UNIVERSIDAD DE ALCALÁ

ESCUELA POLITÉCNICA SUPERIOR

GRADO EN INGENIERÍA ELECTRÓNICA DE COMUNICACIONES

TRABAJO FIN DE GRADO

DISEÑO Y DESARROLLO DE MÓDULOS HARDWARE / SOFTWARE PARA UN SISTEMA AVANZADO ROBÓTICO DE ASISTENCIA A LA MOVILIDAD (SARA)

AUTOR: DAVID PINEDO ÁVILA

DIRECTOR/ES: MARTA MARRÓN ROMERA

JUAN CARLOS GARCÍA GARCÍA

TRIBUNAL:

PRESIDENTE: JOSÉ LUIS LÁZARO GALILEA

VOCAL 1º: MARÍA SOLEDAD ESCUDERO HERNANZ

VOCAL 2º: MARTA MARRÓN ROMERA

FECHA: .....

ÍNDICE

Índice de figuras

Resumen

En las últimas décadas, se han producido numerosos avances de la robótica móvil fácilmente extrapolables al ámbito de las tecnologías de apoyo, en concreto las sillas de ruedas inteligentes, pero que no siempre han sido aplicados.

Continuando los trabajos anteriormente realizados en el Departamento de Electrónica, en este TFG se plantea incorporar una de las tecnologías más usadas actualmente en el ámbito de la robótica móvil, ROS (*Robotic Operating System*), a la silla de ruedas SARA cuya estructura de control de bajo nivel fue diseñada en los trabajos previos. En este trabajo se ha logrado modelar la silla para después poder realizar simulaciones de navegación autónoma y leer los valores del bus CAN que comunican los nodos de control de bajo nivel de la silla en tiempo real con el de alto nivel (nodo PC) que realiza las tareas de navegación autónoma, en base a funciones ROS.

Palabras claves: ROS, SARA, *topic*, RVIZ.

Abstract

In recent decades, there have been numerous advances in robotics, to be easily extrapolated to the field of assistive technologies, specifically to intelligent wheelchairs, but they have not always been applied.

Continuing the work developed in the Electronics Department, in this TFG is poses to incorporate new software technology currently used in the field of mobile robotics, ROS (Robotic Operating System), to the wheelchair SARA, whose low-level control and structure was designed in previous works. In this work it has been developed a SARA model that allows to perform simulations of its autonomous navigation, and to read the values of the CAN bus. Thus it is now possible to communicate in real time the wheelchair low-level control nodes with the high-level one (PC), in order to perform the tasks of autonomous navigation based on ROS functions.

Resumen extendido

En este Trabajo Fin de Grado, se ha incorporado un *Framework de Desarrollo de Robótica* (RFS) llamado ROS (*Robotic Operating System*) en un sistema avanzado robótico de asistencia a la movilidad (SARA), diseñado en el Departamento de Electrónica durante los últimos años.

La Figura 1 muestra el diagrama general de SARA, que será necesario incluir en el modelo de comunicación de ROS, de modo que sea posible usar después las múltiples funciones de navegación autónoma sobre SARA.

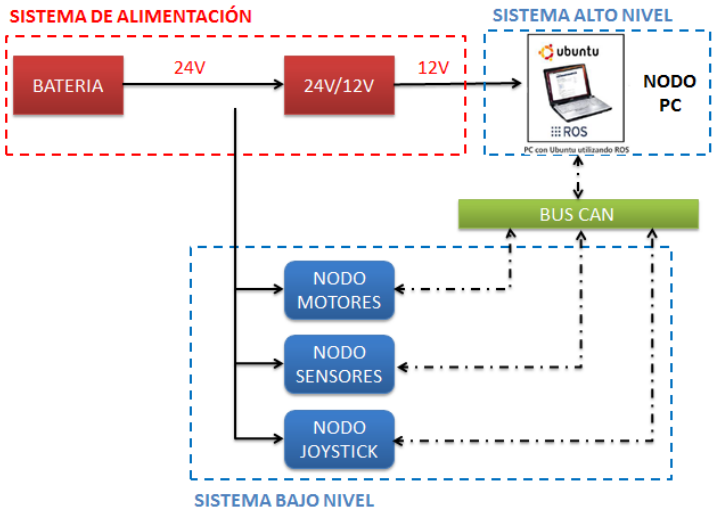

Figura 1 - Esquema general de SARA

ROS

Actualmente ROS es utilizado por una amplia comunidad de diseñadores de aplicaciones robóticas, ya que permite trabajar en diferentes lenguajes de programación (C++, Phyton o Lisp), está basado en una estructura distribuida y es un software de código abierto.

La finalidad de este proyecto, es incorporar ROS en la silla de ruedas SARA, de cara a que en un futuro se consiga implementar un sistema de navegación autónomo con el uso de las funciones de ROS.

Para poder entender los distintos módulos implementados en este Trabajo Fin de Grado, es necesario hablar sobre ROS. A parte de sus principales características, es necesario conocer los conceptos fundamentales para poder trabajar con esta plataforma, así como las herramientas que incluye.

Una vez conocido como funciona ROS, se modelará la silla de ruedas en esta plataforma, con el fin de poder visualizar a SARA tanto en simulaciones como en la navegación en tiempo real. Este modelo se creara mediante el modelo URDF, modelo basado en el lenguaje XML.

Con el modelo creado, se realizará una simulación de SARA en uno de los simuladores 2D más usados en ROS: STDR. En esta simulación, haciendo uso del teclado se podrá dirigir la silla por el mapa y visualizar los valores que leen los sensores de ultrasonidos, sin necesidad de disponer físicamente de la silla.

Hardware

SARA dispone de varios módulos (motores, joystick y sensores) que se comunican entre sí mediante el bus CAN. Los distintos nodos envían mensajes por este bus, cada uno con su propio identificador. Para poder comunicarse con la silla de ruedas, es necesario disponer de un adaptador USB-CAN e implementar un nodo en ROS que realice esta comunicación.

A nivel hardware, para el desarrollo de este TFG se usara todo el hardware de bajo nivel con lo que cuenta actualmente SARA, procedentes de anteriores proyectos realizados por el Departamento de Electrónica de la Universidad de Alcalá, centrándose este TFG en el rediseño de la estructura física con el fin de mejorar su estabilidad, la distribución de los sensores de ultrasonidos y la interfaz con el usuario.

El punto de partida hardware es, por tanto, una silla de ruedas eléctrica a la que le han sido añadidos distintos elementos y circuitería electrónica. Entre otros cabe destacar dos encoders incrementales acoplados a los ejes de los motores, una tarjeta de potencia para controlar cada uno de estos dos últimos, sensores de proximidad, acelerómetros, un ordenador y una pantalla táctil.

La silla de ruedas así construida dispone de varios nodos, comunicados entre sí mediante el bus CAN. Los nodos que actualmente tiene la silla son:

- Nodo joystick: Formado por un joystick analógico, un display, un potenciómetro, varios botones y un teclado.
- Nodo motores: Formado por los motores, encoders y una tarjeta de potencia.
- Nodo sensores: Formado por sensores de ultrasonidos colocados alrededor de la silla y un acelerómetro para detectar colisiones.

Resultados

El paquete correspondiente al modelo de SARA se llama “sara_urdf”, el cual genera y permite visualizar su modelo URDF. (Figura 2).

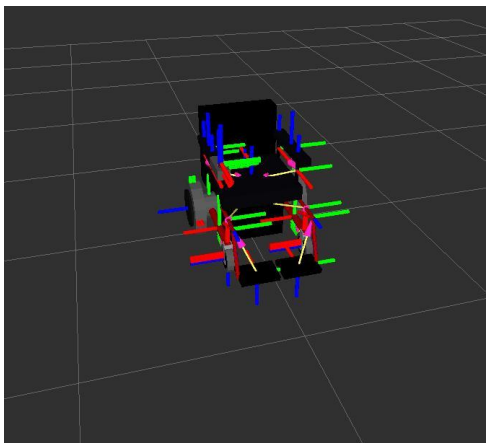

Figura 2 - Modelo URDF de la silla de ruedas

Para la simulación en ROS, se creará el paquete “sara_sim_stdr”. Ejecutándolo se podrá dirigir la silla por el entorno de simulación obteniendo una lectura de los sensores de ultrasonido que hay distribuidos por la silla. (Figura 3).

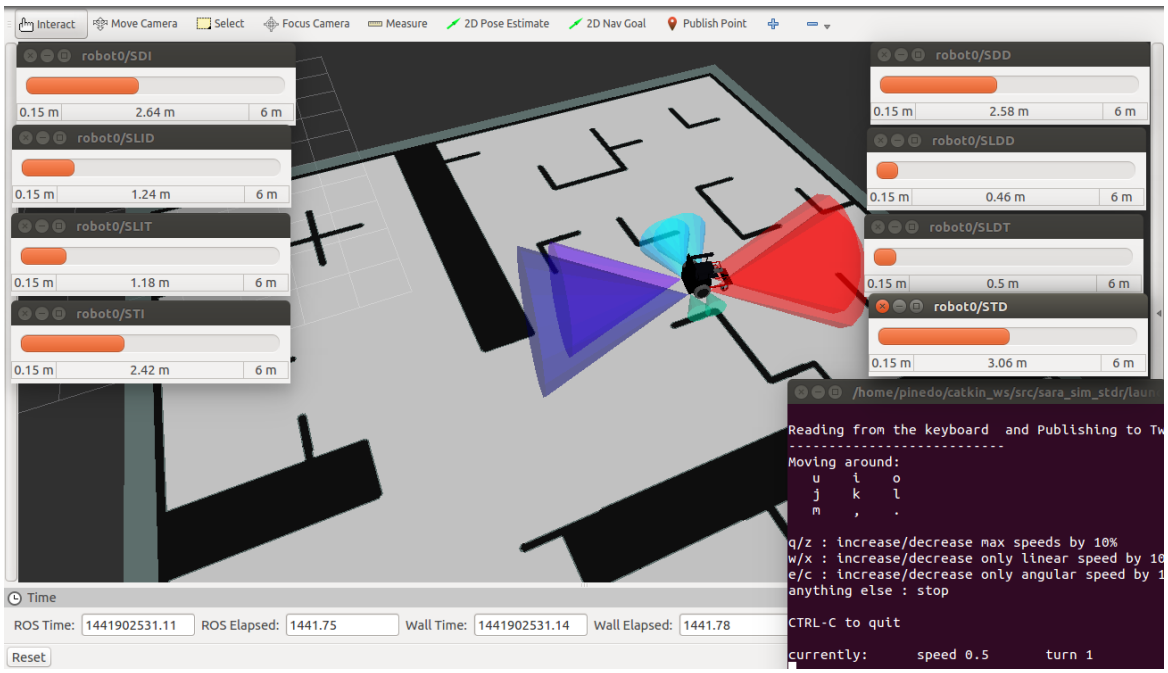

Figura 3 - Simulación de SARA en STDR

La comunicación entre el bus de comunicaciones CAN que usa la silla de ruedas y ROS se ha implementado en el paquete “canusb”. Este se encarga de leer todos los datos que circulan por el bus CAN y de publicarlo en sus correspondientes *topics* (Figura 4)

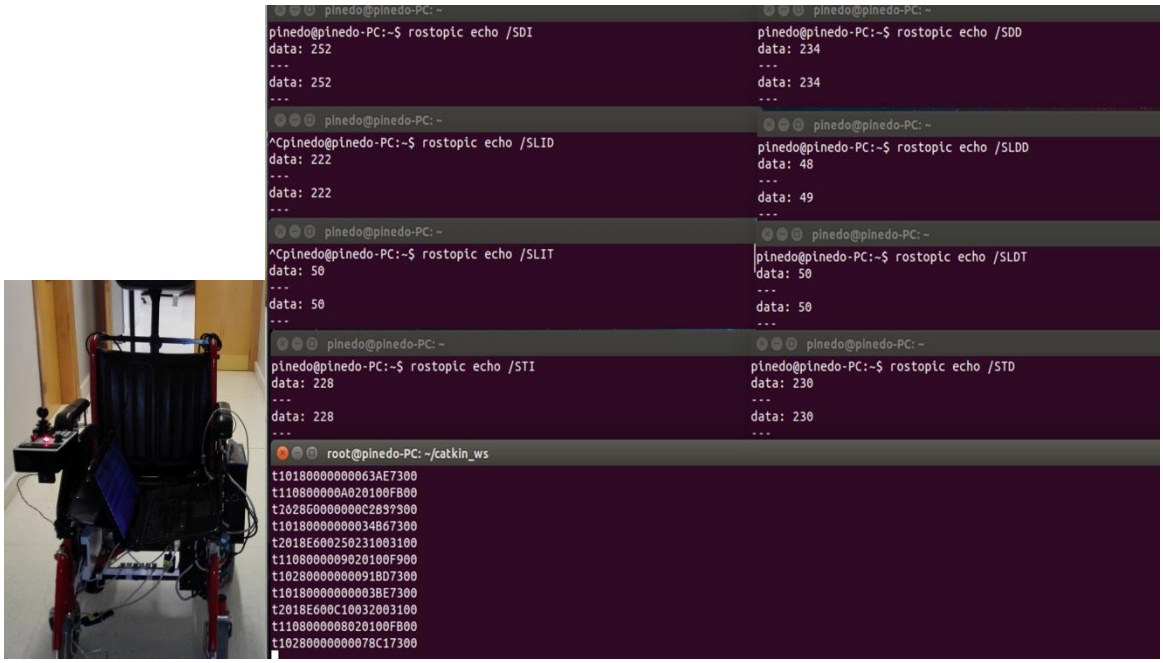

Figura 4 - Resultados comunicación CAN-ROS

Capítulo 1

Introducción

Este Trabajo Fin de Grado permite avanzar en el proyecto "Guiado de una silla de ruedas mediante joystick y soplido por bus CAN" de Juan Bellón Álvarez y Jose Basterrechea García, con el objetivo global de diseñar una silla de ruedas autónoma modulable. En este proyecto se buscaba crear un sistema modulable, de nodos independientes que se comunican entre sí mediante el bus CAN. Se consiguió crear una silla de ruedas avanzada (*Advanced Wheelchairs*, AW) capaz de controlarse mediante un joystick o un sensor de soplo.

En las últimas décadas se han producido múltiples avances en el ámbito de la robótica móvil, tanto en la parte software como en la hardware. En este TFG se busca aprovechar estos avances para convertir la silla de ruedas SARA (Sistema Avanzado Robótico de Asistencia de la movilidad) en una silla de ruedas inteligente. Dado al gran éxito de la plataforma software (framework) de robótica ROS (Robotic Operating System), se ha optado por incorporarlo en la silla de ruedas, ya que entre sus múltiples ventajas, incorpora algoritmos de código abierto para la navegación reactiva de robots, con multitud de ejemplos y aportaciones de comunidades científicas.

1.1. Objetivos

Desde el departamento de Electrónica de la Universidad de Alcalá, de busca hacer de la silla de ruedas SARA una silla de ruedas inteligente (*Smart Wheelchairs*, SW), capaz de navegar de manera autónoma con ayuda de múltiples sensores por un espacio cerrado.

El principal objetivo del trabajo realizado es incorporar un nuevo módulo a la silla de rueda que incorpore ROS, modelando la silla en esta plataforma, realizando simulaciones e implementando la comunicación Bus CAN – ROS.

1.2. Estructura de la memoria

La memoria está distribuida en seis capítulos. En el primero de los capítulos, el capítulo actual, se introduce el trabajo desarrollado, explicando las motivaciones que nos han llevado a realizarlo y los objetivos planteados.

En el Capítulo 2, se hace una introducción sobre las sillas de ruedas básicas (*Basic Wheelchairs*, WB), avanzadas (*Advanced Wheelchairs*) e inteligentes (*Smart Wheelchairs*, SW). También se describirá ROS, explicando sus principales características, herramientas y los conceptos fundamentales que hay que tener claros para trabajar con esta plataforma.

En el Capítulo 3, se expondrá cual es la estructura general del sistema, comentando cuales son los modulos que forman el sistema y como se comunican entre sí mediante el uso del protocolo CAN.

En el Capítulo 4 se explican los tres bloques principales desarrollados en este TFG: el modelado de la silla, la integración virtual del sistema y su integración física. En el primero de ellos, se hará una breve introducción sobre el lenguaje XML en el que se basa el modelo URDF, se explica en que consiste dicho modelo y como se ha creado. En el segundo bloque se explica cómo se ha realizado la simulación de la silla en el simulador STDR y por último, en el tercer bloque, se describe cómo se ha implementado la comunicación en tiempo real entre el Bus CAN de la silla de ruedas y ROS.

En el Capítulo 5 se muestran los resultados obtenidos en este trabajo, comentando los hitos alcanzados y justificándolos.

En el Capítulo 6 se exponen las conclusiones obtenidas al finalizar el trabajo y los posibles trabajos futuros.

Capítulo 2

Marco teórico

En la primer parte de este capítulo, se describen las sillas de ruedas inteligentes. Se comentan cuáles son los tipos de sillas de ruedas inteligentes, haciendo una descripción de cada una de ellas. Dado el bajo número de sillas de ruedas inteligentes que existen en el mercado, se hablará de los problemas existentes que provocan esto y de las posibles soluciones para minimizar estos problemas.

En la segunda parte, se describe la plataforma ROS la cual es la base de este Trabajo Fin de Grado. Se hablará de las principales características de esta plataforma, de las herramientas de simulación y visualización más usadas que incluye y de los conceptos básicos que hay que tener para poder empezar a trabajar con ROS.

2.1. Sillas de ruedas inteligentes (IW-*Intelligent Wheelchair*)

En las últimas décadas ha habido continuos y espectaculares avances en las áreas tecnológicas relacionadas con los sistemas de robótica móvil. Sensores, procesadores y actuadores cada vez mejores y más baratos, unidos a avanzadas estrategias de control, son los actores de estas mejoras. Gracias a estos avances se han venido proponiendo aplicaciones innovadoras a estos sistemas robóticos, algunas de ellas en el campo de las Tecnologías de Apoyo o de la Asistencia.

Ya a comienzos de los años 1980, ciertos trabajos de la Universidad de Stanford [3] proponen utilizar sensores ultrasónicos sobre una Silla de Ruedas adaptada. En los años 1990, otros grupos de investigación alrededor del mundo comenzaron a desarrollar vehículos de Asistencia a la Movilidad innovadores ([4] [5] y [6]). En su concepto inicial, estos vehículos eran similares a cualquier robot móvil estándar: ruedas, motores, baterías y controladores. El propósito de estos trabajos era lograr un vehículo autónomo al que podemos denominar Silla Inteligente (IW - *Intelligent Wheelchair*, Figura 5).

Los términos más usados para describir a una Silla de Ruedas son: *Smart* (SW), *Intelligent* (IW), *Advanced* (AW), *Autonomous* (AuW), *Robotic* (RW). Siguiendo la línea del trabajo descrito en [7], en el que se propone un modelo modular de una silla de ruedas, se entiende por silla básica (BW) una silla construida únicamente por motores y un joystick. Si a BW se le añaden módulos tanto software como hardware que proporcionen a la silla capacidad de comunicación, integración de sensores y conducción asistida entre otras características, se puede hablar de sillas avanzadas (AW). Añadiendo módulos de alto nivel que proporcionen autonomía a AW, como sensores de visión, sensores 3D, conducción autónoma y navegación entre otras, se habla de silla de ruedas inteligente (IW).

Figura 5 - Modelo modular de la silla de ruedas inteligente

La investigación en Sillas de ruedas Inteligentes ha sido un tema de investigación activa en las Tecnologías de Apoyo desde los años 1980. Pero tras más de 30 años de investigaciones casi ninguno de estos prototipos ha llegado a los potenciales usuarios. Como principales motivos destacan la falta de sensores adecuados (en términos de coste y fiabilidad) y la falta de una plataforma estándar en el mercado sobre la que construir una IW. Para detectar obstáculos y reconocer el entorno se requieren sensores muy fiables. La faltan de sensores con la suficiente fiabilidad para todas las posibles situaciones de peligro, mantiene a las IW lejos del mercado por las implicaciones legales resultantes. Actualmente existen sensores modernos con una relación coste/precio que pueden reducir este problema.

SARA se creó usando elementos Electrónicos y de Comunicaciones estándar (USB, CAN, Wifi, etc...) con el objetivo de garantizar su compatibilidad con otros sistemas y servicios presentes en el entorno y facilitar su adaptación a diversos perfiles de usuarios.

2.2. ROS

ROS es una plataforma software que incluye una serie de librerías y herramientas de código abierto que ayuda a los desarrolladores de software a crear aplicaciones para robots. Fue desarrollado originalmente en 2007 bajo el nombre de *switchyard* por el laboratorio de inteligencia artificial de Stanford para dar soporte al proyecto del robot con inteligencia artificial de Stanford (STAIR)[8]. Desde 2008 continua su desarrollo primordialmente en Willow Garaje y en febrero de 2013 se transfiere a la *Open Source Robotics Foundation* (OSRF).

Se desarrolló con la intención de crear un *framework* que permita la integración de la robótica en una gran variedad de situaciones. Además, el hecho de que ROS haya sido

creado totalmente bajo la licencia de código abierto BSD (*Berkeley Software Distribution*) ha ayudado a que actualmente ROS sea uno de los *frameworks* más usados en la comunidad robótica.

2.2.1. Características principales

Las principales características de ROS son las que se listan a continuación:

- **"Peer to Peer".** ROS está basado en un sistema distribuido en el que los diferentes procesos (que pueden estar localizados en distintas máquinas o hosts) se comunican entre sí utilizando una topología "peer to peer". Con esta topología se utiliza un canal nuevo para la comunicación entre dos procesos distintos, evitando tener que usar un servidor central para comunicar todos los procesos.
- **Multilenguaje**. ROS permite trabajar en diferentes lenguajes de programación, como C++, Python y Lisp. También disponen de bibliotecas en Java y Lua, en fase experimental.
- **Basado en herramientas**. ROS proporciona una gran cantidad de herramientas, las cuales permiten realizar diferentes tareas, desde navegar por los ficheros, obtener y modificar los parámetros de configuración de un robot, proceso o driver, visualizar la topología de los procesos en ejecución o realizar la comunicación entre procesos.
- **Código libre y abierto**. ROS es código libre bajo términos de licencia BSD, incluyendo libertad para uso comercial e investigador. El código completo de ROS está disponible al público.

2.2.2. Conceptos fundamentales

Para poder comprender el funcionamiento de ROS, en este apartado se van a comentar los conceptos fundamentales de esta plataforma, a saber: nodos, ROS Master, mensajes, *topics*, servidor de parámetros, servicios y paquetes.

2.2.2.1. Nodos

Son ejecutables que se comunican con otros procesos usando *topics* o servicios. El uso de nodos en ROS proporciona tolerancia a fallos y separa el código del sistema haciéndolo más simple. Un paquete puede contener varios nodos (cada nodo dispone de un nombre único), cada uno de los cuales lleva a cabo una determinada acción.

2.2.2.2. "ROS Master"

Proporciona un registro de los nodos que se están ejecutando y permite la comunicación entre nodos. Sin el maestro los nodos no serían capaces de encontrar al resto de nodos, intercambiar mensajes o invocar servicios.

2.2.2.3. Mensajes

Los nodos se comunican entre sí mediante el paso de mensajes. Los tipos primitivos de mensajes ("integer", "floating point", "boolean", etc.) están soportados y se pueden crear tipos personalizados.

En la Tabla 1 se puede ver un ejemplo de tipos de mensajes que se usan en ROS:

TIPO PRIMITIVO	DECLARACIÓN TIPO	C++	PYTHON
uint16	Entero 16 bits sin signo	uint16_t	Int
float32	Flotante de 32 bits	float	float
String	Cadena de caracteres	Std::string	String

Tabla 1- Tipos de mensaje

2.2.2.4. Topics

Son canales de comunicación para transmitir datos. Los *topics* pueden ser transmitidos sin una comunicación directa entre nodos, significa que la producción y el consumo de datos esta desacoplada. Los nodos pueden comunicarse entre sí mediante *topics*, pudiendo actuar como publicadores (*publisher*) y como suscriptores (*subscriber*).

- **Publicador**. El nodo que actúa como publicador es el encargado de crear el *topic* por el cual va a difundir ciertos mensajes. Estos mensajes podrán ser visibles por los nodos que estén suscritos a este *topic*.
- **Suscriptor**. Este nodo se deberá suscribir a los *topics* correspondientes para poder acceder a los mensajes que hay publicados en ellos.

Un nodo puede publicar o suscribirse a un mensaje a través de un *topic*. En la Figura 6 se puede ver un ejemplo de comunicación entre dos nodos por medio de un *topic*. En este ejemplo, el nodo "*publisher* odometry" publica en el *topic* "Odom" un mensaje del tipo "nav_msgs/odometry" y el nodo "Base_movil" accede a este mensaje suscribiéndose a este *topic*.

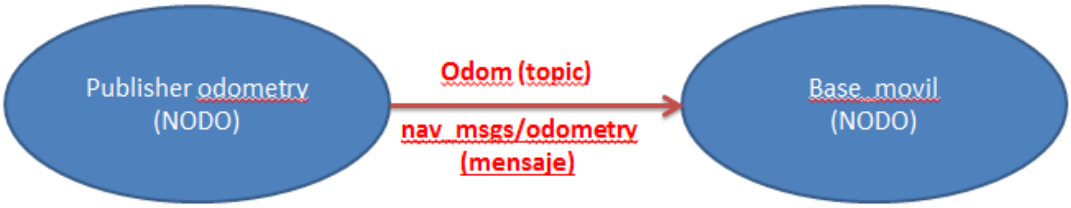

Figura 6 - Comunicación por topics

2.2.2.5. Servidor de parámetros

Es un diccionario nombre-valor compartido que puede ser accedido en toda la red de ROS. Los nodos pueden utilizar este servidor para guardar u obtener parámetros en tiempo de ejecución. Está pensado para guardar datos estáticos y no demasiado complejos, por ejemplo parámetros de configuración.

2.2.2.6. Servicios

Cuando se necesita recibir una respuesta a una comunicación con un nodo, no se puede realizar con *topics*, sino que se necesita hacerlo mediante servicios. En este sistema de comunicación, se dispone de un nodo que actúa como servidor y otro como cliente, los cuales se comunicaran entre sí por medio de mensajes de solicitud/respuesta (ver Figura 7).

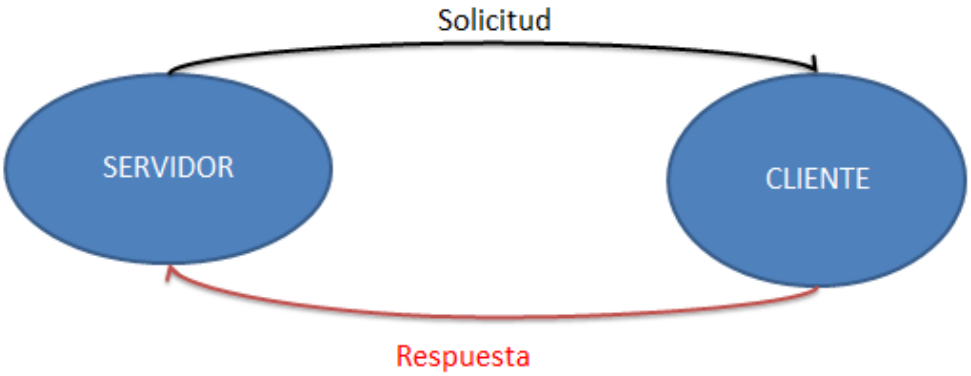

Figura 7 – Comunicación por servicios

2.2.2.7. Paquetes

Los paquetes son la unidad principal de organización en ROS. Un paquete permite agrupar una serie de aplicaciones (nodos), servicios o bibliotecas, facilitando su portabilidad e instalación en diferentes sistemas. La estructura de los paquetes se puede ver en la Figura 8.

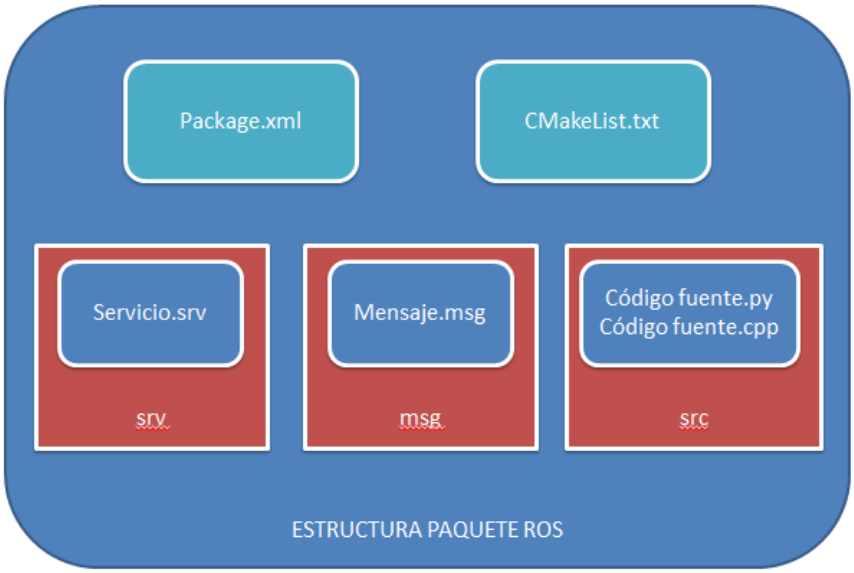

Figura 8 - Estructura paquetes ROS

A continuación, se explicara con más detalle todos los elementos que forman un paquete de ROS:

- **Package.xml.** Fichero utilizado para añadir dependencias e información acerca del paquete.
- **CMakeList.txt.** Fichero que describe cómo se construye el paquete y donde se instala.
- **Directorio src**. Dentro de este directorio se almacenan los códigos fuente del paquete. Estos definen las aplicaciones (nodos) que forman el paquete y pueden estar escritos en diversos lenguajes de programación, p.ej. en C++ (extensión .cpp) o en Python (extensión .py).

- **Directorio srv**. En este directorio se almacenan los servicios que, como se ha comentado, son arquitecturas de solicitud o de respuesta encargadas de realizar la comunicación entre nodos.
- **Directorio msg**. En este directorio se almacenan los ficheros que definen la estructura de los mensajes que los nodos utilizaran para comunicarse.

2.2.3. Herramientas de visualización y simulación.

ROS ofrece un poderoso conjunto de herramientas de desarrollo que permiten obtener información sobre el estado del sistema en tiempo de ejecución.

ROS ofrece herramientas de visualización, que permiten visualizar el intercambio de información entre nodos, ya sea sobre la línea de comandos o sobre entornos 3D. ROS también ofrece herramientas de simulación que permiten probar diferentes algoritmos y comportamientos sin necesidad del robot físico. A continuación, se explicarán las herramientas más usadas en ROS.

2.2.3.1. RVIZ

RVIZ es una de las herramientas más importantes de ROS. Permite visualizar en un entorno 3D la información de los *topics*. Mediante RVIZ se pueden visualizar múltiples tipos de datos, como por ejemplo, imágenes, nubes de puntos, sensores laser, sensores ultrasonidos, odometría, transformadas, mapas, modelos URDF del robot, etc. (ver Figura 9).

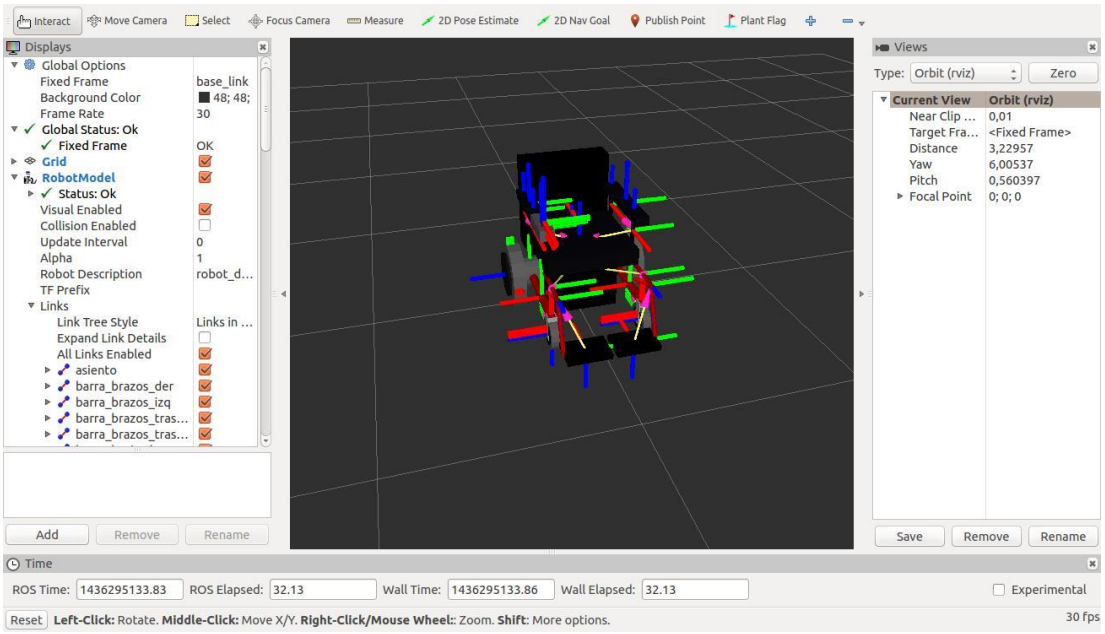

Figura 9 - Herramienta RVIZ

En la ventana llamada “displays” se puede añadir la información que queramos visualizar, siempre y cuando se estén publicando datos del tipo que queremos visualizar en un *topic*.

2.2.3.2. ROS GUI

ROS proporciona una interfaz gráfica de usuario (GUI) utilizando el marco de Qt, un sistema que permite a los usuarios interactuar con el entorno ROS de una manera visual (Figura 10). Está basada en una arquitectura de *plugins*, lo que permite no solo desarrollar nuestras propias herramientas sino también reutilizar las de otros.

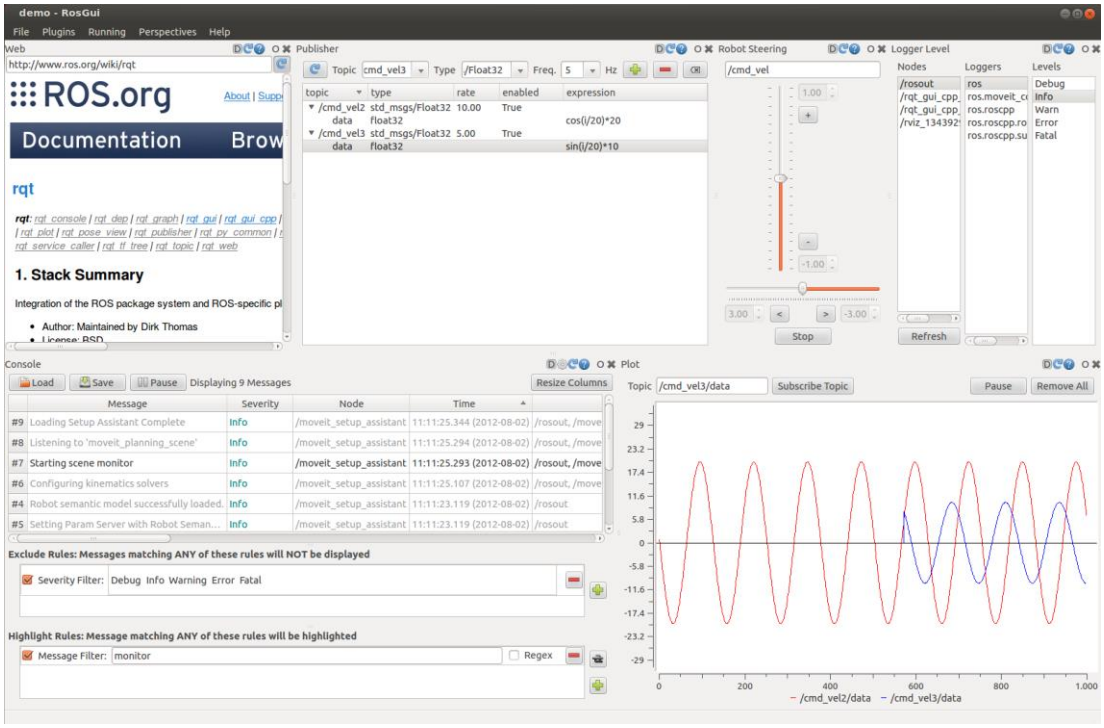

Figura 10 - Interfaz de ROS GUI

A continuación se listan los *plugins* más usados en la interfaz gráfica de usuario de ROS:

- *rqt_graph*. *Plugin* (Figura 11) que muestra los nodos y *topics* que se están ejecutando en tiempo real, lo que permite fácilmente depurar el sistema.

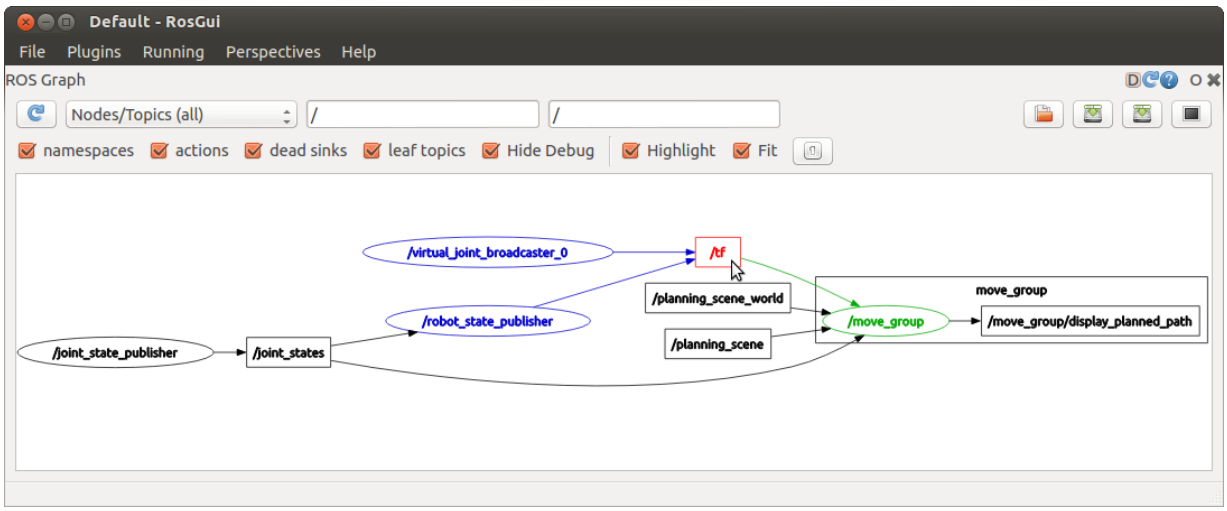

Figura 11 - Ejemplo rqt_graph

- *rqt_plot. Plugin* (Figura 12) que monitoriza diversas variables en función del tiempo de ejecución en una ventana 2D, utilizando diferentes trazados para cada variable.

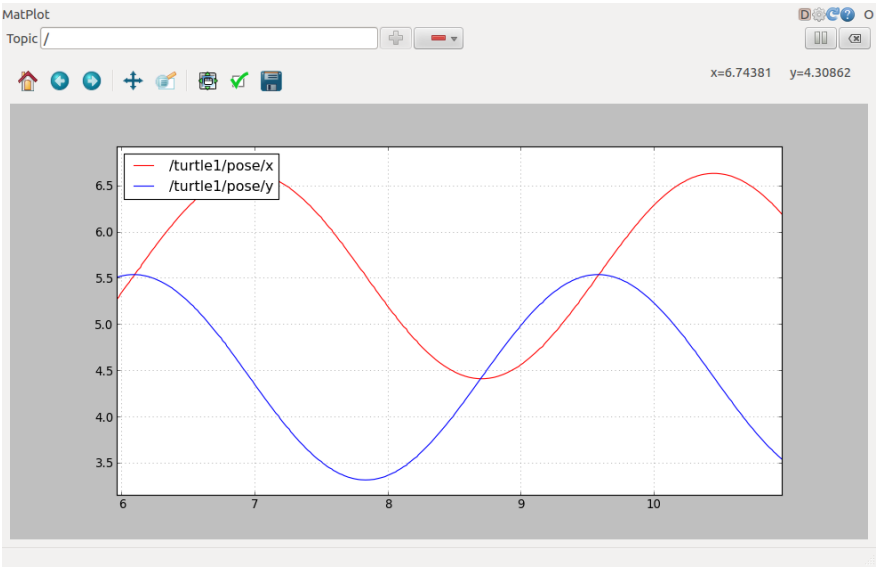

Figura 12 - Ejemplo de rqt_plot

- *rqt_topic. Plugin* que monitoriza información de depuración sobre *topics* de ROS, incluyendo publicadores, subscriptores, tasa de publicación y mensajes sobre estos topics.
- *rqt_bag. Plugin* (Figura 13) que permite visualizar y reproducir archivos del tipo bag de ROS. Los archivos del tipo bag almacenan mensajes grabados en función del tiempo.

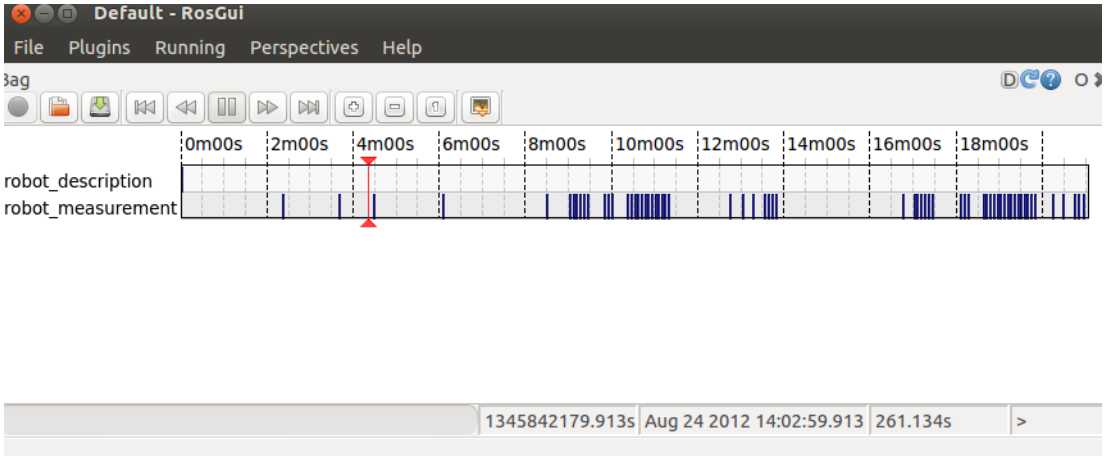

Figura 13 - Ejemplo de rqt_bag

2.2.3.3. STDR

STDR (*Simple Two Dimentional Robot*) es un simulador 2D multi-robot, fácil de usar, flexible y escalable para su uso dentro de ROS ([10]). STDR se creó con dos objetivos:

1) No se buscó que fuese el simulador más realista o el que tiene mayor número de funcionalidades, se buscó poder realizar una simulación tan simple como sea posible,

reduciendo al mínimo las acciones que el usuario tiene que realizar para iniciar su experimento. STDR puede funcionar con o sin un entorno gráfico.

2) Ser compatible con ROS. Cada robot y sus sensores emiten transformaciones TF y todas las medidas son publicadas en *topics* de ROS. De esta manera STDR usa todas las ventajas de ROS.

La GUI del simulador STDR se puede ver en la Figura 14.

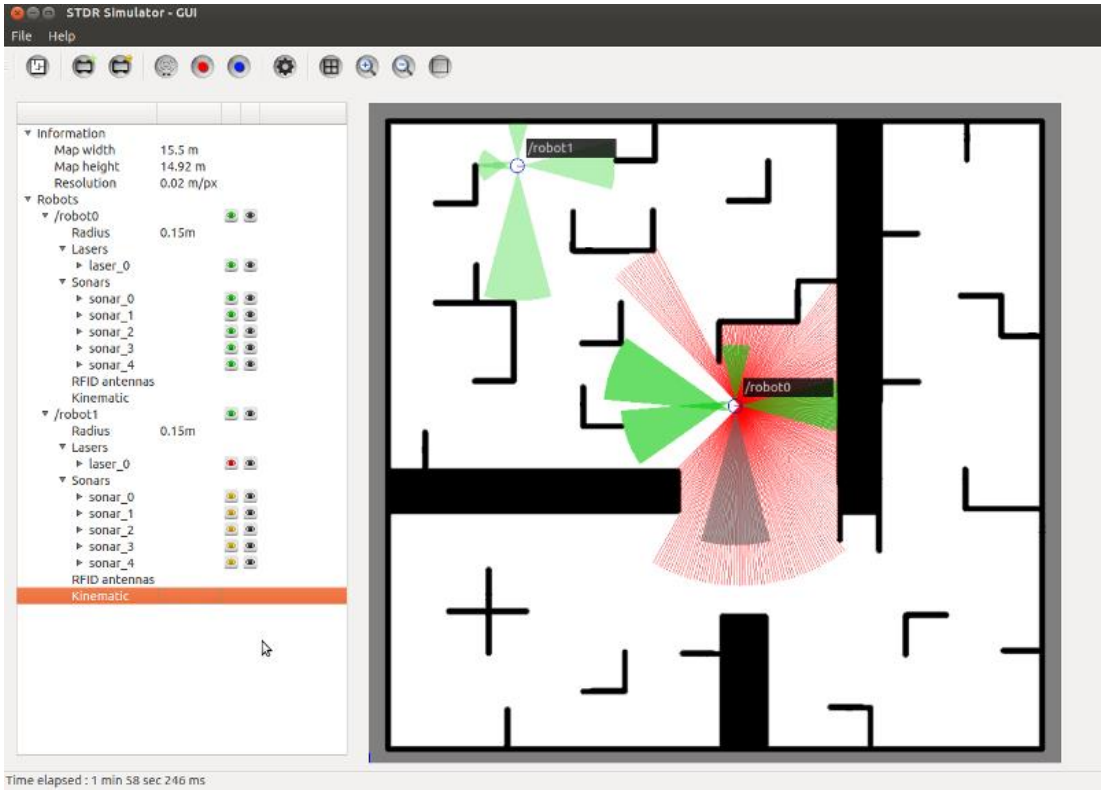

Figura 14 - GUI STDR

La GUI de STDR te permite crear robots circulares con las dimensiones, tipo y número de sensores deseados, siendo totalmente configurables sus parámetros característicos. Una vez creado permite exportar la configuración del robot en un archivo *".yaml"*. También permite cargar mapas del entorno de una forma muy rápida y sencilla.

El principal inconveniente del simulador STDR es que es un poco inestable, ya que está en fase de desarrollo. Aun así, se perfila como el simulador 2D más utilizado en ROS.

2.2.3.4. STAGE

STAGE ofrece una interfaz gráfica en dos dimensiones, que imita al mundo real gracias al uso de una imagen junto con un entorno de simulación definido por el usuario en un archivo ".world". Una de las ventajas de STAGE es que este archivo es bastante sencillo de realizar. En el archivo ".world" se debe especificar:

- Los modelos de los robots que se usarán en la simulación.
- Los dispositivos/sensores que los robots van a poseer.
- La ventana de interfaz gráfica.

- La imagen que servirá como mapa del entorno.

La interfaz gráfica de STAGE se puede ver en la Figura 15.

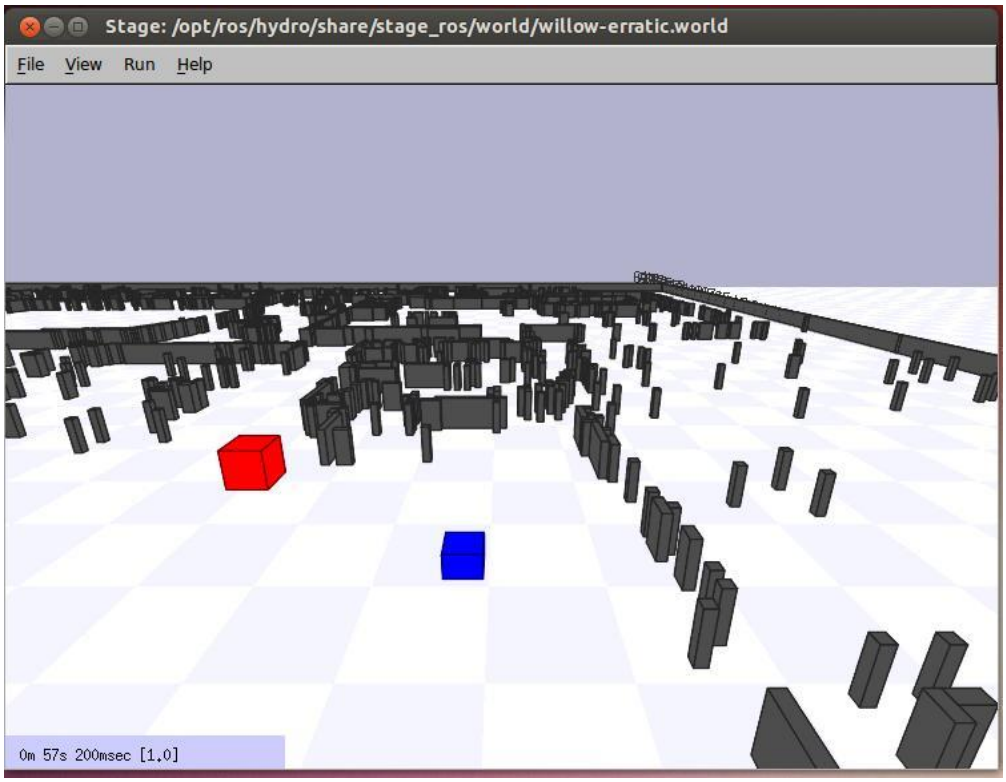

Figura 15 - STAGE

2.2.3.5. GAZEBO

GAZEBO es un simulador 3D multi-robot, con propiedades dinámicas y cinemáticas completas. La integración entre ROS y GAZEBO es proporcionada por un conjunto de *plugins* de GAZEBO que apoyan muchos robots y sensores existentes. Debido a que los *plugins* presentan la misma interfaz de mensaje que el resto del ecosistema ROS, se pueden escribir nodos ROS que son compatibles con la simulación, los datos registrados y el hardware de los robots.

Para mostrar al usuario el entorno de simulación, GAZEBO utiliza el motor gráfico OGRE [11]. Con este software, GAZEBO ofrece una reproducción realista del entorno a simular, usando iluminación de alta calidad, sombras y diferentes texturas. GAZEBO también ofrece compatibilidad con los estándar URDF y SDF, que utiliza el formato de etiquetas XML para crear los objetos, las físicas de simulación y los robots.

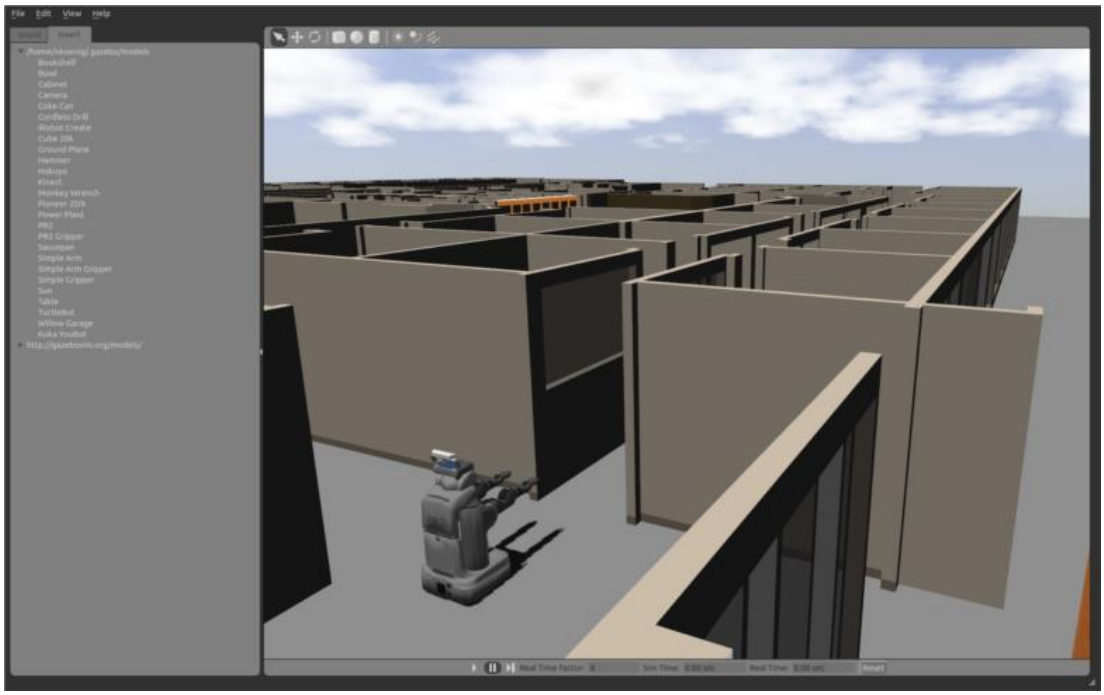

Figura 16 - Ejemplo de simulación en GAZEBO

Capítulo 3

Arquitectura general del sistema

En la Figura 17 se muestran el conjunto de elementos que forman el sistema SARA sobre el que se ha trabajado en este Trabajo Fin de Grado.

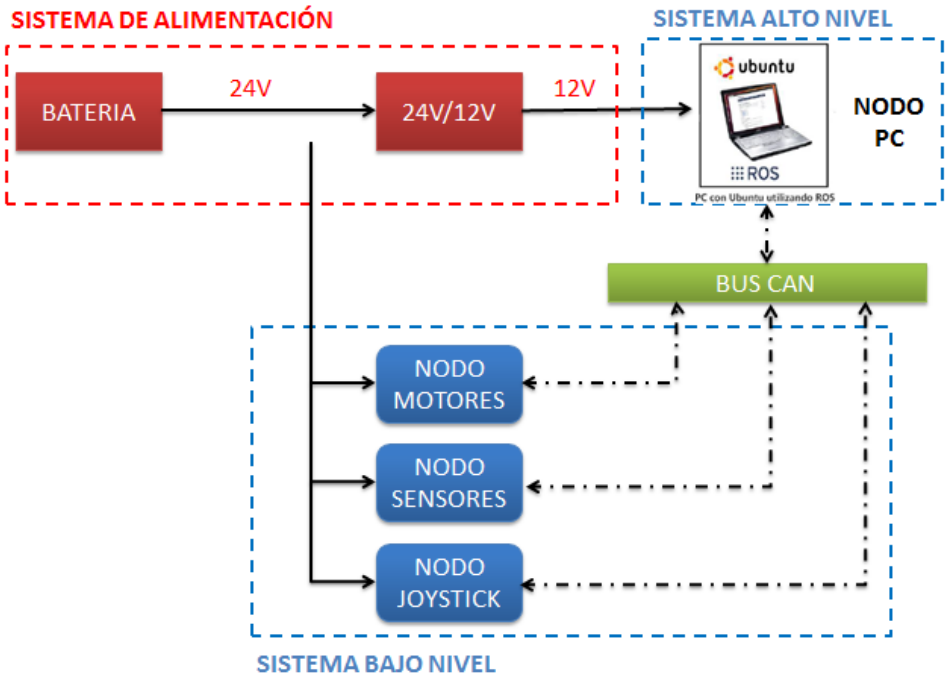

Figura 17 - Esquema general del sistema

Como se puede ver en la imagen anterior, el sistema se basa en una red distribuida dividida en los siguientes tres niveles:

- **Sistema de alimentación**. Se encarga de proporcionar alimentación a los distintos módulos del sistema. Para ello, todos los módulos disponen de fuentes conmutadas

que transforman estos 24V provenientes de las baterías en los 5V necesarios para alimentarlos. Además, mediante un convertidor DC/DC se transforman los 24V de la batería en 12V que se emplean para alimentar otros equipos (pantalla, PC…).

- **Sistema de alto nivel**. En este nivel encontramos el módulo PC.
- **Sistema de bajo nivel**. Formado por los módulos sensores, motores y joystick.

Los distintos módulos que forman el sistema se comunican entre sí por medio de un bus CAN, como se explica a continuación.

3.1 Módulos del sistema.

Como se ha comentado anteriormente, la silla de ruedas (SARA), está formado por cuatro módulos. Tres de estos módulos (sensores, motores y joystick), se implementaron en los trabajos de [1] y [2], el módulo PC es el que se implementa en este TFG.

Cada uno de los módulos joystick, sensores y motores disponen de su propia tarjeta LPC2129 de *Embedded Artists*. Estas tarjetas se usaron en proyectos anteriores con esta silla, ya que tenían un tamaño reducido y disponían de bus de datos CAN, principal medio de comunicación entre los elementos del sistema.

En la Figura 18 se puede ver el diseño de la tarjeta LPC2129 y en la Tabla 2 sus principales características.

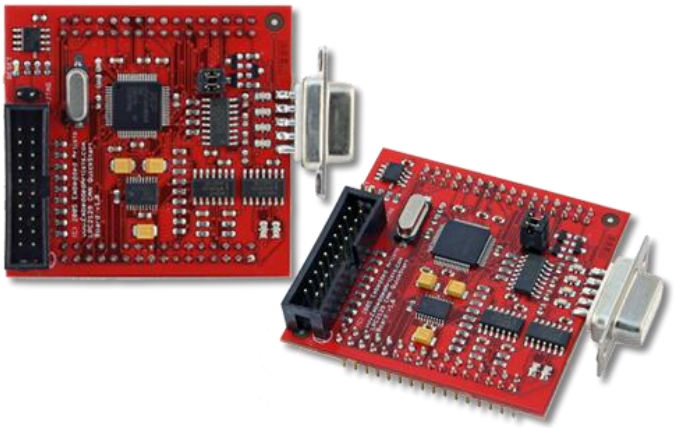

Figura 18 - Tarjeta LPC2129

Procesador	Microcontrolador NXP's ARM7TDMI LPC2129
Memoria Flash	256Mb
Memoria datos	16Kb
CAN	Dos canales CAN con transmisor TJA1040 o TJA1041
Reloj	12Mhz para máxima velocidad de ejecución y velocidades estándar de CAN
Dimensiones	55 x 58 mm
Alimentación	5V DC
Conectores	2 x 16 puertos de entrada/salida, RS232, DSUB-9

Otros	256Kb de memoria E2PROM para I2C, descarga de programas simples y automática por canal serie, 4 capas de PCB para una mejor inmunidad al ruido, fácil de conectar a señales JTAG.

Tabla 2 – Características LPC2129

3.1.1. Módulos sensores

Este módulo es el encargado de leer los datos de los sensores de ultrasonidos y del acelerómetro que incorpora la silla de ruedas para detectar choques.

Para este proyecto, se usaron los sensores de ultrasonidos SRF02 (Figura 19) y no se emplearon los acelerómetros.

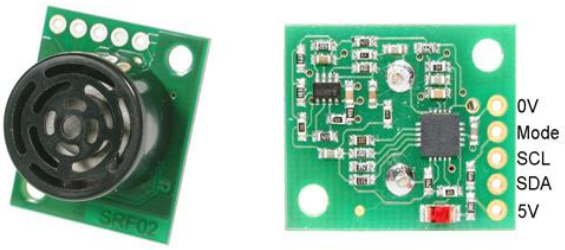

Figura 19 - SRF02

Las principales características de este sensor de ultrasonidos se pueden ver en la Tabla 3.

INTERFACES	**Dos modos regulables por el pin MODO: UART o por bus I2C.**
RANGO DE MEDIDAS	15 a 600 cm
UNIDADES	Centímetros, useg y pulgadas.
ALIMENTACIÓN	5V DC
CONSUMO MEDIO	4mA
DIMENSIONES	24x20x17 mm
PESO	4.6g

Tabla 3- Principales características SRF02

La comunicación entre el microcontrolador (LPC2129) y los sensores de ultrasonidos se realiza mediante el bus I2C, como se muestra en la Figura 20.

3.1.2. Módulos motores

Este módulo está formado por los encoders ópticos HEDS-5500A11, los motores y la tarjeta de potencia AX3500. Se encarga del movimiento de los motores de la silla de ruedas.

Los módulos motores llevan, cada uno, un controlador de bajo nivel integrado, que se encarga, por un lado de hacer de *bridge* de comunicaciones entre el actuador y la red CAN, y por otro de controlar que la consigna de velocidad de rotación de cada rueda se siga con el necesario perfil de aceleración y control de freno [1].

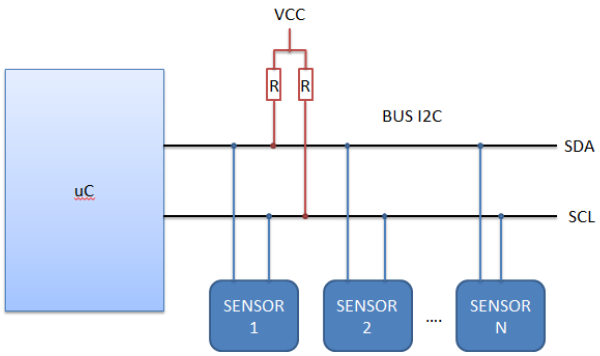

Figura 20 - Comunicación I2C

3.1.3. Módulo joystick

Este módulo proporciona la interfaz hombre-máquina a la silla de ruedas. Está formado por los siguientes elementos:

- Display
- Teclado
- Bocina (Botón negro)
- Luz de encendido
- Joystick analógico de 2 ejes
- Potenciómetro
- Extensión del conector J-Tag

Tal y como se puede ver en la Figura 21, la silla consta de cuatro modos de trabajo:

- Modo joystick. En este modo la silla de ruedas se mueve a través del joystick. Este es el modo que se ha usado en este proyecto.
- Modo soplido. En este modo la silla de ruedas se mueve usando un detector de soplo.
- Modo PC: Este modo fue implementado en los proyectos de [1] y [2], actualmente está en desuso.
- Modo joystick digital. En este modo se controla la silla de ruedas a través del joystick pero de manera digital, es decir, solo hace falta un toque en el joystick para que la silla se mueva hacía una determinada posición.

3.1.4. Módulo PC

Este módulo está formado por un ordenador que trabaja con sistema operativo Linux, con distribución Ubuntu. En este PC se desarrollara la plataforma de navegación basada en ROS que se ha desarrollado en este Trabajo Fin de Grado y que se describe más adelante en esta memoria.

Figura 21 - Modo joystick

3.2 Comunicación entre módulos: Protocolo CAN

Como se ha comentado antes, la comunicación entre los distintos módulos de SARA se realiza mediante el bus CAN. El bus serie CAN, siglas cuyo significado en castellano es "Control de Área de Red", nació en febrero de 1986, cuando el grupo "Robert Bosch GmbH" (más conocido como "Bosch") lo presentó en el congreso de la "Sociedad de Ingeniería de la Automoción". Desde entonces, CAN se ha convertido en uno de los protocolos líderes en la utilización del bus serie.

CAN fue desarrollado inicialmente para aplicaciones en los automóviles y por lo tanto la plataforma del protocolo es resultado de las necesidades existentes en el área de la automoción.

3.2.1. Principales características del bus CAN

Las principales características del bus CAN, que lo hacen el bus distribuido más adecuado para esta aplicación, se listan a continuación [12]:

- Un único bus de transmisión/recepción (half-duplex).
- Velocidad de comunicación de hasta 1 Mbit/s. La velocidad de transmisión puede ser diferente en distintos sistemas.
- Se emplean dos cables por los cuales viajan sendas señales exactamente iguales en amplitud y frecuencia pero completamente inversas en voltaje, generando valores de bus diferenciales CANH-CANL.
- Valores del Bus (dominante o recesivo). Los nodos conectados al bus interpretan dos niveles lógicos denominados:

  - Dominante: la tensión diferencial (CANH – CANL) es del orden de 2V.
  - Recesivo: la tensión diferencial (CANH – CANL) es del orden de 0V.
- Los mensajes CAN poseen un formato fijo que puede tener diferentes longitudes según el tipo de trama.
- Cada mensaje tiene un identificador de mensaje, que debe ser único en toda la red. El contenido de un mensaje se especifica por dicho identificador, que no indica el destino sino que describe el significado del mensaje. Además indica la prioridad del mensaje, lo cual es importante en el momento en el que varios nodos compiten por el acceso al bus.
- La petición de datos remotos se realiza enviando primero una trama remota de petición con un determinado identificador y esta es contestada con otra definida con el mismo identificador.
- Posibilidad de modificar la configuración en cualquier momento.
- Multimaestro:
  - Si el bus está libre, cualquier nodo puede comenzar a transmitir un mensaje.
  - Cuando dos nodos comienzan a transmitir simultáneamente el conflicto de acceso al bus es resuelto por arbitraje utilizando el identificador que determina que mensaje es más prioritario.
  - Cuando una trama de datos y una trama remota se inician al mismo tiempo, prevalece la primera.
- Detección de error y señalización. En todos los nodos CAN se implementan medidas especiales para la detección de errores, señalización y auto-chequeo.

Las implementaciones hardware de CAN cubren de forma estandarizada las capas físicas y de enlace del modelo de comunicaciones OSI (Open Systems Interconection), mientras diversas soluciones de software no estandarizadas cubren también la capa de aplicación.

3.2.2. Funcionamiento bus CAN

El protocolo CAN está orientado hacía el mensaje y no al destinatario. La información en la línea es trasmitida en forma de mensajes estructurados en los que una parte es un identificador que indica la clase de dato que contienen. Cuando las unidades de control reciben el mensaje, verifican el identificador para determinar si el mensaje va a ser utilizado por ellas. Las unidades de mando que necesiten los datos del mensaje lo procesan, y si no lo necesitan, el mensaje es ignorado.

El protocolo dispone de mecanismos para detectar errores en la trasmisión de mensajes. Esto hace que las probabilidades de error en la emisión y recepción de mensajes sean muy bajas, por lo que es un sistema extraordinariamente seguro. Cuando el bus está libre cualquier unidad conectada puede empezar a trasmitir mensaje.

3.2.3. Estructura de las tramas

Las tramas de los mensajes CAN tienen la estructura mostrada en la Figura 22 y cuyos campos se describen a continuación:

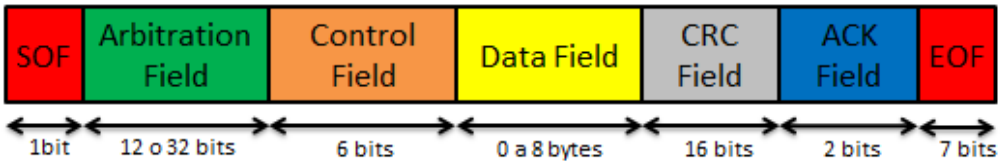

Figura 22 - Estructura trama bus CAN

- **SOF**. Marca el inicio de la trama.
- **Arbitration Field**. Identificador de la trama. En la trama estándar tiene un tamaño de 12 bits y en la trama extendida 32 bits.
- **Control field**. Campo de control. Los dos primeros bits están reservados y los 4 restantes indican el número de bytes de datos.
- **Data Field**. Campo de datos.
- **CRC**. Código de redundancia cíclica. Con este código se comprueba si hay o no errores.
- **ACK**. Celda de reconocimiento. Indica si el mensaje ha sido recibido correctamente.
- **EOF**. Marca el fin de trama.

En nuestro sistema, el empaquetamiento de datos es manual, es decir, se guardan en el mensaje varias variables concatenadas. El contenido de los mensajes se ve en la Tabla 4.

EMISOR	IDENTIFICADOR	MENSAJE	TIPO	TAMAÑO
JOYSTICK	0x110 (272)	Joyx	short	2 bytes
		velI	char	1 byte
		velD	char	1 byte
		Joyy	char	2 bytes
		modo	short	2 bytes
	0x111 (273)	modo(para PC)	short	4 bytes
		----	VACIO	4 bytes
MOTORES	0X101 (257)	encoderAABS	int	4 bytes
		tAabs	int	4 bytes
	0x102 (258)	encoderBABS	int	4 bytes
		tBabs	int	4 bytes
	0x210 (528)	nivel bateria	int	4 bytes
		----	VACIO	4 bytes
SENSORES	0x201 (513)	SDI	int	4 bytes
		STD	int	4 bytes
		SLDD	int	4 bytes
		SLIT	int	4 bytes
	0x202 (514)	SLID	int	4 bytes
		SDD	int	4 bytes
		SLDT	int	4 bytes
		STI	int	4 bytes
	0x203 (515)	EjeZ	short	4 bytes
		SJO	int	4 bytes

		EjeX	short	4 bytes
		EjeY	short	4 bytes
PC	0x120 (288)	DatoD		
		DatoI		
		Lazo		

Tabla 4 – Formato de las tramas del bus CAN en la silla

Como se observa en la Tabla 4, por el Bus CAN se envían los datos que se listan a continuación:

- Valores digitales del ejex y el ejey del joystick (Joyx, Joyy).
- Valores de velocidad de las ruedas (velI, velD).
- El modo de funcionamiento de la silla (modo).
- Valores de los encoders A y B (encoderAABS, encoder BABS).
- El tiempo desde que la puesta en marcha hasta la lectura actual del encoder (tAabs, tBabs).
- El nivel de batería.
- Los datos de los ocho sensores de ultrasonidos que lleva la silla, siendo S:Sensor, D:Delantero o derecho, I:Izquierdo y T:Trasero.
- Lectura acelerómetro en milésimas de g's (EjeX, EjeY y EjeZ).
- Datos de velocidad a los motores (DatoD, DatoI).

Capítulo 4

Desarrollo

En este capítulo de explicará cómo se han creado los tres grandes bloques de este Trabajo Fin de Grado.

En primer lugar se desarrollara como se ha creado el paquete encargado de modelar la silla. Para ello se hará una introducción sobre el lenguaje XML, lenguaje en el cual está basado el modelo URDF y se explicara en que consiste este modelo y como se ha creado.

En segundo lugar se explicará cómo se ha creado el paquete encargado de realizar la simulación, se hablará sobre la estructura de este paquete, las partes que lo forman y como se crean cada una de ellas.

Por último, se explicara cómo se ha creado el paquete encargado de realizar la comunicación bus CAN-ROS, los elementos que lo forman y como se realiza esta comunicación.

4.1 Modelado de la silla

4.1.1. Lenguaje de programación XML.

XML (*extensible Markup Language*), es un lenguaje de marcas desarrollado por el *World Wide Web Consortium* (W3C) utilizado para almacenar datos en forma legible por los sistemas multiplataforma conectados a internet. XML permite definir etiquetas personalizadas para la descripción y la organización de datos ([10]). Las principales ventajas de XML son:

- Es ampliable: Se puede extender añadiendo nuevas etiquetas.
- El analizador es un componente estándar, no siendo necesario crear un analizador específico para cada versión de lenguaje XML. De esta manera se acelera el desarrollo de aplicaciones.

- Al estar basado en una estructura jerárquica, es sencillo entender y procesar, mejorando así la compatibilidad entre aplicaciones.
- Su análisis sintáctico es fácil, debido a que separa el contenido y el formato de la presentación.

4.1.1.1 Estructura de un documento XML:

XML [14] busca expresar información de manera estructurada, de la forma más abstracta y reutilizable posible. Esto quiere decir que un fichero de este lenguaje compone de partes bien definidas, que se componen a su vez de otras partes, manteniendo una estructura jerárquica. Cada parte se denomina elemento y se señala mediante etiquetas.

Por tanto, las etiquetas son marcas hechas en el documento, que señalan una porción de éste como un elemento. Las etiquetas tienen la forma <"nombre">, donde "nombre" es el nombre del elemento que se está señalando.

Un documento XML comienza siempre con una instrucción de proceso. La presencia de esta instrucción indica explícitamente que el documento es XML y con qué versión XML ha sido confeccionado. Un ejemplo de instrucción de proceso puede ser:

```
<?xml version="1.0" encoding="UTF-7" standalone="yes"?>
```

Esta línea indica que el código tras ella está sujeto a la sintaxis de la especificación XML, versión 1.0. Además se especifica el conjunto de caracteres que se van a utilizar en el documento, en Europa se utiliza UTF-7 o ISO-8859-1.

XML como lenguaje, es la combinación de una serie de entidades. A continuación se comentarán brevemente estas entidades.

A) Elementos:

Son el recurso básico que utilizan los lenguajes de marcas como HTML, SGML o XML para identificar y manejar el contenido de los documentos.

Como se ha comentado, la representación práctica de un elemento son las etiquetas, identificadas por la presencia de símbolos "< >". Un elemento comienza con una etiqueta de inicio ("<EJEMPLO>") y finaliza con una etiqueta de cierre ("</EJEMPLO>").

Un elemento puede tener contenido o ser un elemento vacío. Por contenido entendemos texto u otros elementos jerárquicamente inferiores, que a su vez, pueden tener contenido o estar vacíos.

B) Atributos

Los atributos son una manera de incorporar características o propiedades a los elementos de un documento. Deben ir entre comillas.

C) Entidades predefinidas

Las entidades se usan para representar caracteres especiales, como por ejemplo “<”, para que de esta forma no sean interpretados como marcado en el procesador XML.

D) Secciones CDATA

Es una construcción en XML para especificar datos utilizando cualquier carácter sin que se interprete como marcado XML.

E) Comentarios

Los comentarios comienzan con “<!--“ y terminan con “-->”. Se pueden emplazar comentarios en cualquier lugar del documento, excepto dentro de las propias etiquetas o dentro de otros comentarios, y son ignorados por los procesadores.

<!-- Esto es un comentario -->

4.1.2 XML Robot Description Format (URDF).

Unified Robot Description Format (URDF) es la especificación XML para describir un el modelo de un robot que se usa en ROS. Además URDF también ofrece conversiones y compatibilidad con otros formatos de modelado de robots, como el formato SDF (*Simulation Description Format*, usado por el simulador GAZEBO) o el formato COLLADA (más genérico para definir modelos en 3D). La relación de los distintos formatos de modelado de robots en ROS se muestra en la Figura 23.

El formato URDF se usa para definir el robot mediante una estructura en forma de árbol, donde los elementos del robot deben ser estructuras rígidas conectadas por articulaciones (las estructuras flexibles no son compatibles). URDF define la descripción cinemática y dinámica del robot, la representación visual de éste y su modelo de colisiones.

La descripción de un robot en formato URDF consiste en un conjunto de *links*, que definen un cuerpo rígido con inercia y ciertas características visuales, y un conjunto de *joints*, que conectan los *links* entre sí y definen características cinemáticas y dinámicas de la articulación además de limitar los posibles giros y movimientos.

En la Figura 24 se muestra un ejemplo de una posible estructura de árbol formada por cuatro *links* con sus tres *joints*.

La descripción URDF típica de un robot tiene el siguiente formato:

```
<robot name=”ejemplo”>
    <link1> … </link1>
    <link2> … </link2>
    <joint> … </joint>
</robot>
```

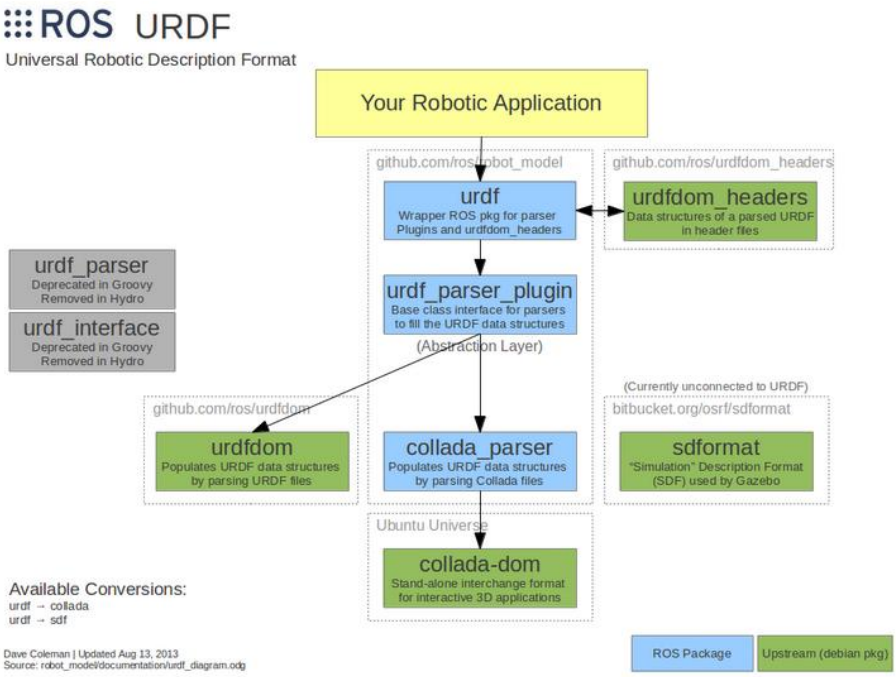

Figura 23- Relación entre los diferentes formatos de modelado de robots en ROS

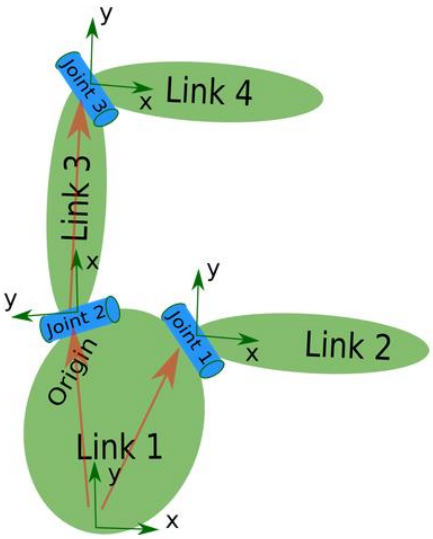

Figura 24 - Ejemplo de estructura URDF

Como se puede ver en el ejemplo anterior, en la descripción de cualquier robot de ROS hay un elemento raíz del formato URDF: el elemento <robot>.

A continuación se comentará brevemente como se describen los *links* y los *joints* y el tipo de propiedades que pueden tener cada uno.

4.1.2.1 Elemento link

El elemento *link* describe un cuerpo rígido con una inercia y características visuales:

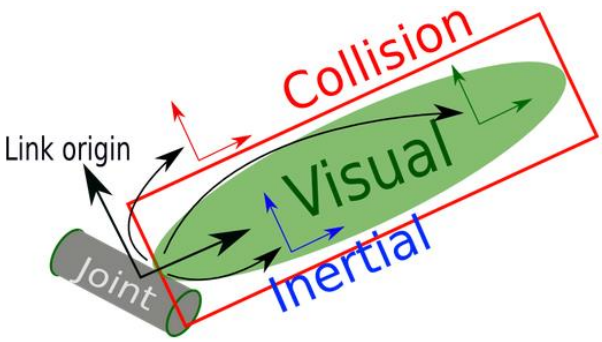

Figura 25 - Elemento link

Dentro del elemento *link* se describen tres elementos más:

- **Elemento inercial**. Mediante la matriz de rotación inercial 3x3 y la masa del *link* se representa la inercia de éste. También es necesario definir el origen de este elemento inercial con respecto al origen del *link*. A continuación se puede ver un ejemplo de la estructura del elemento inercial:

```
<inertial>
<origin xyz="0 0 0.5" rpy="0 0 0"/>
<mass value="1"/>
<inertia ixx="100"  ixy="0"  ixz="0" iyy="100" iyz="0"
izz="100" />
</inertial>
```

- **Elemento visual.** Define las propiedades visuales del *link*. Especifica el origen de coordenadas del elemento visual con respecto al origen del *link*, la forma del *link* (caja, cilindro, etc.) y su color, como se muestra en el ejemplo siguiente.

```
<visual>
  <origin xyz="0 0 0" rpy="0 0 0" />
  <geometry>
    <box size="1 1 1" />
  </geometry>
  <material name="Cyan">
    <color rgba="0 1.0 1.0 1.0"/>
  </material>
</visual>
```

- **Elemento de colisión.** Define las propiedades de colisión del link. El elemento de colisión suele ser un objeto más grande que el elemento visual, de esta manera detecta que el link ha colisionado antes de que lo haga realmente. A continuación se muestra un ejemplo de uso de este elemento:

```
<collision>
  <origin xyz="0 0 0" rpy="0 0 0"/>
  <geometry>
```

```
      <cylinder radius="1" length="0.5"/>
    </geometry>
  </collision>
</link>
```

La información completa de la definición de los modelos puede encontrarse en la web de URDF dentro de la de ROS [15].

4.1.2.2 Elemento joint

El elemento *joint* (articulación) describe las propiedades dinámicas y cinemáticas, además de los límites de seguridad de la articulación, respecto a los dos links que se denominan padre e hijo ("parent" y "child"), ver Figura 26.

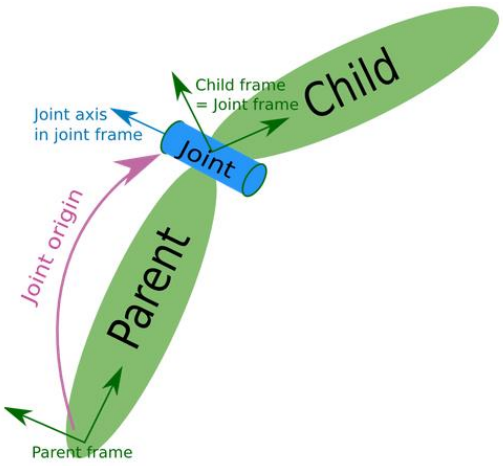

Figura 26 - Elemento Joint

El elemento *joint* tiene dos atributos, tal y como se muestra en el siguiente ejemplo, el nombre y el tipo de articulación que se van a definir (fixed, continuous, floating, planar, etc.…).

```
<joint name="my_joint" type="floating">
```

Dentro del elemento *joint* se pueden definir varios elementos:

- <origin>, en el definimos el origen de coordenadas del *joint* con respecto al origen de coordenadas del link padre.

  ```
  <origin xyz="0 0 1" rpy="0 0 3.1416"/>
  ```

- <parent link> y <child link>, se debe especificar que *link* es el padre y que *link* es el hijo. Cuando se defina el origen de coordenadas del *link* hijo, se tomará como referencia el origen de coordenadas del *joint*.

  ```
  <parent link="link1"/>
  <child link="link2"/>
  ```

- <calibration> define las posiciones de referencia de la articulación para calibrar su posición absoluta.

  ```
  <calibration rising="0.0"/>
  ```

- <dynamics>, especifica las propiedades físicas de la articulación. Estos valores se utilizan para especificar las propiedades de modelado de la articulación.

```
<dynamics damping="0.0" friction="0.0"/>
```

- <limit>, define los límites de movimiento de la articulación.

```
<limit effort="30" velocity="1.0" lower="-2.2" upper="0.7" />
```

- <safety_controller>, especifica los límites de seguridad de la articulación.

```
<safety_controller k_velocity="10" k_position="15"
soft_lower_limit="-2.0" soft_upper_limit="0.5" />
```

La información completa de la definición de los modelos puede encontrarse en la web de URDF dentro de la de ROS [15].

A la hora de crear un modelo URDF complejo, con un gran número de *links* y de *joints*, el código puede crecer demasiado y volverse complicado de entender. Para evitar ese problema se usa XACRO.

XACRO es un lenguaje de macros XML que permite crear archivos XML más cortos y más legibles mediante el uso de macros.

Por ejemplo, en el caso del modelado de SARA, para definir las ruedas traseras de la silla y sus motores, se tendrían que crear dos *links* para las ruedas, dos *links* para los motores y dos *joints* que unan cada par de *links* rueda-motor.

Para facilitar esta tarea, se puede crear una macro a la que se le pase como parámetro el lado en el que está situada la rueda/motor y un parámetro llamado posición, que tomara valor “-1” o “1” en función del lado en el que se esté, tal y como se indica a continuación.

```
<xacro:macro name="motoryruedas" params="lado posicion">
```

De esta forma, dentro de la macro se puede crear en función de estos dos parámetros, un *link* para las dos ruedas, otro para los dos motores y un *joint* para las dos uniones, de este modo:

```
<!--Link Ruedas motores -->
<link name="rueda_tras_${lado}">
      <visual>
    <geometry>
      <cylinder length="${anchura_rueda}" radius="${radio_rueda}"/>
    </geometry>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <material name="rueda_tras">
      <color rgba="0.25 0.25 0.25 1"/>
    </material>
      </visual>
    </link>


<!-- Link para motores ambos lados -->
<link name="motor_${lado}">
   <visual>
    <geometry>
```

```
    <cylinder length="${anchura_motor}" radius="${radio_motor}"/>
    </geometry>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <material name="motor">
      <color rgba="0.25 0.25 0.25 1"/>
    </material>
   </visual>
</link>

<!--Union Ruedas motores y motor -->
    <joint name="motor_to_rueda_tras_${lado}" type="fixed">
      <parent link="motor_${lado}"/>
      <child link="rueda_tras_${lado}"/>
      <origin xyz="0 0 ${-
posicion*(anchura_motor/2.0+anchura_rueda/2.0)}" rpy="0 0 0"/>
    </joint>
</xacro:macro>
```

XACRO también permite crear propiedades y asignarlas un valor para más tarde ser usadas dentro del MACRO.

```
<xacro:property name="anchura_motor" value="0.1016" />
```

Una vez descrita la macro, se llama de este modo:

```
<xacro:motoryruedas lado="der" posicion="-1" />
<xacro:motoryruedas lado="izq" posicion="1" />
```

La información completa de la definición de macros con XACRO puede encontrarse su web dentro de la de ROS [16].

4.1.3 Modelado y visualización de la silla en ROS.

A partir de las herramientas descritas, para realizar el modelado de la silla de ruedas SARA en ROS, se ha creado el paquete “sara_urdf”, estructurado como se muestra en la Figura 27.

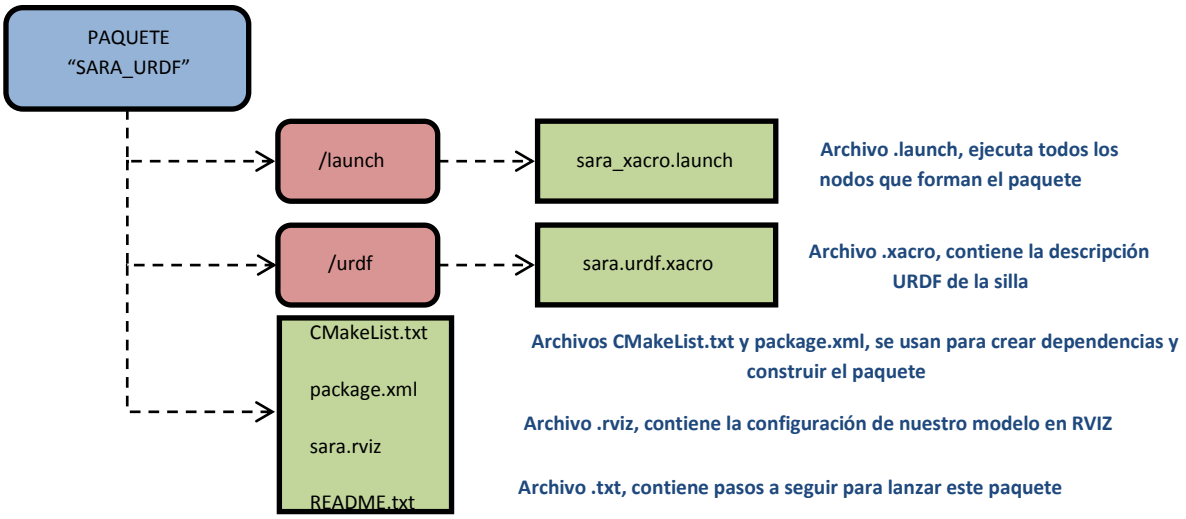

Figura 27 – Estructura de SARA con URDF

A continuación se comentarán los archivos más importantes del paquete ”sara_urdf” diseñado.

4.1.3.1 “sara.rviz”.

Este archivo contiene la configuración de la herramienta RVIZ y guarda las propiedades visuales situadas en la columna de la izquierda de RVIZ para que se visualicen automáticamente.

4.1.3.2 “sara.urdf.xacro”.

Este archivo contiene el modelo visual URDF de la silla, modelada usando el lenguaje de macros XACRO, explicado en el apartado anterior. Comienza con la instrucción de proceso *<?xml version="1.0"?>.*

El modelo del robot (Figura 28), debe colgar del elemento “robot”, cuya declaración es la siguiente:

```
<robot name="sara_v1" xmlns:xacro="http://www.ros.org/wiki/xacro" >
```

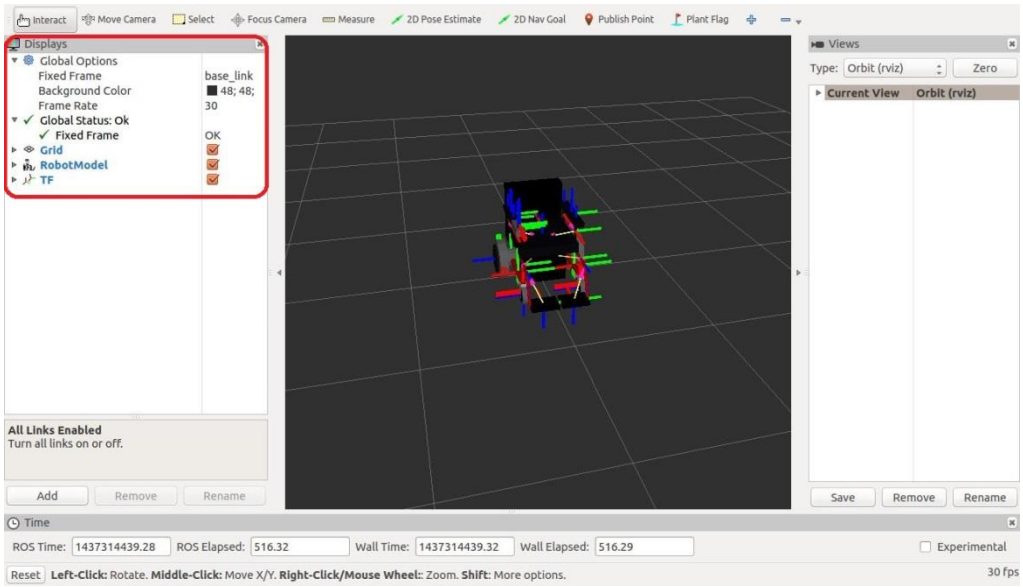

Figura 28 – Interfaz gráfica herramienta RVIZ

El modelo URDF de la silla se ha ido creando por bloques a continuación se explicarán con más detalle cada uno de ellos. En el Capítulo 7 (Diagramas) se puede ver el árbol completo del modelo, el cual ha sido generado con la herramienta “urdf_to_graphiz”. Este modelo está basado en el paquete “smart_wheelchair” de CWRU (*Case Western Reserve University's*) *Robotics* [17].

A) Links principales: ““base_link”” y “batería”.

Antes de empezar a crear los *links* y *joint* del modelo, se deben definir las variables que se utilizarán en el modelo XACRO. En este bloque se define el tamaño del *link* llamado batería y la separación de la batería con el asiento

```
<!-- Tamano bateria entre las dos ruedas -->
```

```
  <xacro:property name="long_bateria" value="0.4318" />
 <xacro:property name="ancho_bateria" value="0.3302" />
 <xacro:property name="altura_bateria" value="0.254" />
 <xacro:property name="separacion_bateria" value="0.04445" />
```

El árbol del modelo URDF debe contener un elemento padre, por tanto se crea este situado en el origen de coordenadas, llamado "base_link". Del elemento padre cuelga el *link* llamado "bateria", el elemento central del modelo, del cual colgarán el resto de elementos de la silla.

```
<!-- Link base-->
 <link name="base_link">
  </link>
  <!-- Main bateria-->
  <!-- link de la bateria -->
  <link name="bateria">
    <visual>
      <geometry>
        <box size="${long_bateria} ${ancho_bateria} ${altura_bateria}"/>
      </geometry>
      <material name="black">
      <color rgba="0 0 0 1"/>
      </material>
    </visual>
  </link>
```

Para unir estos dos *links* se ha creado el *joint* "base_to_batería". En él se define cual es el origen de coordenadas del *link* hijo ("bateria") con respecto al padre ("base_link").

```
<!--Union bateria y base. El centro de la bat esta 0.04445 mas
alto del centro de la bateria -->
  <joint name="base_to_bateria" type="fixed">
    <parent link="base_link"/>
    <child link="bateria"/>
    <origin xyz="0 0 ${altura_bateria/2.0+separacion_bateria}"/>
  </joint>
```

B) *Links* y *joints* de los motores y ruedas traseras.

En este bloque se han modelado los motores y las ruedas traseras de la silla, con sus respectivos ejes. Para ello, se han definido las siguientes propiedades con las dimensiones de las ruedas, motores y ejes:

```
<!-- MOTORES Y RUEDAS -->
  <xacro:property name="radio_rueda" value="0.1651" />
  <xacro:property name="anchura_rueda" value="0.0635" />
  <xacro:property name="radio_eje" value="0.11" />
  <xacro:property name="anchura_eje" value="0.05" />
  <xacro:property name="radio_motor" value="0.0762" />
  <xacro:property name="anchura_motor" value="0.1016" />
  <xacro:property name="long_barra_horiz" value="0.38" />
  <xacro:property name="radio_barra_horiz" value="0.02" />
```

Para facilitar la creación y depuración del código, se ha creado una macro para realizar este bloque. Con el uso de esta macro, podemos crear un *link* y un *joint* para cada par de motores, ejes y ruedas, pasando como argumentos las variables lado y posición. Cuando "lado" tome el valor "izq", se crearan los *links* "motor_izq", "eje_izq" y "rueda_tras_izq",

cuando “lado” tome el valor “der”, se crearan los *links* “motor_der”, “eje_der” y “rueda_tras_der”. La variable posición se utiliza para orientar el eje de coordenadas en el lado correcto, cuando “posición” tome el valor -1, situara el *link* a la derecha del su *link* padre, cuando tome el valor 1, lo situará a la “izq”.

A continuación se puede ver un fragmento del código de este bloque, en el cuál se crea los motores colgando del link “bateria” y sus correspondientes uniones. El resto de *joints* y *links* se crean siguiendo esta metodología.

```
<xacro:macro name="motoryruedas" params="lado posicion">
  <!-- Link para motores ambos lados -->
      <link name="motor_${lado}">
      <visual>
    <geometry>
      <cylinder length="${anchura_motor}" radius="${radio_motor}"/>
    </geometry>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <material name="motor">
      <color rgba="0.25 0.25 0.25 1"/>
    </material>
      </visual>
    </link>
<!-- union motores con la bateria -->
    <joint name="bateria_to_motor_${lado}" type="fixed">
      <parent link="bateria"/>
      <child link="motor_${lado}"/>
      <origin xyz="0 ${posicion*(ancho_bateria/2.0+anchura_motor/2.0)} 0"
rpy="${pi/2} 0 0"/>
    </joint>
```

Una vez descrita la macro, se llama de este modo:

```
<!--Llamamos al macro de los motores y ruedas -->
<xacro:motoryruedas lado="der" posicion="-1" />
<xacro:motoryruedas lado="izq" posicion="1" />
```

En la Figura 29 se puede ver el modelo visual de este bloque junto al bloque anterior.

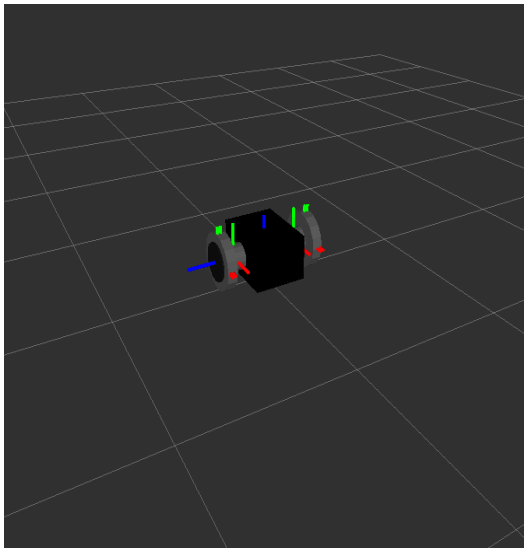

Figura 29 - Modelo URDF: Motores y ruedas traseras

C) Ruedas y barras delanteras.

Para crear las ruedas delanteras, se ha creado una macro, el cual pasándole la posición y el lado, crea los *links* de las ruedas delanteras, sus soportes y las barras que unen estas ruedas con el resto de la silla. También se han definido las propiedades visuales de estas piezas usando variables XACRO.

```
<!--RUEDAS DELANTERAS -->
  <xacro:property name="radio_rueda_del" value="0.0762" />
  <xacro:property name="ancho_rueda_del" value="0.0381" />
  <xacro:property name="radio_eje_rueda_del" value="0.055" />
  <xacro:property name="ancho eje rueda del" value="0.04" />
  <xacro:property name="casterheight" value="0.15" />
```

A continuación se muestra cómo se crean las ruedas (Figura 30), sus soportes y la unión de ambos en función de las variables lado y posición. El significado de estas variables es el mismo que en el bloque anterior.

```
<!--Macro ruedas delanteras -->
  <xacro:macro name="ruedas_delant" params="lado posicion">
<!--Soporte ruedas delanteras -->
    <link name="soporte_rueda_${lado}">
      <visual>
    <geometry>
      <box size="0.04445 0.04445 0.12"/>
    </geometry>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <material name="gris">
      <color rgba="0.45 0.45 0.47 1"/>
    </material>
      </visual>
   </link>
<!--junta entre barra vert y soporte -->
    <joint name="barra_vert_to_soporte_${lado}" type="fixed">
      <parent link="barra_vertical_${lado}"/>
      <child link="soporte rueda ${lado}"/>
      <origin xyz="0 0 -0.02" rpy="0 0 0"/>
    </joint>
<!--Ruedas delanteras -->
    <link name="rueda_del_${lado}">
      <visual>
    <geometry>
      <cylinder length="${ancho_rueda_del}"
radius="${radio_rueda_del}"/>
    </geometry>
    <material name="rueda del">
      <color rgba="0.25 0.25 0.25 1"/>
    </material>
      </visual>
    </link>
<!--junta entre soporte y rueda -->
    <joint name="soporte_to_rueda_del_${lado}" type="fixed">
      <parent link="soporte_rueda_${lado}"/>
      <child link="rueda del ${lado}"/>
      <origin xyz="0 0 0.035" rpy="0 ${pi/2} 0"/>
    </joint>
 </xacro:macro>
```

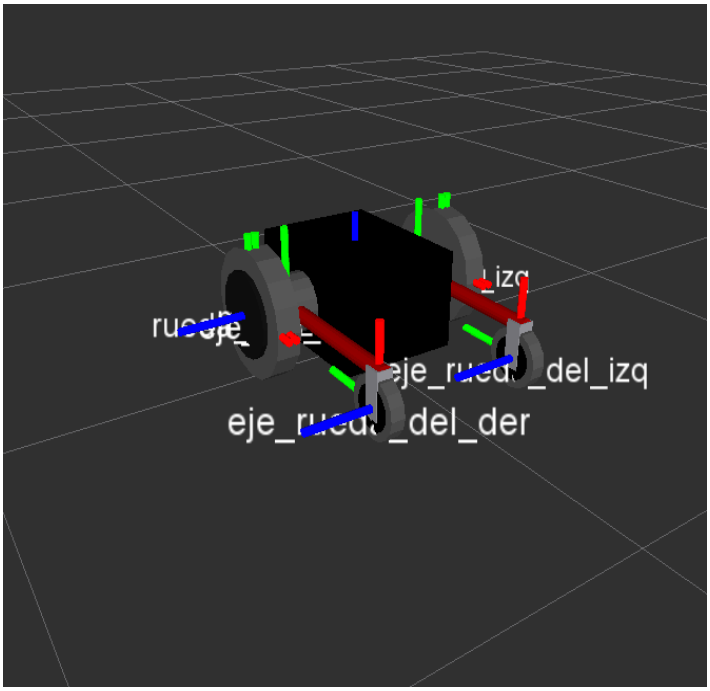

Figura 30 – Modelo URDF: Ruedas delanteras

Una vez descrita la macro, se llama de este modo:

```
<xacro:ruedas_delant lado="der" posicion="-1" />
<xacro:ruedas_delant lado="izq" posicion="1" />
```

D) Respaldo, asiento, apoyabrazos y joystick.

Colgando del *link* batería, se crean los *links* para el asiento y el respaldo de la silla, así como sus correspondientes *joints*. A continuación se puede ver como se ha creado el *link* "respaldo" y su unión con el *link* "batería":

```
<!-- RESPALDO -->
  <xacro:property name="long_resp" value="0.1143" />
  <xacro:property name="ancho_resp" value="0.4318" />
  <xacro:property name="altura_resp" value="0.4572" />
  <link name="respaldo">
    <visual>
      <geometry>
    <box size="${long_resp} ${ancho_resp} ${altura_resp}"/>
      </geometry>
      <material name="black">
    <color rgba="0 0 0 1"/>
      </material>
    </visual>
  </link>
<!-- Junta respaldo y bateria -->
  <joint name="bateria_to_respaldo" type="fixed">
    <parent link="bateria"/>
    <child link="respaldo"/>
    <origin xyz="${-long_bateria/2+0.06} 0 0.3"/>
  </joint>
```

Una vez creado el respaldo y asiento de la silla, se crean los apoyabrazos de ambos lados. Para ello se crea un macro llamado “apoya_brazos”, que al igual que en el resto de macros, te crea un *link* para cada lado en función de la variable lado.

```
<xacro:property name="long_barra_brazos" value="0.18" />
  <xacro:property name="radio barra brazos" value="0.01" />
  <xacro:property name="ancho_soporte_brazos" value="0.0635" />
  <xacro:property name="espesor_soporte_brazos" value="0.04" />
  <xacro:macro name="apoya_brazos" params="lado
long_soporte_brazos separacion_barras">
```

Estos apoyabrazos se unirán con el *link* “asiento” mediante unas barras verticales, tal y como se puede ver en la Figura 31.

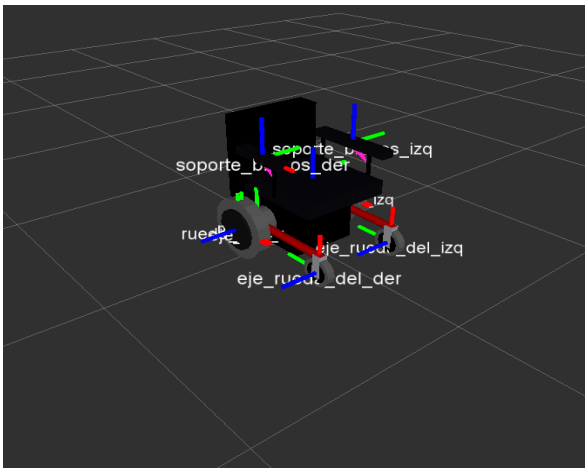

Figura 31 – Modelo URDF: Respaldo y apoyabrazos

Colgando del apoyabrazos derecho, se ha creado el *link* “joystick”, el cual emula el modelo del joystick que incorpora la silla actualmente. El joystick está formado por los *links* “joystick”, “bola_joystick” y “palo_joystick”. En la Figura 32 se puede ver la representación del joystick en la herramienta RVIZ.

Figura 32 – Modelo URDF: Joystick

E) Reposapiernas

El último bloque a crear, es el de los reposapiernas derecho e izquierdo. Para ello, se crea el soporte para el reposapiernas mediante barras horizontales y verticales. Colgando de las barras verticales se crean los soportes para los pies (Figura 33).

Los *links* de este bloque se han creado usando macros, se ha creado el macro "reposa_piernas", el cual crea todos los *links* y *joints* de este bloque pasándole como argumento el lado y la posición.

```
<xacro:macro name="reposa_piernas" params="lado posicion">
```

A continuación se muestra como se ha creado los *links* de las barras horizontales y su uniones con el *link* padre, batería. El resto de *links* y *joints* se crean de manera similar.

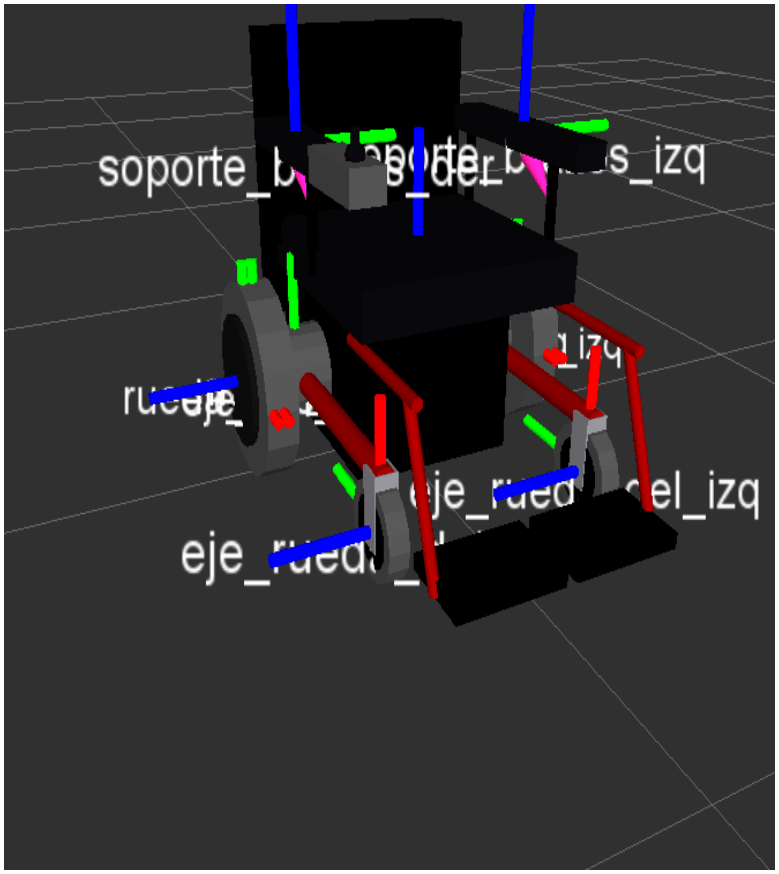

Figura 33 – Modelo URDF: Reposapiernas

```
<!-- Reposa piernas horiz -->
    <link name="reposapiernas_horiz_${lado}">
      <visual>
    <geometry>
      <cylinder length="${altura_reposapiernas}"
radius="${radio_tubo_reposapiernas}"/>
    </geometry>
    <origin xyz="0 0 0" rpy="0 0 0"/>
    <material name="granate">
      <color rgba="0.5 0 0 1"/>
    </material>
```

```
      </visual>
    </link>
    <!-- Union reposa piernas horiz y bateria -->
    <joint name="bateria_to_reposapiernas_horiz_${lado}"
type="fixed">
      <parent link="bateria"/>
      <child link="reposapiernas_horiz_${lado}"/>
      <origin xyz="0.42 ${posicion*0.21} 0.15" rpy="0 ${pi/2} 0"/>
    </joint>
```

Por último, llamamos al macro “reposa_piernas”:

```
<xacro:reposa_piernas lado="der" posicion="-1" />
<xacro:reposa_piernas lado="izq" posicion="1" />
```

4.1.3.3 “sara_xacro.launch”

Una vez creado el modelo URDF de la silla, se crea el archivo “.launch” para lanzarlo y visualizarlo en RVIZ. El código del archivo “sara_xacro.launch” es el siguiente:

```
<launch>
    <arg name="model" default="$(find
sara URDF)/sara URDF.xacro"/>
    <arg name="gui" default="False" />
    <arg name="rvizconfig" default="$(find sara_URDF)/sara.rviz"
/>
    <param name="robot description" command="$(find
xacro)/xacro.py $(arg model)" />
    <param name="use_gui" value="$(arg gui)"/>
    <node name="joint_state_publisher" pkg="joint_state_publisher"
type="joint_state_publisher" />
    <node name="robot state_publisher" pkg="robot_state_publisher"
type="state_publisher" />
    <node name="rviz" pkg="rviz" type="rviz" args="-d $(arg
rvizconfig)" required="true" />
</launch>
```

Para visualizar el modelo de la silla es necesario lanzar los siguientes nodos:

- “joint_state_publisher”. Este nodo es el encargado de publicar mensajes del tipo “sensor_msgs/JointState”. El nodo lee el parámetro “robot_description” y publica un mensaje del tipo *JointState* con todas sus articulaciones definidas. El modelo XACRO de la silla se define como un argumento, para poder lanzar el modelo que queramos desde la línea de comandos sin necesidad de cambiar el código del archivo “.launch”.
- “robot_state_publisher”. Este nodo se encarga de publicar las transformaciones TF de los marcos del robot. Requiere una fuente que publique la posición de cada *joint* (“joint_state_publisher”) y la descripción URDF del robot.
- RVIZ. Para visualizar el modelo de la silla se utiliza la herramienta RVIZ, en la cual se carga la configuración de RVIZ definida en el archivo “sara.rviz”.

Para lanzar el archivo “.launch”, se debe introducir la siguiente línea en el terminal de Linux:

```
roslaunch sara_urdf sara_xacro.launch model:=urdf/sara.urdf.xacro
```

Estamos indicando que se lance el archivo “sara_xacro.launch”, situado en la carpeta “sara_urdf” y cargando el modelo “sara.urdf.xacro” situado en la carpeta “/urdf.

4.2 Integración del sistema

4.2.1 Integración visual

Como se ha comentado en el Capítulo 2, ROS proporciona varios simuladores, tanto en 2D (STDR, STAGE) como en 3D (GAZEBO). De todos estos, sin duda el simulador más potente y realista es GAZEBO, pero para poder usarlo se requiere el modelo dinámico y cinemáticos de la silla, cosa que en este TFG no se ha implementado. En este TFG se ha simulado el comportamiento de la silla en STDR, ya que de los simuladores 2D es el que más se usa actualmente, debido a que STAGE se está quedando obsoleto. Aun así, STDR es algo inestable ya que está en fase de desarrollo.

Para ello, se ha creado el paquete llamado “sara_sim_stdr”. Este paquete tiene la estructura que se muestra en la Figura 34.

A continuación, se explicarán los archivos más importantes contenidos en este paquete.

4.2.1.1 Mapas del entorno

En el directorio /maps se guardan todos los mapas que se quieren usar en la simulación. Para poder cargar un mapa en STDR, se requiere que el mapa este contenido en un archivo “.yalm”. En este archivo se especifica la imagen que se usará como mapa y diversas propiedades de este, como por ejemplo, la resolución, donde queremos colocar la imagen en el simulador, etc...

```
image: nombre_imagen.png
resolution: 0.02
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.6
free_thresh: 0.3
negate: 0
```

4.2.1.2 Modelo del robot para la simulación

En el directorio /robot se guardan los robots que se quieren utilizar en la simulación. Al igual que el mapa, STDR necesita que el modelo del robot este en formato “.yaml”. Hay dos maneras de crear el modelo del robot, gráficamente a través de la GUI de STDR o modificando el archivo “.yaml”. A continuación se explicarán ambos métodos.

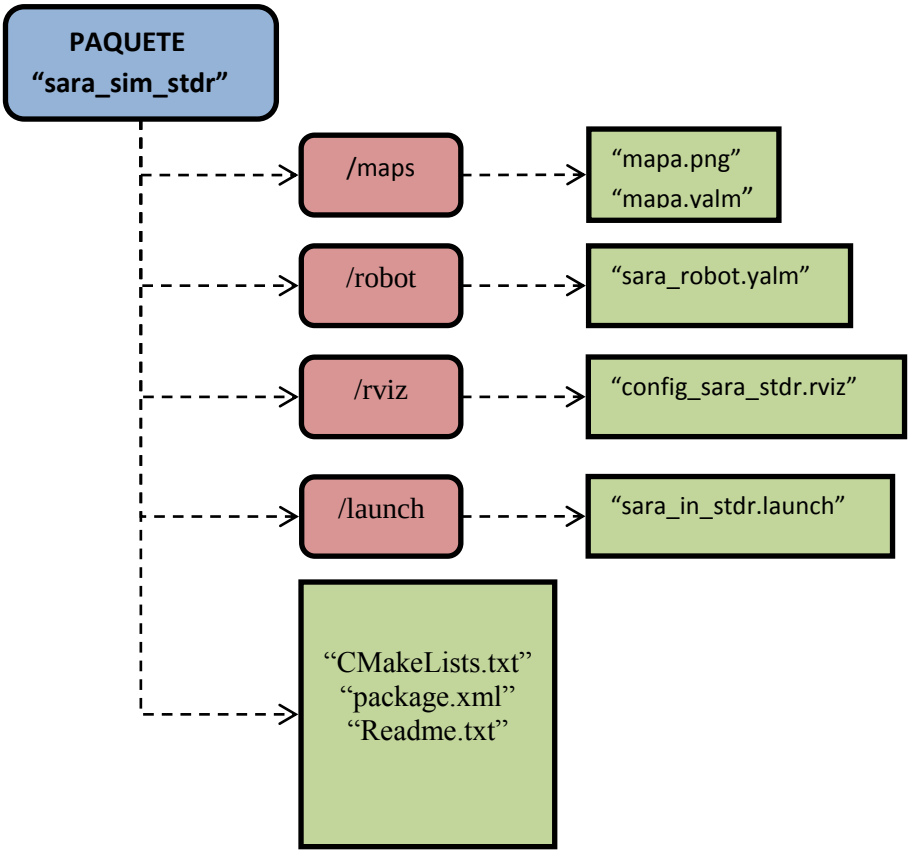

Figura 34 - Estructura del paquete “sara_sim_stdr”

A) Creación del modelo del robot a través de la GUI STDR.

Una vez lanzada la GUI del simulador STDR, se tiene que seleccionar la opción *Create robot*, situada en la barra de herramientas, tal y como se indica en la Figura 35.

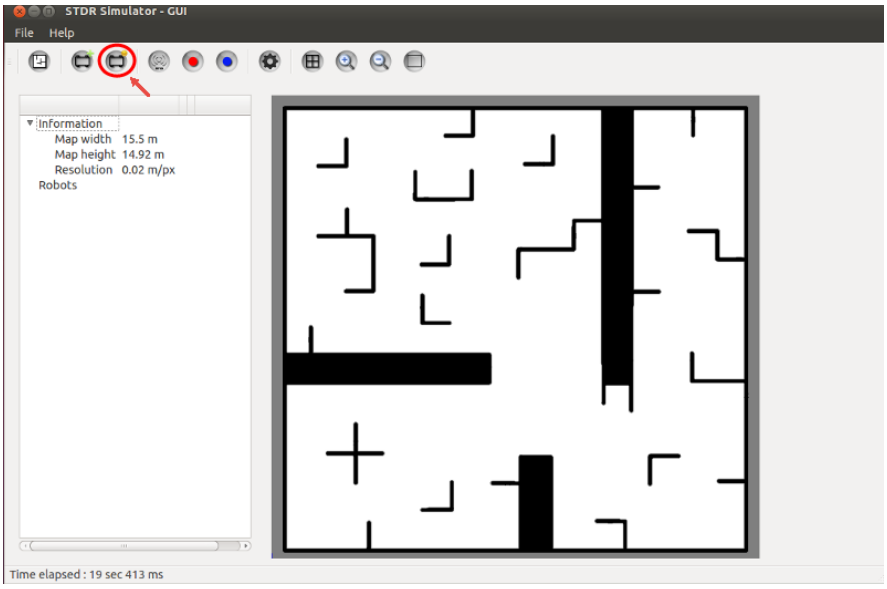

Figura 35 - GUI Simulador STDR

Una vez abierto el STDR Robot Creator, ya se puede proceder a crear el robot. La interfaz gráfica de este se puede ver en la Figura 36.

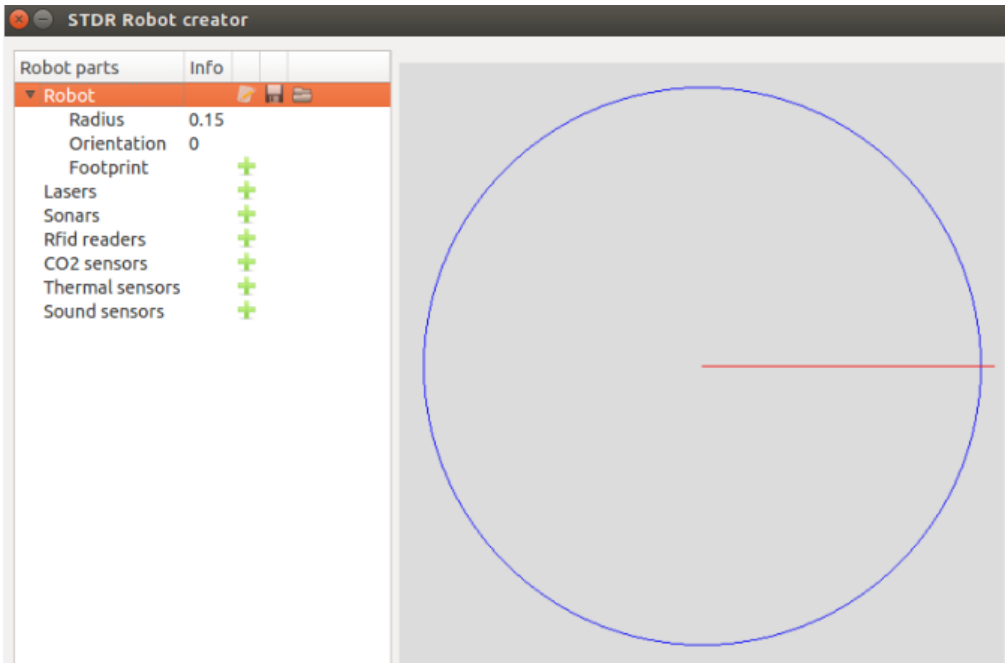

Figura 36 - STDR Robot Creator

STDR permite crear robots circulares, con el radio que se especifique en la opción Radius. Una vez creado el robot hay que añadirle los sensores que tenga el robot a simular. STDR permite añadir sensores de ultrasonidos, laser, térmicos y de sonido entre otros. Una vez añadido un sensor, hay que definir sus principales propiedades.

En el caso de los sensores de ultrasonido, tal y como se muestra en la Figura 37, se pueden configurar los siguientes parámetros:

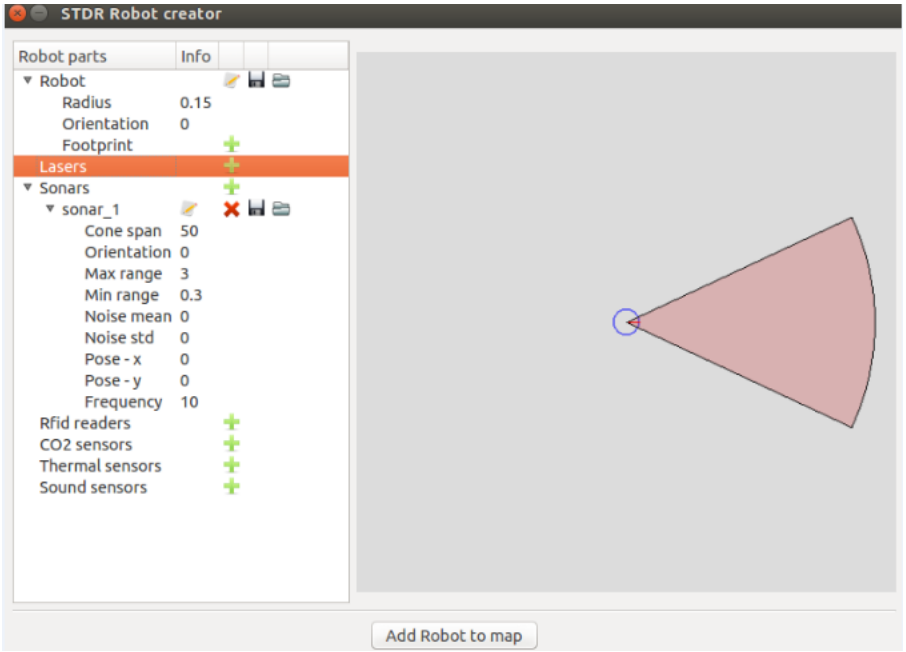

Figura 37 - STDR Robot Creator: SONAR

- Cone_span: Aquí se configura el radio de visión del sensor.
- Orientation: Orientación del sensor con respecto al centro del robot.

- Max/min Range: Rango de visión del sensor.
- Noise mean y noise std: Con estas propiedades podemos introducir ruido en el sensor.
- Posse –x/-y : Posición del sensor respecto al centro del robot.
- Frequency: Frecuencia de muestreo del sensor.

Una vez definido el robot y sus propiedades, seleccionando la opción Add Robot to map colocamos el robot en el simulador STDR.

La GUI de STDR tiene la opción de exportar el modelo del robot en un fichero ".yaml".

B) Creación del modelo del robot con un fichero .yaml.

La estructura del modelo .yaml es la siguiente:

```
robot:
    robot_specifications:
      - footprint:
          footprint_specifications:
                      radius: 0.200000002980232
                      points:
                        []
        - initial_pose:
                 x: 0
                y: 0
                theta: 0
        - sonar:
                 sonar_specifications:
                 cone_angle: 0.870000004768372
                 max_range: 3
                 min_range: 0.150000005960464
                 frequency: 10
                 frame_id: sonar_0
                 pose:
                          x: 0.2
                          y: 0
                          theta: 0
                 noise:
                    noise_specifications:
                       noise_mean: 0.100000001490116
                       noise_std: 0.00999999977648258
```

Como en el caso de la GUI STDR, el modelo del robot tiene una serie de especificaciones, el *footprint* (radio del robot), la posición inicial que ocupa en el mapa de la simulación, y cada uno de los sensores que lleva incorporados. Las propiedades a definir en el caso de los sensores de ultrasonidos son las mismas que en el caso de la GUI STDR.

4.2.1.3 Configuración de RVIZ

Como se explicó en el Capítulo 2, RVIZ es una herramienta que proporciona ROS para visualizar todo tipo de *topics* y robots. Dentro del fichero "config_sara_stdr.rviz" está guardada la configuración de dicha herramienta.

4.2.1.4 Archivo ".launch" del paquete

Este archivo es el encargado de lanzar todos los nodos necesarios para realizar la simulación con una única orden. A continuación se explicara su código por partes.

Antes de lanzar los nodos necesarios para la simulación, se deben definir los argumentos globales en este archivo. En este caso, se ha definido la pose inicial del robot en el mapa.

```
<launch>
  <arg name="initial pose x" default="2.0"/>
  <arg name="initial pose y" default="2.0"/>
  <arg name="initial_pose_a" default="0.0"/>
```

El primer nodo que hay que lanzar, es *robot_manager.* Este es el encargado de realizar la simulación.

```
<!--  ******************** Stdr********************  -->
  <include file="$(find stdr_robot)/launch/robot_manager.launch"
/>
```

Para cargar el mapa, hay que lanzar el nodo *stdr_server,* pasándole como argumento el mapa en formato ".yaml".

```
 <!-- Run STDR server with a prefedined map-->
  <node pkg="stdr server" type="stdr server node"
name="stdr_server" output="screen" args="$(find
sara_sim_stdr)/maps/sparse_obstacles.yaml"/>
```

Para cargar el robot, hay que lanzar el nodo *stdr_robot,* pasándole como argumento el modelo del robot en formato .yaml y la posición del mapa en el que lo queremos colocar.

```
 <!--Spawn new robot at init position 2 2 0-->
  <node pkg="stdr_robot" type="robot_handler" name="$(anon
robot_spawn)" args="add $(find
sara_sim_stdr)/robot/sara_robot.yaml $(arg initial_pose_x) $(arg
initial_pose_y) 0"/>
```

Para lanzar la GUI STDR, hay que lanzar el ".launch" del nodo stdr_gui.

```
 <!-- Run Gui  -->
  <include file="$(find stdr_gui)/launch/stdr_gui.launch"/>
```

Con el fin de visualizar el modelo URDF a la hora de simular, se lanza el paquete "sara_urdf" explicado en el apartado anterior, pasándole como argumento el modelo XACRO del robot.

```
   <!--  ***************** Robot Model *****************  -->
  <include file="$(find sara_urdf)/launch/sara_xacro.launch">
    <arg name="model" value="$(find
sara_urdf)/urdf/sara.urdf.xacro" />
  </include>
```

A continuación se lanza el nodo RVIZ pasándole como argumento la configuración de este en un archivo ".rviz".

```
<!-- ********** Visualisation ********** -->
 <node name="rviz" pkg="rviz" type="rviz" args="-d $(find
sara_sim_stdr)/rviz/config_sara_stdr.rviz"/>
```

Para poder visualizar el modelo URDF en la simulación, hay que coordinar los orígenes de coordenadas del modelo URDF (base_link) y del robot creado en la simulación (robot0). Para ello lanzamos el nodo static_transform_publisher, indicándole cuales son los marcos padre e hijo para realizar la transformación.

```
<!--  ** Transformada tf entre robot0 y base_link **********-->
  <node pkg="tf" type="static_transform_publisher"
name="sara_broadcaster" args="0 0 0 0 0 0 1 robot0 base_link 100"
/>
```

Por último, se ha hecho uso del nodo "teleop_twist_keyboard", a través del cual se puede controlar el robot con las teclas del teclado en la simulación.

```
<!--  ********** Movimiento robot **********  -->
  <node pkg="teleop twist keyboard"
type="teleop twist keyboard.py" name="teleop" output="screen">
      <remap from="cmd_vel" to="robot0/cmd_vel"/>
  </node>
```

4.2.2 Integración física

4.2.2.1 Montaje Hardware

Como se ha comentado en capítulos anteriores, la interconexión de los distintos módulos se realiza principalmente por medio del bus CAN. Tal y como se explica en los trabajos de [1] y [2] la conexión de todos los buses CAN se realiza a través de una regleta con varios conectores RJ-45 hembra (Figura 38).

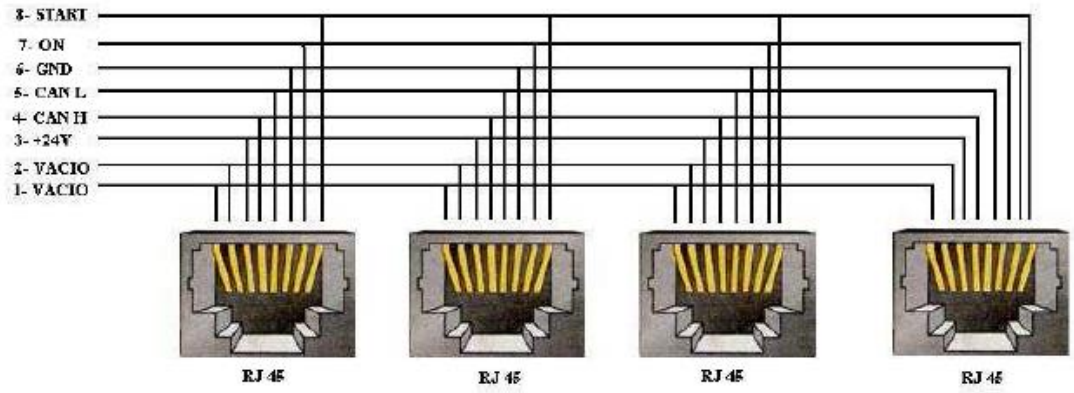

Figura 38- Regleta de conexión del bus CAN

La conexión entre el Alto Nivel (PC) y Bajo nivel (resto de módulos) se realiza por medio de un conversor CAN-USB. El conversor elegido es el de la casa LAWICEL (Figura 39), el mismo que se usaba en los trabajos realizados anteriormente con SARA.

Para la conexión del adaptador con la regleta de conectores RJ45 se ha creado un cable como el de la Figura 40, cableando únicamente las líneas CANH, CANL y GND. En el extremo del conector DB9 hembra, se debe conectar una resistencia terminadora de unos 120 ohm entre las líneas CANH y CANL, tal y como indica el fabricante del adaptador.

Figura 39 - Adaptador CAN/USB LAWICEL

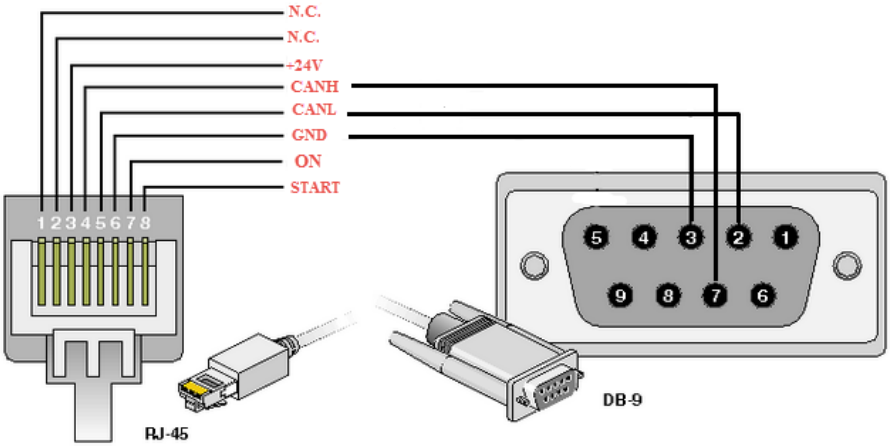

Figura 40 - Cable RJ45 a DB9

4.2.2.1 Software

Como se ha comentado en el Capítulo 3.1, SARA está formada por distintos módulos que se comunican entre sí por medio del bus CAN. La idea de este TFG es crear un nuevo módulo que se comunique con el bus CAN de la silla y envíe/reciba datos de ROS.

Para ello, tomando como referencia una serie de paquetes del Instituto tecnológico de Kyoto [18], se ha creado el paquete “canusb”, el cual tiene la estructura que se indica en la Figura 41.

En la carpeta “/msg” se guardan los mensajes que se usarán en este paquete. El más importante es el mensaje “tramaCAN”. Este mensaje está formado por cada una de las variables que se van a leer del bus CAN y por el identificador del mensaje CAN. Para poder usar este mensaje, es necesario incluirlo en el archivo “CMakeList.txt”

```
uint16 stdId
int32 extId
int16 modo
int16 joyx
int16 joyy
int16 velD
int16 velI
int32 tencA
int64 encA
int32 tencB
int32 encB
int32 bat
int16 SLDD
int16 SLIT
int16 SDI
int16 STD
int16 SLDT
int16 SLID
int16 SDD
int16 STI
int32 ejex
int32 ejey
int32 ejez
int32 SJO
```

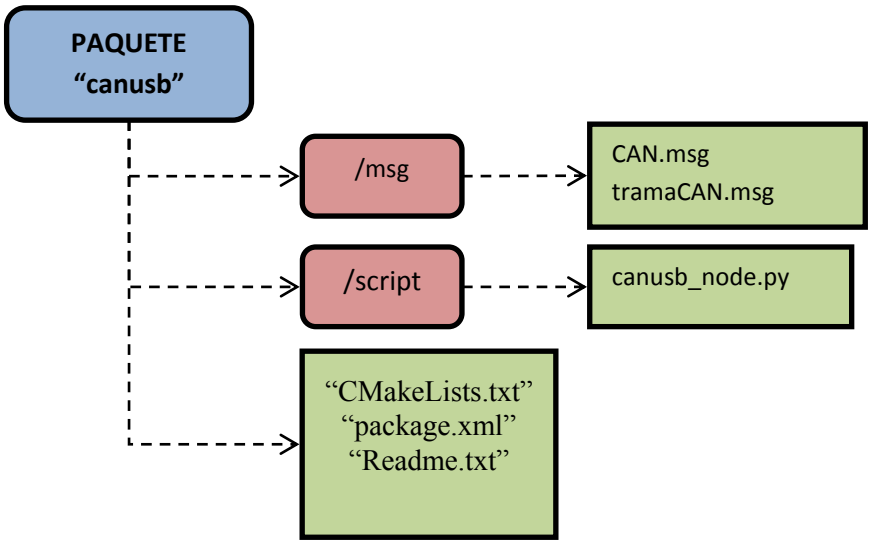

Figura 41 - Estructura del paquete "canusb"

En la carpeta /script está el nodo encargado de realizar la comunicación USB-CAN: “canusb_node.py”. El código de este nodo se puede dividir en varios bloques, cada uno de los cuales se explicarán a continuación.

A) Main.

Lo primero que se hace en el main del nodo es inicializarlo (“rospy.init_node”). Una vez inicializado el programa espera recibir el puerto serie con el cual se quiere realizar la conexión. En el momento que lo recibe crea el objeto CAN, de la clase “CanUSB”, la cual dispone de todos los métodos que se usarán en este programa.

A continuación llama a los métodos “init()”, “versión()” y “serial()” de la clase can para inicializar la comunicación y obtener la versión/número de serie del adaptador CANUSB. Estos dos parámetros se obtienen con el fin de comprobar que se ha realizado correctamente la conexión con el adaptador CANUSB.

Una vez que la conexión ha sido realizada con éxito, espera recibir la velocidad de transmisión y una vez recibida llama al método “setBaud(Sx)” para fijarla. Al adaptador hay que enviarle un comando en el formato *Sx*, siendo x=0,1,2,3,4,5,6,7,8 dependiendo de la velocidad a la que se quiera trabajar.

Con la conexión realizada y la velocidad de transmisión fijada ya se puede iniciar la comunicación, para ello llama al método “start()” del objeto can. A continuación se llama al método “crear_publisher()” el cual crea todos los *publisher* necesarios para publicar los valores leídos por el bus CAN en *topics.*

Por último lanzamos el método *talker*, el cual se encargará de leer y analizar las tramas CAN. En la Figura 42 se muestra el diagrama de bloques del main.

B) Clase “CanUSB”.

Esta clase dispone de todos los métodos necesarios para realizar la comunicación entre el PC y el bus CAN en ROS. Estos métodos son:

- Readline(self). Lee los datos que circulan por el puerto serie.
- Init(self). Se encarga de limpiar la cola de mensajes y de cerrar la conexión en el caso de que estuviese abierta. Para cerrar la comunicación llama al método close().
- version(self). Obtiene la versión del adaptador enviándole el comando “V”.
- serial(self). Obtiene el número de serie del adaptador enviándole el comando “N”.
- setTimestamp(self, enable=0). Se encarga de establecer la marca de tiempo.
- setBaud(self, baud). Establece la velocidad de transmisión enviando el comando Sx.
- Start(self). Abre el puerto e inicia la comunicación enviando el comando “O”.
- Close(self). Cierra la comunicación enviando el comando “C”.
- talker(). Se encarga de llamar al método readline(line) para leer los valores que hay en el puerto serie. Una vez leída llama al método tramas() para que las analice y finalmente llama al método publicar(msg) para que publique los datos en sus correspondientes *topics*.
- crear_publisher(). Crea todos los publisher para publicar los datos que circulan por el bus CAN.

```
pubmodo = rospy.Publisher('/modo', Int16, queue_size=1)
pubjoyx = rospy.Publisher('/joyx', Int16, queue_size=1)
pubjoyy = rospy.Publisher('/joyy', Int16, queue_size=1)
```

```
pubvelD = rospy.Publisher('/velD', Int16, queue_size=1)
pubvelI = rospy.Publisher('/velI', Int16, queue_size=1)
pubtencA = rospy.Publisher('/tencA', Int32, queue_size=1)
pubencA = rospy.Publisher('/encA', Int64, queue_size=1)
pubtencB = rospy.Publisher('/tencB', Int32, queue_size=1)
pubencB = rospy.Publisher('/encB', Int32, queue_size=1)
pubbat = rospy.Publisher('/bat', Int32, queue_size=1)
pubSLIT = rospy.Publisher('/SLIT', Int16, queue_size=1)
pubSLDD = rospy.Publisher('/SLDD', Int16, queue_size=1)
pubSTD = rospy.Publisher('/STD', Int16, queue_size=1)
pubSDI = rospy.Publisher('/SDI', Int16, queue_size=1)
pubSLID = rospy.Publisher('/SLID', Int16, queue_size=1)
pubSDD = rospy.Publisher('/SDD', Int16, queue_size=1)
pubSLDT = rospy.Publisher('/SLDT', Int16, queue_size=1)
pubSTI = rospy.Publisher('/STI', Int16, queue_size=1)
pubSJO = rospy.Publisher('/SJO', Int32, queue_size=1)
pubejex = rospy.Publisher('/ejex', Int32, queue_size=1)
pubejey = rospy.Publisher('/ejey', Int32, queue_size=1)
pubejez = rospy.Publisher('/ejez', Int32, queue_size=1)
```

Para crear un *publisher* se usa “rospy.Publisher”. En primer lugar se le pasa el *topic* en el que cual se va a publicar el dato y el tipo de dato a publicar.

C) Tramas “CanUSB”.

Una vez creado el *publisher*, la forma de realizar la comunicación es mediante la función específica:

```
tramas(self, str).
```

Esta función se encarga de analizar la trama que llega por el puerto serie. Las tramas que llegan tienen el formato de la Figura 43. Lo primero que se hace es crear un mensaje del tipo trama_CAN. Una vez creado se guarda el identificador del mensaje en la variable “stdId”, sabiendo que ocupa los bits del 1 al 4. Evaluando este identificador, guardamos el dato en las variables que corresponda.

A continuación se analizara cada una de las tramas que llegan por el bus CAN. En la Tabla 4 se pueden ver cuál es el identificador de cada trama y que datos la forman.

- Trama con *Id* 272 (Figura 44): En esta trama se almacena el modo del joystick, los valores x e y del joystick y la velocidad de las dos ruedas.

  Para formar los datos, se concatenan los bytes que forman la trama.

```
self.msg.modo = int(str[15:17] + str[13:15],16)
self.msg.joyx = int(str[11:13] + str[9:11],16)
self.msg.joyy = int(str[19:21] + str[17:19],16)
self.msg.velD = int(str[5:7],16)
self.msg.velI = int(str[7:9],16)
```

- Trama con *Id* 257 (Figura 45): En esta trama se almacena el valor del encoderA y el tiempo en el que está funcionando.

  Los datos se obtienen concatenando bytes de uno en uno.

```
self.msg.tencA = int(str[11:13] + str[9:11] + str[7:9] + str[5:7], 32)
self.msg.encA = int(str[19:21] + str[17:19] + str[15:17] + str[13:15],
32)
```

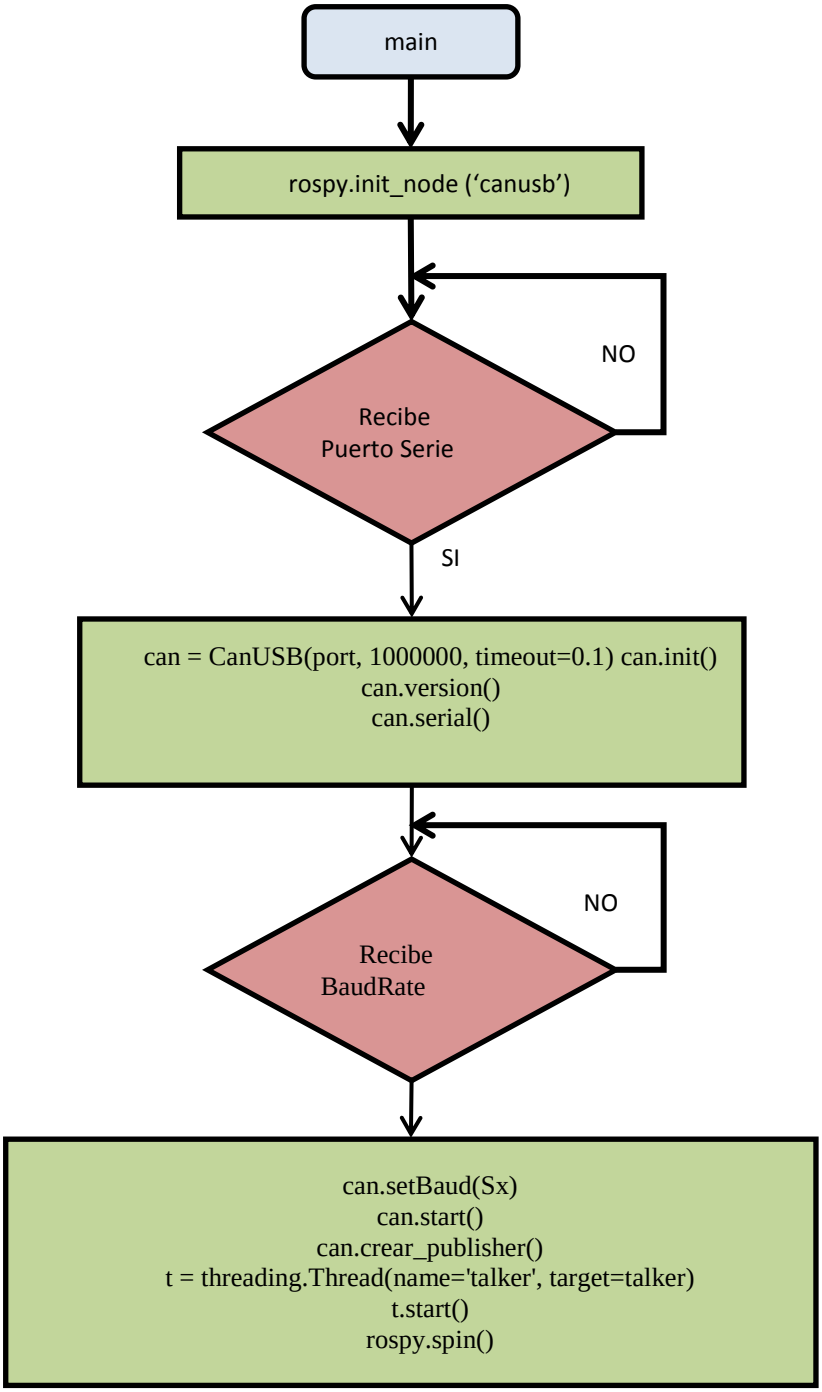

Figura 42 - Diagrama de bloques main

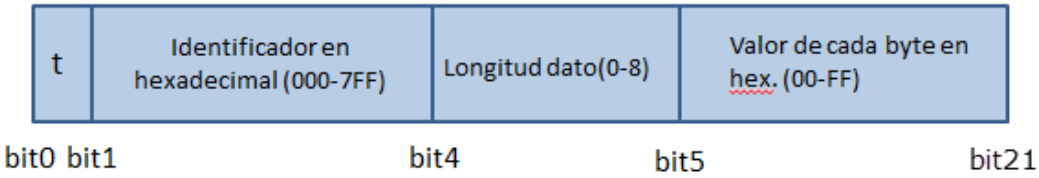

Figura 43 - Formato de las tramas CAN

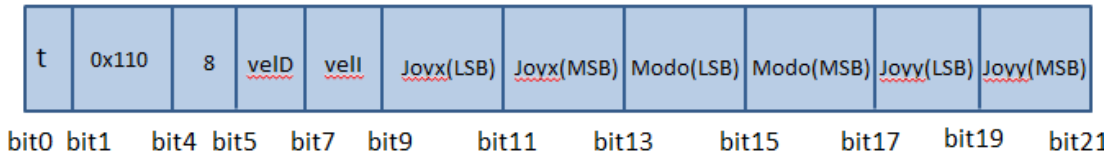

Figura 44 - Formato de la trama con *Id*=272

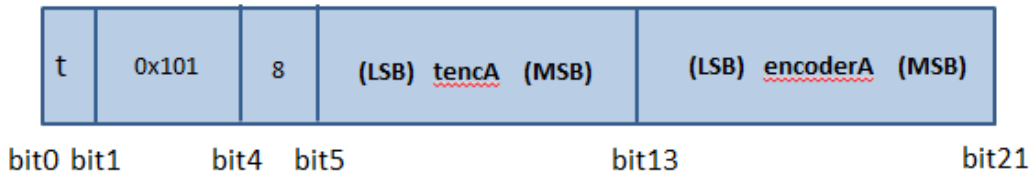

Figura 45 - Formato de la trama con *Id*=257

- Trama con *Id* 258 (Figura 46): En esta trama se almacena el valor del encoderB y el tiempo en el que está funcionando.

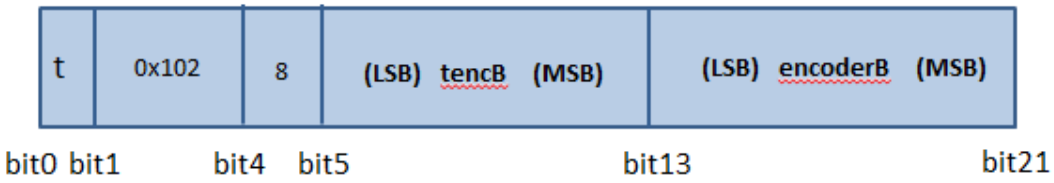

Figura 46 – Formato de la trama con *Id*=258

Los datos se obtienen de la misma forma que en los demás apartados.

```
Self.msg.tencB = int(str[11:13] + str[9:11] + str[7:9] + str[5:7], 32)
self.msg.encB = int(str[19:21] + str[17:19] + str[15:17] + str[13:15],
32)
```

- Trama con *Id* 528 (Figura 47). En esta trama se almacena el valor de la batería en voltios.

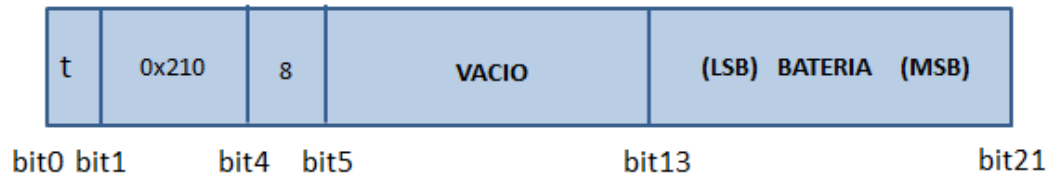

Figura 47 - Formato de la trama con *Id*=528

```
self.msg.bat = int(str[19:21] + str[17:19] + str[15:17] + str[13:15], 32)
```

- Trama con *Id* 513 (Figura 48). Los sensores de ultrasonidos se leen en dos tandas con el fin de que nunca se lean al mismo tiempo dos sensores adyacentes, ya que podrían interferir unos con otros. En esta trama están almacenados los sensores de la primera lectura.

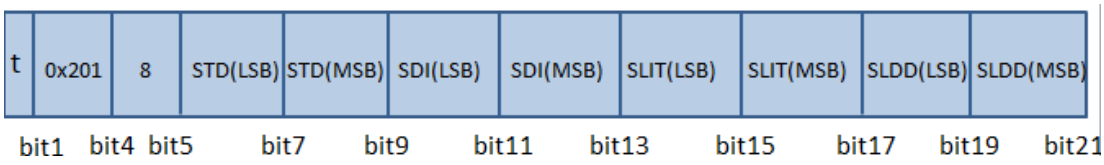

Figura 48 – Formato de la trama con *Id*=513

```
self.msg.STD = int(str[7:9] + str[5:7], 16)
self.msg.SDI = int(str[11:13] + str[9:11], 16)
self.msg.SLIT = int(str[15:17] + str[13:15], 16)
self.msg.SLDD = int(str[19:21] + str[17:19], 16)
```

- Trama con *Id* 514 (Figura 49). En esta trama están almacenados los sensores de la segunda lectura.

```
self.msg.SLDT = int(str[19:21] + str[17:19], 16)
self.msg.STI = int(str[15:17] + str[13:15], 16)
self.msg.SLID = int(str[11:13] + str[9:11], 16)
self.msg.SDD = int(str[7:9] + str[5:7], 16)
```

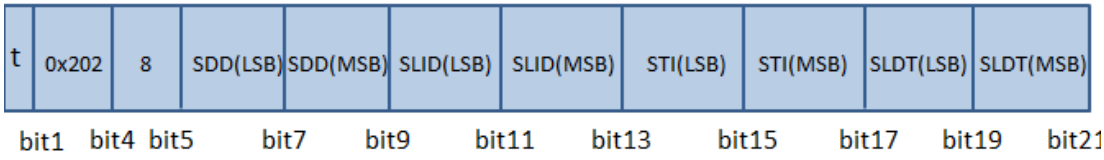

Figura 49 – Formato de la trama con *Id*=514

- Trama con *Id* 515 (Figura 50). En esta trama se almacena el valor del sensor que está colocado en el joystick y del ejex, ejey, ejez del acelerómetro.

```
self.msg.ejez = int(str[19:21] + str[17:19], 32)
self.msg.SJO = int(str[15:17] + str[13:15], 32)
self.msg.ejex = int(str[11:13] + str[9:11], 32)
self.msg.ejey = int(str[7:9] + str[5:7], 32)
```

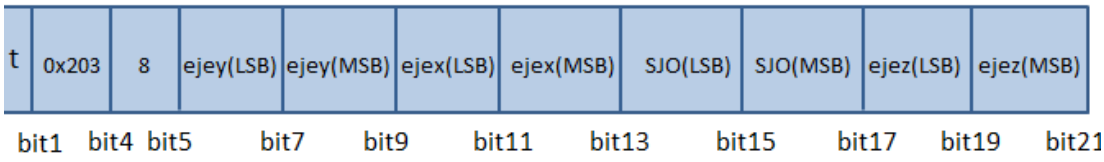

Figura 50 - Formato de la trama con *Id*=515

D) Publicación de tramas “CanUSB”.

Finalmente, para publicar la trama “CanUSB” se utiliza una última función que se muestra y se describe a continuación:

Publicar(self,msg).

Esta función recibe el mensaje del tipo “tramaCAN” y publica, en función de la *Id* de este el dato, en su correspondiente *topic.* Su código se puede ver a continuación.

```
if self.msg.stdId == 272:
  pubmodo.publish(self.msg.modo)
  pubjoyx.publish(self.msg.joyx)
  pubjoyy.publish(self.msg.joyy)
  pubvelD.publish(self.msg.velD)
  pubvelI.publish(self.msg.velI)
elif self.msg.stdId == 257:
  pubtencA.publish(self.msg.tencA)
  pubencA.publish(self.msg.encA)
  pubtencB.publish(self.msg.tencB)
  pubencB.publish(self.msg.encB)
elif self.msg.stdId == 528:
  pubbat.publish(self.msg.bat)
elif self.msg.stdId == 513:
  pubSLIT.publish(self.msg.SLIT)
  pubSLDD.publish(self.msg.SLDD)
  pubSTD.publish(self.msg.STD)
  pubSDI.publish(self.msg.SDI)
elif self.msg.stdId == 514:
  pubSLID.publish(self.msg.SLID)
  pubSDD.publish(self.msg.SDD)
  pubSLDT.publish(self.msg.SLDT)
  pubSTI.publish(self.msg.STI)
elif self.msg.stdId == 515:
  pubSJO.publish(self.msg.SJO)
  pubejex.publish(self.msg.ejex)
  pubejey.publish(self.msg.ejey)
  pubejez.publish(self.msg.ejez)
```

Capítulo 5

Resultados experimentales

En este capítulo se muestran los resultados obtenidos tanto de la simulación como de la integración física de la incorporación de ROS a SARA.

En primer lugar se tratará la simulación, explicando cómo se lanza el correspondiente paquete diseñado para modelar SARA y analizando los resultados obtenidos en varias situaciones.

Por último se mostrará la integración física, explicando cómo conectarse con el adaptador CAN/USB a la silla de ruedas, como ejecutar el paquete que se encarga de implementar la comunicación con ésta y cómo acceder a los datos leídos del bus CAN en ROS.

5.1. Resultados de la integración visual

En este apartado se expondrán los resultados de la simulación de la silla de ruedas en la plataforma ROS.

Como se explicó en apartado 4.2.1, para realizar la simulación en STDR, es necesario tener un mapa del entorno de simulación en formato ".yalm" y un modelo del robot en formato ".yalm". El mapa usado en la simulación es el llamado "sparse_obstacles.yalm" (Figura 51).

El modelo de SARA en el simulador STDR se ha creado por el método visual, explicado en el apartado 4.2.1. Este modelo llamado "*robot0"*, está formado por 8 sensores de ultrasonidos, dos de ellos situados en la parte delantera, dos en la trasera y otros dos a ambos lados de la silla, de modo equivalente a como se ubican en SARA.

Los sensores de ultrasonido se han configurado, tal y como especifica su, con una distancia máxima de medida de 6m, una distancia mínima de 0.15m y un cono de apertura de 30 grados.

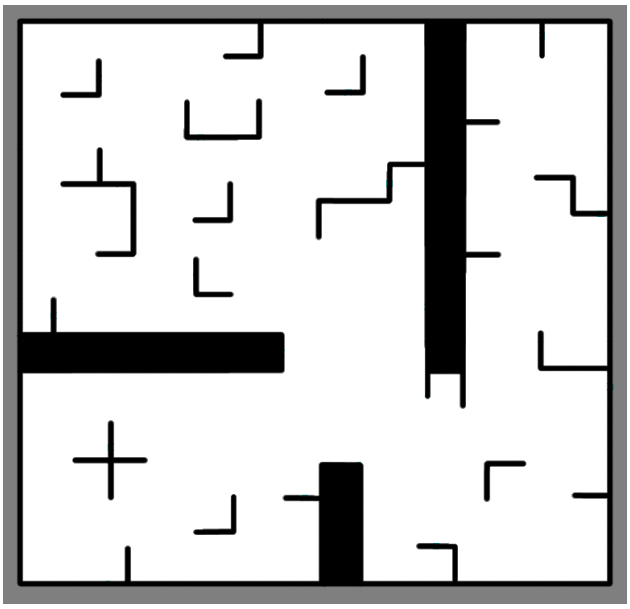

Figura 51 - Mapa de entorno para la simulación STDR

Una vez creado se exporta al archivo “sara_robot.yaml”.

Para arrancar la simulación, hay que lanzar el archivo “sara_in_stdr.launch”, introduciendo la siguiente línea en el terminal de Linux:

```
roslaunch sara_sim_stdr sara_in_stdr.launch
```

Como respuesta a dicho comando, ROS muestra por el terminal qué nodos pasan a ejecutarse al lanzar el archivo “.launch”, tal y como se observa en la Figura 52.

Una vez lanzado éste, se abrirá la GUI del simulador STDR y la herramienta de visualización RVIZ. En la GUI de STDR se ve la simulación, mostrando el modelo “robot0” y el mapa de entorno definido. De la pestaña robot0 cuelgan los sensores creados con todas sus características.

La herramienta RVIZ sirve para visualizar la simulación y con ella se puede ver el modelo URDF del robot moviéndose por el mapa cargado, incluyendo los sensores de ultrasonidos definidos en el modelo “robot0”.

Para que aparezca el modelo URDF de la silla en la visualización de RVIZ es necesaria la transformación TF entre los marcos de coordenadas “robot0” (origen de coordenadas del robot en STDR) y “base_link” (origen de coordenadas del modelo URDF).

```
<node pkg="tf" type="static_transform_publisher" name="sara_broadcaster"
args="0 0 0 0 0 0 1 robot0 base_link 100" />
```

```
pinedo@pinedo-PC: ~/catkin_ws/src/sara_sim_stdr
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://pinedo-PC:35893/

SUMMARY
========

PARAMETERS
 * /robot_description: <?xml version="1....
 * /rosdistro: indigo
 * /rosversion: 1.11.13
 * /use_gui: False

NODES
  /
    joint_state_publisher (joint_state_publisher/joint_state_publisher)
    robot_manager (nodelet/nodelet)
    robot_spawn_pinedo_PC_5338_5238801547685876721 (stdr_robot/robot_handler)
    robot_state_publisher (robot_state_publisher/state_publisher)
    rviz (rviz/rviz)
    sara_broadcaster (tf/static_transform_publisher)
    stdr_gui_node_pinedo_PC_5338_57863396209432504 (stdr_gui/stdr_gui_node)
    stdr_server (stdr_server/stdr_server_node)
    teleop (teleop_twist_keyboard/teleop_twist_keyboard.py)

auto-starting new master
process[master]: started with pid [5353]
ROS_MASTER_URI=http://localhost:11311
```

Figura 52 - Listado de nodos lanzados en la simulación

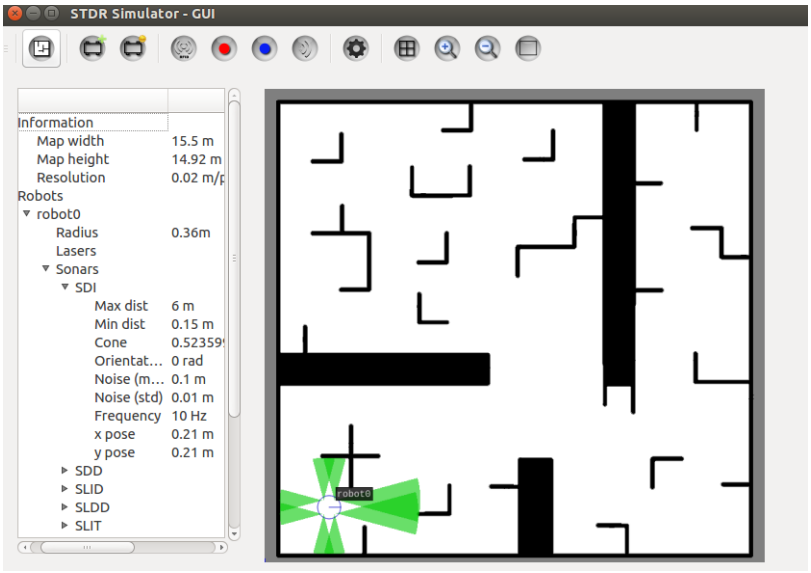

Figura 53 - Simulación SARA en STDR

Como se usa “static_transform_publisher” para realizar una transformación TF entre ambos marcos, el marco “base_link” pasa a ser hijo de “robot0”, tal y como se muestra en la Figura 55. Con esto se consigue que a la vez que se mueve “robot0”, lo haga “base_link”.

Durante la simulación, STDR publica los datos de los sensores de ultrasonidos en *topics*, y por tanto es posible visualizarlos en la herramienta RVIZ usando el modelo “range”. Una vez añadido el modelo es necesario especificar en que *topic* está el dato del tipo “range” a visualizar (Figura 56).

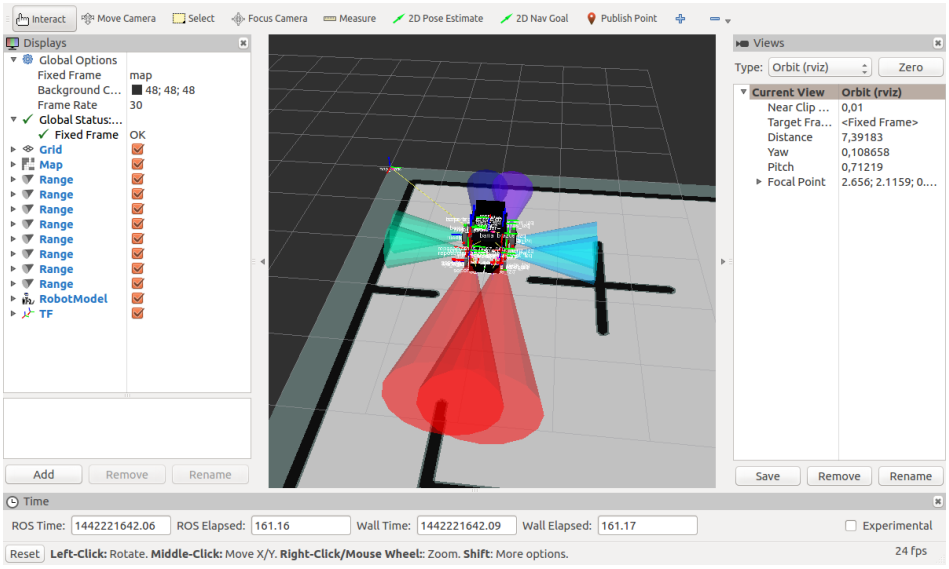

Figura 54 - Visualización de la simulación en RVIZ

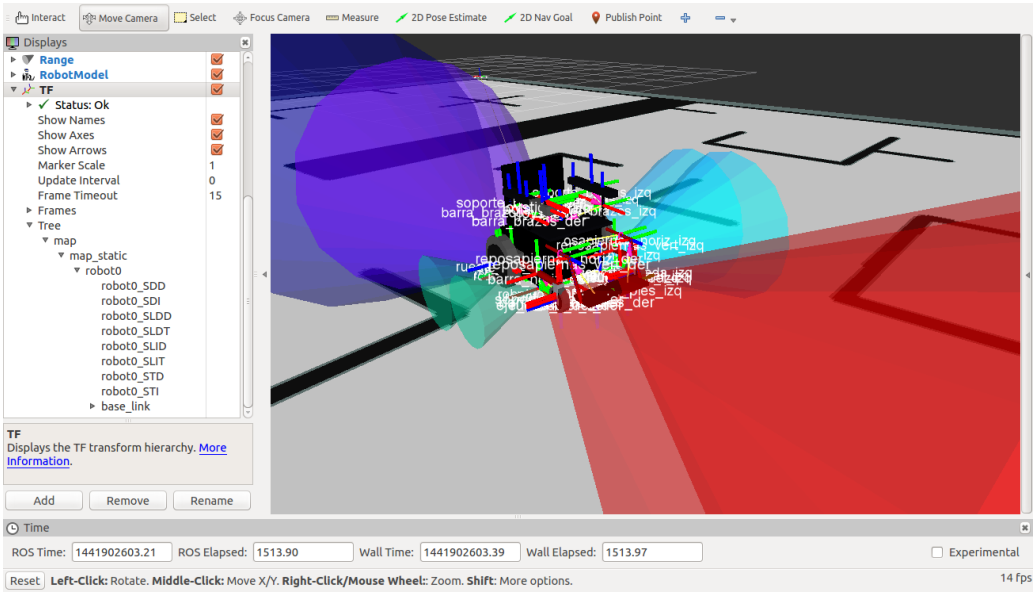

Figura 55 - Representación de la transformación TF entre marcos

Para mover a SARA en el simulador STDR, se ha usado el paquete "teleop_twist_keyboard". Este paquete permite mover el robot por el entorno de simulación con el uso del teclado, enviando comandos de velocidad al simulador STDR. Como se observa en la Figura 57, el paquete permite mover el robot pudiendo aumentar/reducir la velocidad. Para mover el robot es necesario usar las teclas indicadas con el terminal abierto.

Otra opción de gran utilidad que proporciona el simulador STDR, es la de ver la información que están leyendo cada uno de los sensores de ultrasonido en cada momento. El la GUI de STDR, los sensores disponen de dos botones con forma de ojo. El primero de ellos es para visualizar los sensores y el segundo es para visualizar las lecturas de los sensores (Figura 58).

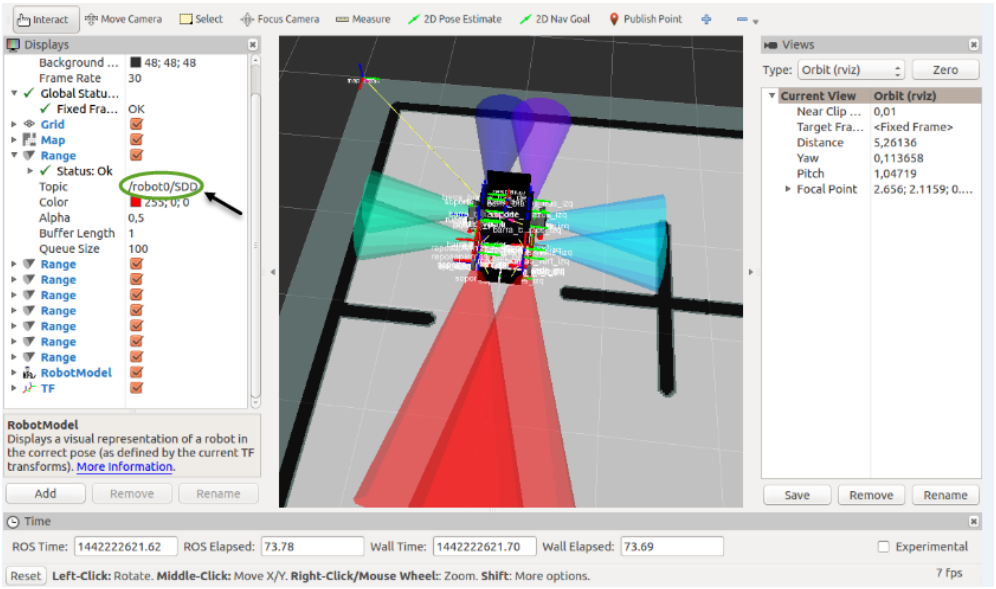

Figura 56 - Modelo "range" en herramienta RVIZ

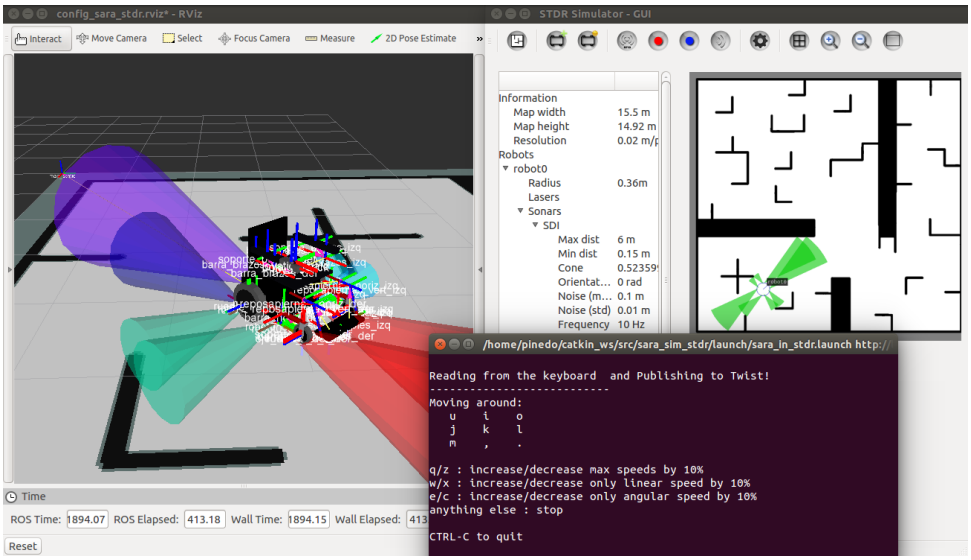

Figura 57 - Simulación STDR: Teleop twist keyoard

De esta forma, se podrá mover a SARA por el mapa de simulación y visualizarlo tanto en RVIZ como en la GUI del simulador STDR, viendo allí los datos que leen los sensores de ultrasonido.

Las pruebas en esta simulación se han centrado en observar como de exactos y realistas son los sensores de ultrasonidos del simulador. Para ello se han capturado varios momentos en la simulación, que se analizan a continuación.

Antes de analizar la simulación, en la Figura 59 se puede ver un esquema donde aparecen situados cada uno de los sensores que incorpora la silla.

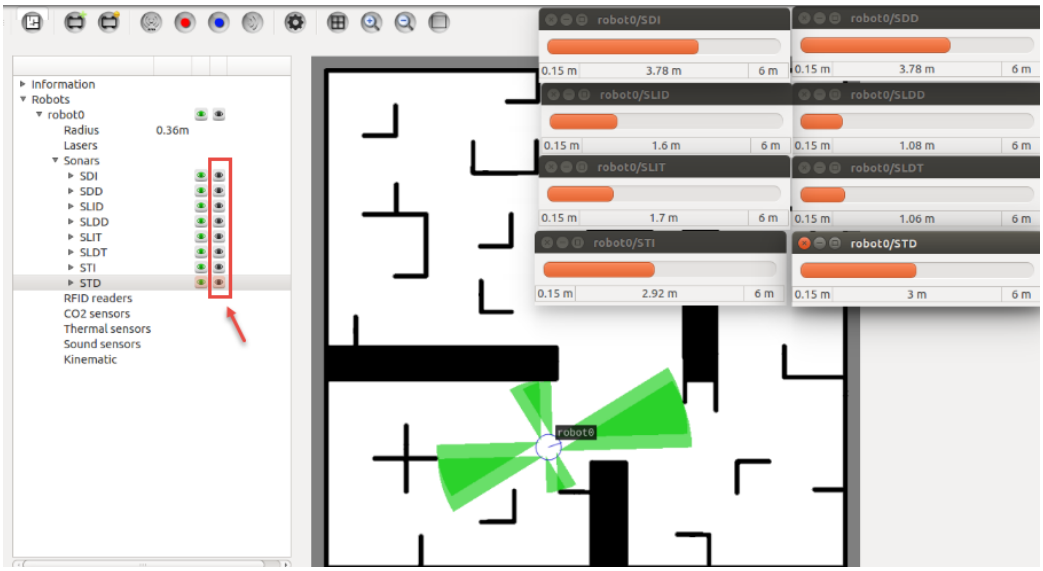

Figura 58 - Lectura de los sensores en GUI STDR

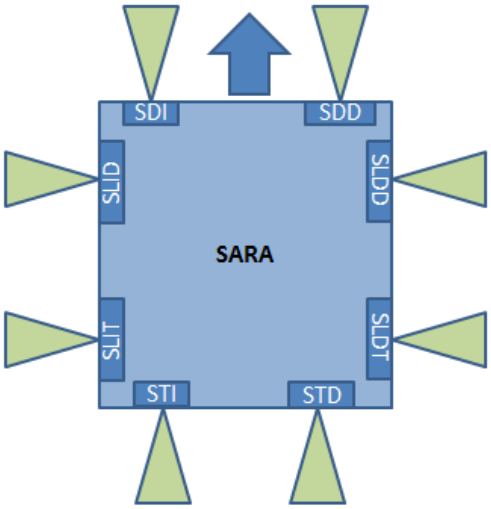

Figura 59 - Colocación de los sensores en SARA

En la Figura 60 se puede ver una primera captura de la simulación de SARA. Los sensores aparecen representados por un cono con la distancia medida.

Los sensores de ultrasonidos envían un haz ultrasónico que rebota al alcanzar un obstáculo. En función del tiempo que pase desde que se envía el haz hasta que vuelve al sensor, se calcula la distancia al objeto. Esto supone que un sensor de ultrasonidos no detecta bien las esquinas ni los objetos que están alejados de la trayectoria del sensor.

En este primer caso, se observa como los sensores laterales izquierdos detectan la pared correctamente. Sin embargo, el sensor delantero derecho detecta la pared lateral cuando en el caso real no lo haría, ya que el eco ultrasónico no volvería al sensor al no rebotar perpendicularmente con este objeto. Lo mismo ocurre con los sensores laterales derechos.

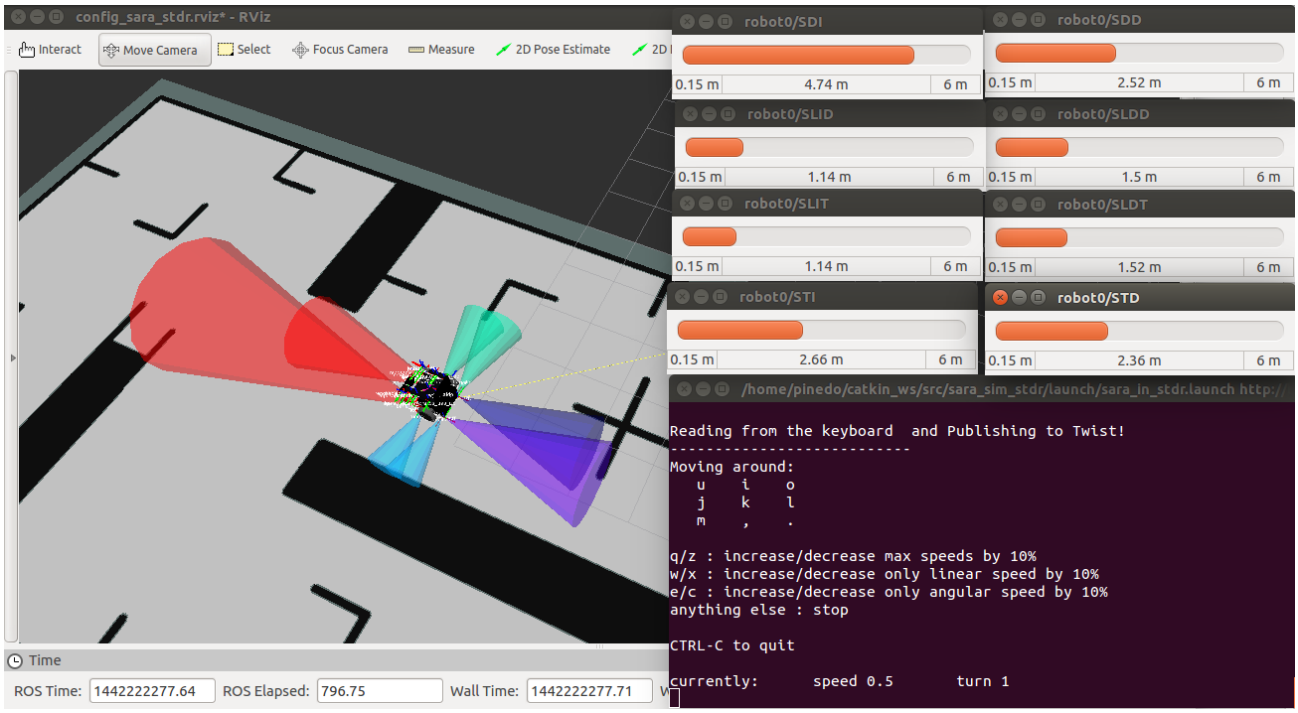

Figura 60- Primera captura de la simulación STDR

En la Figura 61 se observa una segunda captura de la simulación. En este caso los sensores delanteros y los laterales derechos detectan bien la pared, pero los laterales izquierdos en el caso real no lo harían, ya que ambos están detectando esquinas de las paredes.

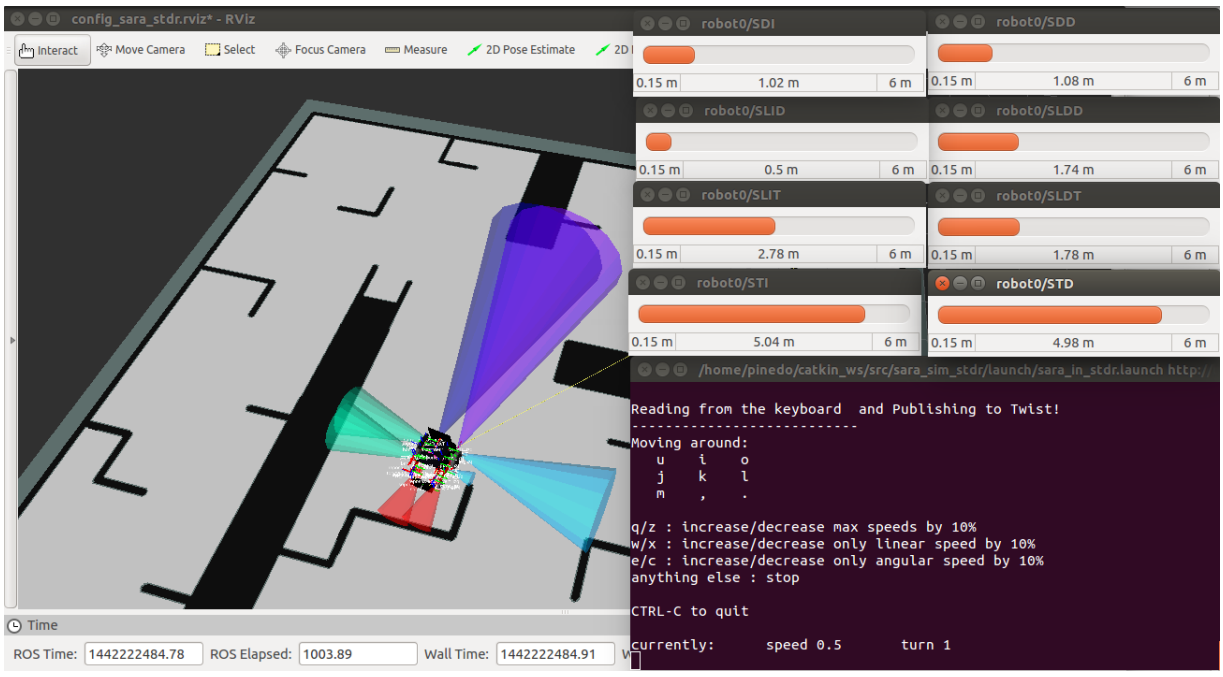

Figura 61 - Segunda captura de la simulación STDR

5.2. Resultados de la integración física

En este apartado se expondrán los resultados de la comunicación entre el bajo nivel de SARA, a través del bus CAN, y ROS. Antes de ejecutar el paquete "canusb", hay que ejecutar el Ros Master e indicarle a ROS cual es el puerto con el que se quiere realizar la conexión y con qué velocidad de transmisión se quiere trabajar.

Para ejecutar el ROS Master hay que introducir en el terminal de Linux el comando "roscore", para lo cual es necesario dar permisos de super-usuario en Linux a la ejecución con el comando "sudo –s".

Para establecer el puerto serie, hay que fijar el parámetro "port", usado en el nodo "canusb_node". Para ello, hay que introducir por el terminal de Linux el siguiente comando:

```
rosparam set /canusb/port /dev/ttyUSB0
```

Para después, fijar la velocidad de transmisión, como sigue:

```
rosparam set /canusb/baud 1m
```

Y finalmente ejecutar el nodo, de este modo:

```
rosrun canusb canusb_node.py
```

Al ejecutar este nodo se muestra por el terminal de Linux la versión y el número de serie del adaptador USB/CAN, los mensajes de depuración y las tramas que se reciben por el puerto serie (Figura 62)

Todos los datos que llegan a partir de ese momento por el bus CAN están siendo publicados en ROS.

Para comprobarlo, viendo la lista de *topics* publicados en cada momento se usa el comando "rostpic list" (Figura 63). Por otro lado, para leer los datos publicados en los *topics* se usa el comando "rostopic echo /topic". Por ejemplo si se desea leer el valor del sensor delantero izquierdo se ha de introducir el comando "rostopic echo /SDI", obteniendo un resultado como el mostrado en la Figura 64.

Finalmente, y de cara a comprobar que las lecturas del bus CAN son correctas, se ha colocado la silla de ruedas en un pasillo y se han leído los valores de los sensores de ultrasonidos. Como se aprecia en la Figura 65, el sistema ROS recibe perfectamente la detección de las paredes del pasillo.

```
root@pinedo-PC:~/catkin_ws# rosrun canusb canusb_node.py
[INFO] [WallTime: 1442239958.945403] Parameter /canusb/port has value /dev/ttyUSB0
Version: V1011
Serial Number: NX345
[INFO] [WallTime: 1442239959.061317] Disable timestamp: okk!
[INFO] [WallTime: 1442239959.064178] Parameter /canusb/baud has value 1m
[INFO] [WallTime: 1442239959.065197] Start: Puerto abierto OK
[INFO] [WallTime: 1442239959.066069] Crea publisher
t110802020E020100FC00
t1028AB7C050062EF1A00
t20387100FDFFD5FFF2FF
t2028C2006900AB001D00
t101899EE0400D7EF1A00
t110802020E020100FC00
t1028EC7C05004BF31A00
t101899EE0400BFF31A00
t2018AC00A30167006300
t110802020F020100FC00
t10282E7D050032F71A00
```

Figura 62 - Ejecución del paquete "canusb"

```
pinedo@pinedo-PC:~$ rostopic list
/SDD
/SDI
/SJO
/SLDD
/SLDT
/SLID
/SLIT
/STD
/STI
/bat
/ejex
/ejey
/ejez
/encA
/encB
/joyx
/joyy
/modo
/rosout
/rosout_agg
/tencA
/tencB
/velD
/velI
```

Figura 63 - Rostopic list

```
pinedo@pinedo-PC:~$ rostopic echo /SDI
data: 261
---
data: 260
---
data: 260
---
data: 260
```

Figura 64 - Lectura del sensor SDI bus CAN

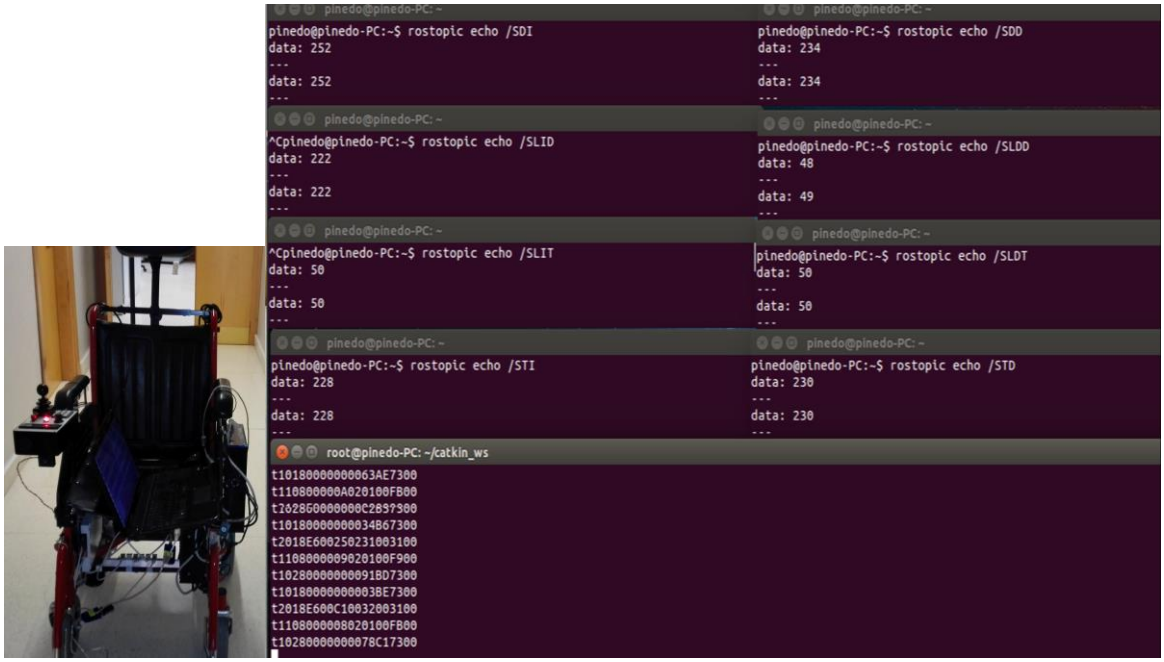

Figura 65 - Lectura sensores de ultrasonidos

Capítulo 6

Conclusiones y trabajos futuros

En este capítulo se exponen las conclusiones obtenidas al finalizar el trabajo y los posibles trabajos futuros.

6.1. Conclusiones

Este TFG surgió con la idea de añadir un nuevo nodo a la silla, en el que se pudieran usar paquetes software típicos en sistemas de navegación autónoma. Para lograr este objetivo se eligió emplear ROS.

ROS es un framework en pleno crecimiento, siendo utilizado cada vez más por grupos tecnológicos. Esta plataforma pone a disposición del investigador cientos de paquetes, algoritmos, robots y una gran cantidad de tutoriales que permiten aprender sus nociones básicas. Aun así, es una plataforma en desarrollo, por lo que hay muchos paquetes, herramientas y algoritmos muy poco documentados.

Tener la silla de ruedas modelada en ROS puede resultar bastante útil para futuras líneas de trabajo, ya que es la base para poder realizar el modelo completo con sus propiedades cinemáticas y dinámicas de cara a poder realizar simulaciones bastante realistas con GAZEBO.

En cuanto a la simulación, se probaron varios simuladores 2D en los que simular el comportamiento de SARA. Se optó por STDR ya que es bastante sencillo de usar y funciona bastante bien. La contra de STDR es, como se ha comentado anteriormente, que es algo inestable en simulaciones más potentes, ya que está en fase de desarrollo.

Pero sin duda lo más importante de este TFG era implementar la comunicación entre el bus CAN y ROS. Conseguir leer los datos que circulan por el bus CAN y publicarlos en *topics* permite al investigador poder empezar a incorporar algoritmos de navegación reactiva en SARA.

6.2. Trabajos futuros

A partir del desarrollo de este TFG se proponen las siguientes mejoras o modificaciones futuras:

- Modificar el modelo URDF, incluyendo sus propiedades cinemáticas y dinámicas.
- Con el modelo URDF completado, se puede empezar a realizar simulaciones más potentes en GAZEBO.
- Una vez se tienen los datos de la velocidad de las ruedas en topics, se puede calcular la odometría de la silla y representar en RVIZ la silla moviéndose. También se podría representar en esta herramienta los datos de los sensores, publicándolos como un tipo “Ranger” en vez de un tipo “Integer”.
- Con los tres paquetes creados se pueden empezar a probar algoritmos de navegación tanto en simulaciones como en tiempo real.

Diagramas

En este apartado se muestra el diagrama del modelo URDF de la silla de ruedas.

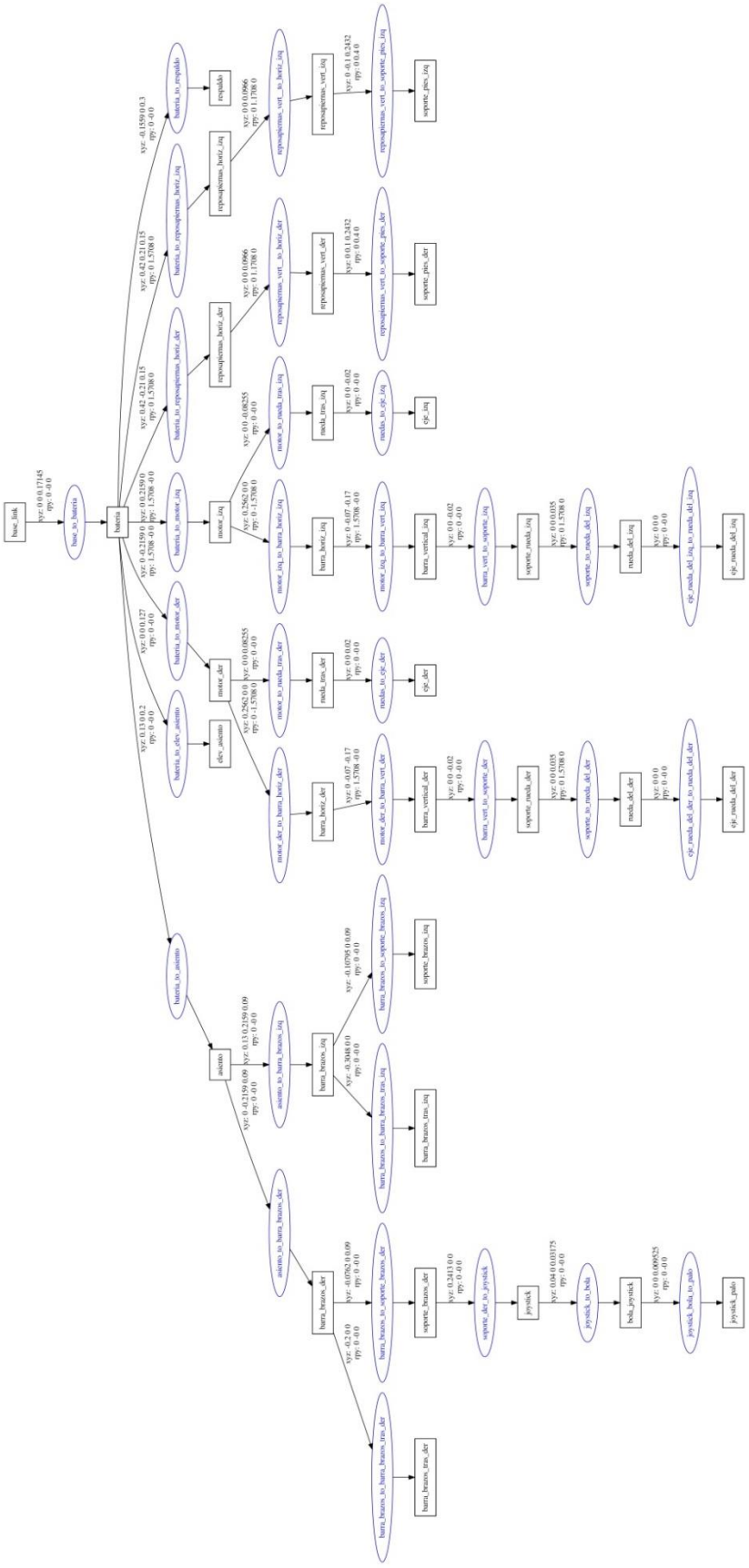

Figura 66 - Diagrama completo modelo URDF

Presupuesto

En esta parte del proyecto se hará una estimación del coste total que supone la ejecución del mismo. En los apartados siguientes aparecen los gastos agrupados según su origen, y en el último apartado se detalla el presupuesto total.

7.1. Recursos hardware

CONCEPTO	PRECIO UNIT.	CANTIDAD	SUBTOTAL
Ordenador portátil Intel Core, i7-4510U(2.0 GHz, 4 MB)	750,00€	1	750,00€
Silla de ruedas SARA	16.000,00€	1	16.000,00€
		TOTAL	16.750,00€

7.2. Recursos software

CONCEPTO	PRECIO UNIT.	CANTIDAD	SUBTOTAL
Microsoft Office Word 2010	60,00€	1	60,00€
Material de oficina	120,00€	1	120,00€
		TOTAL	180,00€

7.3. Coste de la mano de obra

La realización de este proyecto ha sido llevada a cabo por las siguientes personas:

CONCEPTO	PRECIO UNIT.	CANTIDAD	SUBTOTAL
Ingeniero	30,00€	500	15.000,00€
		TOTAL	15.000,00€

7.3. Presupuesto de ejecución material

Es la suma total de los importes del coste de materiales y de la mano de obra.

CONCEPTO	PRECIO
Coste recursos hardware	16.750,00€
Coste recursos software	180,00€
Coste mano de obra	15.000,00€
TOTAL	31.930,00€

7.4. Importe de la ejecución por contrata

A continuación se incluyen los gastos derivados del uso de las instalaciones donde se ha llevado a cabo el proyecto, cargas fiscales, gastos financieros, tasas administrativas y derivados de las obligaciones de control del proyecto. De igual forma se incluye el beneficio industrial. Para cubrir estos gastos se establece un recargo del 22% sobre el importe del presupuesto de ejecución material.

CONCEPTO	PRECIO
Coste total de ejecución material	31.930,00€
22% gastos financieros, beneficios, etc…	6.386,00€
TOTAL	38.316,00€

7.5. Honorarios facultativos

Se ha fijado en este proyecto un porcentaje del 7% sobre el coste total de ejecución por contrata.

Honorarios facultativos……2.682,12€

7.6. Presupuesto total

CONCEPTO	PRECIO
Presupuesto de ejecución material	31.930,00€
Importe de la ejecución por contrata	38.316,00€
Honorarios facultativos	2.682,12€
TOTAL (sin iva)	72.928,12€
IVA (22%)	16.044,18€
TOTAL	88.972,30€

El presupuesto total del proyecto asciende a la cantidad de ochenta y ocho mil novecientos setenta y dos euros con treinta centimos.

Alcalá de Henares a 21 de Septiembre de 2015.

Firmado: David Pinedo Ávila

Graduado en Ingeniería Electrónica de Comunicaciones

Manual de usuario

Actualmente existen varias distribuciones de ROS disponibles: ROS *JadeTurtle*, ROS *Indigo Igloo*, ROS *Hydro Medusa*, ROS *Groovy Galapagos*, etc…

Se recomienda instalar ROS Indigo, ya que junto a Jade es de los últimos que se desarrollaron y es el más estable. En este apartado se puede encontrar un manual de usuario que explica cómo instalar y configurar ROS para poder ejecutar los paquetes creados correctamente.

8.1. Configuración previa de Ubuntu

Antes de instalar ROS, es recomendable realizar una configuración previa de Ubuntu.

Para ello, se debe configurar el equipo para aceptar el software desde "packages.ros.org":

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

8.2. Instalación ROS Indigo

Para instalar ROS Indigo hay que seguir los siguientes pasos:

1) Se debe poner al día el índice de paquetes debían.

```
sudo apt-get update
```

2) Se instala la versión completa de ROS Indigo.

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

3) Una vez instalado, hay que inicializar rosdep. Permite instalar fácilmente las dependencias del sistema que desea compilar y que se requieren para ejecutar algunos componentes básicos en ros.

```
sudo rosdep init
rosdep update
```

4) A continuación se configuran las variables de entorno

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

5) Por último se recomienda instalar “rosinstall”. Esta herramienta permite descargar muchos árboles de código fuente para los paquetes ROS con un comando.

```
sudo apt-get install python-rosinstall
```

8.3. Configuración directorio de trabajo (ROS Workspace)

Una vez instalado ROS, hay que configurar el directorio en el que se trabajará con ROS.

Antes de crear el directorio de trabajo, es recomendable añadir la siguiente línea en el fichero “.bashrc” :

```
source /opt/ros/indigo/setup.bash
```

El directorio de trabajo usado en ROS se conoce como “catkin_ws” y está formado por tres carpetas: /build , /devel y /src.

Para crear el directorio de trabajo se deben introducir los siguientes comandos en el terminal de Linux:

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
```

Una vez creado, se compila con la orden “catkin_make”.

```
$ cd ~/catkin_ws/
$ catkin_make
```

Referencias

[1] Javier Bellón, “Guiado de una silla de ruedas mediante joystick y soplido por bus CAN”, Trabajo Fin de Carrera Escuela Politécnica Superior, UAH, (Septiembre 2009.)

[2] José Baterrechea, “Guiado semiautomático de una silla de ruedas comandada por un joystick de cabeza a través de bus CAN”, Trabajo Fin de Carrera Escuela Politécnica Superior, UAH, (Septiembre 2009.)

[3] David L. Jaffe. An ultrasonic head position interface for wheelchair control. Journal of Medical Systems, 6 (4) (1982), 337-342. DOI: 10.1007/BF00992877.

[4] H.A. Yanco, et al. Initial Report on Wheelesley: A Robotic Wheelchair System. Proc. of the Workshop on Developing AI Applications for the disabled, Montreal, Canada (1995). DOI: 10.1007/BFb0055983.

[5] S.P. Levine, D.A. Bell, L.A. Jaros, R.C. Simpson, Y. Koren, & J. Borenstein. The NavChair assistive wheelchair navigation system. IEEE Transactions on Rehabilitation Engineering, 7(4) (1999) 443-451.

[6] M. Mazo, J.C. Garcia, et al. Experiences in assisted mobility: the SIAMO project, Proceedings of the 2002 IEEE International Conferences on Control Applications, 2, (2002) 766-771.

[7] Juan Carlos Garcia, Marta Marron, Jesús Ureña, David Gualda. Intelligent Wheelchairs: Filling the Gap between Labs and People. Assistive Technology Research Series. Volume 33: Assistive Technology: From Research to Practice. 2013, Pp. 202-209. DOI: 10.3233/978-1-61499-304-9-202..

[8] Morgan Quigley, Eric Berger, Andrew Y. Ng (2007), STAIR: Hardware and Software Architecture, AAAI 2007 Robotics Workshop.

[9] The Robot Operating System (ROS), http://www.ros.org/ (consultado en septiembre de 2015)

[10] Simulador STDR, http://stdr-simulator-ros-pkg.github.io/ (consultado en septiembre de 2015)

[11] Gestor de gráficos 3D open source, http://www.ogre3d.org/ (consultado en septiembre de 2015)

[12] Conrado Grandal, "Desarrollo de un controlador CANOPEN y su integración en un robot móvil". Universidad Carlos III de Madrid (2012)

[13] Extensible Markup Lenguage (XML), https://es.wikipedia.org/wiki/Extensible_Markup_Language (consultado en septiembre de 2015)

[14] Definición de marcadores XML, http://www.upf.edu/hipertextnet/numero-1/xml.html#4.3 (consultado en septiembre de 2015)

[15] Modelo URDF (ROS): Especificaciones XML, http://wiki.ros.org/urdf/XML/link (consultado en septiembre de 2015)

[16] Modelo XACRO, http://wiki.ros.org/xacro (consultado en septiembre de 2015)

[17] Repositorio github CWRU (*Case Western Reserve University's*) *Robotics. https://github.com/cwru-robotics* (consultado en septiembre de 2015)

[18] Repositorio github Spiralray, Instituto tecnológico de Kyoto. https://github.com/spiralray (consultado en septiembre de 2015)