este comando nos permite publicar mensajes en un topic en tiempo real. Esta es una herramienta bastante utilizada, ya que nos permite comprobar en tiempo real las respuesta de nuestro sistema a diferentes argumentos.
- **rostopic hz [topic]**: usando este comando podemos obtener el tiempo de publicación de los mensajes en un topic determinado.

### 3.2.5. Servicios y parámetros

#### 3.2.5.1. Servicios

A parte de la comunicación mediante topics existe otro método menos convencional, los **servicios**. “Los servicios son otra forma en que los nodos pueden comunicarse entre sí. Los servicios permiten a los nodos enviar una **solicitud** (request) y recibir una **respuesta** (response)” [45].

El motivo de la existencia de este método de comunicación se debe a que en ocasiones no queremos que un cierto nodo esté publicando información de manera continua o que cualquiera pueda acceder a esa información en cualquier momento, si no que podemos querer que un nodo desempeñe una tarea puntual y luego deje de funcionar durante un tiempo, es aquí donde entran en juego los servicios [28]. Para entender los servicios debemos conocer dos términos fundamentales:

- **Servidor**: nodo que ejecuta una tarea concreta
- **Cliente**: nodo que necesita de la tarea ejecutada por el servidor.

Para trabajar con los servicios, ROS nos ofrece una herramienta propia llamada **rosservice**. Al igual que el resto de herramientas por comandos que hemos visto con anterioridad, existen múltiples opciones pero solo vamos a ver las más relevantes:

- **rosservice list**: nos devuelve la lista de servicios que se encuentran funcionando en ese mismo momento.
- **rosservice type [service]**: nos devuelve el tipo de los mensajes que un servicio maneja.
- **rosservice call [service] [mensaje]**: mediante el uso de este comando podemos enviar un mensaje a un servicio.

### 3.2.5.2. Parámetros

Cuando uno tiene una aplicación ROS montada con una gran estructura de paquetes y nodos, suele ser normal que quiera establecer una configuración global para el sistema. Un ejemplo de elementos a configurar podrían ser:

- El nombre por el que conoces a tu plataforma.
- La velocidad por defecto de las ruedas.
- La frecuencia con la que se leen los sensores.
- Un parámetro que indique si la plataforma está funcionando sobre un entorno real o es una prueba de simulación.

Estos son solo algunos ejemplos de elementos que puedes querer tener en tu configuración global. De esta manera evitas tener que ir nodo a nodo indicando qué configuración debe seguir. Es aquí donde entra en juego el **servidor de parámetros ROS**. Después de lanzar el nodo maestro, un servidor de parámetros se crea automáticamente dentro de este nodo [5]. Este servidor de parámetros es, básicamente, un gran diccionario de variables accesible por cualquier nodo que se encuentre en la estructura del nodo maestro.

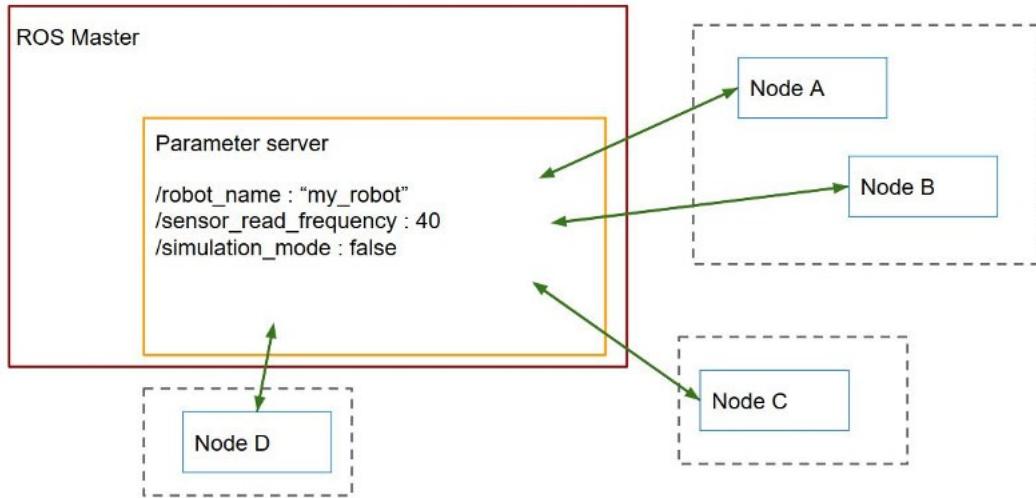


Figura 3.4: Funcionamiento del servidor de parámetros [12].

En la figura 3.4, se puede ver una pequeña implementación de un servidor de parámetros. Este contiene tres parámetros:

- El nombre de la plataforma (string).
- La frecuencia de lectura del sensor (int).
- Flag que indica si se ejecuta sobre un entorno de simulación (boolean).

Como indica la figura, cada nodo dentro de la arquitectura del nodo maestro puede acceder a cualquiera de estos parámetros sin necesidad de tener un contacto intermedio con nadie. A parte, los nodos también son capaces no solo de leer los parámetros del servidor, si no de modificarlos o crear unos nuevos.

Al igual que con los servicios, ROS nos proporciona una herramienta para trabajar con el servidor de parámetros, **rosparam**. Vamos a pasar a ver las opciones de comando más útiles y que nosotros hemos utilizado durante el desarrollo de este trabajo fin de grado [45]:

- **rosparam list**: devuelve todos los parámetros que se encuentran actualmente en el servidor de parámetros.
- **rosparam set [nombre del parámetro] [valor]**: cambia el valor de un parámetro a uno que nosotros le demos.
- **rosparam get [nombre del parámetro]**: devuelve el valor del parámetro que nosotros le indiquemos.

### 3.2.6. Roslaunch

Una aplicación real utilizando ROS puede tener una estructura de unos 20 o 30 nodos, por lo que ir ejecutando cada nodo de manera individual no es una opción. Es por ello que ROS incluye una herramienta llamada **roslaunch** que nos facilita enormemente este trabajo. **Roslaunch** nos va a permitir ejecutar múltiples nodos de manera automática y establecer parámetros en el servidor de parámetros [22].

Esto se lleva a cabo mediante el uso de archivos xml que tienen como extensión **.launch**. Todos los archivos launch que nos encontramos van a tener el mismo formato, por lo que no es necesario aprender a programar en xml. También hay que destacar que al lanzar un archivo launch automáticamente se va a ejecutar el nodo maestro ROS, por lo que no será necesario volver a lanzar roscore.

A continuación vamos a ver un ejemplo de lo que podría ser un archivo launch real:

```

1 <launch>
2   <rosparam param="/LOLA/name"> LOLA </rosparam>
3   <node name="nodo_1" pkg="LOLA_package" type="LOLAnode">
4     <node name="nodo_2" pkg="LOLA_package" type="LOLAnode">
5       <include file="$(find lola_pkg)/launch/rutina_movimiento.launch">
6 </launch>
```

- En la primera línea definimos el valor de un parámetro del servidor de parámetros con nombre “name” y le ponemos el valor “LOLA”.
- En la segunda línea iniciamos el “nodo 1” que se encuentra dentro del paquete “LOLA package”.
- En la tercera línea iniciamos el “nodo 1” que se encuentra dentro del paquete “LOLA package”.
- En la cuarta línea incluimos el archivo “rutina movimiento”, que a su vez es otro archivo launch. De esta manera al ejecutar este archivo launch ejecutaremos a su vez otro archivo launch.

Todos los archivos launch que nos encontremos van a seguir la misma estructura que acabamos de describir, con más o menos líneas dependiendo de la estructura que tenga el proyecto. Esta es una herramienta muy útil que nos ha ayudado mucho a la hora de desarrollar la plataforma robótica asistencial utilizando ROS.

### 3.3. Paquetes que se han utilizado

Como ya hemos comentado en la introducción de este capítulo, ROS es un sistema que tiene como máxima la reutilización de código para así crear una gran comunidad que colabore en distintos proyectos relacionados con el desarrollo de plataformas robóticas. Es por ello que desde el nacimiento de ROS la comunidad ha ido creando diferentes paquetes que buscaban solucionar ciertos problemas o proveer de ciertas herramientas de las que ROS no disponía. Muchos de estos paquetes y bibliotecas han alcanzado tal popularidad que ROS los ha adquirido como propios y se encuentran en su web oficial.

A la hora de plantearnos cómo íbamos a implementar la navegación autónoma de la plataforma manejamos varias opciones, pero al final terminamos por implementar dos paquetes que trabajarían de forma conjunta, el **Navigation Stack** y **RVIZ**.

#### 3.3.1. Navigation Stack

La idea principal que impulsa el desarrollo del Navigation Stack es la creación de un conjunto de paquetes que permitan a una plataforma robótica moverse por un entorno real con obstáculos estáticos y dinámicos, sin chocarse y llegando a su destino con una ruta óptima y en el menor tiempo posible. Para poder conseguir esto, el Navigation Stack recoge muchos tipos de información distinta procedentes de diversos topics y nodos [15]. Es un sistema tan potente que aunque la plataforma se quede atascada a lo largo del recorrido, puede recuperarse una vez vuelva a encontrarse en el mundo real y recalcule una nueva ruta. El Navigation Stack es uno de los puntos más fuertes que tiene ROS como

sistema y, a día de hoy, es impensable que una plataforma que navegue autonomamente y utilice ROS no implemente el Navigation Stack.

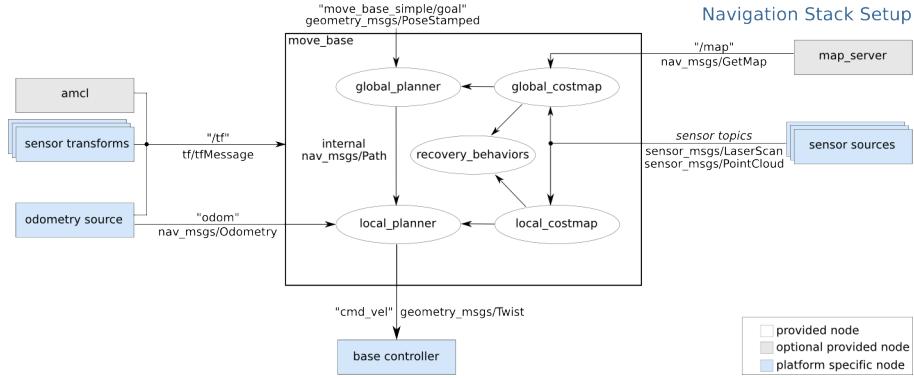


Figura 3.5: Vista de alto nivel del nodo move base y su interacción con otros componentes [35].

En la figura 3.5, tenemos la jerarquía de nodos y topics que el Navigation Stack requiere para funcionar. Los nodos en azul pueden variar según la plataforma que tengamos, los grises son totalmenteopcionales y los blancos son obligatorios, ya que forman la base de la navegación.

A nivel conceptual es bastante simple entender qué es y qué hace el Navigation Stack, pero conseguir que este funcione en cualquier plataforma es más complicado. Hay una serie de requisitos que debe cumplir cualquier plataforma para poder lanzar el Navigation Stack:

- El sistema debe tener ROS instalado y funcionando.
- Tener un árbol de transformación **tf**.
- Publicar los datos de los sensores utilizando los tipos de mensajes ROS correctos.

A parte de estos tres puntos citados, el Navigation Stack necesita ser configurado para la forma y dinámica de la plataforma en la que va a ser ejecutado, ya que cada cual tiene unas dimensiones distintas.

El Navigation Stack está diseñado para ser lo más general posible, pero al hablar del hardware los requisitos son más estrictos:

- Está diseñado para plataformas con ruedas de accionamiento diferencial y holonómicos.

- Se asume que la base móvil se controla enviando los comandos de velocidad deseados en formato: velocidad x, velocidad y, velocidad theta.
- Requiere un láser plano montado sobre la base móvil de la plataforma. Este láser es utilizado para construcción y localización de mapas.
- El Navigation Stack fue desarrollado en una plataforma robótica cuadrada, por lo que su rendimiento será óptimo con plataformas de dimensiones similares. También funciona en plataformas con distintos tamaños, pero puede tener dificultades con robots rectangulares grandes, en espacios estrechos como puertas.

### 3.3.2. RVIZ

Otro de los paquetes que hemos utilizado para la implementación de la navegación autónoma de la plataforma ha sido **RVIZ**. Este es un paquete de visualización 3D que nos permite visualizar información por pantalla utilizando una gran cantidad de topics. RVIZ nos va a permitir ver el mapa 2D de la planta de la escuela Politécnica de Alcalá de Henares sobre la que se va a efectuar la navegación. A parte, esta librería permite interactuar directamente con el mapa, pudiendo seleccionar el punto de origen de la plataforma y hacia donde queremos que se dirija. Su uso es muy sencillo, ya que solo requiere de unos pocos clicks para poner la plataforma en marcha.

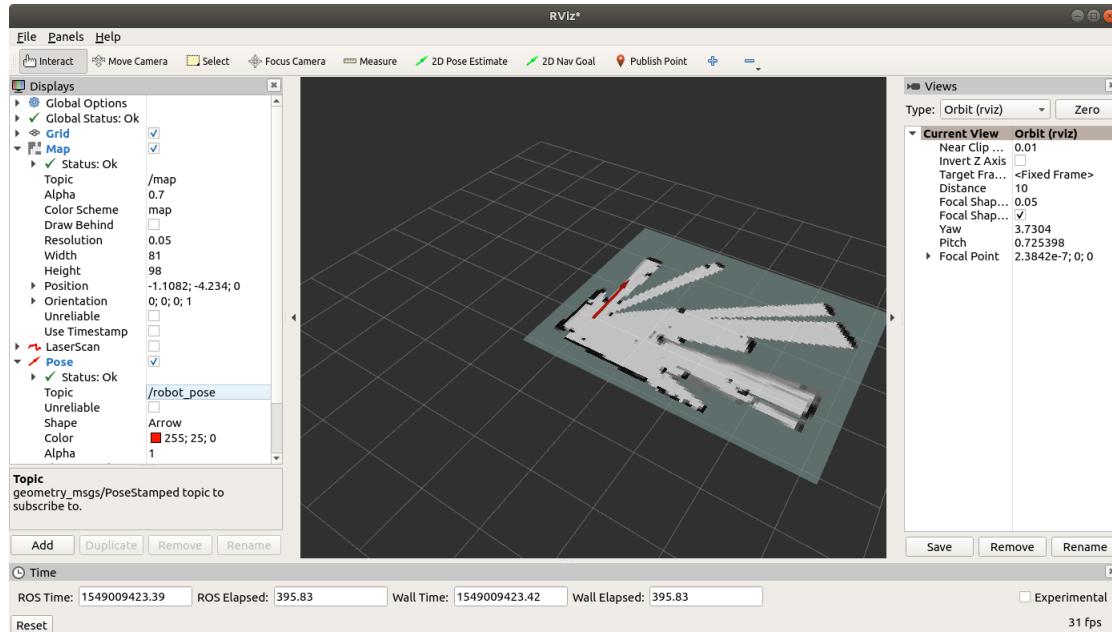


Figura 3.6: RVIZ en funcionamiento [11].

### 3.4. ROS2

A lo largo de los años han sido muchas las versiones de ROS que se han lanzado al mercado. Lo normal era tener una versión de ROS por año, pero esto cambió radicalmente cuando en la ROSCon del año 2014 fue anunciado ROS2. El motivo del lanzamiento de esta nueva API es sencillo, la robótica, como cualquier otro campo dedicado al desarrollo tecnológico, está en constante evolución por lo que ROS debía adaptarse a las nuevas demandas del mercado.

Esto no significa que ROS haya dejado de ser de utilidad después del lanzamiento de ROS2, ya que solo hace tres años desde la salida de una versión estable de ROS2. Será una transición orgánica en la que ROS pasará a un segundo plano debido a las ventajas que irá presentando ROS2 sobre su hermano pequeño.

Las principales diferencias que nos encontramos entre los dos sistemas son las siguientes [36]:

- **Soporte de lenguajes:** ahora mismo la forma más común de trabajar con ROS es usando Python 2.7 o C++ 11. Si se está un poco al corriente de las versiones que manejan a día de hoy ambos lenguajes de programación es fácil observar que estas versiones se encuentran un poco anticuadas. Es por ello que ROS2 permite trabajar, entre otros lenguajes, con Python 3.5 y C++ 17, esto es una clara ventaja sobre ROS.
- **Soporte de sistemas operativos:** a día de hoy ROS funciona exclusivamente en Linux (no comentaremos su funcionamiento en Mac OS ya que sigue siendo un tema de debate), mientras que ROS2 es capaz de correr en Windows 10. Es cierto que la mayoría de desarrollos se basan en una arquitectura Linux, pero que sea capaz de funcionar en Windows abre la puerta a un nuevo mundo de posibilidades.
- **Modelo distribuido:** como hemos comentado anteriormente en la sección de *funcionamiento*, ROS se basa en un modelo centralizado en el que el “master” es el encargado de informar a cada nodo de la existencia del resto de nodos que forman la arquitectura. En ROS2 el “master” desaparece, dando lugar a un modelo distribuido en el que cada nodo se encarga de informar a los otros de su existencia.
- **Protocolo de transporte:** en ROS la comunicación se establecía mediante el protocolo TCPROS basado en el protocolo TCP/IP. El protocolo sobre el que se basa la comunicación en ROS2 es DDS, el motivo de este cambio es la necesidad de permitir comunicaciones en tiempo real, a parte de que otorga gran flexibilidad y eficiencia a las comunicaciones.

Para este trabajo de fin de grado decidimos utilizar la primera versión de ROS, ya que las necesidades que teníamos que cubrir podían ser resueltas sin necesidad de entrar en ROS2.

# Capítulo 4

## Plataforma LOLA

A lo largo de este capítulo veremos cómo se ha conseguido integrar el sistema operativo ROS y modelos de inteligencia artificial en la plataforma robótica asistencial LOLA. Explicaremos paso a paso todo el trabajo hecho a lo largo de estos meses para que cualquiera que siga esta memoria pueda replicar el trabajo hecho y consiga poner en marcha la plataforma.

Para ello este capítulo se componen de seis secciones:

- **Elementos físicos de la plataforma, sección 4.1:** sección en la que se describirán todos los elementos que conforman la plataforma LOLA obteniendo una idea detallada de qué elementos se necesitan para poder llevar a cabo tareas de navegación autónoma y reconocimiento de acciones.
- **Arduino, sección 4.2:** comentaremos cuales son las funciones que desempeña esta placa de Arduino en el conjunto de elementos que conforman la plataforma, hablaremos de como está programada y de cómo se pone en funcionamiento paso a paso.
- **ROS, sección 4.3:** ya hemos dedicado un capítulo entero a introducir este sistema operativo y ahora describiremos paso a paso cómo ha sido la integración de este. Iremos poco a poco, desde cómo instalar ROS en un sistema Ubuntu, hasta la explicación de cómo se ha programado toda la estructura para que la navegación autónoma funcione correctamente.
- **Primeros pasos con ROS y Python, sección 4.4:** teniendo ya ROS instalado será el momento de ver como se puede programar, mediante el uso de Python, una pequeña arquitectura de nodos ROS que permita navegar mediante el uso de las teclas de un teclado a nuestra plataforma.
- **Navegación autónoma, sección 4.5:** una vez comprobado que la placa de Arduino y ROS se encuentran funcionando correctamente, pasaremos a ver cómo se

lanza todo el sistema de navegación autónoma. Aprenderemos a entender la información proporcionada por ROS en tiempo real y a ajustar ciertos parámetros que son completamente circunstanciales.

- **Reconocimiento de acciones online, sección 4.6:** a parte del desarrollo del sistema de navegación de la plataforma, se ha llegado a implementar un nodo ROS remoto capaz de recoger imágenes y pasárselas a una CPU de mayor potencia capaz de detectar la acción que se está desarrollando en tiempo real. Analizaremos cómo se ha incorporado este nodo ROS remoto, para que en un futuro alguien pueda añadir otras tareas complejas.

## 4.1. Elementos físicos de la plataforma

La plataforma robótica asistencial en la que se han implementado los modelos de inteligencia artificial y el sistema operativo ROS tiene como nombre LOLA. Este robot ha sido desarrollado por el grupo de investigación GRAM.

Este es un robot diferencial con ruedas equipado con dos motores y sus correspondientes encoders, ambos controlados por una placa de Arduino Mega. Todo el diseño mecánico y eléctrico ha sido desarrollado por el grupo de investigación. La estructura interna está construida en madera y metal, y la capa exterior (la cual imita a una persona con esmoquin) ha sido realizada mediante impresión 3D.

La plataforma funciona con dos baterías e incluye una interfaz de controlador eléctrico que permite un mejor gestión de la interconexión de las diferentes partes del sistema y de la gestión de energía. Todo el sistema eléctrico se puede alimentar con 24 o 12 Voltios. Para las necesidades que requiere nuestro sistema hemos decidido optar por una alimentación de 12 Voltios, reduciendo así el máximo de velocidad que pueden proporcionar los motores.

Las dos ruedas motorizadas tienen 190 mm de diámetro y 590 mm de distancia al eje. La plataforma completa mide aproximadamente 800 mm, es un poco más alta que una mesa.



Figura 4.1: Sensores de la plataforma LOLA.

Para poder llevar a cabo el sistema de navegación es necesario incluir una serie de sensores que proporcionen la suficiente información del entorno como para permitir la navegación:

- **LIDAR:** este sensor LIDAR nos va a permitir mapear el entorno, siendo capaz de detectar obstáculos y mejorando la navegación de la plataforma. Este sensor es imprescindible para que la navegación funcione, ya que aún teniendo el mapa sobre el que se va a producir la navegación el robot debe de ser capaz de adaptarse a obstáculos dinámicos, más siendo esta una plataforma asistencial pensada para interactuar con personas con capacidades especiales. En concreto hemos utilizado el modelo **RPLIDAR A1** de la empresa Slamtec, aquí están las especificaciones [6]:

- **Rango de medición:** 0.15mm - 12mm
- **Frecuencia de muestreo:** 8K
- **Velocidad de rotación:** 5.5Hz
- **Resolución angular:** 1°
- **Dimensiones:** 96.8 x 70.3 x 55mm
- **Voltaje del sistema:** 5V
- **Corriente del sistema:** 100mA
- **Consumo de energía:** 0.5W
- **Rango angular:** 360°

No es necesario instalar ningún tipo de paquete para poder hacer funcionar el LIDAR, simplemente hay que conectarlo mediante USB a la computadora que este usando la plataforma y se pondrá en funcionamiento inmediatamente. El precio de este LIDAR es de unos 100 euros.

- **Pantalla táctil:** actualmente este sensor está incorporado en la plataforma pero no tiene gran utilidad, de momento despliega un gif de una cara sonriente para dar a la plataforma un toque más amigable. También sirve para acceder a la computadora del robot, sirviendo esta pantalla como monitor.

En un futuro se espera otorgar a esta pantalla de más utilidad haciendo que esta sea un elemento de contacto directo entre el paciente y la plataforma, de manera que la persona que esté interactuando con el robot pueda indicarle alguna acción a través de esta pantalla táctil.

- **Cámara frontal:** para que la plataforma pueda navegar no es necesario que lleve una cámara frontal, si no que esta va a servir para capturar imágenes que, posteriormente, serán enviadas mediante un nodo ROS remoto a otra GPU más potente

y procesadas, obteniendo así la acción que se está llevando a cabo por la persona que se encuentra en frente del robot.

Puede que a la hora de hacer funcionar la cámara sea necesario instalar algún tipo de paquete si la versión de Ubuntu no lo trae por defecto. En nuestro caso instalamos **Cheese**, este es un paquete que habilita la captura de vídeo e imágenes si se tiene una cámara conectada [37].

El comando de instalación de este paquete es el siguiente:

```
1 $ sudo apt-get install cheese
```

Si se ha instalado correctamente simplemente tendremos que poner el comando:

```
1 $ cheese
```

y se nos desplegará una ventana con las imágenes que la cámara está capturando.

- **Arduino Mega**: la funcionalidad y el fin de esta placa de Arduino lo comentaremos en la siguiente sección, ya que es tal su importancia que requiere de un análisis de mayor profundidad. Esta es una placa Arduino Mega:

- **Voltaje del sistema**: 5V
- **Voltaje de entrada (recomendado)**: 7 - 12V
- **Voltaje de entrada (límite)**: 6 - 20V
- **Pines de E/S digital**: 54 (de los cuales 15 proporcionan salida PWM)
- **Pines de entrada analógica**: 16
- **Corriente CC por pin de E/S**: 20mA
- **Corriente CC para pin de 3.3V**: 50mA
- **Memoria flash**: 256KB (de los cuales 8KB los utiliza el gestor de arranque)
- **SRAM**: 8KB
- **EEPROM**: 4KB
- **Velocidad del reloj**: 16MHz
- **LEDS**: 13
- **Dimensiones**: 101.52 x 53.3 mm
- **Peso**: 37g

Esta placa tiene un coste de unos 35 euros, por lo que cumple con el objetivo del desarrollo de una plataforma de bajo coste.

- **NVIDIA Jetson TX2:** al igual que con la placa de Arduino, la funcionalidad y el fin de este elemento se comentará más adelante, en concreto en la sección de ROS. Actualmente la plataforma cuenta con este elemento en su estructura, pero es cierto que la mayoría de pruebas han sido realizadas en un portátil de características similares en cuanto al software. El motivo de esto es que durante el desarrollo e implementación de ROS y de los modelos de inteligencia artificial nos dimos cuenta de que era necesario el uso de Ubuntu 18.04, mientras que en la Jetson TX2 se tenía instalado la versión de Ubuntu 16.04.

Hubo un problema con una de las conexiones de la placa y no pudimos cambiar el sistema operativo, por lo que tuvimos que utilizar un portátil como sustitución a este elemento. En la práctica esto no cambia nada de la arquitectura de la plataforma, ya que todo lo que se explica en esta memoria es replicable en cualquier computador que corra Ubuntu 18.04.

Las especificaciones técnicas de la NVIDIA Jetson TX2 son [16]:

- **GPU:** 256-core NVIDIA Pascal™ GPU architecture with 256 NVIDIA CUDA cores
- **CPU:** Dual-Core NVIDIA Denver 2 64-Bit CPU Quad-Core ARM® Cortex®-A57 MPCore
- **Memoria:** 8GB 128-bit LPDDR4 Memory 1866 MHx - 59.7 GB/s
- **Almacenamiento:** 32GB eMMC 5.1
- **Alimentación:** 7.5W / 15W
- **Dimensiones:** 50mm x 87mm

El precio de esta placa es de 399 dólares. Viendo las especificaciones técnicas podemos darnos cuenta fácilmente que está totalmente preparada para cualquier desarrollo que implica algo de potencia computacional, como puede ser reconocimiento de imágenes y acciones o redes neuronales.

## 4.2. Arduino

### 4.2.1. Objetivos

La placa de Arduino es un elemento fundamental dentro de la arquitectura de nuestra plataforma. Esta permite establecer la comunicación bidireccional entre los motores del robot y el computador de abordo, ya sea la NVIDIA Jetson TX2 o un ordenador portátil.

La necesidad de esta comunicación entre estos dos elementos viene dada por la integración del sistema de navegación autónoma. Para poder integrar este sistema es necesario tanto enviar comandos de velocidad a los motores del robot como recibir información de los encoders para así crear un sistema de control.

Esto lo conseguimos gracias a la inclusión de la placa de Arduino. Esta se va a encargar de enviar la información necesaria a los motores y al computador de abordo. Para ello ha sido programada siguiendo las necesidades de nuestro trabajo. Nosotros simplemente hemos adaptado alguna función del código de Arduino, ya que en ocasiones recibíamos mensajes de control en forma de cadenas que la plataforma interpretaba de manera errónea y, por tanto, no conseguía navegar correctamente.

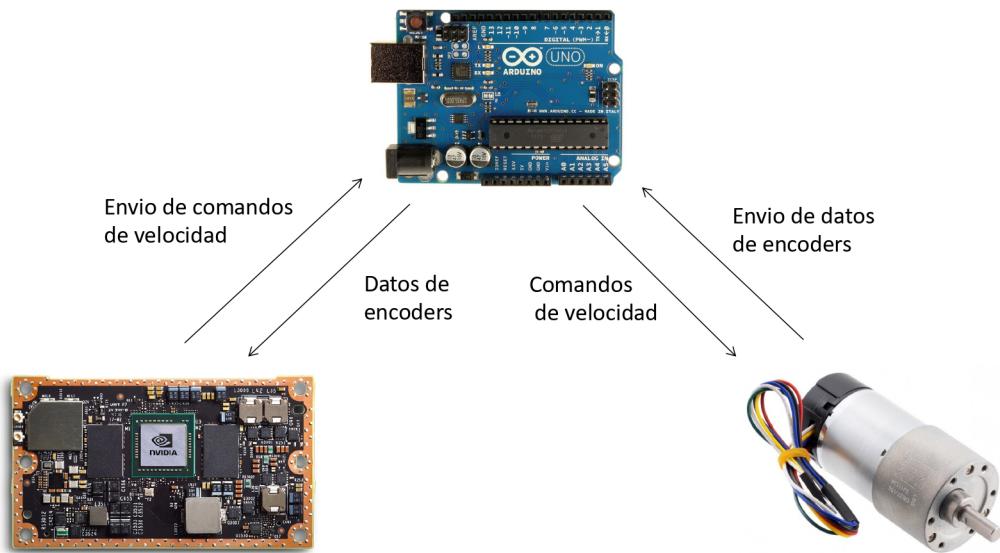


Figura 4.2: Comunicación entre Jetson y motor mediante Arduino.

#### 4.2.2. Explicación del funcionamiento

Para entender el funcionamiento de esta placa hay que pararse a explicar ciertos comandos que se han creado con el fin de mejorar la comunicación entre estos elementos. Todos los comandos que se van a comentar a continuación han sido creados según nuestras necesidades y, por lo tanto, son susceptibles de cambios a medida que el desarrollo de la plataforma avance.

Son muchos los comandos que se han implementado, pero en esta memoria solo vamos a citar los que hemos tenido que utilizar para el desarrollo de la navegación autónoma:

- **4XXXX**: mueve la plataforma hacia delante XXXX mm a velocidad máxima.
- **5XXXX**: mueve la plataforma hacia atrás XXXX mm a velocidad máxima.
- **?:** para los motores de la plataforma y recalcula la posición.

- **VXXXXYYY**: mueve cada rueda a una velocidad determinada, XXX es la velocidad para la rueda derecha y YYY para la izquierda. Las velocidades son interpretadas en pulsos por cada 100 ms y estos, a su vez, están relacionados con radianes. Las velocidades van de 000 a 256, siendo de 000 a 128 velocidad hacia delante y de 129 a 256 velocidad hacia atrás.
- **N**: devuelve el valor absoluto de los encoders y el time-stamp.

A parte de estos comandos hay muchos otros que no se han comentado ya que solo hemos necesitado de estos para la implementación de la navegación autónoma.

El funcionamiento es el siguiente:

1. Se inicia la estructura de nodos ROS y el script principal. En este script, el cual explicaremos en la siguiente sección, se establece una conexión mediante el puerto USB con la placa de Arduino. Para ello se ha utilizado el paquete de Python **serial**, con el que podemos abrir la conexión con el puerto USB de Arduino y enviar y recibir datos según nuestras necesidades.

```
1 self.arduino = serial.Serial('/dev/ttyUSB0', 115200)
```

Esta es la línea de código que establece la conexión con la placa de Arduino. El primer argumento de la función indica el dispositivo USB al que queremos conectarnos y el segundo es el baud rate, en nuestro caso es de 115200 pero esto puede variar según el tipo de arquitectura que tengamos.

2. Se envía el comando **N** a la placa de Arduino, la cual recogerá los datos de los encoders y los devolverá al script como una cadena de texto. Veremos más tarde cuando hablemos del script principal cómo se depura esta cadena de texto.

```
1 self.arduino.write(str('N').encode())
2 comienzo = self.arduino.readline()
```

La primera línea envía el comando **N** a la placa de Arduino, por lo que esta empezará a enviar los datos de los encoders hasta recibir otra orden. Con la siguiente línea de código guardamos esa cadena de texto correspondiente a los encoders en una variable llamada “comienzo”, la cual será procesada para sacar la información útil de los encoders.

3. Una vez recogidos los datos de los encoders estos se utilizarán para calcular las velocidades de cada rueda y se enviará el comando **VXXXXYYY** con las velocidades que se han calculado.

```
1 self.arduino.write('V'+format(int(datod), '03d') +
2 format(datoi, '03d'))
```

Mediante esta línea de Python enviamos a los motores la velocidad que cada rueda debe adquirir. Las variables “datod” y “datoi” contienen las velocidades en ese formato de 000 a 256 que hemos comentado anteriormente. Este dato se tiene que formatear si el valor de la velocidad es “1” la variable no puede contener un solo carácter, es por eso que se tiene que formatear el dato, mediante la función **format** y el parámetro 03d, para que siempre sea una variable de tres caracteres, obteniendo así “001”.

4. Cuando se requiera la parada de la plataforma, bien porque se encuentre con algún obstáculo o porque ya haya llegado a sus destino, se enviará el comando ? haciendo que los motores dejen de funcionar.

```
1     self.arduino.write(str('?'))
2     rospy.loginfo("Parada por no recibir datos")
```

A parte de enviar el comando de parada a los motores, se enviará un mensaje de control ROS para indicar que el motivo de la parada de la plataforma no ha sido accidental, si no por falta de datos.

#### 4.2.3. Instalación y pruebas

Lo primero que necesitaremos para poder poner en funcionamiento esta placa será un ordenador con el IDE de Arduino, este va a ser necesario para cargar en la placa todo el software desarrollado por el grupo de investigación GRAM. También utilizaremos el IDE para probar el correcto funcionamiento de la placa y de la conexión de esta con los motores de la plataforma robótica. Da igual el sistema operativo que tengamos, ya que este se encuentra disponible para Linux, Windows y Mac OS. Este es el link para descargar el ejecutable de instalación: <https://www.arduino.cc/en/software> 4.3.

Dentro de la web simplemente tendremos que elegir nuestro sistema operativo y se nos descargará automáticamente. Como ya he dicho el IDE se encuentra disponible para cualquier sistema operativo pero, para que este trabajo de fin de grado funcione, es necesario que se monte sobre un computador con Ubuntu 18.04.

Una vez descargado el ejecutable de instalación simplemente hay que lanzarlo y pasados unos instantes se habrá completado la instalación y nos aparecerá el IDE en pantalla 4.4.

Ahora pasemos a probar que existe una conexión real entre la placa de Arduino y los motores de la plataforma. Para ello habrá que conectar la placa al computador mediante USB A - B. Una vez conectado el cable nos iremos a la sección de “herramientas o tools” y nos detendremos en la opción de “Serial Monitor” 4.5. Esta opción nos habilitará una pequeña ventana gráfica en la que podremos mandar mensajes de manera manual a la placa de Arduino y, si esta nos devuelve algún tipo de mensaje, se mostrará en esta ventana. Si

## Downloads

The open-source Arduino Software (IDE) makes it easy to write code and upload it to the board. This software can be used with any Arduino board.

Refer to the [Getting Started](#) page for Installation instructions.

**SOURCE CODE**

Active development of the Arduino software is [hosted by GitHub](#). See the instructions for [building the code](#). Latest release source code archives are available [here](#). The archives are PGP-signed so they can be verified using [this gpg key](#).

**DOWNLOAD OPTIONS**

- Windows** Win 7 and newer
- Windows** ZIP file
- Windows app** Win 8.1 or 10 [Get](#)
- Linux** 32 bits
- Linux** 64 bits
- Linux** ARM 32 bits
- Linux** ARM 64 bits
- Mac OS X** 10.10 or newer

[Release Notes](#) [Checksums \(sha512\)](#)

Figura 4.3: Página de descarga del IDE de Arduino.

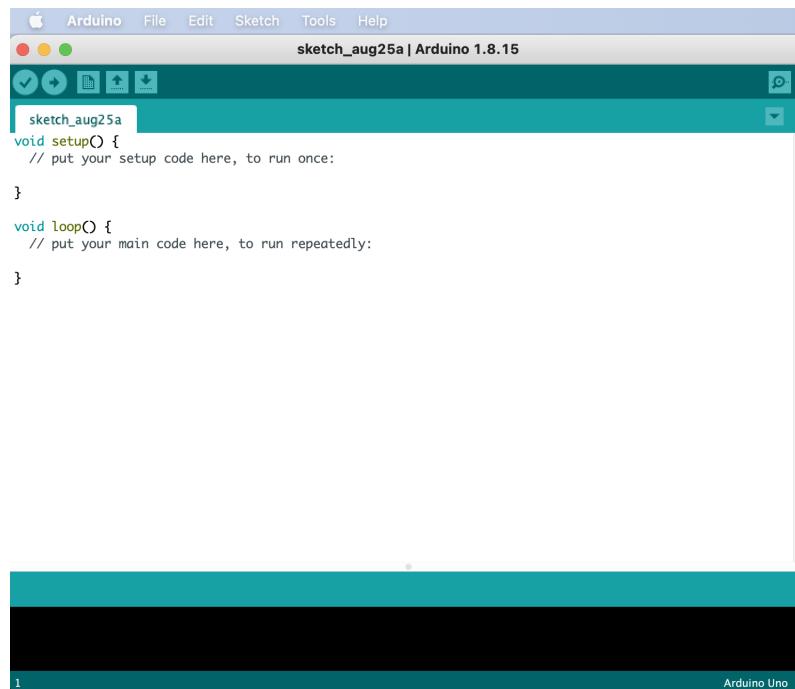


Figura 4.4: IDE de Arduino.

por el contrario no aparece ningún tipo de ventana gráfica significa que ha habido un problema con la conexión USB, ya sea porque la placa de Arduino no se encuentra operativa, porque el puerto se encuentra bloqueado o porque el cable USB no funciona correctamente.

Una vez comprobado que la conexión se ha establecido correctamente vamos a pasar a ver cómo instalar el software de la placa desarrollado por el grupo de investigación GRAM.

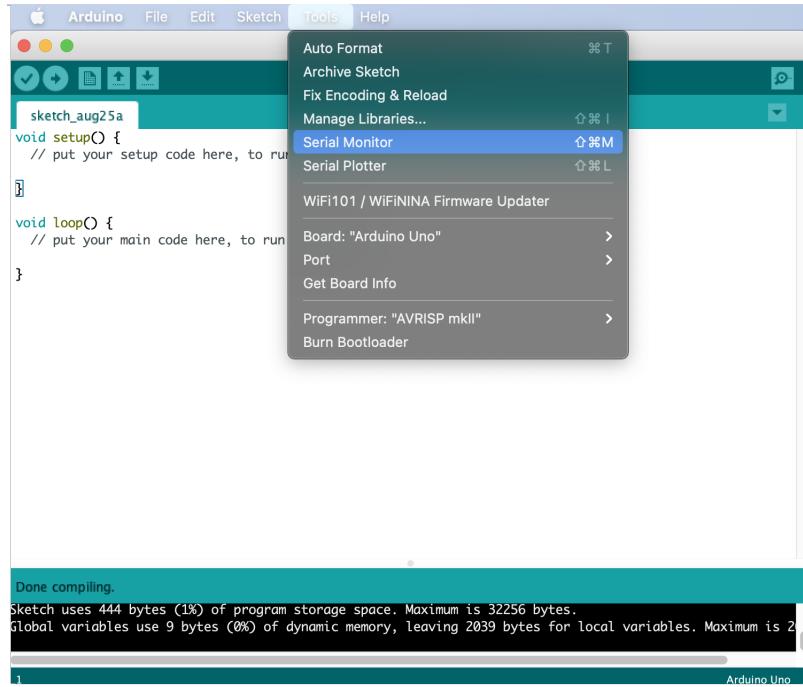


Figura 4.5: Abrir un Serial Monitor en el IDE de Arduino.

Lo primero que tendremos que hacer será descargar el paquete con todo el código de Arduino, para ello hay que acceder al siguiente repositorio: <https://github.com/TheArmega/LOLA2-Autonomous-Navigation.git>. Este es el repositorio donde se encuentra todo el software necesario para el correcto funcionamiento de la plataforma por tanto, debemos descargarlo. Una vez que descomprimamos el zip del repositorio nos encontraremos con otro archivo zip llamado “SRE.zip”, este es el archivo que contiene el software que hará funcionar a la placa de Arduino.

Una vez descomprimido este archivo veremos que dentro nos encontramos con un directorio en el que a su vez hay múltiples archivos. Cada uno de estos archivos es necesario, ya que controlan diferentes partes de la plataforma. No vamos a entrar en que hay dentro de cada archivo y como están programados, ya que no entra dentro de los objetivos de este trabajo de fin de grado.

El siguiente paso es abrir todo el proyecto en el IDE, para ello simplemente tendremos que darle a la sección “File” y a la opción “Open”. Inmediatamente después se nos abrirá una ventana en la que podremos navegar por los diferentes directorios y archivos de nuestro computador. Tendremos que dirigirnos a la carpeta descargada del repositorio, en concreto a la carpeta descomprimida anteriormente que contiene el proyecto de Arduino, y abrir el archivo “**SRE.ino**”.

Solo es necesario abrir este archivo, ya que el IDE de Arduino interpretará las depen-

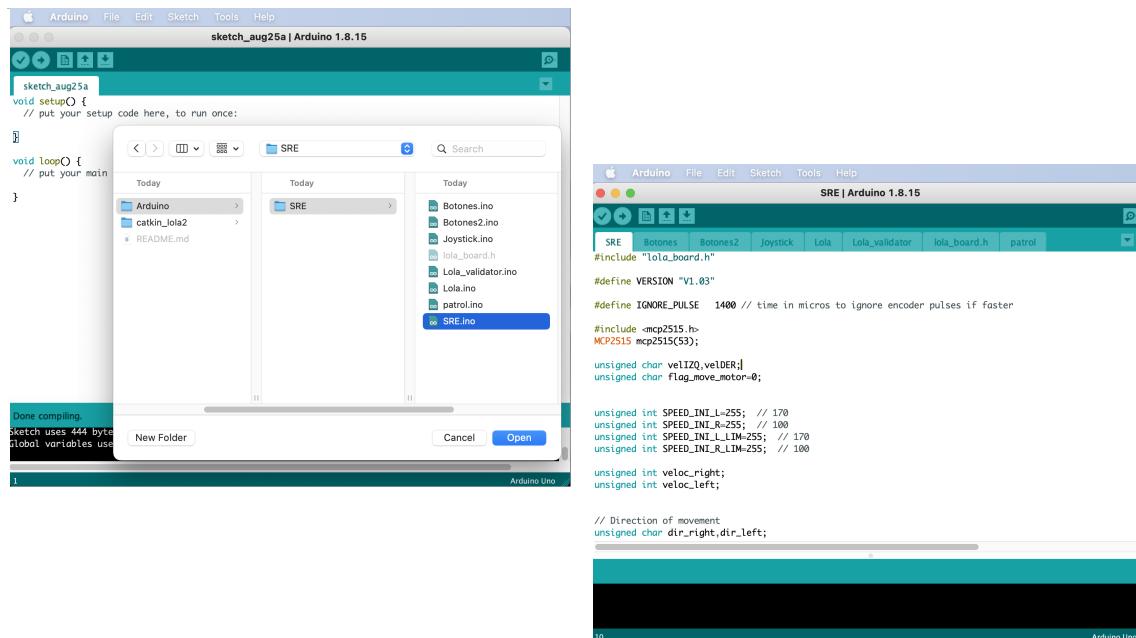


Figura 4.6: Abrir un proyecto de Arduino en el IDE de Arduino.

dencias de este y abrirá el resto del proyecto. Una vez abierto todo el proyecto el siguiente paso es compilarlo y ver si hay algo que está mal programado o si falta algún paquete.

Para ello hay que darle a la opción de “Verify” (símbolo de check). Si todo esta bien nos aparecerá un mensaje en la consola del IDE. Si por el contrario nos aparece un mensaje de error es que hay algo que no se ha realizado de la forma que se ha explicado.

Puede que nos de un error por la falta de un paquete llamado “mcp2515.h”, si este es el caso deberemos instalarlo manualmente. El enlace de descarga de la librería es el siguiente: <https://www.arduino.cc/reference/en/libraries/107-arduino-mcp2515/>. Una vez descargado el archivo zip (da igual la versión que descarguemos) pasaremos a instalarlo en nuestro IDE de Arduino.

Para instalar el paquete debemos irnos a la sección de “Sketch” y seleccionar la opción “Include Library/Add .ZIP Library...”. Cuando seleccionemos esta opción se nos abrirá una ventana de navegación en la que tendremos que seleccionar el archivo .zip descargado anteriormente. Una vez seleccionado este archivo simplemente le daremos a “Open” y, si todo ha salido según lo planeado, el paquete estará instalado en nuestro IDE **4.7**.

Una vez que hayamos verificado que el proyecto está compilado correctamente y que la conexión entre la computadora y la placa de Arduino está establecida, pasaremos a cargar todo el software en la placa.

Para ello seleccionaremos la opción “Upload” (símbolo de la flecha). Tal como en la opción de verificación, si se ha cargado correctamente a la placa nos aparecerá un mensaje

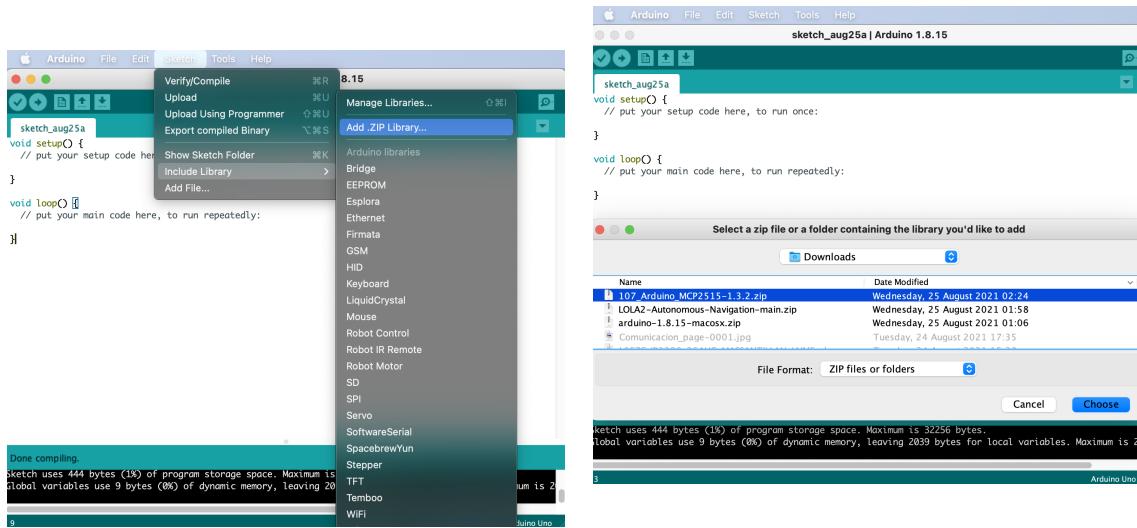


Figura 4.7: Instalación del paquete mcp2515 de Arduino en el IDE.

de okey en la consola del IDE. Si por el contrario no nos aparece este mensaje significa que hay algo que no se ha llevado a cabo correctamente.

El último paso para verificar que todo lo relativo a la placa de Arduino se encuentra funcionando es abrir un “Serial Monitor” que establezca una conexión con la placa y mandarle un comando de los que hemos visto con anterioridad cuando hemos explicado los que hemos usado para la integración de la navegación autónoma de la plataforma.

Normalmente, las pruebas que hemos realizado nosotros han sido con el comando **“40500”** con el que conseguimos que la plataforma avance medio metro de frente a velocidad máxima. Si al introducir este comando vemos la plataforma moverse significará que hemos llevado a cabo la instalación de manera correcta.

## 4.3. ROS

### 4.3.1. Objetivos

El uso del sistema operativo ROS es el eje principal de este trabajo de fin de grado. A la hora de plantear los objetivos de este trabajo se decidió que la mejor manera de poder implementar un sistema de navegación autónoma en una plataforma de bajo coste era mediante el uso de ROS como elemento conductor.

Este sistema operativo, que ya hemos descrito en el capítulo dedicado a él, nos ha permitido crear una estructura de nodos y directorios fácilmente escalable a otras plataformas que tengan unos elementos similares a los nuestros.

El objetivo principal de ROS es calcular las velocidades que tienen que tener cada

rueda para llevar a cabo la navegación autónoma. Dicho así suena sencillo, pero ROS se encarga de otros muchos elementos del sistema, ya que para conseguir calcular las velocidades es necesario desarrollar un sistema de control que tenga en cuenta los datos de todos los sensores que hemos descrito con anterioridad.

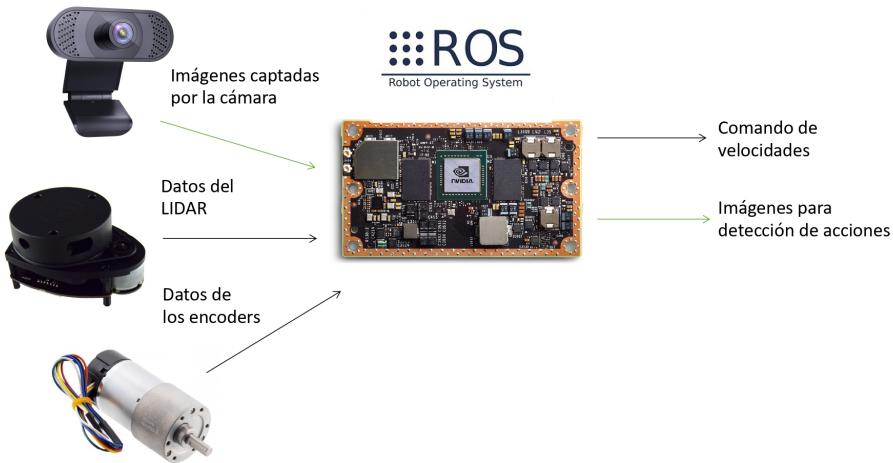


Figura 4.8: Conexión de ROS con el resto de sensores.

ROS va a estar instalado en la placa NVIDIA Jetson TX2 pero, como ya hemos comentado, también tendremos un ordenador portátil con la misma versión de ROS, por lo que el funcionamiento no se tiene que ver alterado dependiendo de la computadora que utilicemos.

#### 4.3.2. Instalar ROS en la Jetson TX2

La instalación del sistema operativo ROS en esta placa NVIDIA Jetson TX2 es un poco diferente a como se haría con un computador tradicional que corra la versión 18.04 de Ubuntu. El motivo por el que la instalación de ROS en esta placa es diferente se debe a que la GPU tiene una microarquitectura PASCAL, siendo diferente a la arquitectura ARM convencional. Es por ello que el proceso de instalación tiene que ser distinto al dado por el soporte oficial de ROS.

Gracias a la comunidad **Jetson Hacks** [14] existe un método sencillo de llevar a cabo la instalación sin quebraderos de cabeza. Para proceder a la instalación lo primero que tenemos que hacer es acceder al repositorio público de Jetson Hacks <https://github.com/jetsonhacks/installROS>. Este repositorio contiene dos scripts que llevarán a cabo la instalación de ROS Melodic en nuestra NVIDIA Jetson TX2.

El script principal es el “installROS.sh”, este archivo contiene una serie de líneas de

código interpretables por la shell de Linux, evitándonos a nosotros como usuarios tener que ir comando por comando arreglando todas las dependencias que la instalación generaría debido al cambio de arquitectura.

1. Abrimos una terminal y accedemos al directorio home:

```
1 $ cd ~
```

2. Clonamos el repositorio en el directorio home:

```
1 $ git clone https://github.com/jetsonhacks/installROSTX2.git
```

3. Accedemos al directorio clonado:

```
1 $ cd installROSTX2
```

4. Ejecutamos el script de instalación. En este caso tenemos tres opciones distintas:

- ros-melodic-ros-base
- ros-melodic-desktop
- ros-melodic-desktop-full

Nosotros recomendamos instalar la última opción, ya que esta instalará la versión completa de ROS Melodic con todos los paquetes. Es preferible instalar la versión completa aunque no vayamos a utilizar todo lo que traiga, ya que así nos quitamos problemas de dependencias de paquetes que a la larga serán más difíciles de resolver. Por tanto el comando de instalación es:

```
1 $ ./installROS.sh -p ros-melodic-desktop-full
```

Para comprobar que ROS se ha instalado correctamente simplemente tenemos que abrir una nueva terminal e introducir el comando:

```
1 $ roscore
```

Si bash es capaz de interpretarlo y darnos una salida significará que la instalación se ha llevado a cabo correctamente. Más adelante veremos que es este comando y como se interpreta la salida por pantalla del mismo.

Existe otro script dentro del repositorio que hemos descargado, **setupCatkinWorkspace.sh**. Este script no es necesario para completar la instalación de ROS, si no que su objetivo es la creación de un directorio Catkin y su correspondiente configuración.

Antes de ver como crear nuestro directorio Catkin vamos a comentar brevemente qué es esto. Catkin combina macros de CMake y scripts de Python con el fin de proporcionar funcionalidades adicionales al flujo de trabajo de CMake. Catkin fue diseñado para ser más convencional que rosbuild (predecesor de Catkin), lo que permite una mejor distribución de paquetes, un mejor soporte de compilación cruzada y una mejor portabilidad.

En resumen, Catkin va a ser la herramienta principal que utilicemos a la hora de controlar la compilación de nuestro proyecto ROS. También gestionará la disposición y creación de los directorios.

Para la creación de nuestro directorio Catkin simplemente tenemos que echar mano de este segundo script que acabamos de comentar:

```
1 $ ./setupCatkinWorkspace.sh [nombre del directorio]
```

#### 4.3.3. Instalar ROS en cualquier computador

Entendiendo que la arquitectura de nuestro computador es ARM y no PASCAL, la instalación de ROS que hemos desarrollado anteriormente no nos vale. En este caso hay que seguir la fuente oficial de ROS:

1. Configurar nuestro computador para que acepte paquetes de la dirección packages.ros.org:

```
1 $ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

2. Instalar **curl** si no está ya instalado:

```
1 $ sudo apt install curl # if you haven't already installed curl
```

3. Configurar las keys para acceder al dominio de ROS:

```
1 $ curl -s https://raw.githubusercontent.com/ros/rosdistro/master/ros.asc | sudo apt-key add -
```

4. Actualizar la lista de paquetes de los repositorios:

```
1 $ sudo apt update
```

5. Instalar ROS Melodic:

```
1 $ sudo apt install ros-melodic-desktop-full
```

Al igual que con la NVIDIA Jetson TX2, aquí disponemos de tres versiones de ROS Melodic, pero seguimos puntuizando la importancia de instalar la versión completa para ahorrarnos problemas en el futuro.

6. Una vez instalado ROS Melodic toca configurar el entorno, para ello necesitaremos utilizar los siguientes comandos:

```
1 $ echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
2 $ source ~/.bashrc
```

7. Al igual que en el apartado anterior debemos comprobar que la instalación se ha llevado a cabo correctamente:

```
1 $ roscore
```

8. dependencias

```
1 $ sudo apt install python-rosdep python-rosinstall python-
    rosinstall-generator python-wstool build-essential
```

9. rosdep

```
1 $ sudo apt install python-rosdep
```

#### 4.3.4. Creación espacio de trabajo Catkin

A diferencia de en la Jetson, existe otra forma de crear nuestro espacio de trabajo Catkin sin usar el script anterior:

1. Creamos el directorio Catkin con el nombre que queramos (por convenio es catkin ws) y el subdirectorio src:

```
1 $ mkdir -p ~/catkin_ws/src
```

La opción -p del comando mkdir nos permite crear subdirectorios de un directorio todavía no creado.

2. Accedemos al espacio de trabajo Catkin:

```
1 $ cd ~/catkin_ws/
```

3. Ejecutamos el comando catkin make por primera vez que generará el archivo CMakeLists.txt dentro de la carpeta src:

```
1 $ catkin_make
```

4. Si echamos un vistazo al espacio de trabajo Catkin, veremos que se han creado dos nuevos directorio: **build** y **devel**. Dentro del directorio “devel” podemos ver una serie de archivos setup.\*sh. Es necesario hacer un source de un archivo .sh concreto:

```
1 $ source devel/setup.bash
```

5. Para asegurarse de que el espacio de trabajo está correctamente superpuesto por el script de configuración, hay que asegurarse de que la variable de entorno **ROS PACKAGE PATH** incluya el directorio en el que se encuentra, para ello utilizaremos el siguiente comando:

```
1 $ echo $ROS_PACKAGE_PATH
```

El cual debe devolvernos algo del tipo:

```
1 /home/youruser/catkin_ws/src:/opt/ros/melodic/share
```

## 4.4. Primeros pasos con ROS y Python

Una vez que la instalación de ROS se haya llevado a cabo correctamente y se haya creado un espacio de trabajo Catkin es hora de pasar a ver como podemos realizar nuestro primer proyecto con ROS y Python. Todo lo que se va a describir en esta sección fue realizado como parte del aprendizaje antes de realizar la integración de los modelos de inteligencia artificial en el sistema.

Antes de ponernos a picar código es necesario instalar una librería que nos permita escribir sentencias de ROS en el lenguaje de programación Python, este paquete es **Rospy**. Para instalarlo tenemos que introducir el siguiente comando:

```
1 $ sudo apt install python-rospy
```

Una vez instalado este paquete ya tendremos todas las herramientas necesarias para ponernos a desarrollar aplicaciones usando ROS y Python. Es posible que se necesiten algunos paquetes más de Python, a medida que aparezcan durante la explicación de esta sección comentaremos como instalarlos mediante el uso de **Pip**, el cual es un instalador de paquetes de Python que nos ayudará a instalar y manejar paquetes adicionales que no son parte de las librerías estandar de Python.

El primer desarrollo que comenzamos fue una pequeña estructura de nodos en la que cuando se pulsase una tecla del teclado la plataforma se movie. Las teclas, y su función son las siguientes:

- Tecla **w**: hace avanzar a la plataforma 10 metros hacia delante.

- Tecla **s**: hace avanzar a la plataforma 10 metros hacia atrás.
- Tecla **a**: hace girar a la plataforma 90 grados hacia la derecha.
- Tecla **d**: hace girar a la plataforma 90 grados hacia la izquierda.
- Tecla **x**: hace pararse a la plataforma inmediatamente, da igual que este ejecutando algún comando de los anteriores.

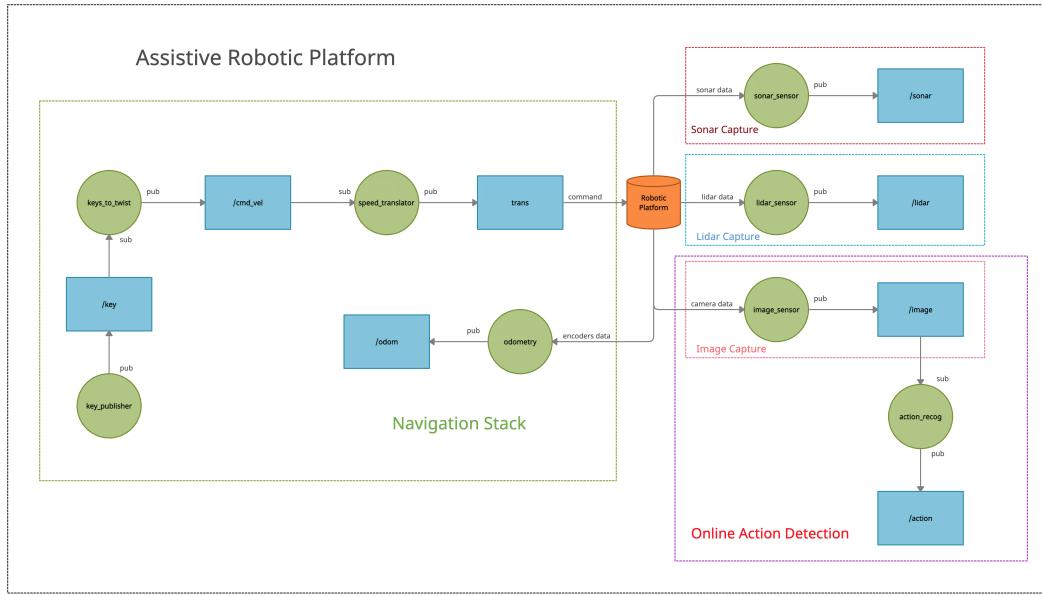


Figura 4.9: Arquitectura del primer programa desarrollado con ROS y Python.

Esta estructura forma parte de una arquitectura temprana de la plataforma LOLA que incluía tres módulos más. Estos tres módulos adicionales se corresponden con el control de los sonar, del LIDAR y del reconocimiento de acciones.

El sonar fue eliminado de la plataforma, por lo que no va a ser comentado en esta memoria. El manejo del LIDAR se explicará más adelante, ya que es usado en la arquitectura que controla la navegación autónoma de la plataforma. Y el reconocimiento de acciones online tiene su propia sección en la que veremos detalladamente como se ha conseguido y los motivos de hacer que sea online. Se puede apreciar con más detalle la arquitectura completa en la figura 4.9.

#### 4.4.1. Módulo Navigation Stack

Este módulo, delimitado por un rectángulo verde en la figura 4.9, tiene como objetivo recoger la pulsación de las teclas del teclado conectado a la plataforma, comprobar si es una de las teclas que indican un comando de movimiento y, si es el caso, enviar un comando de velocidad a esta mediante la placa de Arduino.

El módulo se compone de cuatro nodos y, por tanto, de cuatro archivos Python en los que se ha programado la funcionalidad de cada nodo, desde la función que tiene que desarrollar hasta en que topic tiene que publicar la información computada para que el siguiente nodo de la arquitectura la pueda utilizar.

#### 4.4.1.1. key\_publisher

Este primer nodo es el encargado de detectar las pulsaciones de las teclas. Cuando detecta que se ha pulsado una nueva tecla, automáticamente la publica en el topic **keys**.

El funcionamiento es muy sencillo, ya que el objetivo principal durante el desarrollo de esta pequeña arquitectura ha sido separar cada función para que un nodo se encargue de ella.

```

1 import sys, select, tty, termios
2 import rospy
3 from std_msgs.msg import String
4
5
6 def key_publisher():
7     key_pub = rospy.Publisher('keys', String, queue_size=1)
8     rospy.init_node("key_publisher")
9     rate = rospy.Rate(100)
10    old_attr = termios.tcgetattr(sys.stdin)
11    tty.setSIGPOLL(sys.stdin.fileno())
12    print("Publishing keystrokes. Press Ctrl-C to exit...")
13    while(not rospy.is_shutdown()):
14        if(select.select([sys.stdin], [], [], 0)[0] == [sys.stdin]):
15            :
16            key_pub.publish(sys.stdin.read(1))
17            rate.sleep()
18    termios.tcsetattr(sys.stdin, termios.TCSADRAIN, old_attr)
```

Todo el código que aparece en este nodo es código Python, el hecho de que se utilice ROS no afecta a la sintaxis de ninguna manera, simplemente mediante la librería **Rospy** podemos interactuar con ROS de una manera más sencilla.

Por ejemplo en la **línea 8** se llama a la librería Rospy y se utiliza la función “`init_node`”, la cual va a crear un nodo ROS sobre el archivo Python en el que estamos. Hay que destacar que ROS no es capaz de mantener dos nodos en un mismo archivo de código, por lo que siempre tendrá que haber, al menos, tantos archivos Python como nodos.

Al mismo tiempo, en la **línea 7** se crea un topic nuevo llamado **keys**, este va a publicar mensajes de tipo String y solo va a poder tener un mensaje simultáneamente, es decir,

cada vez que llegue un mensaje nuevo al topic este sobre escribirá el anterior. Es en este topic donde se va a publicar que tecla ha sido pulsada.

La parte principal del programa se encuentra en el bucle while, **línea 14 - 16**, en donde se evalúa si ROS sigue funcionando y, si es el caso, se comprueba si hay alguna tecla que se este pulsando. Si este es el caso publicará en el topic **keys** cual es.

El resto de líneas que hay en el archivo son sentencias de Python comunes que utilizan las librerías **sys**, **select**, **tty** y **termios** para poder detectar las pulsaciones de las teclas del teclado.

Mediante este archivo conseguimos crear un nodo de nombre **key\_publisher** capaz de detectar cuando pulsamos una tecla del teclado bluetooth conectado a la plataforma y que publica la información de esta a un topic llamado **keys**.

#### 4.4.1.2. keys\_to\_twist

El segundo nodo que nos encontramos es el encargado de transforma la tecla pulsada a un valor que el sistema pueda entender. También va a comprobar si la tecla que se ha pulsado es la w, s, a, d o x, y si es el caso la publicará en un nuevo topic creado escpecificamente para publicar estos nuevos valores.

```

1
2 import rospy
3 from std_msgs.msg import String
4 from geometry_msgs.msg import Twist
5
6 key_mapping = { 'w': [0, 1], 'x': [0, -1],
7                 'a': [-1, 0], 'd': [1, 0],
8                 's': [0, 0] }
9
10 def keys_cb(msg, twist_pub):
11     if(len(msg.data) == 0 or not key_mapping.has_key(msg.data[0])):
12         return # unknown key
13     vels = key_mapping[msg.data[0]]
14     t = Twist()
15     t.angular.z = vels[0]
16     t.linear.x = vels[1]
17     twist_pub.publish(t)
18
19 def keys_to_twist():
20     rospy.init_node('keys_to_twist')
21     twist_pub = rospy.Publisher('cmd_vel', Twist, queue_size=1)
```

```

22     print("Publishing speeds. Press Ctrl-C to exit...")
23     rospy.Subscriber('keys', String, keys_cb, twist_pub)
24     rospy.spin()

```

Este archivo esta formado por dos funciones distintas, **keys\_to\_twist** es la encargada de crear el nodo y el topic, **cmd\_vel**. Como se puede ver en la **línea 23** este nodo está suscrito al topic **keys** visto en el archivo anterior. En este topic se encuentran los mensajes de tipo String que indican la tecla que se ha pulsado. A parte, cuando nos suscribimos al topic se llama a la otra función del archivo **keys\_cb**. En la **línea 11** se valida si realmente se ha pulsado alguna tecla o, si por el contrario, la tecla que se ha pulsado pertenece a alguna de las que hemos habilitado para el movimiento. Para ello utilizamos el diccionario **key\_mapping**, (**línea 6**).

Si no se ha pulsado una tecla o ésta no es correcta no se publicará nada en el topic **cmd\_vel**, pero si ésta es correcta se publicará un mensaje de tipo Twist (mensajes ROS utilizados para la publicación de velocidades lineales y angulares) con los valores indicados en el diccionario según el tipo de tecla. La codificación que hemos seguido es el primer valor para la velocidad lineal en x, y el segundo para la velocidad angular en z.

#### 4.4.1.3. speed\_translator

El siguiente nodo que veremos es el encargado de recoger esos mensajes de tipo Twist de los que acabamos de hablar, analizarlos y transformarlos en comandos de Arduino para así conseguir que nuestra plataforma se mueva.

```

1
2 import serial, time
3 import rospy
4 from std_msgs.msg import String
5 from geometry_msgs.msg import Twist
6
7 print("Publishing connection to Arduino. Press Ctrl-C to exit...")
8 arduino = serial.Serial('/dev/ttyUSB0', 115200)
9
10 print(arduino)
11
12 def send_to_arduino(msg, trans_pub):
13
14     vel_x = msg.linear.x
15     vel_z = msg.angular.z
16
17     if(int(vel_x) == 1 and int(vel_z) == 0):
18         trans_pub.publish('Forward')

```

```

19         time.sleep(1)
20         arduino.write(str(49999).encode())
21     elif(int(vel_x) == 0 and int(vel_z) == 0):
22         trans_pub.publish('Backward')
23         time.sleep(1)
24         arduino.write(str(59999).encode())
25     elif(int(vel_x) == 0 and int(vel_z) == -1):
26         trans_pub.publish('Left')
27         time.sleep(1)
28         arduino.write(str(2090).encode())
29     elif(int(vel_x) == 0 and int(vel_z) == 1):
30         trans_pub.publish('Right')
31         time.sleep(1)
32         arduino.write(str(3090).encode())
33     elif(int(vel_x) == -1 and int(vel_z) == 0):
34         trans_pub.publish('Stop')
35         time.sleep(1)
36         arduino.write(str('?').encode())
37
38 def speed_translator():
39     rospy.init_node('speed_translator')
40     trans_pub = rospy.Publisher('trans', String, queue_size=10)
41     print('Established connection with Arduino...')
42     rospy.Subscriber('cmd_vel', Twist, send_to_arduino, trans_pub)
43     rospy.spin()
44     arduino.close()
45     print('Finishing connection with Arduino...')
```

Antes de hablar del código es necesario parar en la **línea 1**. Este paquete que importamos llamado **serial** nos va a permitir abrir una conexión mediante el puerto USB con la placa de Arduino y, por lo tanto, enviar los comandos que sean necesarios para así conseguir el correcto desplazamiento de la plataforma.

Para instalar este paquete lo primero que tenemos que hacer es instalar pip en nuestro computador:

1. Actualizamos los paquetes del sistema:

```
1 $ sudo apt update
```

2. Instalamos pip:

```
1 $ sudo apt install python-pip
```

3. Comprobamos que se ha instalado correctamente viendo la versión que hemos instalado:

```
1 $ pip --version
```

Una vez instalado pip podemos instalar la librería **serial** utilizando este instalador:

```
1 $ sudo pip install pyserial
```

Una vez tengamos esta librería instalada pasaremos a comentar el código. Al igual que el anterior archivo, aquí nos volvemos a encontrar con dos funciones. La función **speed\_translator** es la encargada de crear el nodo ROS y el topic en el que se van a publicar las direcciones que tiene que tomar la plataforma. Este topic, **trans**, contendrá mensajes de tipo String y podrá almacenar como máximo diez, para así nosotros poder ver que ruta ha estado tomando.

Este nodo está suscrito al topic **cmd\_vel** en el cual tenemos los valores Twist de las teclas pulsadas. Cuando se suscriba a este topic se llamará a la función **send\_to\_arduino** la cual, dependiendo de las velocidades lineales, mandará un comando u otro. Por ejemplo, en la **línea 17** podemos ver como comprueba si se tiene que la velocidad lineal en x es 1 y si la angular en z es 0. Según la codificación que hemos hecho en el archivo anterior esto significa que el robot de avanzar hacia delante diez metros por lo que, en la **línea 20** se envía el comando **49999** por el USB a la placa de Arduino. A parte, se publicará un mensajes en el topic **trans**, **línea 18**, con la dirección que va a tomar el robot, en este caso “Forward”.

#### 4.4.1.4. odometry

Este último nodo es un estándar en cualquier proyecto ROS que nos encontremos. El nodo **odometry** es el encargado de publicar en el topic **odom** todos los datos relativos a la odometría del robot. Para ello utilizará mensajes ROS de tipo Odometry. Este tipo de mensajes están formados por cuatro mensajes a su vez:

1. Header header.
2. string child\_frame\_id.
3. geometry\_msgs/PoseWithCovariance Pose.
4. geometry\_msgs/TwistWithCovariance Twist.

Cada uno de estos mensajes forman en conjunto un mensaje de tipo Odometry, formado por una cabecera, un id, unos valores de posición y otros valores de velocidades lineales y angulares.

```
1 import serial, time
2 import rospy
3 import tf
4 from nav_msgs.msg import Odometry
5 from geometry_msgs.msg import Point, Pose, Quaternion, Twist, Vector3
6
7 arduino = serial.Serial ('/dev/ttyUSB0', 115200)
8
9
10 def odometry():
11
12     rospy.init_node('odometry')
13
14     odom_pub = rospy.Publisher("odom", Odometry, queue_size=50)
15     odom_broadcaster = tf.TransformBroadcaster()
16
17     x = 0
18     y = 0
19     th = 0
20
21     vx = 0.1
22     vy = -0.1
23     vth = 0.1
24
25     current_time = rospy.Time.now()
26     last_time = rospy.Time.now()
27
28     r = rospy.Rate(1.0)
29
30     while(not rospy.is_shutdown()):
31
32         current_time = rospy.Time.now()
33
34         odom_quat = tf.transformations.quaternion_from_euler(0, 0,
35                                               th)
36
37         arduino.write(str('A').encode())
38         time.sleep(1)
39         data = arduino.readline()
```

```

40         data = data.split()
41         print(data)
42
43         x = int(data[1])
44         y = int(data[3])
45         th = int(data[5])
46
47         odom_broadcaster.sendTransform(
48             (x, y, th),
49             odom_quat,
50             current_time,
51             "base_link",
52             "odom"
53         )
54
55         odom = Odometry()
56         odom.header.stamp = current_time
57         odom.header.frame_id = "odom"
58
59         odom.pose.pose = Pose(Point(x, y, th), Quaternion(*
60                               odom_quat))
61
62         odom.child_frame_id = "base_link"
63         odom.twist.twist = Twist(Vector3(vx, vy, 0), Vector3(0, 0,
64                               vth))
65
66         odom_pub.publish(odom)
67
68         last_time = current_time
69         r.sleep()
70
71     arduino.close()

```

Todo el código de este archivo se encuentra concentrado en una única función llamada **odometry**. Lo primero que hacemos en esta función es crear el nodo **odometry** y el topic **odom**, **líneas 14 y 15**. El trabajo principal de este nodo se desarrolla en el bucle while que se encuentra en la **línea 30**. Este bucle se mantendrá activo hasta que ROS siga en funcionamiento, por lo que hasta que apaguemos el nodo maestro de ROS se seguirán publicando mensajes de odometría.

En la **línea 36** mandamos el comando **A** a la placa de Arduino. Este comando nos va a devolver las coordenadas de la plataforma robótica, las cuales necesitamos para crear

nuestro mensaje de odometría. Una vez que enviamos el mensaje esperamos un pequeño tiempo y leemos lo que la placa de Arduino nos está enviando por el puerto USB. Esto es importante, ya que en la parte de la navegación autónoma haremos algo muy parecido esto a la hora de leer los datos de los encoders.

Una vez recibido y depurados los datos de las coordenadas de la plataforma se procede a crear el mensaje de odometría. La cabecera , **líneas 56 y 57**, se compone de la marca de tiempo y del nombre **odom**. El id, **línea 61**, siempre contiene la cadena **base\_link**. La posición, **línea 59**, se calcula mediante las coordenadas recibidas por la placa de Arduino y el uso de quaterniones. Y por último, las velocidades, **línea 62**, en este caso son simbólicas ya que los comandos que utilizamos para que el robot se desplace son de velocidad fija y, por tanto, la velocidad de la plataforma siempre será la misma.

#### 4.4.1.5. Directorios y Archivos

Ahora que sabemos qué es lo que hay dentro de estos cuatro archivos de Python, vamos a pasar a ver como está estructurado esta arquitectura en distintos directorios y archivos. Existen una gran cantidad de archivos y directorios que no vamos a comentar debido a que estos no han de ser modificados por nosotros. La mayoría son archivos propios de la configuración de Catkin o ROS, por lo que tendremos que ignorarlos. La estructura que nos queda si ignoramos todos estos archivos y directorios que no afectan al diseño del programa es la correspondiente a la figura 4.10.

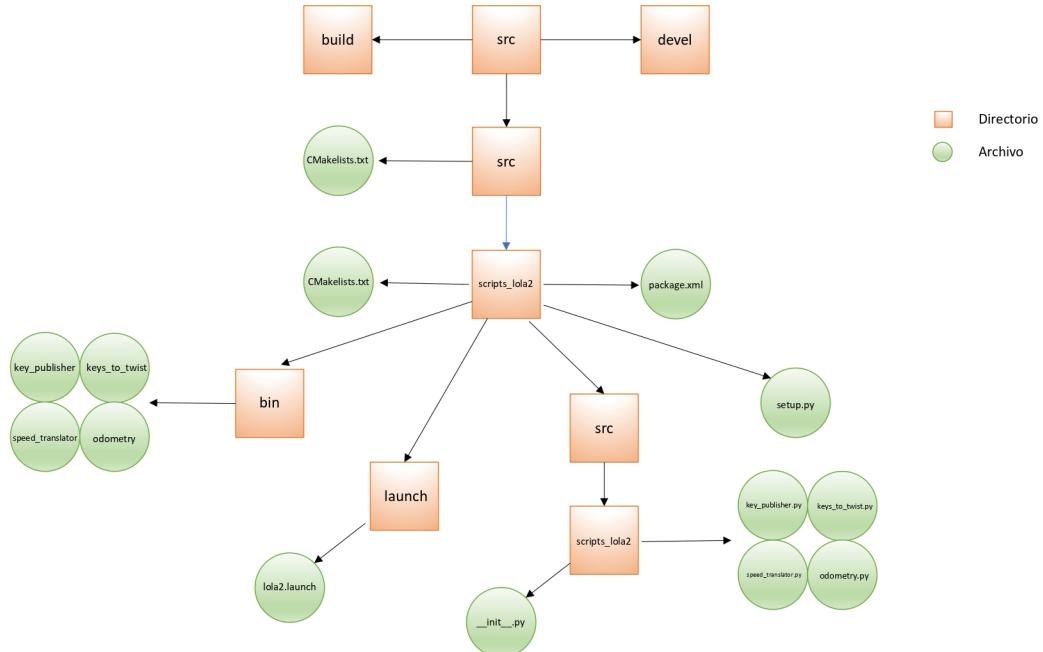


Figura 4.10: Estructura de directorios y archivos de la teleoperación mediante teclado.

Se sobre entiende que el primer directorio **src** que muestra la arquitectura se encuentra dentro de nuestro espacio de trabajo Catkin. Los tres directorios principales a los que más vamos a recurrir y sobre los que tendremos que ahcer modificaciones son:

- **src/src/scripts\_lola2/src/scripts\_lola2:** dentro de esta carpeta se encuentran todos los archivos de Python correspondientes a cada uno de los nodos de la arquitectura. Si se quisiera ampliar las funcionalidades de este proyecto añadiendo más nodos, estos tendrían que ir en esta carpeta.
- **src/src/scripts\_lola2/bin:** aquí se encuentran los ejecutables que llamará el launch. Por cada nodo que queramos ejecutar tiene que haber su ejecutable, o main, en esta carpeta.
- **src/src/scripts\_lola2/launch:** en esta carpeta tendremos que tener los archivos .launch que queramos.

El archivo **CMakelists.txt** con ruta `src/src/scripts_lola2` hay que tenerlo en cuenta a la hora de ampliar con más nodos la arquitectura. En la **línea 162** del archivo hay que indicar en este CMakelists todos los archivos ejecutables, de lo contrario Catkin no será capaz de encontrarlo y no se podrá lanzar los nuevos nodos que sean incluidos.

A parte de estos directorios y archivos comentados, no hay que tener en cuenta nada más a la hora de ponerse a desarrollar cualquier proyecto con ROS y Python. Que no se comenten en esta memoria no significa que no sirven para nada, si no que todos ellos son archivos de configuración y necesarios para que Catkin sea capaz de compilar el proyecto y entender como está estructurado nuestro proyecto.

#### 4.4.1.6. Lanzamiento de la navegación mediante teclado

Para poner en funcionamiento esta pequeña arquitectura de cuatro nodos tenemos dos opciones, ir ejecutando nodo a nodo o crear un archivo .launch que lance los cuatro nodos a la vez. En primer lugar vamos a ver como se lanza la arquitectura mediante el uso de un archivo .launch:

1. Lo primero que hay que hacer es acceder al directorio donde se encuentra nuestro proyecto:

```
1 $ cd /catkin_ws
```

2. Una vez en el espacio de trabajo se tiene que compilar el proyecto entero:

```
1 $ catkin_make
```

3. Si la compilación del proyecto no ha dado ningún problema se pasa a hacer un source del archivo setup.bash:

```
1 $ . devel/setup.bash
```

4. Accedemos al directorio src:

```
1 $ cd src
```

5. Lanzamos el launch que ejecutará el nodo maestro de ROS y los cuatro nodos descritos anteriormente:

```
1 $ rosrun scripts_lola2 lola2.launch
```

Si hemos realizado bien todos los pasos descritos nos deberá aparecer una terminal. En el capítulo dedicado a ROS ya vimos como podíamos ver la información que cada topic contenía para una mejor depuración y entendimiento de lo que está pasando.

Si se pulsa cualquiera de las teclas w, s, a, d se debería apreciar como la plataforma empieza a desplazarse. Si no es el caso, se debería ir comprobando nodo a nodo si este se encuentra funcionando correctamente. Es por eso que vamos a ver la otra forma de lanzar la arquitectura, nodo a nodo:

1. Abrir 5 terminales.
2. En una de ellas lanzar el nodo maestro:

```
1 $ roscore
```

3. En el resto de terminales se debe seguir la siguiente secuencia de comandos:

```
1 $ cd /catkin_ws
2 $ catkin_make
3 $ . devel/setup.bash
4 $ cd src
```

4. Una vez lista cada terminal se deberá ejecutar cada nodo en cada una de las terminales:

```
1 $ rosrun scripts_lola2 [nombre del archivo que contenga el nodo]
```

Al ejecutar cada nodo veremos que la información relativa a este aparecerá en la terminal desde la que se ha ejecutado, por lo que se puede ir nodo por nodo ejecutando la arquitectura e ir viendo qué mensajes se publican al pulsar las teclas del teclado. De esta manera se puede entender perfectamente la arquitectura y se puede comprobar si hay algún error al intercambiar los mensajes ROS.

Uno de los errores más comunes que nos podemos encontrar al intentar lanzar esta estructura de nodos, si hemos copiado tal cual el proyecto, es que los archivos ejecutables que se encuentran en la carpeta bin no tengan permisos de ejecución. Para solucionar esto simplemente hay que abrir una terminal, dirigirse hasta el directorio bin donde se encuentran todos estos archivos y mediante el comando chmod darle a cada uno de ellos permisos de ejecución.

## 4.5. Navegación autónoma

El objetivo principal de este trabajo de fin de grado siempre ha sido llegar a construir un sistema que sea capaz de navegar autónomamente. Durante todo este capítulo se han explicado que pasos hay que seguir para tener la plataforma funcionando correctamente, y es en esta sección en la que abordaremos como se ha conseguido navegar de manera autónoma gracias a la ayuda de RVIZ.

Todo este sistema de navegación ha sido rescatado de trabajos previos que se han realizado en el grupo de investigación GRAM. Lo que se ha hecho para esta plataforma es adaptar el sistema de navegación que una plataforma previa tenía. Hay que destacar que aquella plataforma tenía una arquitectura muy distinta a la nuestra actual, la parte más destacable es la presencia de un USB Can. Puede parecer que este pequeño elemento no afecta en gran medida a la comunicación de los elementos de la plataforma, pero cambia radicalmente la comunicación entre el computador y la placa de Arduino.

En la figura 4.11 se puede ver el diseño que se hizo para la plataforma KATE (plataforma previa a LOLA). En la conexión con el script principal, hwinterface\_script\_kate, se puede ver como la comunicación no se establece mediante una placa de Arduino, si no que esta tiene que pasar por tres nodos distintos antes de llegar a este script. Cuando vimos esta estructura para adaptarla a nuestra plataforma nos dimos cuenta de que estos tres nodos iban a ser sustituidos por nuestra placa de Arduino, facilitando así enormemente la comunicación. En la figura 4.12 se puede ver la reorganización de toda la estructura.

### 4.5.0.1. hwinterface

Si vemos la nueva estructura que hemos adaptado para nuestra plataforma LOLA, figura 4.12, podemos apreciar que el sistema de navegación esta formado por cuatro módulos. El módulo Arduino ya ha sido explicado en secciones anteriores y por tanto no hace falta hacer una nueva implementación de este.

Con el módulo nav\_stack y el módulo intermedio que contiene el ros\_control, el kate\_hw\_interface y el kate\_control\_loop simplemente basta con importar los ficheros dentro de nuestro espacio de trabajo Catkin. Estos módulos son completamente independientes de la plataforma ya que no entran en aspectos técnicos del sistema.

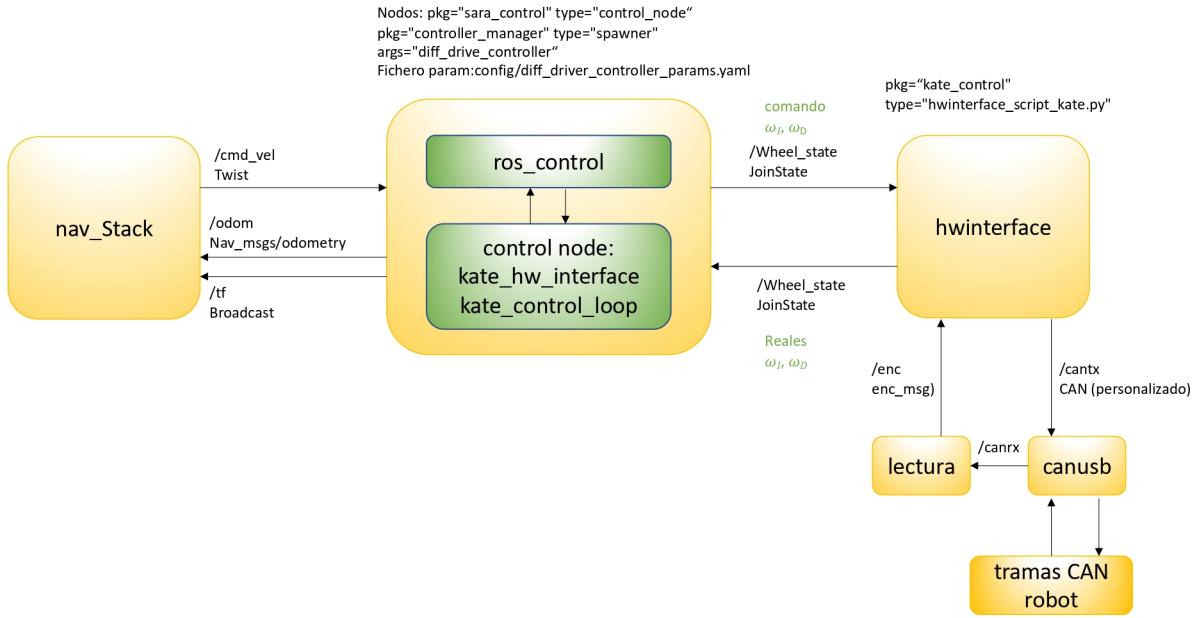


Figura 4.11: Estructura de la plataforma KATE.

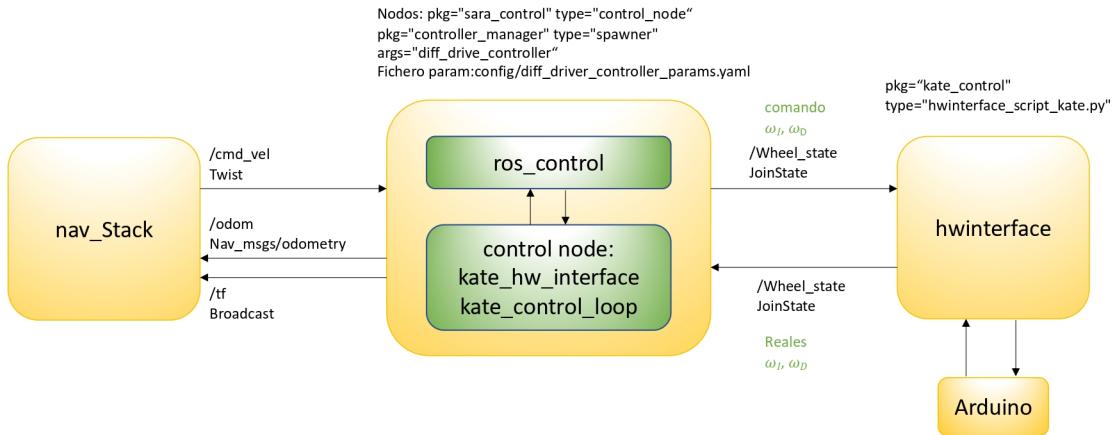


Figura 4.12: Adaptación de la arquitectura KATE a la plataforma LOLA.

El módulo hwinterface es el enlace entre nuestra placa de Arduino y los dos módulos anteriormente nombrados. Este es el módulo clave de todo el sistema de navegación y el cuál hemos tenido que reescribir casi por completo para que pudiese funcionar como enlace en nuestra arquitectura.

El código de nuestro archivo hwinterface (escrito en Python) es el siguiente:

```
1  #!/usr/bin/env python
2
3
4 import serial, time
5 import struct
6 import rospy
7 from math import copysign, pi
8 from sensor_msgs.msg import JointState
9
10
11
12 class HwClass:
13     # Function: __init__
14     #
15     # Comments
16     # -----
17     # Constructor of the class HW_CLASS
18     # Define system constants.
19     #
20     # Parameters
21     # -----
22     #
23     # Returns
24     # -----
25     def __init__(self):
26
27         # Constant parameters
28         self.pasos_vuelta= 356.3*2
29         self.left = 0
30         self.right = 1
31         self.kdato = 61.95
32         self.twistTimeout = 5
33         self.read_encoder_freq = 2
34
```

```
35     #Encoders data
36     self.time_enc_left_last=0
37     self.time_enc_right_last=0
38     self.steps_enc_left_last=0
39     self.steps_enc_right_last=0
40
41     # Serial port
42     self.arduino = serial.Serial('/dev/ttyUSB0', 115200)
43     time.sleep(0.01)
44
45
46     def __del__(self):
47         self.arduino.write(str('?').encode())
48         self.arduino.close()
49
50 # Function: main
51 #
52 # Comments
53 # -----
54 # Set up the node, the publications and subsciptions.
55 #
56 # Parameters
57 # -----
58 #
59 # Returns
60 # -----
61     def main(self):
62         rospy.init_node('hwinterface')
63         rospy.loginfo("Iniciado nodo LOLA_interface")
64
65         # Publications
66         self.data_pub = rospy.Publisher("wheel_state", JointState,
67                                         queue_size=10)
68
69         #Check the status of the robot
70         data = self.arduino.readline()
71         rospy.loginfo(data)
72         r = rospy.Rate(self.read_encoder_freq)
73         self.lastTwistTime = rospy.get_time()
74
```

```
75     # Subscriber
76     rospy.Subscriber("cmd_wheel", JointState, self.callback_vel,
77                         queue_size=1)
77     r.sleep()
78     while not rospy.is_shutdown():
79         self.check()
80         self.read_encoders()
81         r.sleep()
82
83
84 # Function: check
85 #
86 # Comments
87 # -----
88 # Check if there has been no velocity cmd_wheel message in a defined time
89 # if so, stop the robot
90 # Parameters
91 # -----
92 #
93 # Returns
94 # -----
95     def check(self):
96         if (rospy.get_time() - self.lastTwistTime) > self.twistTimeout:
97             #TODO: Enviar comando de parada a la silla
98             selfarduino.write(str('?'))
99             rospy.loginfo("Parada por no recibir datos")
100
101
102 # Function: callback_vel
103 #
104 # Comments
105 # -----
106 # Callback of the cmd_wheel topic.
107 # Convert the velocity in rad/s to data.
108 # [0-127] for positive velocity.
109 # [255-129] for negative velocity (129 maximun velocity).
110 # 0 and 128 stop the wheel.
111 #
112 # Parameters
113 # -----
114 # msg: Data of the topic
```

```

115 #
116 # Returns
117 # -----
118     def callback_vel(self,msg):
119         sign = lambda x: int(copysign(1,x))
120
121         # Save variables
122         comandD = msg.velocity[self.right] #rad/s
123         comandI = msg.velocity[self.left] #rad/s
124
125         # Convert from rad/s to data 0-127
126         datod = int(round(comandD * self.kdato))
127         datoI = int(round(comandI * self.kdato))
128
129         # Saturation of data
130         if abs(datod) > 127:
131             datod = 127 * sign(datod)
132         if abs(datoI) >127:
133             datoI = 127 * sign(datoI)
134
135         # Minimum velocity asigned
136         if (abs(datod)<10 and datod <> 0):
137             datod = 10 * sign(datod)
138         if (abs(datoI)<10 and datoI <> 0):
139             datoI = 10 * sign (datoI)
140
141         # Negative velocity convert
142         if datod<0:
143             datod=256 + datod
144         if datoI<0:
145             datoI=256 + datoI
146
147         # Send speed command
148         self.arduino.write('V'+format(int(datod),'03d')+format(datoI,'03d'
149                         ))
150
151         # Check the timeouts
152         self.lastTwistTime = rospy.get_time()
153         rospy.loginfo('V'+format(int(datod),'03d')+format(datoI,'03d'))
154

```

```
155  
156  
157 # Function: read_enc  
158 #  
159 # Comments  
160 # -----  
161 # Read the encoders of the robot  
162 # Calculate the position and velocity of the wheels with the encoders  
# data  
163 # and publish in a JointState message  
164 #  
165 # Parameters  
166 # -----  
167 #  
168 # Returns  
169 # -----  
170     def read_encoders(self):  
171         rospy.loginfo("funcion lectura")  
172  
173         # Command that gives back the encoders value  
174         self.arduino.write(str('N').encode())  
175  
176         # Read the encoders, chain of 16 bits  
177         comienzo = self.arduino.readline()  
178         steps_enc_left = struct.unpack('I',self.arduino.read(4))[0]  
179         time_enc_left = struct.unpack('I',self.arduino.read(4))[0]  
180         steps_enc_right = struct.unpack('I',self.arduino.read(4))[0]  
181         time_enc_right = struct.unpack('I',self.arduino.read(4))[0]  
182         final = self.arduino.readline()  
183         data = JointState()  
184  
185         '''CALCS ON THE RIGHT SIDE'''  
186  
187         # Increment calc  
188         dt_right = float((time_enc_right - self.time_enc_right_last) *  
# (10**-6))  
189         dsteps_right = float(steps_enc_right - self.steps_enc_right_last)  
190         self.time_enc_right_last = time_enc_right  
191         self.steps_enc_right_last = steps_enc_right  
192  
193
```

```

194     # Position
195     posD = (steps_enc_right/self.pasos_vuelta) * 2 * pi
196
197     # Angular speed
198     wd = ((dsteps_right/self.pasos_vuelta) * 2 * pi) / dt_right
199
200     # Save data to publish
201     data.name = ["RIGHT"]
202     data.position = [posD]
203     data.velocity = [wd]
204     data.effort = [0]
205     data.header.stamp = rospy.Time.now()
206     data.header.frame_id = "base_link"
207
208     # Publish topic
209     self.data_pub.publish(data)
210     rospy.loginfo("Pasos encoder: "+str(steps_enc_right))
211     rospy.loginfo("ENCD: "+ str(dsteps_right) + " TIEMPD: "+str(
212         time_enc_right))
213
214     '''CALCS ON THE LEFT SIDE'''
215
216     # Increment calc
217     dt_left = float((time_enc_left-self.time_enc_left_last)*(10**-6))
218     dsteps_left = float(steps_enc_left-self.steps_enc_left_last)
219     self.time_enc_left_last = time_enc_left
220     self.steps_enc_left_last = steps_enc_left
221
222     # Position
223     posI = (steps_enc_left/self.pasos_vuelta) * 2 * pi
224
225     # Angular speed
226     wi = ((dsteps_left/self.pasos_vuelta) * 2 * pi) / dt_left
227
228     # Save data to publish
229     data = JointState()
230     data.name = ["LEFT"]
231     data.position = [posI]
232     data.velocity = [wi]
233     data.effort = [0]
234     data.header.stamp = rospy.Time.now()

```

```

234     data.header.frame_id = "base_link"
235
236     # Publish topic
237     self.data_pub.publish(data)
238     rospy.loginfo("Pasos encoder: "+str(steps_enc_left))
239     rospy.loginfo("ENCI: "+ str(dsteps_left) + " TIEMPI: "+str(
240         time_enc_left))
241
242 if __name__ == '__main__':
243     foo = HwClass()
244     foo.main()

```

Todo el código de este script, que consigue establecer la comunicación entre la arquitectura diseñada para la plataforma KATE y nuestra placa de Arduino, está comentado destacando las líneas más importantes del programa. El script se compone de varias funciones:

- **`__init__`**: constructor de la clase HW\_CLASS encargado de definir las constantes del sistema tales como el puerto USB mediante el que se establece la conexión con la placa de Arduino (**línea 25**).
- **`main`**: función de puesta en marcha del nodo. El nombre de este será **hwinterface**, las publicaciones las hará sobre el topic **wheel\_state** y estará suscrito al topic **cmd\_wheel** (**línea 61**).
- **`check`**: detecta si no se ha enviado un mensaje de velocidad en un tiempo determinado y, si es el caso, manda un mensaje de parada a los motores entendiendo así que la navegación ha concluido
- **`callback_vel`**: función que recoge los datos de velocidad del topic cmd\_wheel en radianes y los convierte al sistema de unidades que maneja la plataforma LOLA, siendo de 1 a 127 velocidad positiva y de 255 a 129 velocidad negativa. Los valores 0 y 128 se utilizan para parar las ruedas (**línea 118**).
- **`read_encoders`**: esta es la función más importante del programa. Es la encarga de leer los valores de los encoders de los motores y parsear la información recibida por la placa de Arduino. Para ello se manda el comando **N** a la placa de Arduino y así esta empezará a enviar una cadena de texto con los datos de ambos encoders.

La cadena de texto está formada por 16 bytes, dividiéndose de la siguiente manera:

- 1 - 4: valor del encoder izquierdo.
- 5 - 8: marca de tiempo del encoder izquierdo.

- 9 - 12: valor del encoder derecho.
- 13 - 16: marca de tiempo del encoder derecho.

Una vez recogidos los datos de los encoders se calcula, tanto para la rueda derecha como para la izquierda, la posición y la velocidad angular. Una vez calculados estos valores se publican en el topic para poder calcular las nuevas velocidades que necesitan tomar las ruedas para proseguir con la navegación.

## 4.6. Reconocimiento de acciones online

Uno de los objetivos de este trabajo de fin de grado siempre ha sido el de implementar un sistema de reconocimiento de acciones. Este sistema ha sido desarrollado, dentro del grupo de investigación GRAM, por el doctor Marcos Baptista Ríos.

Durante la implementación de este sistema nos dimos cuenta de que la placa NVIDIA Jetson TX2 no era capaz de correr este sistema debido a la potencia computacional requerida por este, es por ello que decidimos implementar un nodo ROS online.

Este nodo ROS se encarga de publicar las imágenes recogidas por la cámara de la plataforma y de publicarlas en un topic. Este topic tiene la particularidad de que puede ser visto por otros nodos que no se encuentren en la misma arquitectura, es por ello que decimos que es un nodo capaz de publicar de manera online.

Otro nodo ROS ejecutado en una GPU más potente se encarga de recoger las imágenes publicadas en este topic y de mandarlas a procesar al programa de detección de acciones. Una vez detectada la acción que se está realizando, este mismo nodo va a publicar el nombre de la acción en otro topic online. De esta manera la plataforma va a poder saber qué acción está realizando el usuario y, en el futuro, ser capaz de adaptar su comportamiento en función de la acción.

### 4.6.1. Configuración para ejecutar nodos ROS en múltiples máquinas

Para poder implementar el sistema de reconocimiento de acciones online hay que configurar antes las dos máquinas. De ahora en adelante vamos a suponer que tenemos un ordenador A con usuario userA y un ordenador B con usuario userB. Supondremos que la dirección del ordenador A es 192.168.1.10 y la del ordenador B es 192.168.1.20 (en una implementación real cada uno tiene que poner su usuario y dirección IP reales).

A continuación vamos a ver los pasos que hay que seguir para llevar a cabo la configuración en las dos máquinas, lo primero es la configuración de servicios SSH:

1. Comprobar si tenemos instalado openSSH en ambos ordenadores. Si no es así, instalar.
2. En el ordenador A editamos el fichero hosts. Esto lo tenemos que hacer cuando cambiemos de IP en este ordenador o en el B:
  - Introducir el comando:

```
1 userA@ordenadorA:~$ sudo nano /etc/hosts
```
  - Aparecerá por lo general algo del tipo:

```
1 127.0.0.1 localhost
2 127.0.0.1 ordenadorA
```
  - Tenemos que modificarlos a lo siguiente:

```
1 127.0.0.1 localhost
2 192.168.1.10 ordenadorA
3 192.168.1.20 ordenadorB
```
3. Repetir la misma operación en el ordenador B sustituyendo esas líneas por el mismo código que en A.
4. En el ordenador A generaremos un sistema de claves públicas y privadas:

```
1 userA@ordenadorA:~$ ssh-keygen t rsa
```
5. Copiamos la clave pública a nuestro ordenador B:

```
1 userA@ordenadorA:~$ ssh-copy-id -i ~/.ssh/id_rsa.pub
userB@ordenadorB
```
6. Probamos la conexión, pero es importante que lo hagamos la primera vez mediante el siguiente comando:

```
1 userA@ordenadorA:~$ ssh -oHostKeyAlgorithms=ssh-rsa
userB@ordenadorB
```
7. Las siguientes veces que nos queramos conectar será suficiente con poner:

```
1 $ ssh userB@ordenadorB
```
8. Repetimos todo el proceso del ordenador A en el ordenador B, al que ahora nos podemos conectar por SSH sin contraseña.

Una vez realizada la configuración pasemos a ver la creación de ficheros de entorno:

1. Lo primero es decidir en qué máquina va a lanzarse el roscore. En este ejemplo va a lanzarse en la máquina A. Creamos en el ordenador A el siguiente fichero y lo guardamos, por ejemplo como:

```
1 multi_A.sh en /home/userA/catkin_ws/devel
```

2. El contenido del archivo debe ser el siguiente:

```
1 #!/bin/bash
2 source /home/userA/catkin_ws/devel/setup.bash
3 export ROS_IP=ordenadorA
4 export ROS_MASTER_URI=http://ordenadorA:11311
5 export ROS_HOSTNAME=ordenadorA
6
7 exec $@
```

3. En el ordenador B debemos crear otro fichero que guardaremos, por ejemplo, como:

```
1 multi_remoteB.sh en /home/userB/catkin_ws/devel
```

4. El contenido del archivo debe ser el siguiente:

```
1 #!/bin/bash
2 source /home/userB/catkin_ws/devel/setup.bash
3 export ROS_IP=ordenadorB
4 export ROS_MASTER_URI=http://ordenadorA:11311
5 export ROS_HOSTNAME=ordenadorB
6
7 exec $@
```

5. La clave está en decirle que el ordenador que tiene el roscore es el ordenador A y que él publicite su IP.

Una vez que tenemos nuestro entorno preparado, ya podemos crear ficheros launch que ejecuten los nodos en diferentes, este sería un ejemplo:

```
1 <launch>
2 <machine
3   name="MA"
4   address="ordenadorA"
5   user="userA"
6   env-loader="/home/userA/catkin_ws/devel/multi_A.sh"
```

```

7          />
8 <machine
9   name="MB"
10  address="ordenadorB"
11  user="userB"
12  env-loader="/home/userB/catkin_ws/devel/multi_remoteB.sh"
13          />
14 <node
15   machine="MB"
16     name="talker"
17     pkg="ejemplo"
18     type="talker.py"
19     output="screen"
20 />
21 <node
22 machine="MA"
23 name="subscriber1"
24 pkg="ejemplo"
25 type="subscriber1.py"
26 output="screen"
27         />
28 </launch>
```

El script principal que hemos creado para poder enviar las imágenes a través de un nodo ROS es el siguiente:

```

1 import rospy
2 from sensor_msgs.msg import Image
3 import cv2
4 from cv_bridge import CvBridge, CvBridgeError
5 from distutils.version import LooseVersion
6 import numpy as np
7
8 def opencv():
9
10    br = CvBridge()
11
12    rospy.init_node('opencv')
13    pub = rospy.Publisher('video_frames', Image, queue_size=10)
14
15    rate = rospy.Rate(10)
16
```

```

17     # Load the cascade
18     face_cascade = cv2.CascadeClassifier('/home/nvidia/catkin_ws/src/
19         scripts_lola2/src/scripts_lola2/haarcascade_frontalface_default
20         .xml')
21
22     # Capture video
23     cap = cv2.VideoCapture(0)
24
25     while not rospy.is_shutdown():
26
27         # Read the frame
28         ret, frame = cap.read()
29
30         if ret == True:
31             # Print debugging information to the terminal
32             rospy.loginfo('publishing video frame')
33
34             # Publishing image
35             # The 'cv2_to_imgmsg' method converts an OpenCV
36             # image to a ROS image message
37             pub.publisher(br.cv2_to_imgmsg(frame))
38
39             #cv2.imshow('img', frame)
40
41             # Stop if escape key is pressed
42             k = cv2.waitKey(30) & 0xff
43             if k==27:
44                 break
45
46             # Release the VideoCapture object
47             cap.release()
48             cv2.destroyAllWindows()
49
50     """
51     # Software License Agreement (BSD License)
52     #
53     # Copyright (c) 2011, Willow Garage, Inc.
54     # Copyright (c) 2016, Tal Regev.
55     # All rights reserved.
56     #
57     # Redistribution and use in source and binary forms, with or without

```

```

56 # modification, are permitted provided that the following conditions
57 # are met:
58 #
59 # * Redistributions of source code must retain the above copyright
60 # notice, this list of conditions and the following disclaimer.
61 # * Redistributions in binary form must reproduce the above
62 # copyright notice, this list of conditions and the following
63 # disclaimer in the documentation and/or other materials provided
64 # with the distribution.
65 # * Neither the name of Willow Garage, Inc. nor the names of its
66 # contributors may be used to endorse or promote products derived
67 # from this software without specific prior written permission.
68 #
69 # THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS
70 # "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
71 # LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS
72 # FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE
73 # COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT,
74 # INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
75 # BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
76 # LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER
77 # CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
78 # LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN
79 # ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
80 # POSSIBILITY OF SUCH DAMAGE.

81
82 import sensor_msgs.msg
83 import sys
84
85
86 class CvBridgeError(TypeError):
87     """
88     This is the error raised by :class:`cv_bridge.CvBridge` methods when
89     they fail.
90     """
91
92
93 class CvBridge(object):
94     """
95     The CvBridge is an object that converts between OpenCV Images and ROS

```

```

Image messages.

96
97 .. doctest::
98 :options: -ELLIPSIS, +NORMALIZE_WHITESPACE
99
100    >>> import cv2
101    >>> import numpy as np
102    >>> from cv_bridge import CvBridge
103    >>> br = CvBridge()
104    >>> dtype, n_channels = br.encoding_as_cvtype2('8UC3')
105    >>> im = np.ndarray(shape=(480, 640, n_channels), dtype=dtype)
106    >>> msg = br.cv2_to_imgmsg(im) # Convert the image to a message
107    >>> im2 = br.imgmsg_to_cv2(msg) # Convert the message to a new
108        image
109    >>> cmprsmsg = br.cv2_to_compressed_imgmsg(im) # Convert the
110        image to a compress message
111    >>> im22 = br.compressed_imgmsg_to_cv2(msg) # Convert the
112        compress message to a new image
113    >>> cv2.imwrite("this_was_a_message_briefly.png", im2)

114 """
115
116 def __init__(self):
117     import cv2
118     self.cvtype_to_name = {}
119     self.cvdepth_to_numpy_depth = {cv2.CV_8U: 'uint8', cv2.CV_8S: '
120         int8', cv2.CV_16U: 'uint16',
121                         cv2.CV_16S: 'int16', cv2.CV_32S:'int32
122                         ', cv2.CV_32F:'float32',
123                         cv2.CV_64F: 'float64'}
124
125     for t in ["8U", "8S", "16U", "16S", "32S", "32F", "64F"]:
126         for c in [1, 2, 3, 4]:
127             nm = "%sC%d" % (t, c)
128             self.cvtype_to_name[getattr(cv2, "CV_%s" % nm)] = nm
129
130     self.numpy_type_to_cvtype = {'uint8': '8U', 'int8': '8S', 'uint16':
131         '16U',
132                         'int16': '16S', 'int32': '32S', ,
133                         float32: '32F',
134                         'float64': '64F'}

```

```

129     self.numpy_type_to_cvtype.update(dict((v, k) for (k, v) in self.
130                                         numpy_type_to_cvtype.items()))
130
131     def dtype_with_channels_to_cvtype2(self, dtype, n_channels):
132         return '%sC%d' % (self.numpy_type_to_cvtype[dtype.name],
133                            n_channels)
133
134     def cvtype2_to_dtype_with_channels(self, cvtype):
135         from cv_bridge.boost.cv_bridge_boost import CV_MAT_CNWrap,
136                                         CV_MAT_DEPTHWrap
136         return self.cvdepth_to_numpy_depth[CV_MAT_DEPTHWrap(cvtype)],
137                                         CV_MAT_CNWrap(cvtype)
137
138     def encoding_to_cvtype2(self, encoding):
139         from cv_bridge.boost.cv_bridge_boost import getCvType
140
141         try:
142             return getCvType(encoding)
143         except RuntimeError as e:
144             raise CvBridgeError(e)
145
146     def encoding_to_dtype_with_channels(self, encoding):
147         return self.cvtype2_to_dtype_with_channels(self.
148                                         encoding_to_cvtype2(encoding))
148
149     def compressed_imgmsg_to_cv2(self, cmprs_img_msg, desired_encoding = "
150                                 passthrough"):
150         """
151             Convert a sensor_msgs::CompressedImage message to an OpenCV :cpp:
152             type: 'cv::Mat'.
153
154             :param cmprs_img_msg: A :cpp:type: 'sensor_msgs::CompressedImage'
155                                         message
156             :param desired_encoding: The encoding of the image data, one of
157                                         the following strings:
158
159                 * 'passthrough'
160                 * one of the standard strings in sensor_msgs/image_encodings.h
161
162             :rtype: :cpp:type: 'cv::Mat'
163             :raises CvBridgeError: when conversion is not possible.

```

```

161
162     If desired_encoding is "passthrough", then the returned image
163         has the same format as img_msg.
164     Otherwise desired_encoding must be one of the standard image
165         encodings
166
167     This function returns an OpenCV :cpp:type:`cv::Mat` message on
168         success, or raises :exc:`cv_bridge.CvBridgeError` on failure.
169
170     If the image only has one channel, the shape has size 2 (width and
171         height)
172     """
173     import cv2
174     import numpy as np
175
176     str_msg = cmptrs_img_msg.data
177     buf = np.ndarray(shape=(1, len(str_msg)),
178                      dtype=np.uint8, buffer=cmptrs_img_msg.data)
179     im = cv2.imdecode(buf, cv2.IMREAD_ANYCOLOR)
180
181     if desired_encoding == "passthrough":
182         return im
183
184     from cv_bridge.boost.cv_bridge_boost import cvtColor2
185
186     try:
187         res = cvtColor2(im, "bgr8", desired_encoding)
188     except RuntimeError as e:
189         raise CvBridgeError(e)
190
191     return res
192
193 def imgmsg_to_cv2(self, img_msg, desired_encoding = "passthrough"):
194     """
195     Convert a sensor_msgs::Image message to an OpenCV :cpp:type:`cv::
196         Mat`.
197
198     :param img_msg: A :cpp:type:`sensor_msgs::Image` message
199     :param desired_encoding: The encoding of the image data, one of
200         the following strings:

```

```

196     * "passthrough"
197     * one of the standard strings in sensor_msgs/image_encodings.h
198
199 :rtype: :cpp:type:`cv::Mat`
200 :raises :exc:`cv_bridge.CvBridgeError` when conversion is not possible.
201
202 If desired_encoding is "passthrough", then the returned image
203 has the same format as img_msg.
204 Otherwise desired_encoding must be one of the standard image
205 encodings
206
207 This function returns an OpenCV :cpp:type:`cv::Mat` message on
208 success, or raises :exc:`cv_bridge.CvBridgeError` on failure.
209
210 If the image only has one channel, the shape has size 2 (width and
211 height)
212 """
213 import cv2
214 import numpy as np
215 dtype, n_channels = self.encoding_to_dtype_with_channels(img_msg.
216     encoding)
217 dtype = np.dtype(dtype)
218 dtype = dtype.newbyteorder('>' if img_msg.is_bigendian else '<')
219 if n_channels == 1:
220     im = np.ndarray(shape=(img_msg.height, img_msg.width),
221                     dtype=dtype, buffer=img_msg.data)
222 else:
223     im = np.ndarray(shape=(img_msg.height, img_msg.width,
224                         n_channels),
225                     dtype=dtype, buffer=img_msg.data)
226 # If the byte order is different between the message and the system
227 #
228 if img_msg.is_bigendian == (sys.byteorder == 'little'):
229     im = im.byteswap().newbyteorder()
230
231 if desired_encoding == "passthrough":
232     return im
233
234 from cv_bridge.boost.cv_bridge_boost import cvtColor2
235
236 try:

```

```

230         res = cvtColor2(im, img_msg.encoding, desired_encoding)
231     except RuntimeError as e:
232         raise CvBridgeError(e)
233
234     return res
235
236 def cv2_to_compressed_imgmsg(self, cvim, dst_format = "jpg"):
237     """
238     Convert an OpenCV :cpp:type:`cv::Mat` type to a ROS sensor_msgs::
239     CompressedImage message.
240
241     :param cvim: An OpenCV :cpp:type:`cv::Mat`
242     :param dst_format: The format of the image data, one of the
243     following strings:
244
245         * from http://docs.opencv.org/2.4/modules/highgui/doc/reading\_and\_writing\_images\_and\_video.html
246         * from http://docs.opencv.org/2.4/modules/highgui/doc/reading\_and\_writing\_images\_and\_video.html#Mat imread\(const string& filename, int flags\)
247             * bmp, dib
248             * jpeg, jpg, jpe
249             * jp2
250             * png
251             * pbm, pgm, ppm
252             * sr, ras
253             * tiff, tif
254
255     :rtype: A sensor_msgs.msg.CompressedImage message
256     :raises CvBridgeError: when the ``cvim`` has a type that is
257     incompatible with ``format``
258
259     import cv2
260     import numpy as np
261     if not isinstance(cvim, (np.ndarray, np.generic)):
262         raise TypeError('Your input type is not a numpy array')
263     cmprs_img_msg = sensor_msgs.msg.CompressedImage()

```

```

264     cmprs_img_msg.format = dst_format
265     ext_format = '.' + dst_format
266     try:
267         cmprs_img_msg.data = np.array(cv2.imencode(ext_format, cvim)
268                                     [1]).tostring()
269     except RuntimeError as e:
270         raise CvBridgeError(e)
271
272     return cmprs_img_msg
273
274 def cv2_to_imgmsg(self, cvim, encoding = "passthrough"):
275     """
276     Convert an OpenCV :cpp:type:`cv::Mat` type to a ROS sensor_msgs::
277     Image message.
278
279     :param cvim: An OpenCV :cpp:type:`cv::Mat`
280     :param encoding: The encoding of the image data, one of the
281                     following strings:
282
283         * `""passthrough""`  

284         * one of the standard strings in sensor_msgs/image_encodings.h
285
286     :rtype: A sensor_msgs.msg.Image message
287     :raises CvBridgeError: when the ``cvim`` has a type that is
288                           incompatible with ``encoding``
289
290     If encoding is `""passthrough""`, then the message has the same
291     encoding as the image's OpenCV type.
292     Otherwise desired_encoding must be one of the standard image
293     encodings
294
295     This function returns a sensor_msgs::Image message on success, or
296     raises :exc:`cv_bridge.CvBridgeError` on failure.
297     """
298
299     import cv2
300     import numpy as np
301     if not isinstance(cvim, (np.ndarray, np.generic)):
302         raise TypeError('Your input type is not a numpy array')
303     img_msg = sensor_msgs.msg.Image()
304     img_msg.height = cvim.shape[0]
305     img_msg.width = cvim.shape[1]

```

```

298     if len(cvim.shape) < 3:
299         cv_type = self.dtype_with_channels_to_cvtype2(cvim.dtype, 1)
300     else:
301         cv_type = self.dtype_with_channels_to_cvtype2(cvim.dtype, cvim
302             .shape[2])
302     if encoding == "passthrough":
303         img_msg.encoding = cv_type
304     else:
305         img_msg.encoding = encoding
306         # Verify that the supplied encoding is compatible with the
307             type of the OpenCV image
307     if self.cvtype_to_name[self.encoding_to_cvtype2(encoding)] !=
308         cv_type:
309         raise CvBridgeError("encoding specified as %s, but image
310             has incompatible type %s" % (encoding, cv_type))
310     if cvim.dtype.byteorder == '>':
311         img_msg.is_bigendian = True
311     img_msg.data = cvim.tostring()
312     img_msg.step = len(img_msg.data) // img_msg.height
313
314     return img_msg

```

Este archivo se encarga de crear un nodo ROS de nombre **opencv**, este va a publicar todas las imágenes que la cámara capta en el topic de nombre **video\_frames**. Todo el código de este script ha sido desarrollado usando Python 3 y la librería **cv2**. No vamos a entrar en profundidad de como se ha usado esta librería ya que no entra dentro de las competencias de este trabajo de fin de grado. La mayoría de sentencias que nos podemos encontrar son de la librería **rospy** que ya hemos visto anteriormente durante el desarrollo del sistema de navegación, el resto son propias de la librería **cv2**.

Antes de lanzar este nodo en nuestra plataforma LOLA tendremos que lanzar el roscore en el otro computador y, una vez que este en funcionamiento, podremos lanzar este nodo y el programa de detección de acciones.

# Capítulo 5

## Resultados

En esta sección vamos a pasar a ver como lanzar el sistema de navegación autónoma y a comentar los resultados que hemos obtenido según las últimas pruebas que hemos realizado.

Debemos seguir los siguientes pasos para lanzar la navegación:

1. Abrir un terminal y escribir la siguiente secuencia de comandos:

```
1 $ cd ~/catkin_lola2
2 $ catkin_make
3 $ . devel/setup.bash
4 $ cd src
5 $ rosrun lola2_global hwinterface_script_lola2
```

2. Abrir otro terminal y escribir la siguiente secuencia de comandos:

```
1 $ cd ~/catkin_lola2
2 $ catkin_make
3 $ . devel/setup.bash
4 $ cd src
5 $ roslaunch lola2_global rviz_navigation.launch
```

El motivo por el que tenemos que ejecutar todo desde dos terminales se debe a que si no ejecutamos primero el **hwinterface\_script\_lola2.py** este se quedará esperando a que termine el **rviz\_navigation.launch**. Es por ello que lanzamos primero el **hwinterface\_script\_lola2.py** y luego el **rviz\_navigation.launch** que desplegará toda la estructura de nodos que necesitamos para la navegación.

Una vez que lancemos todo nos aparecerá la ventana de RVIZ que podemos ver en la figura 5.1.

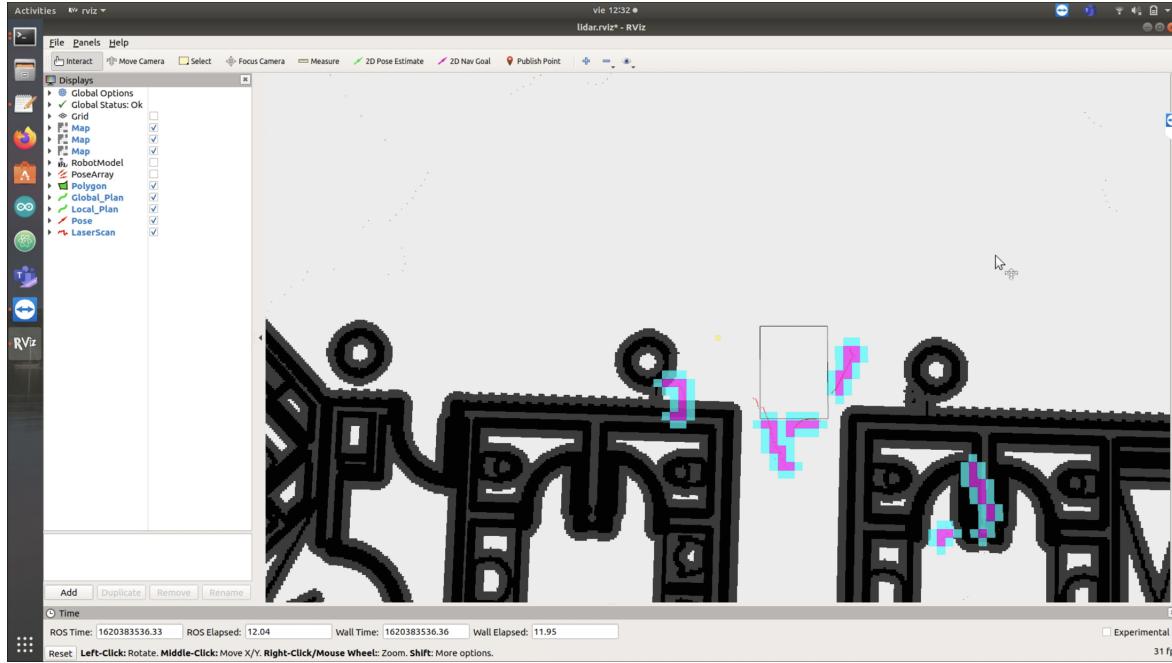


Figura 5.1: Despliegue de RVIZ para iniciar la navegación.

Lo normal será que nuestra plataforma LOLA no se encuentre en la posición en la que aparece en el mapa, por lo que habrá que indicarle a RVIZ donde se encuentra. Para ello seleccionaremos la opción **2D Pose Estimate** y, haciendo click y arrastrando el cursor, le indicaremos la posición y orientación en la que se encuentra la plataforma.

Cuando tengamos la plataforma en donde se encuentra en el mundo real, solo tenemos que indicar el punto donde queremos que vaya. Para ello solo tenemos que seleccionar la opción **2D Nav Goal** y hacer clic en el mapa donde queremos que vaya la plataforma. Mientras haces clic en el mapa para seleccionar el punto tienes que arrastrar el cursor indicando la orientación que quieras que tenga la plataforma cuando llegue al punto.

En las figuras 5.2 podemos ver como la plataforma navega. En estas capturas de un vídeo completo (el cual se pude ver siguiendo el siguiente enlace <https://www.youtube.com/watch?v=qjXZxAmTKXk>) se puede observar como la plataforma sigue la línea de ruta indicada en RVIZ y como esta va detectando los distintos obstáculos que se encuentran. La última captura corresponde al final de la ruta de navegación, llegando hasta el punto indicado en frente de la persona.

Si nos paramos a analizar las distintas pruebas de navegación que hemos llevado a cabo podemos concluir que hemos llevado a cabo la correcta integración del sistema de navegación autónoma en la plataforma de bajo coste LOLA. La mayoría de las pruebas han dado un resultado favorable, elegiendo la mejor ruta para llegar a su destino y parándose en el punto adecuado con la orientación adecuada.

Cabe destacar que este sistema tiene sus limitaciones, ya que hay ciertas rutas que

la plataforma no puede realizar debido al tamaño de esta. Es por ello que todavía queda mucho trabajo que hacer para que esta puede navegar por pasillos estrechos.

Otro punto a mejorar es el tiempo que tarde la plataforma en navegar hasta su destino. Este es un problema que no se ha conseguido resolver y que puede ser un muy buen punto a seguir dentro del desarrollo de la plataforma. Hemos detectado que hay un pequeño delay entre mensajes ROS que hace que la plataforma se quede esperando a ciertos mensajes que tardan bastante en llegar, retrasando así la navegación.

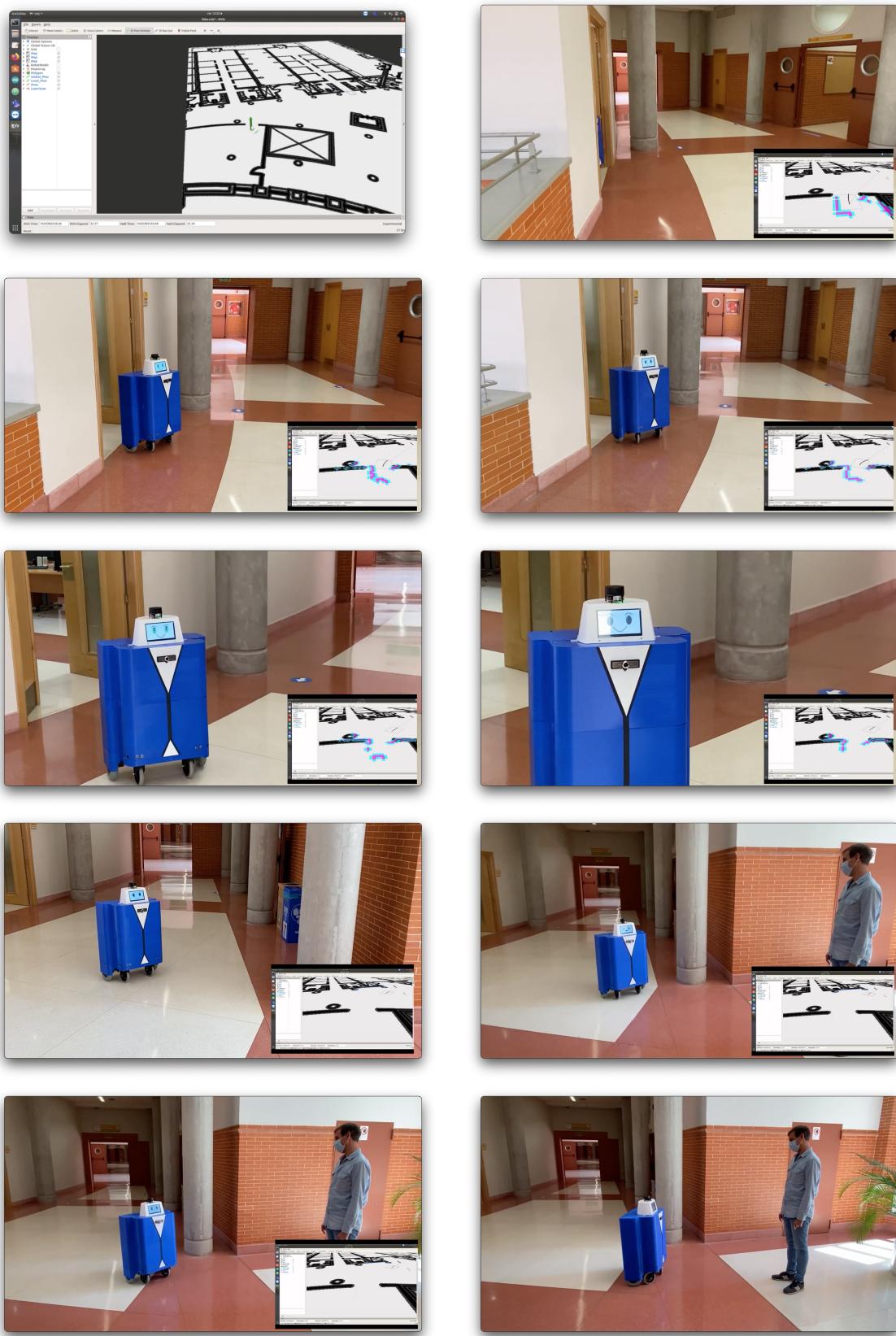


Figura 5.2: Plataforma LOLA navegando.

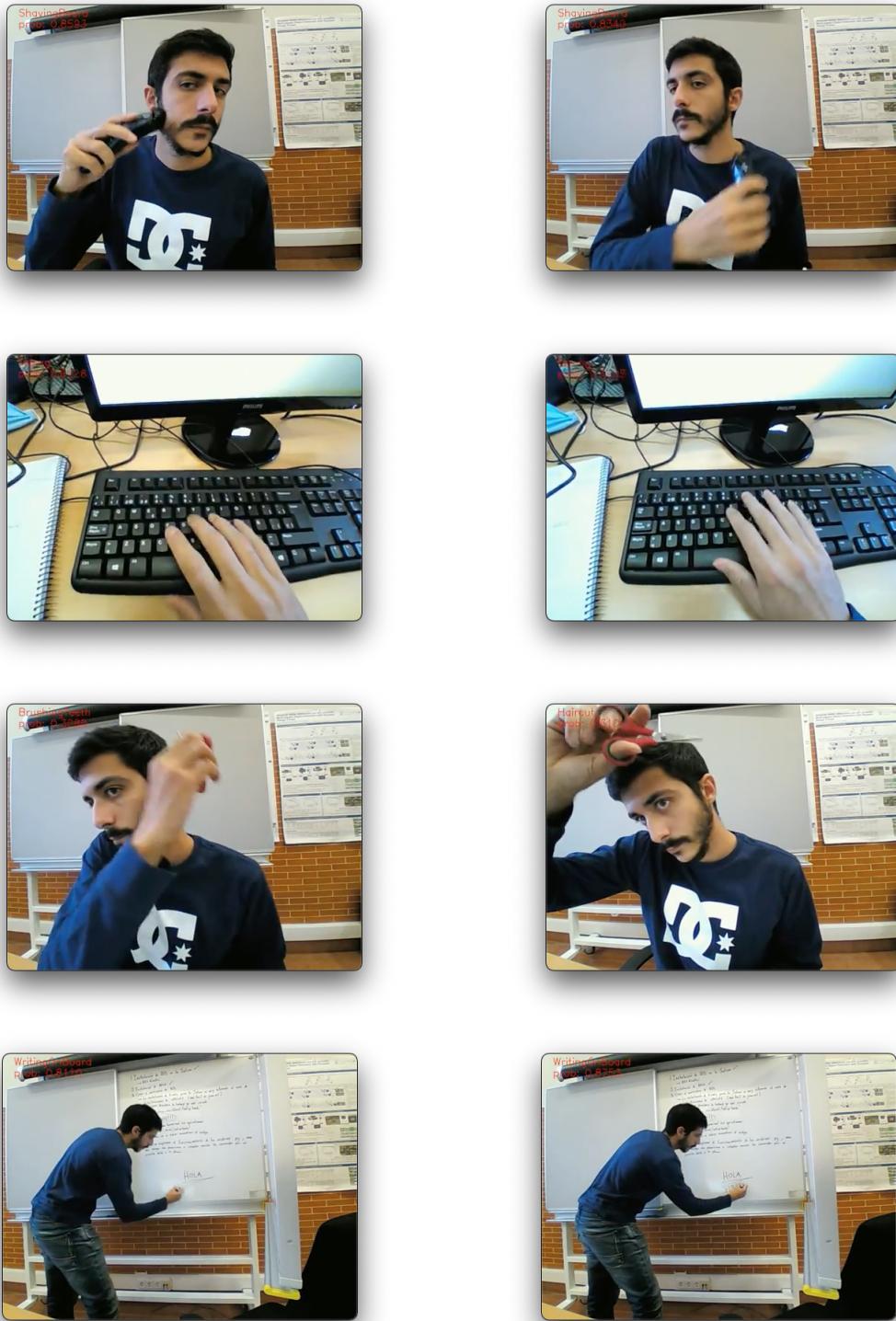


Figura 5.3: Sistema de detección de acciones online en funcionamiento.



# Capítulo 6

## Conclusiones

Actualmente el objetivo principal de este trabajo de fin de grado ha sido llevado a cabo correctamente, se ha conseguido desarrollar una plataforma robótica asistencial de bajo coste en la que se han integrado modelos de inteligencia artificial capaces de conseguir una navegación autónoma por un entorno dado previamente mediante el uso de ROS, y también es capaz de enviar imágenes captadas por cámara a una GPU externa y que estas sean analizadas para deducir qué tipo de acción se está desarrollando en ellas.

Durante el desarrollo e implementación nos hemos topado con dificultades, sobre todo a la hora de encontrar una concordancia entre las diversas herramientas utilizadas y la versión de las mismas. Es el caso de la NVIDIA Jetson TX2, placa que tuvo que ser sustituida de la arquitectura debido a la dificultad de implementar ciertos paquetes que Python3 requería pero que eran inviables para la versión de Ubuntu 16.04 que corre la placa. Esto nos hizo darnos cuenta de la importancia de preparar un entorno de trabajo estable en el que tuviésemos cada elemento software controlado. Así fue como decidimos restablecer el desarrollo en una computadora con Ubuntu 18.04 donde sabíamos que los problemas de versiones iban a desaparecer.

El estudio y uso de ROS nos ha abierto las puertas a una infinidad de posibilidades que, de otra manera, serían mucho más difíciles de llevar a cabo. El concepto de nodos y topics es muy simple, pero a la vez muy potente, ya que nos permite crear una gran arquitectura intercomunicada entre sí pero a la vez robusta. Son grandes las virtudes de ROS y es por ello por lo que un gran paso a dar (siguiendo con el desarrollo de esta plataforma) sería el estudio e implementación de ROS2. Como hablamos en el capítulo dedicado a ROS, esta nueva versión trae una serie de mejoras que podrían beneficiar a la plataforma.

Los pasos a seguir ahora que hemos conseguido completar el objetivo principal de este trabajo de fin de grado sería la mejora de los tiempos de navegación. Aunque es cierto que la plataforma es capaz de navegar autonomamente, esta suele requerir un tiempo moderado para recalcular la ruta cuando se encuentra con ciertos elementos durante la

navegación. Es por ello que una buena continuación de este trabajo sería la mejora de la navegación aumentando la eficiencia de esta, tanto en tiempo como en recorrido.

# Bibliografía

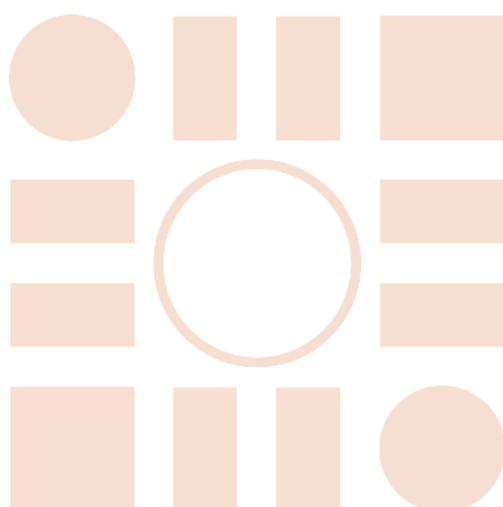
- [1] Presentación. URL <https://www.hisparob.es/presentacion/>. 13
- [2] Different types of robot programming languages. URL <https://www.plantautomation-technology.com/articles/different-types-of-robot-programming-languages>. 6, 7
- [3] Maggie: futuro, autonomía y diversión. URL [https://portal.uc3m.es/portal/page/portal/actualidad\\_cientifica/actualidad/reportajes/archivo\\_reportajes/Maggie\\_futuro\\_autonomia\\_diversion](https://portal.uc3m.es/portal/page/portal/actualidad_cientifica/actualidad/reportajes/archivo_reportajes/Maggie_futuro_autonomia_diversion). 11
- [4] Nvidia jetson: The ai platform for edge computing. URL <https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/>. 8
- [5] What is a ros parameter? URL <https://roboticsbackend.com/what-is-a-ros-parameter/>. 23
- [6] Rplidar a1. URL <https://www.slamtec.com/en/Lidar/A1Spec>. 31
- [7] Definition of robot. URL <https://www.lexico.com/definition/robot>. 5
- [8] Arduino a000052. URL [https://www.mouser.es/ProductDetail/Arduino/A000052?qs=JYLCs0CWXI0EuuE9AXNMFg=&mgh=1&vip=1&gclid=Cj0KCQjw-NaJBhDsARIaAja6dNJK0y\\_NPF9f76kk37zyBFmlsMdQcLUqOR0xMCuDmMnti9PeBmprMaAn7SEALw\\_wcB](https://www.mouser.es/ProductDetail/Arduino/A000052?qs=JYLCs0CWXI0EuuE9AXNMFg=&mgh=1&vip=1&gclid=Cj0KCQjw-NaJBhDsARIaAja6dNJK0y_NPF9f76kk37zyBFmlsMdQcLUqOR0xMCuDmMnti9PeBmprMaAn7SEALw_wcB). IX, 9
- [9] Maggie: futuro, autonomía y diversión. URL [https://portal.uc3m.es/portal/page/portal/actualidad\\_cientifica/actualidad/reportajes/archivo\\_reportajes/Maggie\\_futuro\\_autonomia\\_diversion](https://portal.uc3m.es/portal/page/portal/actualidad_cientifica/actualidad/reportajes/archivo_reportajes/Maggie_futuro_autonomia_diversion). IX, 12
- [10] Raspberry pi 3 model b+ - placa de base. URL <https://www.amazon.es/Raspberry-Pi-Modelo-Placa-Color/dp/B07BFH96M3>. IX, 9
- [11] Sending commands from rviz. URL <https://ardupilot.org/dev/docs/ros-rviz.html>. IX, 27

- [12] What is a ros parameter? URL <https://roboticsbackend.com/what-is-a-ros-parameter/>. IX, 23
- [13] Exchange data with ros publishers and subscribers. URL <https://es.mathworks.com/help/ROS/ug/exchange-data-with-ros-publishers-and-subscribers.html>. IX, 21
- [14] Jetsonhacks developiong for nvidia jetson, 2014. URL <https://www.jetsonhacks.com/>. 41
- [15] Navigation stack review, 2016. URL <https://edu.gaitech.hk/turtlebot/navigation-stack.html>. 25
- [16] Jetson tx2 module, 2017. URL <https://developer.nvidia.com/embedded/jetson-tx2>. 33
- [17] Encuentro con takanori shibata: El caso de nuka, el robot foca, nov 2018. URL <https://espacio.fundaciontelefonica.com/evento/encuentro-con-takanori-shibata-el-caso-de-nuka-el-robot-foca/>. 12
- [18] Encuentro con takanori shibata: El caso de nuka, el robot foca, nov 2018. URL <https://espacio.fundaciontelefonica.com/evento/encuentro-con-takanori-shibata-el-caso-de-nuka-el-robot-foca/>. IX, 13
- [19] Teo, el robot que se puede comunicar en lengua de signos, jul 2019. URL <https://www.europapress.es/epsocial/igualdad/noticia-teo-robot-puede-comunicar-lengua-signos-20190708141410.html>. 11
- [20] Master, nov 2019. URL <http://library.isr.ist.utl.pt/docs/roswiki/Master.html>. IX, 20
- [21] Getting started with the nvidia jetson platform, sep 2019. URL <https://www.flir.es/support-center/iis/machine-vision/application-note/getting-started-with-the-nvidia-jetson-platform/>. IX, 8
- [22] Ros parameter server y roslaunch, sep 2020. URL <http://rostutorial.com/6-rosparam-y-roslaunch/>. 24
- [23] El robot copito facilitará la vida a los mayores de los roiales, apr 2021. URL <https://sorianoticias.com/noticia/2021-04-28-el-robot-copito-facilitara-la-vida-a-los-mayores-de-los-royales-77680>. 14

- [24] 2021 developer survey, may 2021. URL <https://insights.stackoverflow.com/survey/2021#developer-profile-demographics>. 7
- [25] El robot copito facilitará la vida a los mayores de los royaless, apr 2021. URL <https://sorianoticias.com/noticia/2021-04-28-el-robot-copito-facilitara-la-vida-a-los-mayores-de-los-royales-77680>. IX, 14
- [26] C. Balaguer. Teo humanoid, jun 2021. URL <http://roboticslab.uc3m.es/roboticslab/robot/teo-humanoid>. 10
- [27] C. Balaguer. Teo humanoid, jun 2021. URL <http://roboticslab.uc3m.es/roboticslab/robot/teo-humanoid>. IX, 11
- [28] BASQUEGEEK. Ros – servicios (i), mar 2018. URL <https://geekgasteiz.wordpress.com/2018/03/17/ros-servicios-i/>. 22
- [29] M. A. P. M. T. M. T. A. D. A. M. A. R. M. F. D. L. Castro. Un marco ético-político para la robótica asistencial. *ArtefaCToS*, page 21, apr 2019. URL <https://revistas.usal.es/index.php/artefactos/article/download/art20198197117/20284>. 10
- [30] darkcrizt. Nvidia jetson nano: una computadora para la implementación de aplicaciones ai, 2018. URL <https://blog.desdelinux.net/nvidia-jetson-nano-una-computadora-para-la-implementacion-de-aplicaciones-ai/>. 8
- [31] E. R. de Luis. De cero a maker: todo lo necesario para empezar con raspberry pi, jul 2018. URL <https://www.xataka.com/makers/cero-maker-todo-necesario-para-empezar-raspberry-pi>. 9
- [32] Y. Fernández. Qué es arduino, cómo funciona y qué puedes hacer con uno, aug 2020. URL <https://www.xataka.com/basics/que-arduino-como-funciona-que-puedes-hacer-uno>. 9
- [33] T. Foote. Topics, feb 2019. URL <http://wiki.ros.org/Topics>. 20, 21
- [34] N. Fragale. move\_base, sep 2020. URL [http://wiki.ros.org/move\\_base](http://wiki.ros.org/move_base). 19
- [35] J. S. Gill. Setup and configuration of the navigation stack on a robot, jul 2018. URL <http://wiki.ros.org/navigation/Tutorials/RobotSetup>. IX, 26
- [36] I. R. Herrero. Ros2 vs ros (1) – ¿migramos?, nov 2018. 28
- [37] C. Imes. Webcam, aug 2017. URL <https://help.ubuntu.com/community/Webcam>.

- [38] jjromeromarras. Conceptos básicos de ros, aug 2014. URL <https://jjromeromarras.wordpress.com/2014/08/27/conceptos-basicos-de-ros/>. 19
- [39] F. J. R. Lera. Diseño, construcción, y validación de una plataforma robótica asistencial, social y de servicios basada en realidad aumentada = design, fabrication, and validation of a socially assistive robot based on augmented reality. *Escuela de Ingenierías Industrial e Informática, Universidad de León*, 2015. URL <https://buleria.unileon.es/handle/10612/6012>. 10
- [40] M. Moon. Robear is a robot bear that can care for the elderly, feb 2015. URL [engadget.com/2015-02-26-robear-japan-caregiver.html?guccounter=1&guce\\_referrer=aHR0cHM6Ly93d3cuZ29vZ2x1LmNvbS8&guce\\_referrer\\_sig=AQAAAD9LqL6doNbIBirUuASjE1jXVQHaAjQVbz3VgfU4rp4oNN9Vwi2YFISbFWgJLQf4snhAMvYwatjLxbMdOHd9eX1Lh8SrHbL2s7G\\_IAq-6z0tiP4\\_aRT](http://engadget.com/2015-02-26-robear-japan-caregiver.html?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2x1LmNvbS8&guce_referrer_sig=AQAAAD9LqL6doNbIBirUuASjE1jXVQHaAjQVbz3VgfU4rp4oNN9Vwi2YFISbFWgJLQf4snhAMvYwatjLxbMdOHd9eX1Lh8SrHbL2s7G_IAq-6z0tiP4_aRT). IX, 1
- [41] M. Moore. ros-logo, mar 2019. URL <http://so.engineering/ros-logo-2/>. IX, 15
- [42] H. Oladepo. Nodes, apr 2018. URL <http://wiki.ros.org/Nodes>. 18
- [43] M. Quigley, B. Gerkey, and W. D. Smart. *Programming Robots with ROS*, volume 425. O'Reilly Media, dec 2015. ISBN 978-1-4493-2389-9. 15
- [44] J. E. Riva. Navegando el sistema de archivos de ros, aug 2021. URL <http://wiki.ros.org/es/ROS/Tutoriales/NavegandoElSistemaDeArchivos>. 17
- [45] J. E. Riva. Comprendiendo servicios (services) y parametros (parameters) ros, aug 2021. URL <http://wiki.ros.org/es/ROS/Tutoriales/ComprendiendoServiciosParametros>. 22, 24
- [46] J. E. Riva. Comprendiendo tópicos ros, aug 2021. URL <http://wiki.ros.org/es/ROS/Tutoriales/ComprendiendoTopicosROS>. 21
- [47] K. Wyrobek. The origin story of ros, the linux of robotics, oct 2017. URL <https://spectrum.ieee.org/automaton/robotics/robotics-software/the-origin-story-of-ros-the-linux-of-robotics>. 16

Universidad de Alcalá  
Escuela Politécnica Superior



ESCUELA POLITECNICA  
SUPERIOR

