Universidad Carlos III de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática
Trabajo Fin de grado

Control de un robot con Raspberry Pi

Autor: David Yagüe Cuevas
Tutor: Ángel García Olaya

Madrid, 24 de septiembre 2018

"Science is more than a body of knowledge. It is a way of thinking; a way of skeptically interrogating the universe with a fine understanding of human fallibility"

— Carl Sagan

Agradecimientos

A menudo damos por hecho ciertas cosas que en verdad son las que más merecen nuestra gratitud, es por eso por lo que en esta sección quiero destacar las personas más importantes que me han ayudado con todos mis objetivos y ambiciones, que me han apoyado en cada momento de necesidad. Y es que, sabed que uno puede devolver un préstamo de oro, mas está en deuda de por vida con aquellos que son amables…

Así, en primer lugar, quiero agradecer a toda a mi familia por el apoyo que siempre me ha dado, por estar ahí en todo momento, confiando en mí y aguantando mis inquietudes y manías. Por su infinita paciencia y comprensión, pues sin su ayuda nada de esto hubiera sido posible.

A todos mis amigos, por ofrecerme distintos puntos de vista y animarme al enfrentarme a la duda y la zozobra. A todos aquellos profesionales dispuestos a aguantar mis dudas, y en particular a aquella que me ha ayudado a mejorar hasta límites insospechados, tanto profesional como personalmente.

A mis compañeros, de carrera, del laboratorio; profesores y alumnos que han hecho que este camino sea más sencillo, y cuyo conocimiento albergo en la más alta estima.

Y, finalmente, a Ángel García Olaya, por sus horas de dedicación como tutor de este proyecto, por su profesionalidad y sus consejos.

“Death is inevitable. Our fear of it makes us play safe, blocks out emotion. It's a losing game, without passion you are already dead” ─ M. Payne

Índice general

Agradecimientos II
Índice general III
Índice de Ilustraciones VII
Índice de tablas IX
Índice de Códigos XI
Prefacio 1
Preface 3
Resumen 5
Abstract 6
Capítulo 1: Introducción 7
1.1 Motivación 8
1.2 Objetivos 9
1.3 Estructura del documento 11
Capítulo 2: Estado del arte 13
2.1 Historia de la Robótica 13
2.2 Robótica móvil (Mobile robots) 18
2.2.1 Introducción 18
2.2.2 P3-DX: especificaciones de Hardware 19
2.2.3 Especificaciones de Software 22
2.3 Raspberry Pi 3 modelo B 22
2.3.1 Introducción 22
2.3.2 Alternativas 23
2.3.3 Especificaciones de Hardware 24
2.3.4 Especificaciones de Software 24
2.4 Sensor Kinect 25
2.4.1 Introducción 25
2.4.2 Alternativas 26
2.4.3 Especificaciones de Hardware 27
2.4.4 Especificaciones de Software 27
2.5 SLAM (Simultaneus Localization And Mapping) 28
2.6 Entorno de desarrollo 29
2.6.1 Sistema operativo 29
2.6.2 Github y GitKraken 30

2.6.3 Máquinas virtuales ... 30
2.6.4 Librerías y herramientas ... 31
2.6.5 Lenguajes de programación ... 38
Capítulo 3: Integración y descripción del sistema ... 41
3.1 Introducción ... 41
3.2 Análisis del sistema ... 42
3.2.1 Características funcionales ... 42
3.2.2 Restricciones del sistema ... 42
3.2.3 Entorno operacional ... 44
3.2.4 Especificación de los casos de uso ... 46
3.2.5 Especificación de los requisitos del sistema ... 53
3.3 Diseño del sistema ... 75
3.3.1 Arquitectura del sistema ... 75
3.3.2 Descripción general del sistema ... 76
3.3.3 Descripción de los componentes ... 81
3.3.4 Topics generados y utilizados por el Sistema ... 88
3.3.5 Diagramas de secuencia ... 89
Capítulo 4: Experimentación ... 91
4.1 Introducción ... 91
4.2 Entorno de pruebas ... 91
4.2.1 Hardware ... 91
4.2.2 Software ... 91
4.3 Especificación de pruebas ... 92
4.3.1 Pruebas unitarias ... 92
4.3.2 Pruebas del sistema ... 95
4.4 Metodología ... 98
4.5 Análisis de consistencia ... 98
4.6 Análisis de resultados ... 100
Capítulo 5: Gestión del proyecto ... 107
5.1 Planificación temporal ... 107
5.1.1 Planificación inicial ... 107
5.1.2 Desarrollo real del proyecto ... 109
5.2 Presupuesto ... 111
5.2.1 Presupuesto total ... 111

5.2.2 Desglose presupuestario 111
5.2.3 Resumen de costes 114
Capítulo 6: Marco regulador y ético 115
6.1 Responsabilidades profesionales y éticas 115
6.1.1 Movimiento, Navegación y Mapas 115
6.2 Privacidad y seguridad 115
6.3 Estándares técnicos: lenguajes y herramientas 115
6.3.1 C++ y XML 116
6.3.2 ROS y su estándar 116
6.3.3 Ubuntu (Linux) 116
6.3.4 VMware 116
6.3.5 Librerías utilizadas 117
6.4 Propiedad intelectual 117
Capítulo 7: Impacto Socioeconómico 119
7.1 Impacto directo 119
7.2 Impacto indirecto 119
7.3 Impacto inducido 120
7.4 Impacto medioambiental 120
Capítulo 8: Conclusiones, resultados y trabajos futuros 121
8.1 Resultados 121
8.2 Conclusiones 122
8.3 Conclusiones generales 123
8.4 Trabajos futuros 124
Referencias 126
Anexo A – Instalación y configuración del sistema 132
1. Configuración de la Raspberry Pi 3 132
2. Configuración del portátil de monitorización 135
3. Configuración de la Kinect 136
4. Configuración de las conexiones 137
4.1. Raspberry-Kinect 137
4.2. Raspberry-Robot 137
4.3. Kinect-Robot 138
4.4. Raspberry-Portátil (Monitorización) 139
Anexo B – Estructura del proyecto 139
Anexo C – Manual 140

Anexo D – Tablas Informativas 142
1. Especificaciones de Hardware 142
1.1. Raspberry Pi 3 142
1.2. Plataforma robótica p3dx (Pioneer 3DX) 143
1.3. Sensor Kinect 144
2. Especificaciones de Software 145
2.1. Nodos de ROS 145
2.2. Librerías utilizadas 145
Anexo E – Acrónimos 145
Anexo F – Glosario 146
Anexo G – Summary 148
1. Introduction 148
1.1. Motivation 149
1.2. Goals 150
1.3. State of the art 151
1.3.1. Mobile robots 151
1.3.2. Raspberry Pi 3 152
1.3.3. Kinect 152
1.3.4. SLAM 153
2. Results 154
3. Conclusions 155
3.1. Main conclusions 155
3.2. General conclusions 156
3.3. Future work 157
Anexo H – Documentos y enlaces de interés 158

Índice de Ilustraciones

Ilustración 1 - Plataformas robóticas móviles ... 8
Ilustración 2 - Caballero mecánico de Leonardo ... 14
Ilustración 3 - Pato de Vaucanson ... 14
Ilustración 4 - Máquina textil de Jacquard ... 14
Ilustración 5 - Hiladora giratoria de Hargreaves ... 14
Ilustración 6 - "El Dibujante" ... 14
Ilustración 7 - Autómata de Maillardert ... 14
Ilustración 8 - Brazo robótico Unimate ... 15
Ilustración 9 - Robot DANTE ... 16
Ilustración 10 - Robot Genghis ... 16
Ilustración 11 - Pasado y presente del proyecto ASIMO ... 16
Ilustración 12 - Robot Nao ... 17
Ilustración 13 - Perro robot Aibo ... 17
Ilustración 14 - Robonaut ... 17
Ilustración 15 - Snapdragon Cargo ... 17
Ilustración 16 - Robot SOFIA ... 18
Ilustración 17 - Dron de reconocimiento ... 18
Ilustración 18 - Medidas y dimensiones P3-DX [24] ... 19
Ilustración 19 - Componentes básicos del P3-DX [24] ... 19
Ilustración 20 - Panel de control del P3-DX [24] ... 20
Ilustración 21 - Anillo de sonares [24] ... 21
Ilustración 22 - Distribución de Bumpers [24] ... 21
Ilustración 23 - De izq. a der.: Arduino, Raspberry y BeagleBone ... 23
Ilustración 24 - Partes de la Raspbery Pi [29] ... 24
Ilustración 25 - Cámaras PTZ ... 26
Ilustración 26 - Asus Xtion ... 26
Ilustración 27 - Partes del sensor Kinect [28] ... 27
Ilustración 28 - Coche autónomo de Google ... 28
Ilustración 29 - Explicación gráfica del "Loop Clousure" ... 29
Ilustración 30 - Mapa 2D visualizado en MobileSim ... 36
Ilustración 31 - Interfaz de Mapper ... 36
Ilustración 32 - Interfaz Rviz ... 37
Ilustración 33 - Esquema del entorno operacional ... 45
Ilustración 34 - Diagrama de los Casos de Uso ... 48
Ilustración 35 - Descripción de los componentes ... 76
Ilustración 36 - Esquema general de la pila de navegación de ROS [71] ... 78
Ilustración 37 - Visualización del servicio de mapeo ... 87
Ilustración 38 - Mapa generado tras finalizar el servicio ... 87
Ilustración 39 - Diagrama de secuencia Inicio de ejecución ... 90
Ilustración 40 - Diagrama de secuencia general del sistema de peticiones de tareas ... 90
Ilustración 41 - Gráfica desempeño de las Pruebas Unitarias ... 101

Ilustración 42 - Gráfica desempeño de las Pruebas del Sistema 101
Ilustración 43 - Mapa generado con el Servicio de mapeo.. 103
Ilustración 44 - Vestíbulo Ala B Sabatini 1º planta... 104
Ilustración 45 - Vestíbulo Ala C Sabatini 1º planta... 104
Ilustración 46 - Pasillo Ala C Edificio Sabatini 104
Ilustración 47 - Pasillo Ala B Edificio Sabatini 104
Ilustración 48 - Plano oficial de la sección del edifico Sabatini.................................. 105
Ilustración 49 - Diagrama de Planificación estimada... 108
Ilustración 50 - Diagrama de Planificación Real.. 110
Ilustración 51 - Gestor de SSOO de PINN [30] 132
Ilustración 52 - Puertos y conexiones del p3dx... 138
Ilustración 53 - Cable para la Raspberry 138
Ilustración 54 - Set up de las conexiones 138
Ilustración 55 - Estructura del proyecto en Github 140
Ilustración 56 - Especificaciones físicas Raspberry Pi [3] 143
Ilustración 57 - Dimensiones p3dx [24] 144

Índice de tablas

Tabla 1 - Formato de tabla para la especificación de casos de uso 48
Tabla 2 - Caso de uso CU-00 49
Tabla 3 - Caso de uso CU-01 49
Tabla 4 - Caso de uso CU-02 50
Tabla 5 - Caso de uso CU-03 51
Tabla 6 - Caso de uso CU-04 51
Tabla 7 - Caso de uso CU-05 52
Tabla 8 - Caso de uso CU-06 53
Tabla 9 - Caso de uso CU-07 53
Tabla 10 - Formato de tabla para la especificación de los requisitos 55
Tabla 11 - Requisito funcional 00 55
Tabla 12 - Requisito funcional 01 56
Tabla 13 - Requisito funcional 02 56
Tabla 14 - Requisito funcional 03 56
Tabla 15 - Requisito funcional 04 57
Tabla 16 - Requisito funcional 05 57
Tabla 17 - Requisito funcional 06 57
Tabla 18 - Requisito funcional 07 58
Tabla 19 - Requisito funcional 08 58
Tabla 20 - Requisito funcional 09 58
Tabla 21 - Requisito funcional 10 59
Tabla 22 - Requisito funcional 11 59
Tabla 23 - Requisito funcional 12 59
Tabla 24 - Requisito funcional 13 60
Tabla 25 - Requisito funcional 14 60
Tabla 26 - Requisito funcional 15 60
Tabla 27 - Requisito funcional 16 61
Tabla 28 - Requisito funcional 17 61
Tabla 29 - Requisito funcional 18 61
Tabla 30 - Requisito funcional 19 62
Tabla 31 - Requisito funcional 20 62
Tabla 32 - Requisito funcional 21 62
Tabla 33 - Requisito no funcional 00 63
Tabla 34 - Requisito no funcional 01 63
Tabla 35 - Requisito no funcional 02 64
Tabla 36 - Requisito no funcional 03 64
Tabla 37 - Requisito no funcional 04 64
Tabla 38 - Requisito no funcional 05 65
Tabla 39 - Requisito no funcional 06 65
Tabla 40 - Requisito no funcional 07 65
Tabla 41 - Requisito no funcional 08 66

Tabla 42 - Requisito no funcional 09 ... 66
Tabla 43 - Requisito no funcional 10 ... 66
Tabla 44 - Requisito no funcional 11 ... 67
Tabla 45 - Requisito no funcional 12 ... 67
Tabla 46 - Requisito no funcional 13 ... 68
Tabla 47 - Requisito no funcional 14 ... 68
Tabla 48 - Requisito no funcional 15 ... 69
Tabla 49 - Requisito no funcional 16 ... 69
Tabla 50 - Requisito no funcional 17 ... 69
Tabla 51 - Requisito no funcional 18 ... 70
Tabla 52 - Requisito no funcional 19 ... 70
Tabla 53 - Requisito no funcional 20 ... 71
Tabla 54 - Requisito no funcional 21 ... 71
Tabla 55 - Requisito no funcional 22 ... 72
Tabla 56 - Requisito no funcional 23 ... 72
Tabla 57 - Matriz de trazabilidad Requisito-Requisito ... 73
Tabla 58 - Matriz de trazabilidad Requisito-Caso de Uso ... 74
Tabla 59 - Topics en el sistema ... 89
Tabla 60 - Formato tabular para las Pruebas unitarias ... 92
Tabla 61 - Prueba Unitaria TU-00 ... 93
Tabla 62 - Prueba Unitaria TU-01 ... 93
Tabla 63 - Prueba Unitaria TU-02 ... 94
Tabla 64 - Prueba Unitaria TU-03 ... 94
Tabla 65 - Prueba Unitaria TU-04 ... 95
Tabla 66 - Formato tabular para las Pruebas del sistema ... 95
Tabla 67 - Prueba del Sistema TS-00 ... 96
Tabla 68 - Prueba del Sistema TS-01 ... 96
Tabla 69 - Prueba del Sistema TS-02 ... 97
Tabla 70 - Prueba del Sistema TS-03 ... 97
Tabla 71 - Tabla de consistencias Requisito/Prueba ... 99
Tabla 72 - Costes de personal ... 112
Tabla 73 - Costes material informático ... 113
Tabla 74 - Costes de material fungible ... 113
Tabla 75 - Costes indirectos ... 113
Tabla 76 - Beneficios ... 114
Tabla 77 - Resumen de costes ... 114
Tabla 78 - Especificaciones de hardware Raspberry Pi 3 [5] ... 142
Tabla 79 - Especificaciones de hardware p3dx ... 144
Tabla 80 - Especificaciones de hardware Kinect [38] ... 144
Tabla 81 - Paquetes de ROS utilizados ... 145
Tabla 82 - Librerías utilizadas ... 145

Índice de Códigos

Sección de código 1 - Lanzador de la arquitectura 77
Sección de código 2 - Definición del mensaje p3dx_sensors 77
Sección de código 3 - Definición del srv 81
Sección de código 4 - Launcher para el mapeo 86
Sección de código 5 - Script para configurar el entorno bash 135
Sección de código 6 - Lanzador para los drivers, la transformada estática y el depth_image_to_laserscan 136
Sección de código 7 - Lanzador para el nodo depth_image_to_laserscan 137

Prefacio

En un mundo donde la robótica está anidando en cada pequeño hueco de nuestras vidas, donde la autonomía de las máquinas que nosotros mismos creamos pone a debate la moralidad y la ética del "frío" autómata que se rige por los números y no por el corazón (o eso es lo que dicen muchos…), debemos plantearnos los horizontes a alcanzar, las motivaciones que surgen para su creación y las aportaciones que en última instancia sistemas autónomos y robóticos son capaces de brindar al ser humano. ¿Cómo se llega a controlar la maquinaria que constituye al robot? ¿Cómo la inteligencia artificial es capaz de modelar los comportamientos y planificar las acciones? ¿Cómo son capaces de alcanzar unas metas, guiarse por unos objetivos y resolver tareas de forma eficaz y robusta?

La *RoboCup* [1] fue una manera inteligente y estratégica de plantear estas preguntas para intentar resolverlas en un dominio universal; y es que, al fin y al cabo, a quién no le gusta o no conoce el fútbol. No obstante, el conocimiento humano no se sacia fácilmente… No fue sino las diferentes modalidades de esta competición, uno de los pilares fundamentales a través del cual la robótica se popularizó; y es que ya no solo pretendemos que robots autónomos puedan ganar a jugadores de la liga profesional de fútbol (propósito para 2050) [2] sino que aspiramos a ir más allá, planteando problemas como las misiones de salvación y rescate donde elementos tan importantes como la odometría o la navegación supondrán en un futuro, si cabe, salvar vidas de una posible catástrofe. La navegación y el mapeado es, por tanto, uno de los puntos más interesantes en este campo; la robótica ya no se limita a la creación de partes articuladas para satisfacer un problema específico como atornillar o pintar chapados de vehículos, intenta ahondar más en las posibilidades con sistemas de navegación robustos, sistemas inteligentes y, en general, arquitecturas capaces de dar soporte a la decisión (intentando no caer en la llamada discriminación logarítmica).

Es por todo esto por lo que nos planteamos algunos problemas y paradigmas que incluso hoy en día no llegan a estar resueltos; es por eso por lo que comunidades enteras como la de ROS (*Robots Operating Systems*) [3] nacen en el seno del mundo académico de investigación proponiendo ideales de alto valor como el desarrollo ágil y comunitario. La comunidad aporta sus creaciones, su visión, su paradigma y propone soluciones a problemas a priori triviales para un humano, pero que para un robot autónomo suponen tareas de gran complejidad. Controladores, accionadores, sensores… todo para culminar en una forma de autonomía artificial capaz de servirnos, dando soporte a decisiones robustas y coherentes, en pos de lo que es correcto en situaciones de necesidad.

El Pioneer 3-DX [4] es una de esas muchas plataformas robóticas para el movimiento autónomo. Es el robot con el que se ha trabajado en este proyecto, pero cuyas posibilidades van más allá de las que se verán en las siguientes páginas. Muchas veces estos robots tienen limitaciones, pero eso pasa también con cualquier ser humano: alguien carente de vista no es capaz de ubicarse correctamente, personas discapacitadas no serán

capaces de caminar; robots sin sensores no se podrán guiar en un entorno controlado, sin ruedas o accionadores de movimiento poco podrán hacer, pero la robótica está aquí para esas personas, para intentar ser un soporte, una ayuda para aquellos que más lo necesiten. El Pioneer 3-DX carece de un ordenador interno potente y es por lo que placas de bajo coste como la Raspberry Pi [5] nos ayudarán a conseguir distintas metas en la navegación y planificación de nuestro ayudante, que si bien tiene sus limitaciones (como cualquier ordenador de su tipo) la potencia que brinda será suficiente para llevar a término las distintas tareas para el control y la navegación autónomos (en un entorno controlado).

Preface

In a world where robotics is nesting in every little hole of our lives, where the autonomy of the machines that we create puts into debate the morality and ethics of the "cold" automaton that is ruled by numbers and not by the heart (or that is what many say ...), we must consider the horizons to reach, the motivations that arise for its creation and the contributions that ultimately autonomous and robotic systems are able to provide to the human being. How do you get to control the machinery that constitutes the robot? How artificial intelligence is able to model behaviors and plan actions? How are they able to achieve goals, be guided by objectives and solve tasks efficiently and robustly?

The RoboCup [1] was an intelligent and strategic way to pose these questions to try to solve them in a universal domain; and, after all, who does not like or does not know soccer. However, human knowledge is not easily satiated ... It was not only the different modalities of this competition one of the fundamental pillars through which robotics became popular; and the thing is that we no longer only want autonomous robots to win players in the professional soccer league (purpose for 2050) [2] but we intend to go further, raising problems such as salvation and rescue missions where elements as important as odometry or navigation will in the future, if possible, save lives from a possible catastrophe. Navigation and mapping is, therefore, one of the most interesting points in this field; Robotics is no longer limited to the creation of articulated parts to satisfy a specific problem such as screwing or painting vehicle plating, it tries to delve further into the possibilities with robust navigation systems, intelligent systems and, in general, architectures capable of supporting the decision (trying not to fall into the so-called logarithmic discrimination).

It is because of all this that we set ourselves some problems and paradigms that even today are not resolved; that is why entire communities such as ROS (Robots Operating Systems) [3] are born in the academic world of research, proposing high-value ideals such as agile and community development. The community contributes its creations, its vision, its paradigm and proposes solutions to problems a priori trivial for a human, but supposing for an autonomous robot tasks of great complexity. Controllers, actuators, sensors ... all to culminate in a form of artificial autonomy capable of serving us, giving support to robust and coherent decisions, in pursuit of what is right in situations of need.

The Pioneer 3-DX [4] is one of those many robotic platforms for autonomous movement. It is the robot that has been used on this project, but whose possibilities go beyond those that will be seen in the following pages. Many times, these robots have limitations, but this also happens with any human being: someone without sight is not able to locate correctly, disabled people will not be able to walk properly; robots without sensors cannot be guided in a controlled environment, without wheels or movement actuators can do little, but robotics is here for those people, to try to be a support, an aid for those who need it most. The Pioneer 3-DX lacks a powerful internal computer and that is why low-cost boards like the Raspberry Pi [5] will help us achieving different goals

in the navigation and planning of our assistant, that although it has its limitations (like any computer of its type) the power it provides will be sufficient to carry out the various tasks for autonomous control and navigation (in a controlled environment).

Resumen

En este proyecto se presenta el desarrollo de un sistema de control de una plataforma robótica móvil a través de una Raspberry Pi. Con ayuda del *framework* de ROS (Robots Operating System), se implementa una interfaz sencilla que permite ejecutar tareas a bajo nivel y leer *topics* de ROS para poder modelar comportamientos relativamente complejos, que serán gobernados por un sistema de alto nivel encargado de gestionar la planificación de las tareas. Esta interfaz se modela partir de una arquitectura cliente-servidor que permite no solo leer la información del estado a bajo nivel del robot en el mundo, sino también ejecutar comandos a través de peticiones como el movimiento autónomo y la evasión de obstáculos, el mapeo de un área (SLAM) y la completa monitorización de la propia plataforma. Para ello, se presenta una arquitectura modular compuesta por diversos nodos de ROS que permiten realizar cada tarea y, haciendo uso de la pila de navegación de ROS y las lecturas de sensores como la Kinect, se realiza un primer acercamiento a un sistema de SLAM en tiempo real que, dinámicamente, actualiza la posición de la plataforma móvil en un mapa de costes que va generando y publicando.

Palabras clave que definen el proyecto: Robótica móvil, Raspberry Pi 3, Robots Operating System, Planificación y Navegación, SLAM, Kinematics controllers.

Abstract

This project presents the development of a control system for a mobile robotic platform using a Raspberry Pi. With the help of ROS framework (Robots Operating System), a simple interface is implemented allowing to execute tasks at low level and to read topics from ROS in order to model relatively complex behaviors, which will be governed by a high level system in charge of managing the planning of the tasks. This interface is modelled with a client-server architecture that allows not only to read the information of the robot's low level status in the world, but also to execute commands through requests such as autonomous movement and obstacle avoidance, area mapping (SLAM) and complete monitoring of the platform itself. To this end, a modular architecture is presented consisting of various ROS nodes that allow each task to be performed and, using the ROS navigation stack and sensor readings from a the Kinect, a first approach is made to a real-time SLAM system that dynamically updates the position of the mobile platform on a cost map that is automatically generated and published.

Keywords related to the project: Mobile robots, Raspberry Pi 3, Robots Operating System, Planning and Navigation, SLAM, Kinematics controllers.

Capítulo 1: Introducción

El ser humano siempre ha tenido el don para el imaginario destructivo, y es que poco tiempo ha pasado desde que la inventiva humana exponía aquellas historias de locura y ciencia ficción en las que los robots sometían al ser humano bajo el yugo de la esclavitud y la tiranía, incluso de aquellas creaciones de propósito general, capaces de ejecutar cualquier tarea como un humano o que sentían y se comportaban como nosotros.

Si bien es cierto que estamos muy lejos de esos conceptos (aunque ha habido bastantes y diversas charlas y discusiones acerca de la inteligencia artificial que hoy en día estamos desarrollando y descubriendo), no estamos tan lejos de una robótica que abre infinidad de puertas a campos de estudio e investigación tan dispares como útiles. La domótica es un claro ejemplo, y es que, gradualmente, nos ha ido enseñando de manera eficaz muchas de sus aplicaciones, a cada cual más imaginativa.

Por otro lado, las grandes factorías han desarrollado nuevos métodos de manufactura apoyados por la robótica. Estos sistemas integrados en las cadenas de producción son capaces de manipular objetos con más rapidez que un humano, mejorando la productividad y eficiencia de la industria en general. Así, la robótica comienza a desplazar al humano de puestos de trabajo que conllevan tareas monótonas y aburridas, brindando la oportunidad de centrarse en otros campos de estudio para el desarrollo del conocimiento. Todos nos acordamos de aquella simpática pareja de *C3PO* y *R2-D2* de la saga cinematográfica de *StarWars* [6], con total autonomía de movimientos, percepción completa de su entorno y razonamiento lógico que, aun discutido por algunos, se consideraría totalmente humano; o *Sonny* [7] aquel robot ágil y autónomo, capaz de cuestionar, razonar y bordear las tres famosas leyes de la robótica.

Desafortunadamente, la robótica sobre la que estamos discutiendo aún se encuentra distante de esto, pero no estamos tan lejos de encontrar un sistema de movimiento autónomo que ofrezca respuesta dinámica y más o menos "inteligente" en un entorno controlado. La industria aeroespacial es un claro ejemplo de esto último y de lo importante que va a ser en un futuro el desarrollo y estudio de la robótica móvil tanto en el abismal espacio como en el dominio terrestre; estamos, sin casi darnos cuenta, sumidos en la "carrera espacial" del siglo XXI. Gran cantidad de avances y líneas de investigación se han abierto y se abrirán. Sí bien es cierto que queda mucho por descubrir, podemos aventurarnos a decir que estamos encaminados a un progreso y desarrollo tecnológico que no se ha visto hasta ahora en este campo.

Si nos paramos y observamos con atención, nos daremos cuenta de que hay muchos tipos de robots móviles, algunos de ellos aparecen en la Ilustración 1. La mayoría de ellos ofrecen características comunes como el movimiento teleoperado o la planificación por objetivos, otros, sin embargo, ofrecen aspectos mucho más específicos como el "*path finding*" o el uso de pinzas o ganchos para manipular objetos (¿Quién no conoce el Dominio de los bloques?). Concretamente para este TFG se ha utilizado un modelo de

robot móvil P3-DX capaz de navegar por un entorno controlado haciendo uso de la planificación de objetivos preestablecidos o dinámicamente fijados. Este es uno de los muchos robots y series de robots que pretender abrir vías de investigación y aplicación en distintos campos en los que, antiguamente, no se tenía presencia de la robótica o inteligencia artificial, y que poco a poco nos estamos dando cuenta de que son realmente útiles y prácticos en ellos.

Ilustración 1 - Plataformas robóticas móviles

En cierta manera, los robots móviles han supuesto (y lo siguen haciendo) un cambio drástico en la mentalidad de programadores y constructores, pues éstos no solo han de tener en cuenta la mera estática de la máquina, sino también su dinámica y la manera con la que ésta interactúa con un entorno que pretende simular el mundo real. Y es que, ¿quién no ha visto un coche conducir o aparcarse solo [8] o un dron sobrevolando su ciudad [9]?

Para ejemplificar todo esto, este año hemos tenido la noticia de Leo [10], un robot destinado a gestionar y cuidar el equipaje de los pasajeros del aeropuerto Comodoro Arturo Merino Benítez de Santiago, o Ada [11], el primer robot humanoide que llega a España para servir de guía a los visitantes del Museo Elder de Ciencia y Tecnología de Las Palmas de Gran Canaria. Unos cuantos ejemplos de la extensión de estos dos campos que son la robótica y la inteligencia artificial, que, en la mayoría de los casos, van de la mano para ofrecernos servicios tan útiles como los anteriormente mencionados.

1.1 Motivación

Muchas veces con un conjunto de recursos escaso se puede llegar a creaciones realmente útiles y eficaces. Lo cierto es que la plataforma P3-DX carece de sistemas de navegación y no posee un ordenador de a bordo, por lo que recursos de bajo coste como la Raspberry Pi o sensores como el Kinect pueden ofrecernos alternativas baratas y capaces para llevar a término tareas complejas y elaboradas. La implementación de estos sistemas muchas veces es solo dependiente de la plataforma robótica, pues en lo que

probablemente pueda diferir es en la adición de sensores de mayor calidad, que, si nos fijamos bien, realmente no suponen un verdadero problema, pues la implementación base está ahí y tan solo habría que modificar el conjunto de señales, ruido etc. que se recibe. Los ideales del sistema serán los mismos, adaptados a las nuevas condiciones.

No obstante, resulta interesante ahondar en ciertos problemas que aparecen en este campo y se muestran recurrentes. No todos tenemos la capacidad de las grandes empresas para el desarrollo de este tipo de arquitecturas, y es aquí donde entra el desarrollo comunitario de ROS o el uso de componentes de bajo coste que, aunque no nos garanticen la mejor versión de los sistemas, sí que son capaces de ser sorprendentemente competentes para culminar en los comportamientos robustos y eficaces que siempre buscamos en la navegación robótica. Todos hemos sido testigos de los problemas que surgen de los coches autónomos; ciertamente, su funcionamiento es mucho mejor del que cabría esperar, pues no solo cuentan con datos en tiempo real de sensores, si no con mapas que ya se han construido y que permiten una actuación más segura y controlada. Incluso si miramos un poco más lejos de casa, los propios Rovers lanzados en Marte suponen también una serie de retos, plataformas robóticas en cuyo conjunto de objetivos está el reconocer y mapear una zona. Y es que el SLAM (*Simultaneus Localization And Mapping*) ha surgido como uno de los problemas más recurrentes en robótica, y resultaría evidente pensar que cuantos más recursos destinemos a estos problemas tan complejos, más rápida y eficazmente idearemos maneras de solucionarlos o, en última instancia, de sortearlos.

Así, de manera anecdótica diremos que no solo encontramos este problema en complejos campos como el de la navegación espacial. Hace poco tuve la oportunidad de discutir cómo ciertos robots limpiadores que están tan de moda, son capaces de navegar por el entorno. Para mi sorpresa, resulta que algunos (aunque no muchos) abordan el problema del SLAM, y si eres un poco curioso podrás ver con las aplicaciones del fabricante cómo son capaces de construir su propio mapa y navegar de manera reactiva. Una pequeña curiosidad convertida en motivación, una práctica que se extiende más allá de lo académico y que se adentra en lo más mundano, nuestros hogares; ¿y quién quiere un sistema de navegación autónoma que no funcione bien?

1.2 Objetivos

El objetivo del proyecto inicialmente era poder controlar una plataforma robótica con una placa Raspberry Pi Modelo 3. La plataforma robótica pronto tomó forma y se eligió el P3DX de *MobileRobots*. Sin embargo, como esto parecía un poco limitado y escaso, se propuso algo que se fundamentaba en la idea principal original, pero que añadía algunas capas de complejidad.

En este contexto, se decidió abordar el problema de la navegación y mapeado en la plataforma robótica. Un navegador generalmente se divide en dos capas principales, la primera es un sistema de control que gestiona la asignación de tareas a alto nivel (que

ha sido realizado mediante planificación automática en otro TFG y que por lo tanto no forma parte de los objetivos del proyecto) y la segunda, implementada en este proyecto, que es la que se encarga de gestionar los recursos a bajo nivel, en el robot; movimientos básicos, sensores, *encoders*, etc. Para ello se escogió el *framework* de ROS y se fueron desarrollando una serie de nodos que permitieran, no solo la integración de la plataforma con el módulo de alto nivel elegido, PELEA [12] (o cualquier arquitectura que suponga la implementación de una interfaz de alto nivel), y con el propio ROS, sino también la comunicación entre el microcontrolador de la plataforma y todos los demás sistemas de gestión, generalmente proporcionados por ROS.

De esta manera, se propondría el desarrollo de una arquitectura a bajo nivel capaz de implementar una interfaz sencilla que permitiera el mapeado y la navegación autónoma del P3-DX a través de la ejecución de comandos simples y la publicación de topics de ROS. Estos últimos se proporcionarían a un sistema de alto nivel para ofrecer la capacidad de gestionar la plataforma y monitorizarla.

A continuación, se exponen los objetivos propuestos en la realización de este proyecto. Nótese que están presentados en orden de relevancia, no obstante, todos se tratan con igual importancia.

1. Demostrar la posibilidad de controlar una plataforma robótica móvil mediante una Raspberry Pi utilizando ROS.
2. Comprobado el punto anterior, proporcionar una interfaz sencilla consistente en una serie de tareas a bajo nivel que se puedan utilizar para el control externo a alto nivel, ya sea de forma natural o mediante paradigmas como el de la planificación automática.
3. Abstraer la arquitectura de ROS y proponer comandos y *topics* sencillos que se puedan utilizar rápidamente por un alto nivel.
4. Desarrollar una primera versión de SLAM, basada en las funcionalidades proporcionadas por ROS y utilizando la Kinect. Asumir las limitaciones para este comportamiento extra y demostrar la factibilidad para el mapeo con la Raspberry Pi.

De esta manera, el sistema final supondrá una interfaz de fácil acceso a cualquier aplicación de alto nivel que pretenda generar un plan compuesto por tareas que deberán transformarse a bajo nivel. Este sistema, de carácter modular, estará implementado en base a una serie de nodos que serán los que ejecuten las tareas para el movimiento de la base robótica y el tratamiento del problema del SLAM (mapeo con la Kinect y localización de la base robótica). Así, para abordar la problemática de la comunicación se utilizará una arquitectura Cliente (Alto nivel) Servidor (Bajo nivel). Para tratar el movimiento de la base robótica utilizaremos el microcontrolador del P3DX, ARIA, y el nodo para conectar con ROS llamado RosAria que nos permitirá mandar comandos de velocidad al robot. La pila de navegación de ROS nos ayudará con la evasión de

obstáculos a partir de las lecturas de la Kinect, que también serán utilizadas con el nodo *Gmapping* y *Depth_image_to_laser_scan* para el mapeo con los escaneos láser proporcionados por la trasformación de las lecturas de la Kinect a cargo de este último nodo. Finalmente, el sistema deberá incorporar herramientas para la gestión de la plataforma, así como para la monitorización de la misma que permitirán, en última instancia, evaluar y comprobar que el comportamiento del mismo es seguro y coherente con las expectativas propuestas para este tipo de arquitecturas. Es por esto que se hará uso de la generación de archivos de log en cada parte y de la herramienta Rviz, programa con el que podremos visualizar el comportamiento de la plataforma.

1.3 Estructura del documento

En esta sección se realizará una descripción de cada uno de los capítulos que se van a exponer en este documento, así como la estructura del mismo.

- **Introducción**: En este capítulo se realizará una descripción de los objetivos y las motivaciones que definen este proyecto. Por otro lado, se expondrá la estructura del propio documento.
- **Estado del arte**: En este apartado se expondrá el contexto relacionado con el proyecto. Se realizará un análisis de las tecnologías, del entorno y las aplicaciones de sistemas similares a los de este trabajo. Por otro lado, se hará mención a las novedades e historia del campo de la robótica, así como las especificaciones y usos de los componentes utilizados para la culminación del proyecto.
- **Integración y descripción del sistema**: Este capítulo pretende ser la introducción, definición y análisis del sistema desarrollado. Se podrán encontrar las funcionalidades y capacidades del mismo, los requisitos y casos de uso y, en general, las necesidades para implementar el sistema final. Por otro lado, se hará una detallada descripción de los diferentes componentes del sistema y cómo interactúan unos con otros, así como el proceso de integración de los mismos con ROS.
- **Experimentación**: Este apartado supone la descripción del plan de pruebas. En él se definirán las pruebas realizadas y se hará un análisis general de los resultados obtenidos con el objetivo de preparar adecuadamente la disposición de los resultados finales del proyecto.
- **Gestión del proyecto**: Este capítulo expone los métodos de planificación utilizados para la organización y gestión del proyecto. A su vez se proporcionará un presupuesto del mismo y se contrastará entre la planificación inicial (estimada) y la real, haciendo un análisis de las posibles eventualidades y/o retrasos.

- **Marco regulador y ético**: Este apartado servirá para realizar un análisis detallado de marco legal y ético que afectaría al sistema dentro de España y Europa.
- **Impacto socioeconómico**: Este capítulo pretende ser un estudio acerca del impacto social y económico del proyecto. De esta manera, se hará un análisis de diferentes aspectos dentro de la economía y la sociedad española, así como del impacto medioambiental derivado de la producción de este proyecto.
- **Conclusiones y trabajos futuros**: Este apartado es la última sección del proyecto y su objetivo es realizar un análisis del sistema y de los resultados obtenidos. Se expondrán una serie de conclusiones que emanarán del pensamiento crítico y los resultados en las diferentes fases de prueba; haciendo un contraste con sistemas similares y literatura científica. Finalmente, se propondrán posibles mejoras para el sistema y nuevas líneas de desarrollo en el futuro.

Capítulo 2: Estado del arte

Esta sección pretende ser una introducción para enmarcar el trabajo desarrollado. En primer lugar, se presentará un resumen de la historia de la robótica desde sus inicios hasta los más recientes avances, a continuación, se ofrecerán las especificaciones y características de los medios utilizados que ayudarán a entender la utilización e integración entre los diversos sistemas, y finalmente, se explicará el entorno de desarrollo donde se darán a conocer todas las herramientas, el software y los lenguajes de programación utilizados para llevar a término el trabajo en cuestión.

2.1 Historia de la Robótica

La historia de robótica, como en casi todos los campos de investigación y experimentación, ha sido convulsa y mistificada. El ser humano siempre se ha empeñado en dar vida a seres artificiales que le facilitaran aquellas tareas complicadas o tediosas; muchos consideran a Adán y Eva como los primeros autómatas, creados y programados por Dios. También podemos mirar en la mitología griega, donde encontramos al titán Prometeo, creador del primer ser humano con aquella prodigiosa mezcla de barro y fuego. Ya con esta base literaria se puede ver subyacente la obsesión por crear vida artificial desde el principio de los tiempos, una inquietud que se ha mantenido hasta nuestros días.

La gran mayoría de la gente tiene cierta idea de lo que representa la robótica, saben sus aplicaciones y su potencial, pero no conocen el origen ni de esta rama ni de la propia palabra “robot”. Ciertamente se sabe que sus orígenes se remontan miles de años atrás. Basándonos puramente en hechos registrados a través de la historia diremos que antiguamente los robots eran conocidos como autómatas y la robótica no era tratada como ciencia (la palabra robot surgió mucho después).

Partiendo desde el principio de los tiempos empezamos, cómo no, con la civilización china, donde recogemos una de las primeras descripciones de autómatas en el texto de *Lie Zi* occidentalizado al *Tratado de la perfecta vacuidad* [13] donde se relata un encuentro con un artífice y el rey *Mu de Zhou*. Avanzamos un poco y nos encontramos con textos de Herón de Alejandría: *Pneumatica* [14] y *Automata* [15] donde tropezamos con descripciones de más de 100 máquinas y autómatas.

Por el año 1206 se hace alusión a un caballero mecánico ideado por Leonardo Da Vinci, el cual no se sabe si llego a existir, pero que a posteriori se creó basándose fielmente en los diseños originales y se demostró que era completamente funcional.

Durante los siglos XVII y XVIII, en plena revolución industrial en Europa, se asentaron los pilares de la robótica con registros de muñecos mecánicos y de inventores como Jacques de Vaucanson, constructor del “Pato con aparato digestivo”,

Ilustración 2 - Caballero mecánico de Leonardo

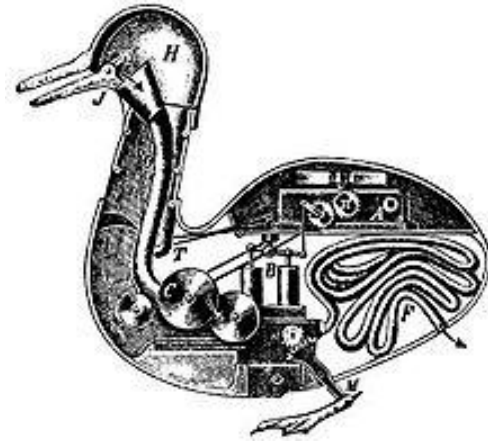


Ilustración 3 - Pato de Vaucanson

James Hargreaves y su hiladora giratoria (1770), Joseph Jacquard y su máquina textil programable (1801),

Ilustración 4 - Máquina textil de Jacquard

Ilustración 5 - Hiladora giratoria de Hargreaves

O la muñeca mecánica de Henri Maillardert, capaz de escribir (1805).

Ilustración 6 - "El Dibujante"

Ilustración 7 - Autómata de Maillardert

Ya sería en el año 1921 cuando aparece el primer autómata de ficción llamado "robot" en el *Rossum's Universal Robots*, obra del escritor checo Karel Capek [16] donde la palabra "robot" se utiliza para referirse a máquinas orgánicas de forma humanoide forzadas a trabajar para el humano. De hecho, en checo, *robota* significa servidumbre o labor forzada.

Posteriormente, en 1942, la revista *Astounding Science Fiction* publica "*Círculo vicioso*" [17] (*Runaround* en inglés) donde Isaac Asimov presenta una historia de ciencia ficción donde da a conocer las famosas Tres leyes de la robótica.

Será a partir de los años 50 del anterior siglo cuando se comience a establecer los fundamentos para la robótica moderna que conocemos hoy en día. Curiosamente uno de los grandes inventos que impulsó el desarrollo de ésta no fue otro que el transistor (1951), con el que William Shockley revolucionó los diseños de circuitos eléctricos y que supuso un increíble crecimiento en el desarrollo computacional y la miniaturización de componentes. A partir de ese momento, la robótica avanza a pasos agigantados: en el año 1956 la compañía Unimation, fundada por George Devol y Joseph Engelberger, lanza el primer robot comercial y en el 61 culminan gran parte de esta iniciativa creando el primer brazo robótico (Unimate) que se instaló en una cadena de montaje de Generals Motors y estableció las bases de la robótica industrial.

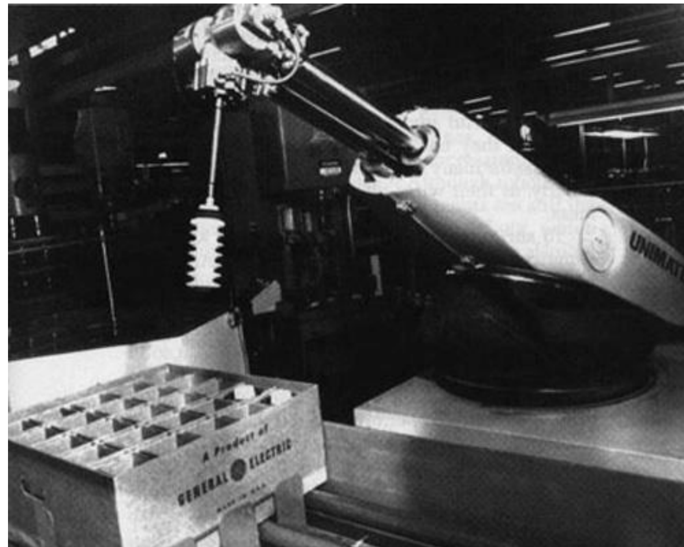


Ilustración 8 - Brazo robótico Unimate

A partir de la década de los 70, el desarrollo y la investigación en el campo de la robótica ha aumentado exponencialmente, consiguiendo diferentes hitos como la creación de PUMA (*Programmable Universal Machine for Assembly*), un brazo robótico para tareas de montaje. En 1982 Isaac Asimov en *El robot completo* [18] (*The complete robot* en inglés) presenta una colección de cuentos de ciencia ficción donde se explican con más ahínco y profundidad moral las tres leyes de la robótica (incluyendo la ley 0, con la se discute la muerte del ser humano a manos de un robot). En el 89 Rodney A. Brooks creó, con ayuda del MIT (*Massachusset Institue of technology*), el

robot Genghis para reconocer la superficie marciana y posteriormente en 1993, ingenieros de la universidad de Carnegie Mellon desarrollan, siguiendo el mismo concepto, el robot DANTE para la exploración del planeta marciano.

Ilustración 9 - Robot DANTE

Ilustración 10 - Robot Genghis

Entre este año y 2005, la empresa nipona, Honda, llegó a desarrollar una serie de humanoides teleoperados, culminando con el ASIMO [19], un robot humanoide capaz de desplazarse de forma bípeda e interactuar con personas.

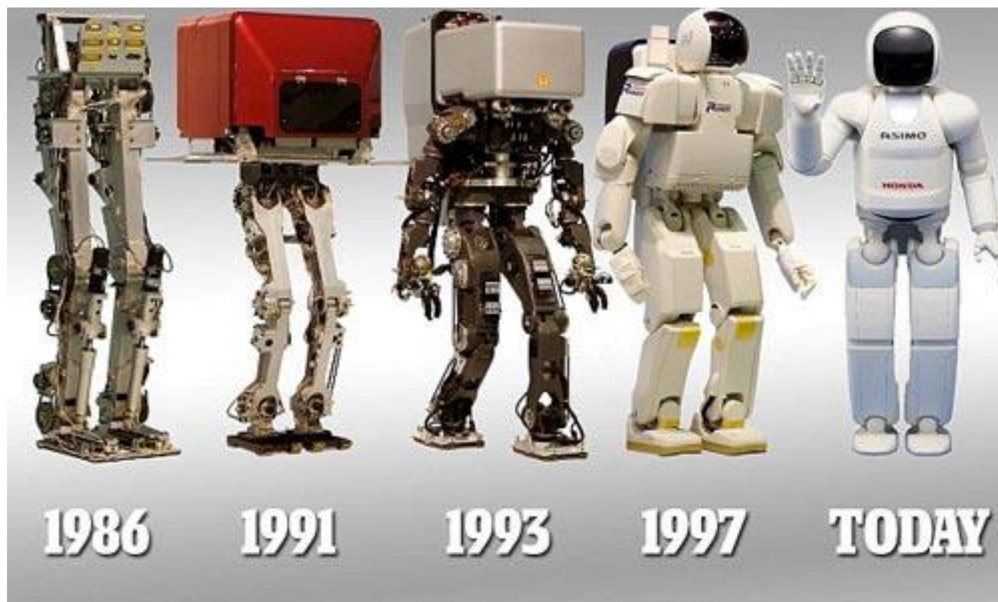


Ilustración 11 - Pasado y presente del proyecto ASIMO

Esto sentaría las bases para que dos años más tarde, concretamente en el año 2007, apareciera el robot humanoide Nao, famoso por su uso en la *RoboCup* que luego pasó a utilizarse para la docencia y la investigación en el campo de la robótica e inteligencia artificial. Un modelo bípedo muy parecido al último ASIMO pero que incluye muchas

mejoras técnicas, ideal para el desarrollo y que acabo sustituyendo al perro robot Aibo de Sony [20] (que fue, de hecho, el primer modelo elegido para la *Robocup*).

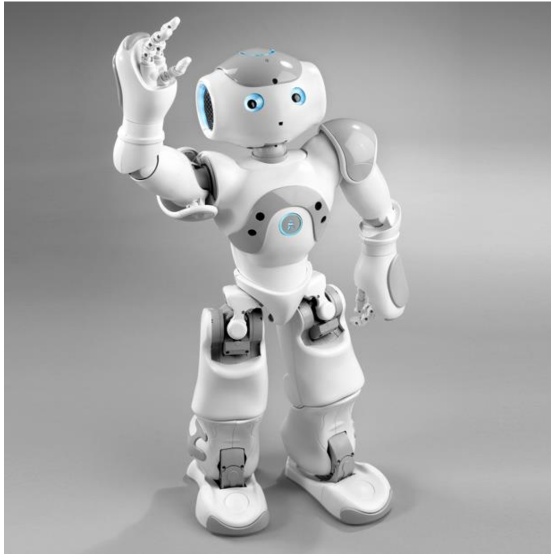


Ilustración 12 - Robot Nao

Ilustración 13 - Perro robot Aibo

Acercándose ya a la primera década de este siglo, en 2011, la NASA envía a la estación espacial internacional al robot *Robonaut* para ayudar a los astronautas residentes allí. En 2013 los medios de comunicación se hicieron eco de la noticia en la que dos robots (*Lingodroids*) diseñaron su propio lenguaje para comunicarse y en 2015 Qualcomm introdujo el *Snapdragon cargo*, un robot que se desplaza por suelo, aire y traslada objetos.

Ilustración 14 - Robonaut

Ilustración 15 - Snapdragon Cargo

Por otro lado, todos hemos conocido el auge de los drones en 2016 y finalmente el hito de SOFIA este 2017 por parte de Hanson Robotics Co. Ltd [21], un robot

humanoide capaz de reconocer y recordar caras, de conversar y producir expresiones; además de ser el primer robot en conseguir una ciudadanía oficialmente, estableciendo formalmente la hipótesis del “Valle inquietante” [22].

Ilustración 16 - Robot SOFIA

Ilustración 17 - Dron de reconocimiento

2.2 Robótica móvil (Mobile robots)

2.2.1 Introducción

Los robots han sido utilizados en la industria desde los años 60 con el objetivo de facilitar diferentes tareas a los seres humanos. Los robots móviles en particular pueden servir para la exploración y el transporte de diferentes mercancías y objetos; sin embargo, hasta el momento de su aparición, no eran tantas las complicaciones que emanaban del propio lugar que ocupaba en el mundo el robot en cuestión. Aunque para el ser humano el hecho de moverse es algo trivial, para el robot móvil no lo es tanto. Un robot móvil necesita saber en todo momento su localización y el espacio que ocupa en un mundo que resulta muchas veces bastante difícil de definir en su lógica. El objetivo final de éstos es, generalmente, lograr seguir consecuentemente una serie de acciones racionales para conseguir realizar una tarea que puede ir desde la simpleza de llegar a un objetivo o posición determinada, hasta mover y colocar una serie de cajas o palés. Esto es, en cierta manera, una de las mayores problemáticas que se pretende solventar en el campo de los robots móviles y donde subyacen la mayoría de los problemas en esta área.

Desde 1995, Omron Adept MobileRobots [23] (anteriormente Adept MobileRobots) ha sido la fuente preferida para la investigación y la educación con robots móviles. Esta empresa está dedicada a la manufacturación de Robots móviles, con todo su hardware, software y accesorios para garantizar la cartografía y navegación autónoma, así como un control flexible y fluido del robot. En este trabajo en particular se va a hacer uso de uno de sus modelos más populares, el P3-DX, cuyas posibilidades y características se describirán en secciones posteriores a esta introducción.

2.2.2 P3-DX: especificaciones de Hardware

Los robots P3-DX son de la familia de los robots móviles. En particular este modelo tiene las siguientes dimensiones:

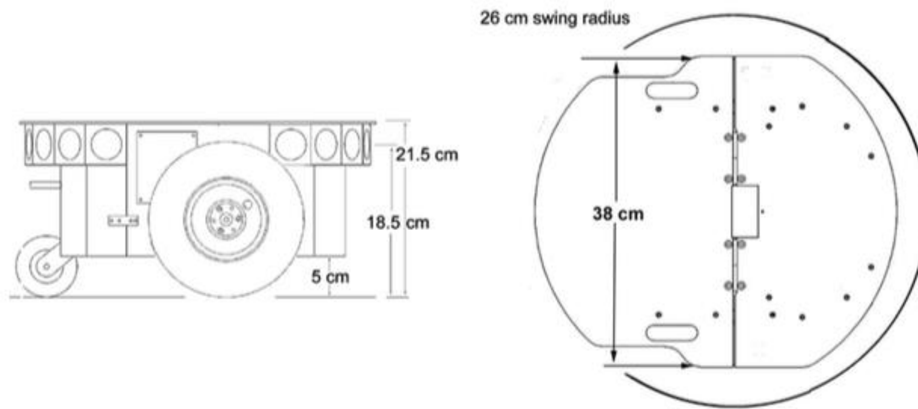


Ilustración 18 - Medidas y dimensiones P3-DX [24]

Adicionalmente el robot está dotado de un conjunto de sistemas y elementos que se enumeran y analizan a continuación:

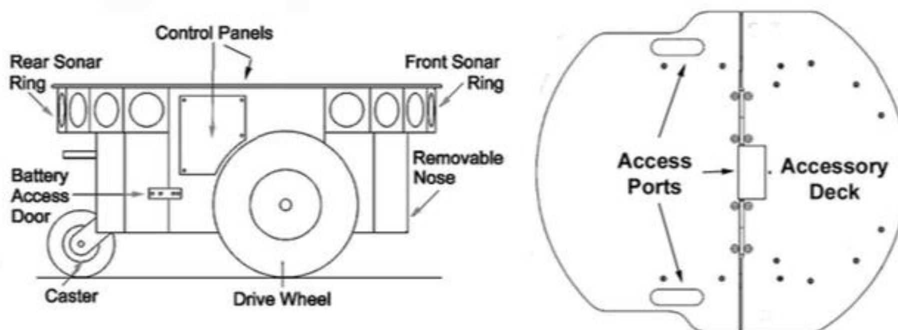


Ilustración 19 - Componentes básicos del P3-DX [24]

Posiblemente el elemento más importante es el panel de control del usuario el cual da acceso a la configuración del microcontrolador basado en el sistema ARCOS. En el panel podemos ver una serie de elementos que se describen a continuación:

- **Serial**: Se trata del puerto de conexión serie, con el que se establece la conexión con el robot. El indicador RX es la entrada de datos mientras que el TX es la salida de los mismos.
- **Stat**: Este indicador muestra el estado del robot. Llega a parpadear lentamente cuando está en espera para conectarse al cliente o rápidamente cuando hay una conexión estable con el mismo.

- **Pwr**: Indica si se está aplicando potencia al robot.
- **Battery**: Ofrece información sobre el nivel de carga de la batería del robot. Siendo el color verde indicativo de una batería cargada complemente (>12.5V) y entre naranja y rojo según se va descargando. Por otro lado, en modo de mantenimiento este led siempre es rojo independientemente de la carga de la batería.
- **Motors**: Botón que permite cambiar el estado de los motores, deshabilitándolos o activándolos. Nótese que este botón generalmente no se va a utilizar ya que su acción puede implementarse con ayuda de ARIA.
- **Reset**: Genera un reinicio inmediato del microcontrolador deshabilitando cualquier dispositivo añadido al robot, incluyendo los motores.
- **Aux1/Aux2**: Botones auxiliares que ofrecen potencia para cualquier accesorio adicional que se quiera añadir al robot, como por ejemplo cámaras.

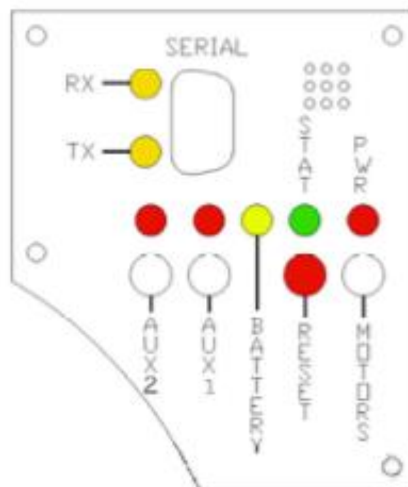


Ilustración 20 - Panel de control del P3-DX [24]

Además de este panel de control el robot posee un panel de acceso, un conjunto de parachoques o bumpers y un anillo de sonares.

- **Bumpers**

Se trata de un conjunto de sensores de contacto "binario" que únicamente son utilizados cuando otros sensores han fallado y el robot ha alcanzado un obstáculo. El termino binario es utilizado para definir la forma de envío de señales; 0 si no hay presión, 1 si la hay. Hay un total de 10 parachoques distribuidos a partes iguales entre la parte delantera y trasera del robot, como se puede observar en la Ilustración 22.

- **Sónares**

El Pioneer 3 soporta hasta 4 arrays de sonares, cada uno con un máximo de 8 transductores que proporcionan detección de objetos e información de rangos para evitar colisiones, reconocimiento de características, localización y navegación. Las posiciones de los arrays, como se observa en la figura, son fijas una en cada lado (0 y 7) y seis mirando hacia fuera a intervalos de 20 grados (1 a 6).

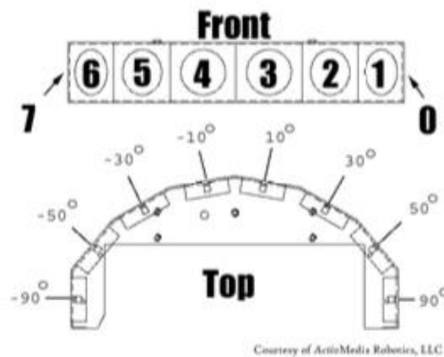


Ilustración 21 - Anillo de sonares [24]

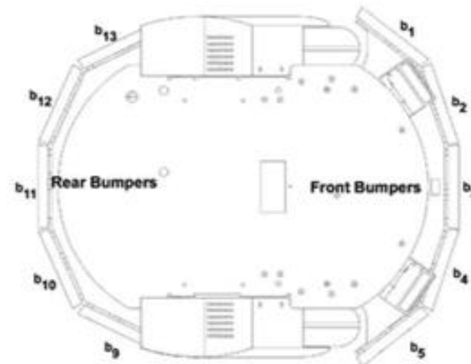


Ilustración 22 - Distribución de Bumpers [24]

Por otro lado, debido a que el robot posee gran cantidad de sistemas eléctricos el P3-DX necesita una batería que le ofrezca cierta autonomía. Las baterías están localizadas en una cavidad en la parte trasera del robot tal y como se puede observar en la Ilustración 19 y, como anteriormente se ha mencionado, la información sobre el estado de éstas es ofrecida por el panel de control de usuario.

Finalmente, llegando a término de este análisis tenemos que hablar sobre el sistema de movimiento del robot. El P3-DX en particular posee un sistema de tracción diferencial ya que tiene dos ruedas motrices a los lados (ruedas conectadas al motor y que son las que aportan movimiento) y una libre en la parte trasera central que ofrece cierta libertad de movimiento. Este sistema permite que las ruedas roten con distintas velocidades haciendo que el robot pueda girar sobre sí mismo a la vez que se mueve. Esto, en verdad, supone muchas ventajas, pero también el gran inconveniente de tener que manejar cada rueda independientemente, teniendo en cuenta que para moverse en línea recta es absolutamente obligatorio que todas ellas tengan el mismo valor de velocidad pues si no se acabaría girando.

Nótese que, la versión utilizada del P3-DX no cuenta con el ordenador de a bordo (su función va a ser sustituida por la Raspberry) por lo que es un elemento que no se va a analizar. Además, una tabla de especificaciones de hardware se podrá encontrar en el Anexo D para más información. Por último, comentar que los elementos de conexión del robot que se van a utilizar se analizarán y explicarán en profundidad en el Capítulo 3: integración y descripción del sistema (Esto se ha

decidido así por que resultará más fácil para el lector entender cuáles son y cómo funcionan).

2.2.3 Especificaciones de Software

El Robot viene de fábrica con un paquete de librerías que facilitan el desarrollo de código en la plataforma, sin embargo, éstas son herramientas externas que necesitan instalarse en el cliente (Raspberry Pi). Dichas herramientas se analizarán en siguientes secciones de este documento. Para hacer un examen del soporte de software del robot únicamente deberemos destacar el firmware que viene implementado en él y que permite su control. Bautizado con el nombre de ARCOS [25] (Advanced Robotics Control and Operations) se trata de un software integrado que ejecuta el microcontrolador del robot y se utiliza como base para el control y la lectura de información de todos los sistemas del mismo. Es un software esencial y necesario, pues incorporado con las demás librerías posibilita un control fluido y sencillo de los mecanismos, movimientos y acciones del robot.

El firmware de ARCOS, adicionalmente, viene con la herramienta de configuración ARCOScf, que permite actualizar la versión del firmware y cambiar los parámetros almacenados alojados en su memoria FLASH para configurar los sistemas internos o accesorios añadidos por el usuario. Nótese que el propio firmware tiene implementado un *watchdog* que producirá una parada del robot si la comunicación entre éste y el cliente no es estable o, en su defecto, es inexistente. Por último, comentar que, de manera particular para este trabajo, el cliente, en nuestro caso la Raspberry Pi, necesita una correcta configuración del SO y el entorno de la distribución de ROS (y sus nodos) para conseguir una comunicación correcta entre plataformas (un manual sobre la configuración del sistema puede encontrarse en el Anexo A).

2.3 Raspberry Pi 3 modelo B

2.3.1 Introducción

Se trata de la tercera generación de Raspberry Pi, un computador de placa reducida de bajo coste desarrollada en Reino Unido por la Fundación Raspberry Pi con el objetivo de estimular la enseñanza de las TIC en escuelas. Es una placa base con unas dimensiones relativamente pequeñas, unas especificaciones muy aceptables y una lista de aplicaciones realmente extensa. El principal atractivo de este tipo de placas no es otro que el precio. Bajo la premisa de ofrecer la potencia de un ordenador a bajo precio, la Raspberry se ha convertido en una de las formas más inteligentes y asequibles de disponer de un hardware especialmente eficiente y dinámico, accesible para cualquiera. Este modelo en particular fue sacado a la luz en el año 2016, y con él, se renueva el procesador, una vez más de la compañía

Broadcom y se añaden nuevas funcionalidades como wifi y Bluetooth sin necesidad de adaptadores.

2.3.2 Alternativas

En este tiempo donde la tecnología está tan avanzada y la economía de un determinado producto disfruta de una sana competencia, siempre tendremos que encontrar alternativas y debatir cuales están más adaptadas a una tarea y cuales a otra. En nuestro caso nuestras alternativas a la placa Raspberry son Arduino [26] y BeagleBone [27]. Lo cierto es que se ha decido utilizar Raspberry por varios motivos: el primero es disponibilidad y el segundo facilidad de uso. Una Raspberry es más completa que el Arduino, eso está claro, mientras que la primera es básicamente un miniordenador completo la segunda no deja de ser un microcontrolador. La Raspberry posee el potencial para correr un sistema operativo, tiene puertos ethernet, de wifi y Bluetooth; algo que nos va a ser útil para establecer una conexión con robot. Además, su potencia es superior y permite la rápida ejecución de ROS y de todos sus nodos.

La última opción que tenemos es utilizar la placa BeagleBone, que comparte ciertos aspectos con la Raspberry Pi, pero, sin embargo, tiene una baja disponibilidad y facilidad de uso para lo que se pretende llevar a cabo en ese trabajo. Tras indagar entre sus posibilidades, sí que es cierto que puede correr un sistema operativo por si sola pero su configuración y adaptabilidad a los sistemas del robot es bastante menor que en la Raspberry. Cabe, finalmente, destacar el apoyo y soporte que el propio fabricante [28] y el sitio oficial de ROS [3] llegan a dar a la Raspberry, otro motivo por el cual se ha optado por esta plataforma.

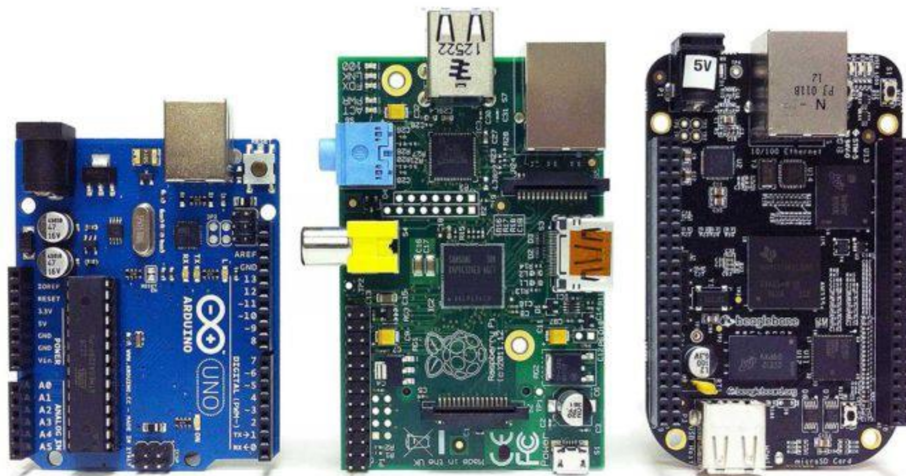


Ilustración 23 - De izq. a der.: Arduino, Raspberry y BeagleBone

2.3.3 Especificaciones de Hardware

El nuevo modelo de Raspberry posee en su circuito un chipset Broadcom BCM2387 de cuatro núcleos y 1.2 GHz. La GPU que trabaja con los gráficos es la Broadcom VideoCore IV, una propuesta Dual Core compatible con Open GL ES 2.0 y OpenVG, permitiendo llegar a resoluciones HD fácilmente. Si bien es cierto que necesita una tarjeta microSD para poder tenerla completamente operativa, incorpora 1 GB de RAM DDR2 que le permite mover con soltura bastantes sistemas operativos como Windows o Linux. Dispone de 4 puertos USB 2.0 que permiten ampliar el dispositivo con todo tipo de periféricos. Las nuevas mejoras incorporan por primera vez conectividad Wifi y Bluetooth, que junto con el puerto Ethernet supone tener gran variedad de conexiones. Por último, se destaca la entrada HDMI que ofrece la visualización de contenido en alta definición y, por consiguiente, la compartición de canales de audio y video. Una tabla de especificaciones de hardware se puede encontrar en el Anexo D donde se resume de manera más detallada cada componente.

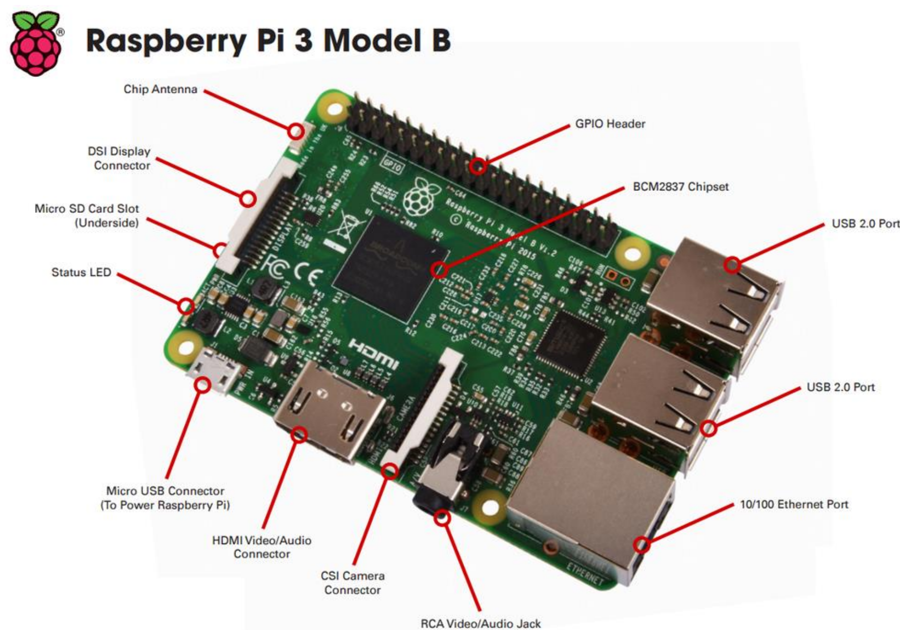


Ilustración 24 - Partes de la Raspbery Pi [29]

2.3.4 Especificaciones de Software

Aunque la Raspberry no incorpora ningún software de fábrica debemos destacar que, para tenerla totalmente funcional, necesita un Sistema Operativo que se instala con un cliente (ya sea oficial o no) desde una tarjeta microSD. En nuestro caso el Sistema Operativo que se ha seleccionado ha sido Linux, más concretamente la distribución Ubuntu mate 16.04 LTS con una arquitectura de 64 bits que posee una versión para Raspberry y está soportada a largo plazo. Está versión de Linux es la más reciente que ha demostrado ser estable y, aunque el escritorio mate puede

que no se llegue a utilizar para algunas cosas, siempre puede ser útil tener un entorno gráfico con una interfaz más o menos simple y sencilla. Para su instalación solo es necesario un cliente llamado PINN [30]. PINN es un programa que intenta mejorar NOOBS [31], el instalador oficial de sistemas operativos para Raspberry; es una versión que añade nuevas funcionalidades y más sistemas operativos para instalar, entre ellos Ubuntu Mate. El proceso es sencillo, se aloja el cliente en la tarjeta microSD y por medio de un instalador se elige el Sistema Operativo que convenga de una lista ofrecida por PINN; nótese que la instalación necesita de red pues en ningún momento PINN posee los archivos de cada Sistema Operativo, sino que solo sirve de interfaz para facilitar su instalación. Tras la instalación del Sistema Operativo, la Raspberry ya es completamente funcional como cualquier ordenador de escritorio.

La instalación e integración de todos estos componentes, así como la conexión a los diferentes sensores y al robot se discutirá en las secciones correspondientes de la memoria (Anexo A).

2.4 Sensor Kinect

2.4.1 Introducción

Kinect para Xbox 360 [32] es un dispositivo que originalmente estaba destinado a la industria del entretenimiento, más concretamente, es una cámara RGB acompañada de un sensor de profundidad, desplegada bajo el nombre en clave “Project Natal” y basado en el diseño de referencia de la compañía PrimeSense. Creado por Alex Kipman y desarrollado en el seno de la compañía Microsoft, permitiría interactuar con la videoconsola Xbox 360 sin necesidad de contacto físico, ni el uso de controladores tradicionales como los *gamepads*. La Kinect es un claro ejemplo de un dispositivo realmente barato pero muy potente, que no llegó a calar hondo en el mundo del entretenimiento, pero que sirvió para que desarrolladores e investigadores con bajo presupuesto pudieran garantizar unas lecturas de sensores muy precisas.

Debido a que en nuestro caso es vital incorporar sensores con lecturas mucho más avanzadas que los sónares (pues estos realmente no se han creado para el mapeado), es muy fácil incorporar este tipo de cámaras 3D que nos garantizan una gran calidad de datos leídos y, por tanto, podremos construir mapas más cercanos a la realidad. Así, en artículos académicos que abordan el problema del SLAM, se proponen sistemas eficientes basados en sensores RGBD [33], y ya que el sensor Kinect es bastante barato y eficaz [34], resulta ser un elección muy buena para ello.

2.4.2 Alternativas

Al igual que ocurre con la Raspberry Pi, resulta coherente pensar que tenemos más de una alternativa a escoger entre todo el mercado de cámaras 3D. De hecho, la Kinect 1, que es la que se ha utilizado en este proyecto, debido a que no obtuvo mucha fama, pasó a ser un producto descontinuado en el año 2016, que luego saldría mejorado con el lanzamiento de la Kinect 2.0 [35] en los años posteriores. No obstante, es importante añadir que hay un gran mercado de cámaras compatibles que no pertenecen obligatoriamente a la marca de Microsoft. El propio proveedor del robot también nos ofrece cámaras PTZ [36] y podemos encontrar, de manera alternativa, gran variedad de cámaras 3D en el mercado como la Asus Xtion [37], la cual está considerada como uno de los "competidores" de la Kinect en este campo.

En este punto, es de vital importancia destacar la versión del sensor utilizado finalmente. Como se ha comentado anteriormente, Microsoft logró lanzar al mercado, tras el pequeño desastre de la primera versión del sensor, la Kinect v2. Se trata de una cámara 3D mucho más potente y precisa, no obstante, no se ha llegado a utilizar porque es incompatible con el sistema propuesto.

La necesidad de un puerto USB 3.0 ha sido un factor determinante en la decisión sobre el sensor. La Kinect v2 necesita un ratio de transferencia asíncrono rápido que solo puede ser proporcionado por un puerto USB 3.0. Debido a que la placa Raspberry Pi 3 no posee este tipo de puertos, es fácil argumentar que este sensor no sería la mejor opción [38]; de hecho, si logramos conectar la Kinect v2 a un puerto USB 2.0 y ejecutamos el comando *lsusb*, podremos ver cómo solo una pequeña parte de los sistemas de la Kinect v2 son detectados, imposibilitando pues, una transferencia correcta de todos los datos necesarios.

Independientemente de la elección del sensor, gracias a ROS, la integración de todas ellas sería muy similar a la de la Kinect (o la elegida finalmente), contando con que exista un buen controlador suficientemente documentado para desarrollar con él. No obstante, debido a la fácil disponibilidad con la que cuenta la Kinect, al hecho de contar con controladores de código abierto y por las restricciones impuestas por el sistema base, que será el que gobierne la arquitectura, se ha decidido continuar con la Kinect versión 1.

Ilustración 25 - Cámaras PTZ

Ilustración 26 - Asus Xtion

2.4.3 Especificaciones de Hardware

La Kinect es una cámara RGB que almacena las lecturas de los tres canales en una resolución de 1280x960, esto hace que sea posible capturar el color de una imagen. Por otro lado, posee un emisor de infrarrojos y un sensor de profundidad, mientras que el primero emite rayos infrarrojos, el sensor toma las lecturas de los rayos reflejados, las procesa y convierte en datos de profundidad midiendo la distancia entre el objeto y el sensor. Adicionalmente, también posee un anillo de micrófonos, que no se van a utilizar en este proyecto, y un acelerómetro en 3 ejes que permitiría detectar la orientación actual del dispositivo [38]. Finalmente, es importante añadir que el sensor viene con una conexión que desafortunadamente es propietaria, por lo que acaba siendo necesario un adaptador para poder conectarlo a cualquier dispositivo que no sea la consola.

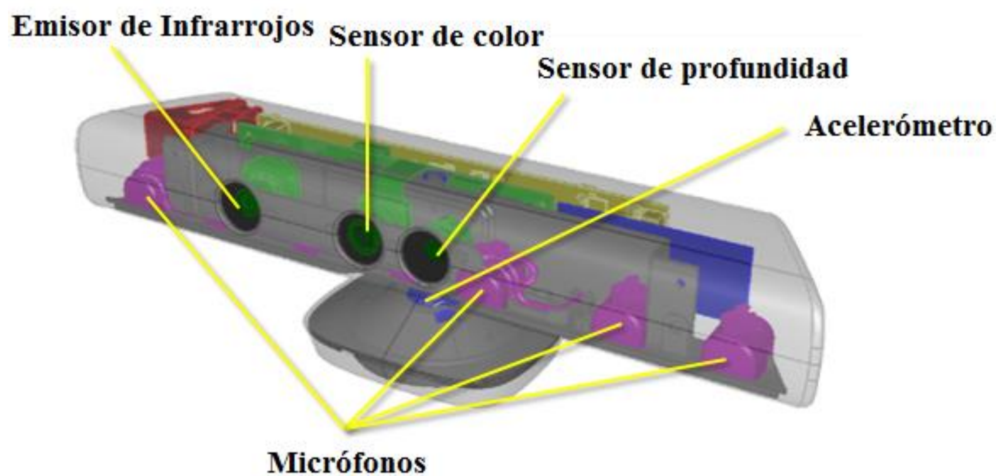


Ilustración 27 - Partes del sensor Kinect [28]

2.4.4 Especificaciones de Software

La cámara posee un SDK implementado también por Microsoft, que permite desarrollar en base a una API de manera rápida y sencilla. No obstante, como nuestro objetivo es integrar el dispositivo con ROS (y dado que ya hemos optado por trabajar en Linux) vamos a realizar todo el desarrollo con la ayuda de otro controlador de código abierto y muy cuidado por la comunidad llamado *libfreenect* [39]. Adicionalmente se utilizarán los nodos de ROS que permiten conectar el sensor al *framework* e integrar las lecturas en mensajes que puedan ser tratados por el propio ROS.

Tanto las modificaciones en las conexiones del sensor con el robot y la placa Raspberry Pi, así como la instalación de estos controladores se explicarán en las secciones correspondientes de esta memoria (Anexo A).

2.5 SLAM (Simultaneus Localization And Mapping)

SLAM son las siglas de *Simultaneus Localization And Mapping* [40]. Se puede definir como el problema, dentro del área de la computación, que supone construir un mapa de un entorno no conocido mientras que se sabe en todo momento y de manera simultánea la localización del agente dentro de éste.

El problema realmente se define de manera matemática basándose en la estadística, en el cálculo estimado de la localización del agente por medio de diversas fuentes de datos como son los sensores o la odometría relativa del robot. Por eso, al fin y al cabo, el objetivo acaba siendo computar. Los datos utilizados pueden ser datos recabados por sensores como un láser o una cámara 3D como el sensor Kinect.

Ilustración 28 - Coche autónomo de Google

Para realmente explicar a fondo lo que es SLAM podríamos dividirlo en varias secciones: el primero es el mapeado, donde se discuten los algoritmos utilizados para computar la localización relativa del agente y de los objetos escaneados (Algoritmos basados en el método Monte Carlo [41] son bastante útiles o abordajes como el de los coches autónomos, los cuales ya cuentan con una base de datos significativa de mapas ya construidos: ciudades, calles, intersecciones etc.). Otro elemento son los sensores, con los que se tratan las fuentes de los datos recibidos para la construcción de mapas. Generalmente cámaras 3D como la Kinect suelen ser muy útiles, también se usan láseres o sensores LIDAR, que tienen una gran precisión… Por otro lado, se trata el problema del movimiento, en el que se discute las velocidades relativas y las acciones del agente sobre el entorno que se mapea. Estos comandos suelen reducirse a velocidades lineares y angulares para llegar a un punto determinado. Y finalmente se aborda el problema de la complejidad y el "*Loop clousure*".

Como todo problema computacional existe cierto grado de complejidad derivado de la capacidad de computación en donde se lleven a cabo los cálculos de posiciones relativas, sobre todo en el contexto de SLAM en 3D [42]. Tenemos un ejemplo en los robots con sistemas embebidos, incapaces de realizar un SLAM eficiente debido a la necesidad de esa capacidad computacional que no tienen. Además de esto, muchos investigadores y expertos siguen afirmando a día de hoy que SLAM supone todavía uno de los desafíos fundamentales dentro de la robótica [43].

De diferente forma, pero igual de importante, es el problema del "*Loop clousure*", muy común en SLAM. Es el problema que surge al reconocer puntos que ya se han visitado anteriormente y actualizar el mapa construido adecuadamente. La corrección de esta cuestión resulta una tarea fundamental en la creación de sistemas SLAM robustos y eficaces, pues su resolución es bastante útil para aumentar enormemente la precisión sobre la odometría calculada del robot, ya que se ha de tener en cuenta que el cálculo de las posiciones es estimado y sujeto a errores que, generalmente, se arrastran con el tiempo, lo que significa que empeoran la calidad del mapa final si no se tratan con cuidado. El abordaje de este tipo de problemas implementa un conjunto de algoritmos de comparación de puntos que en última instancia permitiría reconocer los puntos ya detectados y ajustar odometría y velocidad de forma más precisa.

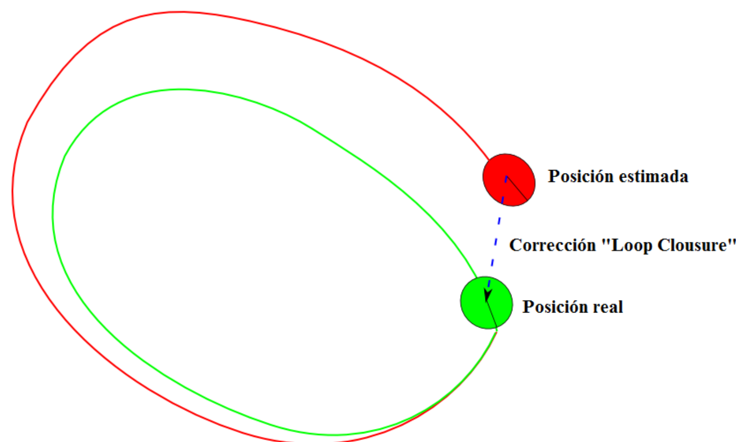


Ilustración 29 - Explicación gráfica del "Loop Clousure"

2.6 Entorno de desarrollo

2.6.1 Sistema operativo

Una de las decisiones más importantes en el trabajo es la elección del sistema operativo, pues es donde va a ejecutar el *framework* de ROS y todos los nodos que van a permitir una conexión estable y segura con el Robot. Si navegamos un poco por la página oficial del fabricante del robot observamos que se tiene preferencia por Linux, más concretamente Ubuntu. Esto mismo ocurre en el sitio oficial de ROS

donde parece que hay más soporte para versiones de Linux en comparación con otros sistemas operativos. Lo cierto es que es normal, Ubuntu ha resultado ser un Sistema Operativo de fácil acceso, sencillo, configurable y muy útil. Es por ello por lo que se ha optado por Ubuntu mate 16.04. La versión se ha elegido por ser LTS, y ser la más reciente y estable. Por otro lado, Ubuntu es bastante fácil de utilizar y es perfecto para el trabajo de cualquier tipo; además, el escritorio Mate ofrece una interfaz gráfica bastante asequible; aunque no se ha elegido mate por ello sino porque es la única oficialmente soportada en la placa Raspberry Pi.

Independientemente de esto último, tengo bastante más experiencia en trabajar con Linux que con otro sistema operativo, asimismo pienso que es más rápido y seguro que, por ejemplo, Windows. Por otro lado, y para finalizar, debemos tener en cuenta que hay versiones de ROS que no son compatibles con algunas distribuciones de Linux, es por ello por lo que la compatibilidad entre el SO y ROS también ha jugado un papel fundamental en la elección, no solo del primero sino también de este último (La relevancia de la distribución de ROS se discutirá en secciones posteriores).

2.6.2 Github y GitKraken

El sistema implementado y todos los aspectos desarrollados se han gestionado con el cliente de git de Github [44] y el gestor GitKraken [45]. El control de versiones no acaba siendo tan importante en un desarrollo individual, pero, aun así, siempre es positivo el uso de este tipo de clientes enfocados al desarrollo colaborativo. Si bien es cierto que permite un mejor impacto y visibilidad de lo que se ha realizado en el transcurso del proyecto.

El repositorio se mantendrá privado hasta la finalización del proyecto para luego ser convertido en público el día de la defensa.

2.6.3 Máquinas virtuales

Todo el código desarrollado y el trabajo realizado en el proyecto se ha completado en diversas máquinas virtuales proporcionadas por el software VMware [46]. En total, dos máquinas virtuales para Linux Xenial donde se desarrolló el código y las simulaciones y una con Windows donde se tuvo acceso a Microsoft Word para crear la memoria.

El uso de las máquinas virtuales es por elección propia, son mucho más fáciles de manejar y transportar y las instantáneas dan la oportunidad de salvar el trabajo de una forma eficaz y rápida.

2.6.4 Librerías y herramientas

En el proceso de desarrollo se ha llegado a utilizar un conjunto de librerías, herramientas y funcionalidades diferentes que han ofrecido cierta flexibilidad y facilidad durante el progreso del trabajo. Todas ellas se tratan a continuación:

- **SSH**

La comunicación con la Raspberry se ha realizado por consola de comandos con una conexión SSH.

Se trata de un protocolo para la gestión remota que ofrece a los usuarios el control sobre sus servidores a través de Internet. La motivación detrás de su creación fue reemplazar el Telnet sin cifrar, añadiendo técnicas criptográficas que garantizasen comunicaciones seguras. Tiene mecanismos para autenticar a un usuario remoto y es capaz de realizar transferencias entre cliente y host de forma bidireccional.

La mayor ventaja que encontramos con SSH es el uso del cifrado para asegurar que la comunicación entre el cliente y el host se realiza de forma segura. Existen tres tecnologías para el cifrado diferentes utilizadas por SSH:

1. Encriptación simétrica
2. Encriptación asimétrica
3. Hashing

Su funcionamiento está basado en un modelo cliente-servidor permitiendo la transferencia y comunicación entre máquinas remotas. SSH trabaja en el puerto TCP 22 (aunque esto es ajustable). El host escucha en el puerto configurado, organiza la conexión segura mediante la autenticación del cliente y la abre en el entorno de shell si la verificación tiene éxito [47].

El cliente inicia la conexión SSH con el protocolo TCP con el servidor, asegurándose de que la identidad que posee el servidor coincide con los registros del almacén de claves RSA y presenta los datos de usuario necesarios para autenticar la conexión. Así pues, hay dos fases en el proceso de establecer una conexión: primero ambos sistemas deben ponerse de acuerdo en los estándares de encriptación para proteger futuras comunicaciones, y segundo, el usuario debe proceder a autenticarse. Si las credenciales coinciden, se concede acceso al usuario.

No obstante, esta herramienta no se ha utilizado por las características de seguridad sino por la capacidad de conexión entre máquinas de forma rápida y sin necesidad de interfaz gráfica. Además, es una herramienta integrada en la mayoría de las distribuciones Linux, de fácil acceso y de poco peso. Esto es fundamental pues, aunque la Raspberry Pi tiene bastante potencia para ser una

placa de bajo coste, debemos tener en cuenta que va a mover muchas más cosas que la conexión al robot.

- **Filezilla**

Se trata de un cliente FTP, FTPS y SFTP multiplataforma, rápido y seguro con muchas funciones útiles y una interfaz gráfica de usuario intuitiva [48]. Esta herramienta tiene una alta disponibilidad al ser un cliente gratuito y facilita la transferencia de archivos de manera segura entre máquinas remotas a través de una interfaz sencilla.

El uso de Filezilla se ha limitado en este proyecto a la transferencia de código fuente de manera puntual entre la Raspberry y la estación de trabajo. Debido a que el desarrollo de las diferentes aplicaciones se ha llevado a cabo en una máquina diferente, que es donde se procedía con la simulación y el testeo, esta herramienta ha facilitado enormemente la transferencia del material desarrollado y los nodos de ROS implementados.

Por otro lado, es más rápido utilizar este tipo de software que ejecutar secuencialmente comandos de transferencia de archivos desde la terminal, y como ya se tenía experiencia con MobaXterm para Guernika se ha optado por la versión libre compatible con Linux.

- **ARIA (Advanced Robot Interface for Applications)**

ARIA [49] es una librería desarrollada por el mismo fabricante (del robot) en C++ que proporciona una interfaz y un *framework* para controlar y recibir datos dinámicamente de todas sus plataformas (MobileRobots). Por una parte, permite recibir estimaciones de posición, lectura de sonares, radares y láseres, mientras que, por otra, posibilita utilizar estos datos de manera dinámica para establecer comportamientos y/o acciones que el robot recibe y ejecuta. Incluye utilidades para desarrollar software de control del robot siendo la base de todas las demás bibliotecas como ARNL o ArNetworkin (tratadas a continuación). ARIA es portable y puede ser recompilada en diferentes plataformas y compiladores, tiene pocas dependencias y un buen soporte para la programación con hilos. ARIA viene con código fuente bajo la GNU General Public License. Esta licencia permite redistribuir el código libremente haciendo que no sea necesario la compra o autorización expresa del fabricante.

- **OpenKinect (Libfreenect)**

OpenKinect [39] es una comunidad de usuarios que tiene el objetivo de hacer uso de los dispositivos de Microsoft, Kinect y KinectV2, en plataformas diferentes a la consola Xbox. Esta comunidad está formada por alrededor de 2000 personas, y la mayoría trabaja y pone sus esfuerzos en el desarrollo y mantenimiento de la librería *Libfreneect*. Esta librería es el pilar fundamental

para la utilización de un dispositivo Kinect en una plataforma diferente a la consola. Se trata de una serie de controladores que actúan como API para poder utilizar el sensor. Libfreenect es un driver de código abierto, libre y es compatible con Windows, Mac y Linux, es por esto que es una de las mejores opciones para poder utilizar el sensor para el proyecto.

- **ArNetworking**

Anteriormente se han mencionado dos nuevas librerías, una de ellas es ArNetworking. Se trata de una librería incluida en ARIA que implementa una infraestructura extensible para operaciones de red remota, interfaces de usuario y otros servicios de red. A través de un servidor que se ejecuta en el ordenador de a bordo del robot, los clientes habilitados para ArNetworking se conectan desde otro equipo a la red para obtener datos y emitir comandos. Se suele utilizar con un cliente dinámico llamado MobileEyes que permite enviar comandos de forma remota y el simulador MobileSims que permite representar al robot y su comportamiento en un entorno virtual controlado (Las herramientas mencionadas serán tratadas en secciones posteriores pues han resultado de gran utilidad). No obstante, cabe decir que la gestión de conexiones se realizará con ROS a través del nodo ROSARIA, nodo en el que se delega toda la interfaz de conexión al robot.

- **ROS y sus módulos**

ROS (Robots Operating System) [3] es un *framework* flexible para desarrollar software de control de robots. Es una colección de herramientas, bibliotecas y convenciones que tienen como objetivo simplificar la tarea de crear un comportamiento robótico robusto y complejo en una variedad de plataformas robóticas (en nuestro caso MobileRobots, ActivMedia). Si observamos un poco la literatura científica podemos encontrar que ya ha habido intentos satisfactorios de adecuar ROS a la plataforma P3DX [50], es por esto que se propone ROS como base para implementar el sistema de este proyecto. ROS por sí mismo ofrece gran valor a la mayoría de los proyectos de robótica, pero también ofrece una oportunidad de trabajar en red y desarrollar con los *roboticistas* de clase mundial. Una de las filosofías de ROS es el desarrollo de componentes comunes, ya que no deja de ser una comunidad muy activa en un campo de investigación relativamente nuevo. Pero, ¿cuál es el motivo de que se utilice ROS? Porque la creación de software de propósito general para cualquier plataforma robótica resulta, en la mayoría de ocasiones, bastante difícil. Desde la perspectiva del robot, los problemas que irónicamente parecen triviales para los seres humanos a menudo varían enormemente entre diferentes instancias de tareas y entornos. Lidiar con estas variaciones es tan complicado muchas veces que ningún individuo, laboratorio o institución puede esperar hacerlo por sí mismo de manera

sencilla y relativamente rápida, pero con ROS solucionamos el problema al ofrecer un sistema de diseño ágil y claro [51].

El principal atractivo de ROS es que permite el desarrollo de software de control a través de nodos que se comunican entre sí. En ROS un nodo [52] es un ejecutable que se utiliza para comunicarse con otros nodos, estas comunicaciones utilizan mensajes que son un tipo de datos de ROS que contienen la información ofrecida por un tópico o *topic* [53] una vez suscrito a él. Un Tópico se puede ver como un conjunto de mensajes y datos que ofrecen cierta información generada por un nodo acerca de, por ejemplo, uno de los sistemas del robot (como el motor, el radar etc.). En el proceso de desarrollo, los nodos se suscriben a otros nodos, de esta manera comparten información vital como el estado del motor, la posición de robot etc. lo que permite que otros nodos gestionen esa información para modelar comportamientos, comandar acciones… Sabiendo esto, un nodo puede ser publicador y/o suscriptor, es decir publica información acerca de un sistema del robot y/o la recibe para utilizarla de una manera especificada en su código fuente.

El conjunto de nodos y archivos de configuración necesarios para desarrollar el comportamiento del robot se distribuyen en un paquete de ROS; en él, mientras que los archivos .cpp (.java o .py) representan el código fuente, los .xml poseen una declaración de dependencias y parámetros necesarios para la construcción correcta del proyecto en el espacio de trabajo. Este conjunto de archivos mencionados anteriormente vienen acompañados, adicionalmente, por archivos de configuración de lanzamiento; extensión .launch [54] que utilizan en verdad una versión de XML específica para ROS y permiten la ejecución conjunta de varios nodos a la vez con gran variedad de parámetros y modos de ejecución. Nótese que la arquitectura de ROS se ve obligada a tener alojado un espacio de trabajo que necesita ser construido con cada nueva implementación (nodo); el constructor (en esta distribución es una construcción por medio del gestor catkin [55], implementado y soportado de forma oficial) se encargará de compilar cada una de las partes y administrar problemas, excepciones y dependencias. Así, será necesario, por cada nuevo proyecto, un archivo CMake en el que se declaran dependencias y trabajos para el compilador y para ROS.

Para terminar con este análisis, anteriormente se ha discutido sobre los posibles problemas de incompatibilidades entre el sistema operativo y ROS, pues la distribución de ROS necesita un SO específico donde ejecutarse. Para la completa compatibilidad de ambas partes se ha decidido utilizar la distribución Kinetic de ROS, única que soporta Ubuntu 16.04 Xenial y que, al igual que ésta, tiene soporte a largo plazo (LTS).

- **IDEs utilizados**

Si bien es cierto que a la hora de programar algo es mejor tener una idea bien establecida, bolígrafo y papel, para la implementación de las diferentes funcionalidades en el proyecto se han utilizado una serie de IDEs que han facilitado el desarrollo y depurado de código, así como la experimentación y testeo del mismo. En primer lugar, debemos destacar el VisualStudo Code [56], un editor de código fuente ligero pero potente que se ejecuta en escritorio y tiene infinidad de extensiones y paquetes de funcionalidad para una miríada de lenguajes de programación, entre ellos C++. Por otro lado, debido a que el Sistema Operativo en el que se ha trabajado es Linux, el editor de texto eMacs [57] también supone una herramienta útil cuando no se tiene acceso a VisualCode en alguna de las plataformas. No es un IDE por definición, pero prefiero tratarlo como tal por lo extensible que llega a ser.

- **Herramientas adicionales**

Para logar diferentes hitos y objetivos en el progreso del proyecto se han llegado a utilizar un conjunto de programas que han simplificado el manejo y la comunicación con el robot ofreciendo interfaces gráficas asequibles para una correcta visualización del sistema a tiempo real. Son las siguientes:

  - **MobileSim**

  Es un software para simular plataformas de MobileRobots, ActivMedia y sus entornos con el objetivo de depurar y experimentar con ARIA. Está basado en el simulador de *Stage*, creado por Richard Vaughan, Andrew Howards y otros, como parte del proyecto Player/Stage. MobileSim [58] utiliza datos registrados en un mapa o archivo .map para simular paredes y obstáculos en el entorno. Dichos archivos son creados con la herramienta Mapper3 cuyos detalles se ofrecen en secciones posteriores. Su uso resulta interesante ya que con estos mapas se puede simular el robot en cuestión y hacer que navegue por el entorno pudiendo obtener una visual de la localización del robot y sus movimientos de manera inmediata y dinámica.

  Si bien es cierto que existen otras herramientas más potentes para la simulación de estos entornos como Gazebo [59], (simulador que viene por defecto en la distribución de ROS) se ha decidido utilizar MobileSim mayormente por motivos de disponibilidad y facilidad de uso, ya que es proporcionado por el fabricante. Además, para la simulación con este último programa es necesario crear una serie de proyectos para construir virtualmente el robot en sí y, en algunos casos, existen incompatibilidades o no hay soporte para la versión del robot con la que se está trabajando.

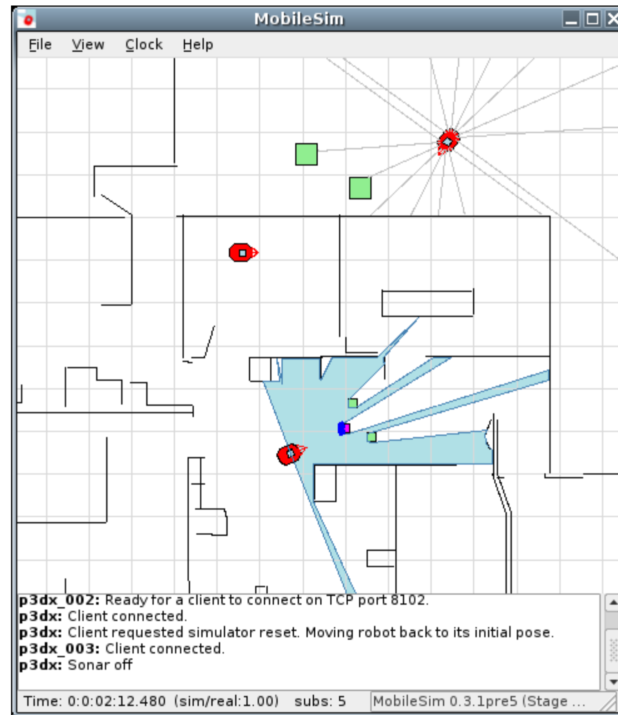


Ilustración 30 - Mapa 2D visualizado en MobileSim

- **Mapper3**

Esta aplicación [60] es, a grandes rasgos, un ayudante para MobileSim. Permite, por un lado, convertir archivos de registro de exploración láser (.2d) en archivos de mapa (.map) y, por otro, crearlos desde cero con la posibilidad de editar, modificar y agregar puntos de inicio y objetivos. Estos mapas se pueden utilizar con librería del fabricante (ARNL o SONARNL) como mapas para que el robot pueda localizar y navegar por un entorno con la ayuda de MobileSims y MobileEyes (otro software del fabricante que no se ha llegado a utilizar). El principal uso en el proyecto es la realización de mapas de prueba para testear diferentes aspectos de la implementación del sistema.

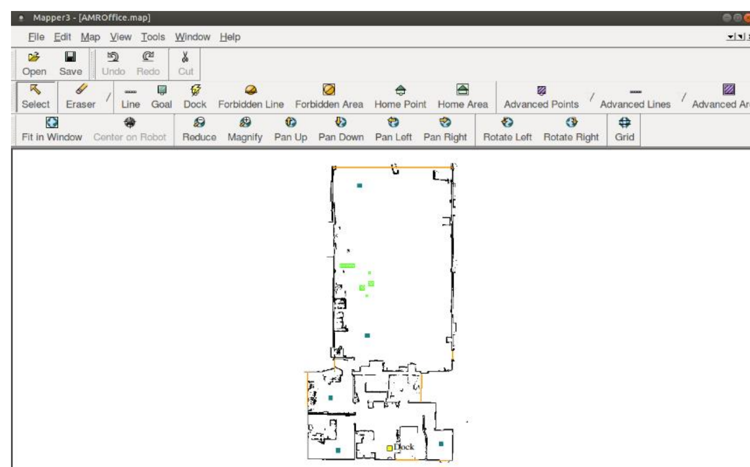


Ilustración 31 - Interfaz de Mapper

- **Rviz**

Rviz [61] es una herramienta de visualización que viene por defecto en la instalación de ROS. Debido a que MobileSim es bastante limitado y solo nos permite simular el robot, no visualizarlo, se decidió monitorizar el trabajo del robot con este sistema.

Como cualquier entorno en 3D (simulado), Rviz permite el movimiento en los tres ejes, esto posibilitará tener una visualización completa tanto del entorno como de cualquier parte de la plataforma robótica que queramos observar en particular.

Rviz trabaja con *displays*, por cada *topic* publicado en ROS se puede añadir un *display* a la interfaz para visualizar el contenido publicado. En este proyecto, por ejemplo, se ha querido crear el modelo del p3dx en el mundo de Rviz; para ello debemos lanzar un nodo que publique en un *topic* los *joint states* y un script (denominado generalmente *spawner*) que cargue la descripción del robot en la interfaz. Esta descripción viene definida por una serie de archivos de extensión URDF (Unified Robot Description Format) [62] que utiliza XML para describir cada componente del robot que se quiere simular. Estos archivos generalmente son muy difíciles de entender, leer y mantener, es por eso que la comunidad de ROS ha creado los archivos .xacro [63]. Estos archivos son otra variante en XML de los anteriores, pero son capaces de permitir macros que facilitan enormemente el depurado y mantenimiento de las descripciones de los diferentes robots que se estén utilizando.

Además, como podemos leer todos los *topics* publicados a través de ROS, somos capaces de añadir displays a la interfaz para interpretar las lecturas de los sónares o los escaneos de la Kinect. Así podemos visualizar en todo momento qué es lo que está viendo el robot y cómo lo ve.

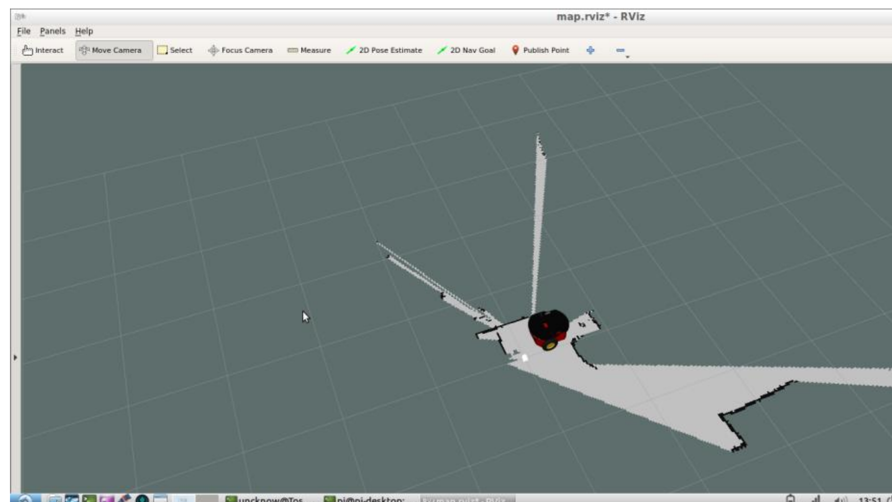


Ilustración 32 - Interfaz Rviz

Esto puede ser especialmente útil para las tareas de mapeo, ya que podemos representar el mapa generado dinámicamente, o simplemente para tener un orden de cada uno de los *frames* de coordenadas que tenga el robot que estamos utilizando. En nuestro caso, el p3dx tiene 5 *frames* de coordenadas, y nuestro futuro sistema trabajará el movimiento en 2D en el *frame* de la odometría */odom* mientras que el mapeo y su visualización se realizará en el *frame* denominado */map*. Además, uno de los puntos atractivos de este sistema es que permite salvar archivos de configuración; estos archivos de extensión .rviz permiten salvar el conjunto de displays seleccionado para poder lanzar la herramienta directamente sin tener que configurar la interfaz.

Así, se pueden desarrollar archivos de configuración Rviz para monitorizar el nodo de teleoperación, para configurar el sensor Kinect o para monitorizar el proceso de mapeo de manera independiente que ahorrarían tiempo al tener la configuración de cada uno ya salvada. Estos son unos pocos ejemplos que se han desarrollado durante el proyecto y que son extensibles a otros muchos aspectos que brinda ROS.

2.6.5 Lenguajes de programación

Durante el progreso del proyecto se han visto involucrados varios lenguajes de programación. Si bien es cierto que ha habido alternativas se ha decidido, por diferentes motivos, que a continuación se exponen, unos sobre otros:

Se tenía la oportunidad de desarrollar en C++, Python o Java (estos dos últimos requerían de *wrappers* [64] para la correcta comunicación con el controlador del robot). Finalmente se eligió C++ al ser un lenguaje dirigido a objetos, extremadamente eficiente en tiempos de compilación y ejecución. Además, por otro lado, también ha habido un motivo de disponibilidad ya que la librería principal del robot, interfaz de comunicación que permite la comunicación con éste, está desarrollada por el propio fabricante en C++ y se ha preferido no hacer uso de medios intermedios, traductores o *wrappers*.

- **C++**

  Se trata de un lenguaje de programación que nace a mediados de los años 80 a manos de Bjarne Stroustrup [65]. Su intención fue extender el lenguaje C con mecanismos para manipular objetos; es por ello por lo que C++ es considerado un lenguaje híbrido. A día de hoy existe un estándar denominado ISO C++ por lo que el desarrollador se puede guiar por una norma. Debido a la proximidad de este lenguaje con C (y que de hecho permite la interpretación y compilación de código con sintaxis en C) es fácil desarrollar y testear cualquier aplicación implementada a bajo nivel. Considerado por muchos como el lenguaje de programación más rápido del mundo (ya que permite al

programador mejorar el tiempo de ejecución mediante diferentes técnicas), se trata de un lenguaje extremadamente útil en proyectos de inteligencia artificial, los cuales al fin y al cabo son susceptibles al tiempo y grandes cantidades de cómputo, necesitando en muchos casos una respuesta inmediata en tiempo de ejecución. Un ejemplo claro es el de la gran mayoría de IA en juegos, desarrollada en C++ ofreciendo un conjunto de tiempos realmente bajo. Si bien es cierto que C++ pretende ser un lenguaje para aplicaciones a gran escala, controladores y firmware en general, su inclusión en el mundo de la inteligencia artificial se ha visto, en relativo poco tiempo, incrementado, poniéndose al mismo nivel con otros como Python o Lisp, los cuales, siendo más flexibles y dinámicos, permiten la manipulación de objetos de una manera fácil y eficaz en detrimento de la velocidad de ejecución [66].

Al ser considerado un lenguaje de nivel intermedio, encapsula muy bien tanto características de lenguajes de alto nivel como de bajo, lo que hace que sea ideal para establecer una comunicación entre hardware y software de manera eficiente y sencilla. Su principal premisa: una colección de clases predefinidas, que, al fin y al cabo, son tipos de datos que se pueden instanciar en múltiples ocasiones [67].

Por su naturaleza, permite la definición de clases por el usuario, funciones miembro para implementar características específicas, una noción de herencia fuertemente establecida y normalizada y conceptos como el polimorfismo, plantillas, punteros o *namespaces*. Aun con todo ello, el motivo de peso para su uso en las diferentes subsecciones de IA (y en este proyecto) no deja de tener relación con el bajo tiempo de cómputo, compilación y ejecución que muchas veces ofrece, así como la adecuación hardware-software.

- **Xml**

Siglas de e**X**tensible **M**arkup **L**anguage, se trata de un meta-lenguaje que permite generar otros lenguajes de marcas, desarrollado por el World Wide Web Consortium (W3C) y utilizado para almacenar datos de forma legible. Proviene del lenguaje SGML y posibilita precisar la gramática de lenguajes específicos para la estructuración de documentos de gran tamaño. Muchas veces referido como tecnología xml, este lenguaje intenta solventar la problemática de expresar información estructurada de la manera más abstracta y reutilizable posible [68].

XML es extensible, el analizador es un componente estándar por lo que no es necesario crear un analizador concreto para cada versión del lenguaje, su estructura es sencilla de procesar y entender y supone la transformación de datos en información, con lo que se obtiene cierta flexibilidad para estructurar documentos. Pero ¿cuál es el motivo de su extensa utilización? Para empezar, es un lenguaje de sintaxis sencilla, fácil de aprender; por otro lado, es un

estándar internacionalmente conocido y no pertenece a ninguna compañía y finalmente porque permite la concreción de etiquetas personalizadas para la descripción y organización de datos [69].

Con respecto a esto último, y como ya se ha mencionado en anteriores secciones, se ha de notar que muchos archivos de configuración de ROS se desarrollan con versiones de XML. Por ejemplo, el archivo de configuración de compilación de paquetes en cada proyecto está en XML y es el que deberemos modificar para decirle al compilador qué tareas debe realizar. Por otro lado, nos encontramos con los ejemplos de los archivos de configuración de los modelos de los robots utilizados en Rviz. Estos modelos son mantenidos y depurados con el lenguaje Xacro que es una versión específica de XML creada por la comunidad de ROS.

Capítulo 3: Integración y descripción del sistema

En esta sección se procederá a realizar una descripción detallada del sistema desarrollado. A través de la declaración de las características funcionales del sistema y de los casos de uso, junto con una explicación pormenorizada de la arquitectura propuesta, se explicarán sus características y su funcionamiento.

3.1 Introducción

Debido a que ROS es un *framework* algo complejo y se han de entender bastantes aspectos a cerca de las características que implementa, hubo un pequeño periodo que se dedicó exclusivamente a familiarizarse con el entorno. Estos nodos, que se fueron implementando como pruebas de concepto, sirvieron como base para el desarrollo del sistema final. En total se desarrollaron 4 nodos como introducción básica a ROS.

- **Publicador**: Debido a que era necesario comunicar a una capa de alto nivel los posibles estados en los que se encontraba el robot dentro del mundo, se desarrolló un publicador. Para ello se creó un mensaje propio de ROS y se implementó un nodo que publicará en un *topic* un mensaje básico formado por la odometría del robot $[x, y, \theta]$.
- **Teleoperación**: Para tener un acercamiento un poco más complejo a lo que sería mandar comandos de velocidad al robot y controlar la plataforma, se implementó un nodo de teleoperación que permitía, conectado un teclado a la Raspberry, leer la entrada y apilar comandos de velocidad en el *stack* de ROS para poder mover la base del robot.
- **Movimiento Básico**: Debido a que era importante familiarizarse con las transformadas y los tiempos (*Durations*) proporcionados por ROS, se desarrolló un nuevo nodo en ROS que implementa una clase en C++ que permite dar órdenes básicas a la base del robot. En total son dos: moverse linealmente teniendo como parámetros la velocidad, la distancia y la dirección (delante o detrás) y girar en torno al eje Z (en nuestro caso el único posible) teniendo como parámetros velocidad, grados y una dirección (a favor de las agujas del reloj o en contra).
- **Cliente-Servidor**: Llegando ya al término del aprendizaje sobre ROS, apareció la necesidad de establecer un puente de comunicación entre el sistema de alto nivel y la plataforma robótica. Para eso se desarrolló la base de una arquitectura cliente-servidor. Primero se implementó un mensaje tipo srv en ROS que permite describir un mensaje específico para usarse en este tipo de arquitecturas. Luego se implementó el nodo cliente que permitiría al alto nivel mandar peticiones al servidor. El último nodo desarrollado es el nodo servidor, capaz de coger como parámetro la petición del cliente y ejecutar aquello que se pide. Una

vez finalizada su ejecución devolvería una respuesta para ser utilizada por el otro sistema.

Esta parte del proyecto ha sido muy importante ya que la mayoría de las características implementadas han sido fundamentales para desarrollar gran parte de los nodos que conforman el sistema final.

3.2 Análisis del sistema

Esta sección servirá para analizar el diseño propuesto de la arquitectura desarrollada durante el proyecto. Se describirá el entorno tecnológico y sus componentes, poniendo especial detalle en estos últimos y en las funciones que llevan a cabo dentro del sistema.

3.2.1 Características funcionales

El sistema desarrollado en este proyecto debe ser capaz de completar eficientemente una serie de tareas a continuación expuestas:

En primer lugar, el sistema debe suponer una interfaz de fácil acceso a cualquier aplicación de alto nivel que pretenda generar un plan de alto nivel. Este plan deberá estar formado por una serie de acciones a completar de forma consecutiva y debería ser transformado a bajo nivel para poder ser utilizado por la arquitectura desarrollada.

En segundo lugar, los nodos implementados deben ser, en todo momento, capaces de comunicarse con el robot, publicar el estado de este al alto nivel y realizar las tareas propuestas del conjunto dado de forma eficiente; a saber, moverse linealmente, girar, moverse en el plano 2D, teleoperar la base del robot e iniciar o finalizar el servicio de mapeo (consecuentemente creando un mapa de la zona). Estas dos últimas tareas se traducirán en el abordaje eficiente del problema de SLAM. Así, la arquitectura será capaz de realizar las tareas propuestas por el cliente con la restricción de aportar *feedback* a este último acerca del mundo y del estado tanto de robot, como de la tarea completada.

Además, el sistema debe ser capaz de gestionar dinámicamente lo que acaece en cada una de las partes involucradas en la ejecución, y reportar de manera fácil y rápida cualquier error o divergencia con respecto al comportamiento que se supone como normal. Esto se traduce en la generación de logs y la gestión de errores, sin embargo, se deja claro que no será competencia del sistema la replanificación o la actuación ante un plan fallido, pues ésta será tarea del alto nivel.

3.2.2 Restricciones del sistema

A lo largo de este apartado se introducirán y explicarán todas aquellas restricciones impuestas en el sistema tanto a nivel de Hardware como de Software.

3.2.2.1 Restricciones de Hardware

El sistema está compuesto por una serie de elementos de hardware y cada uno de ellos posee unas restricciones asociadas que se exponen a continuación:

- Las baterías que posee el P3-DX tienen una autonomía de 5 horas tras su carga completa.
- La velocidad lineal máxima alcanzada por el p3dx es de 1,2 m/s mientras que la velocidad angular máxima es de 300 º/s. No obstante, estas velocidades están limitadas en el sistema desarrollado.
- El P3-DX soporta una carga máxima de 17 kg. Tendremos que tener en cuenta el peso del sensor y la Raspberry.
- La Raspberry debe funcionar con un voltaje del rango [4.5-5.5V] y un amperaje de 2A.
- El sensor Kinect sólo puede funcionar con una alimentación de 12V.
- La placa Raspberry estará alimentada por medio de un cable con un extremo conectado al puerto auxiliar de 5V del P3-DX y el otro a un micro-USB conectado al puerto de alimentación de la placa.
- El sensor deberá estar conectado con un cable al puerto de 12V del P3-DX.
- La placa estará conectada al puerto SERIE del P3-DX mediante el cable Serie, la transmisión de datos se dará desde este puerto a la ranura USB de la placa, y viceversa.
- La Kinect estará conectada a un puerto USB de la placa para la transmisión de datos y a un conector especial para adaptar la conexión propietaria de Microsoft.
- El dispositivo Kinect tiene unos rangos de visión mínimos y máximos, de 1 metro y 3 metros, respectivamente.
- Los sonares del p3dx tiene unos rangos de visión mínimos y máximos, de 10 cm y 5 metros, respectivamente.
- El portátil de monitorización estará conectado a la Raspberry por una red wifi Ad-Hoc sin seguridad, por lo tanto, ambas máquinas deben poseer una interfaz *wlan* configurada.

3.2.2.2 Restricciones de Software

Para establecer las restricciones del Software utilizado, debemos primero destacar diversos puntos interesantes de esta sección. En primer lugar, hay que

comentar que tenemos dos tipos de software: el primero tiene una procedencia de terceros, como, por ejemplo, ROS o ARIA, mientras que el último es el desarrollado durante el proyecto. En esta sección se discutirán las restricciones de los dos tipos de Software comentados, pero solo en relación con la arquitectura implementada, restricciones más allá de este entorno no se abordarán.

Así, encontramos las siguientes restricciones de Software:

- El entorno de desarrollo utilizado se corresponde con el *framework* de ROS.
- La placa Raspberry necesita un sistema operativo para poder correr ROS y comunicarse con el robot.
- El sistema operativo utilizado es Ubuntu Mate Xenial (16.04 LTS).
- La distribución de ROS utilizada debe ser ROS Kinetict (LTS).
- Será necesaria la librería ARIA para el microcontrolador del P3-DX y el nodo ROSARIA para la comunicación con ROS.
- Tanto la gestión del código del proyecto como de los nodos externos de ROS utilizados se ha realizado con el gestor de git GitHub (y GitKraken).
- El lenguaje utilizado para el desarrollo de la arquitectura propuesta es C++, siguiendo el estándar SO/IEC 9899:2011 (c11).
- El driver del sensor Kinect es OpenKinect *(libfreenect)*.
- El portátil de monitorización ha de tener una versión de openGL superior a 4. Esto es para que Rviz no de ningún error.
- Tanto el portátil como la placa deben poseer una versión de la implementación de SSH en Ubuntu para posibilitar la comunicación.

3.2.3 Entorno operacional

Esta sección servirá para describir tanto el Software como el Hardware necesario para el correcto funcionamiento del sistema desarrollado. Este entorno se puede ver en la Ilustración 33.

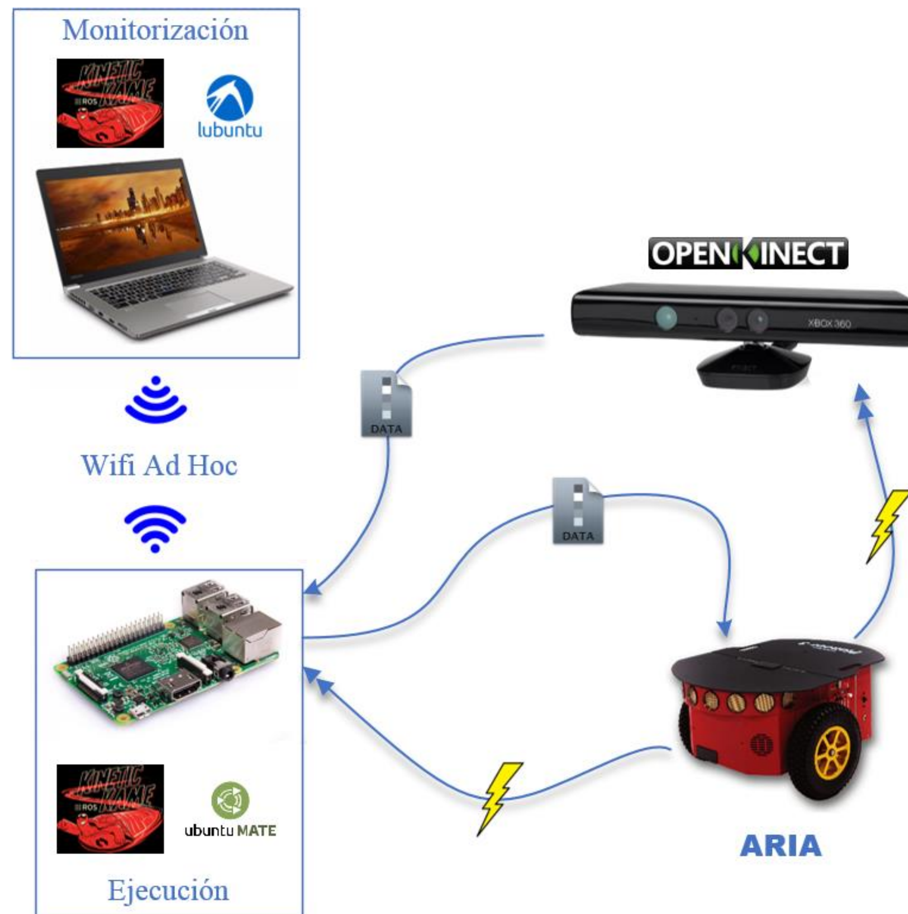


Ilustración 33 - Esquema del entorno operacional

Tratamos a continuación los aspectos del software:

- Será necesario el SO Ubuntu Mate Xenial (16.04 LTS).
- Será necesario la distribución ROS Kinetic (LTS).
- Será necesaria la librería ARIA y el nodo ROSARIA para la comunicación entre el robot y la Raspberry.
- Será necesario el driver *libfreenect* para comunicarse con la Kinect.
- Se usará la pila de navegación de ROS.
- Se utilizará el nodo *depth_image_to_laserscan* para publicar las lecturas de la cámara de manera que puedan utilizarse.
- Se utilizará el nodo *Gmapping (OpenSlam)* para el mapeo de la zona.
- Se utilizarán los modelos URDF para la monitorización proporcionados por el paquete *p2os* de ROS.
- La comunicación entre máquinas se realizará con una arquitectura Cliente-Servidor con el siguiente formato de mensaje (Petición y Respuesta). **Petición**: Código de tarea (*String*), Vector_movimiento,

Vector_giro, Vector_x_y. **Respuesta**: Feedback (*String*) y Eval (*Integer*).

- La publicación del estado a bajo nivel del robot deberá comunicarse con el siguiente formato: Header (*frameRosData*), Motor State (*Integer*), Battery State (*Double*), Posición X (*Double*), Posición Y (*Double*), Posición W (*Double*), Front Bumpers (*Boolean*), Rear Bumpers (*Boolean*).
- Será necesario un servidor y cliente SSH en cada máquina.

Tratamos a continuación los aspectos del hardware:

- Será necesario un robot P3-DX.
- Será necesaria una placa Raspberry y una memoria microSD para almacenar el SO.
- Será necesario un dispositivo Kinect.
- La placa Raspberry necesitará un cable de alimentación de al menos 5V y 2A
- La Kinect necesitará un cable de alimentación de al menos 12V.
- La placa necesitará el cable SERIAL y un adaptador a USB para conectarse al robot y transmitir datos.
- La Kinect necesitará un adaptador para convertir la conexión propietaria de Microsoft a USB para la Raspberry.
- Será útil un portátil de monitorización (aunque no necesario para que el sistema funcione correctamente).
- En el caso de utilizar un portátil para monitorizar, éste y la Raspberry deben tener dos interfaces *wlan* configuradas y estar en la misma red.

3.2.4 Especificación de los casos de uso

Este apartado servirá para establecer las funcionalidades que se busca ofrecer con la arquitectura desarrollada. Mediante la descripción de las diferentes actividades que se puedan llegar a realizar en el sistema (Casos de uso), se dispondrán una serie de escenarios y se extraerán un conjunto de requisitos para formalizar los diferentes objetivos, restricciones y funcionalidades.

3.2.4.1 Descripción de los actores de los casos de uso

El conjunto de actores partícipes en los casos de uso son los siguientes:

- **Interfaz de Alto nivel**: Elemento encargado de generar los planes a alto nivel, es decir la consecución de pequeñas tareas que se pedirán al cliente para completar un objetivo determinado.
- **Monitorización**: Máquina exclusivamente dedicada a la supervisión del sistema. Su funcionalidad principal es la de visualizar la ejecución.
- **Usuario**: Elemento encargado de iniciar todos los servicios del sistema. Es el actor más importante ya que es el encargado de la ejecución de toda la arquitectura.

3.2.4.2 Descripción de los atributos de los casos de uso

Para la correcta formalización de los casos de uso se utilizarán una serie de atributos y una presentación tabular limpia y clara. Esta sección servirá para realizar una descripción de cada uno de estos atributos:

- **Código**: Supondrá una identificación unívoca del caso de uso. La nomenclatura utilizada será CU-YY, siendo YY el número representativo del caso de uso empezando con 00.
- **Nombre**: Denominación del caso de uso tratado. Generalmente resumirá la descripción del requisito.
- **Actores**: Conjunto de entidades que son susceptibles de intervenir en el caso de uso.
- **Descripción**: Especificación de la funcionalidad o funcionalidades del caso de uso.
- **Precondiciones y efectos**: Especificación de las condiciones que se deben satisfacer para la realización de una operación y del estado del sistema tras ésta.
- **Escenario**: Descripción esquemática de las diferentes fases que componen el caso de uso.
- **Condiciones de fallo**: Especificación de errores contingentes y la respuesta del sistema para su recuperación tras éstos.

De esta manera se seguirá el siguiente formato tabular:

Código		Nombre	
Actores			
Descripción			
Precondiciones			
Efectos			
Escenario			
Fallos posibles			

Tabla 1 - Formato de tabla para la especificación de casos de uso

3.2.4.3 Descripción textual de los casos de uso

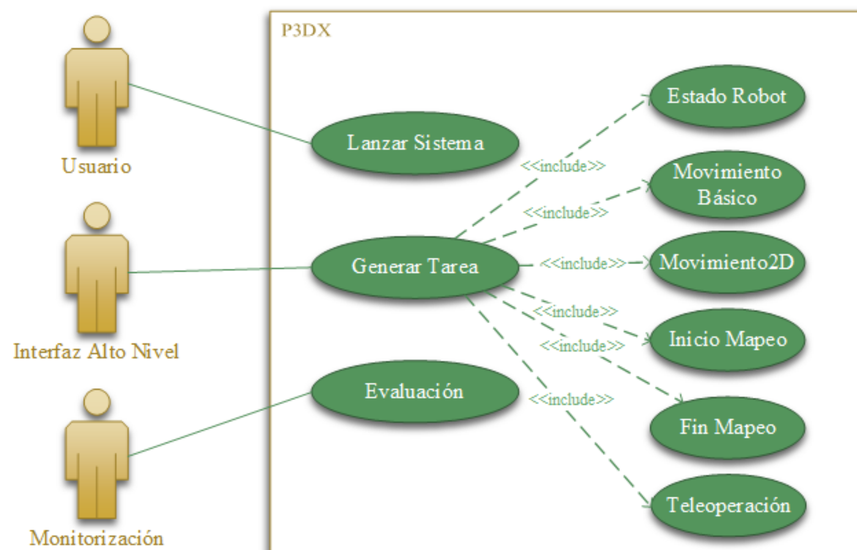


Ilustración 34 - Diagrama de los Casos de Uso

Código	CU-00	Nombre	Lanzar el sistema
Actores	Usuario		
Descripción	El Usuario inicia los controladores del robot, sensores y servidor.		
Precondiciones	El robot debe estar operativo (baterías cargadas), la placa conectada y el sensor Kinect activo.		
Efectos	El sistema se inicia levantando los nodos de ROS: Publicador, servidor, sensors y navegación.		
Escenario	1. Se verifica que el robot está encendido. 2. Se verifica que la placa esté conectada al puerto serie (con permisos de administrador) y al puerto auxiliar de 5v. 3. Se verifica que el sensor está conectado a la placa por USB y al puerto auxiliar de 12v.		

	4. Se crea el entorno bash necesario en la placa y el portátil mediante el lanzamiento del script *start_p3dx.sh*. 5. Se realiza una llamada desde la placa para lanzar el archivo launch que ejecuta la arquitectura: rosaria, servidor, publicador, sensores y navegación.
Fallos posibles	• Error de comunicación entre el robot y rosaria. Se debe a permisos insuficientes en el puerto de conexión serie. • Error que produce una parada del sistema en bucle infinito. Se debe a una mala conexión con el dispositivo Kinect, el sistema esperará hasta que uno se conecte.

Tabla 2 - Caso de uso CU-00

Código	CU-01	Nombre	Petición movimiento de 2D
Actores	Interfaz de Alto nivel		
Descripción	El alto nivel hace una petición para mover la plataforma a una coordenada al servidor de bajo nivel mediante la llamada al cliente.		
Precondiciones	Arquitectura operativa. No hay tarea en ejecución.		
Efectos	La plataforma robótica se moverá al punto indicado y enviará el resultado de la acción al cliente.		
Escenario	1. El cliente establece conexión con servidor haciendo una llamada con el código de tarea *moveTo* y los parámetros $[x, y, \theta]$. 2. El servidor recibe la tarea, procesa si es viable. 3. Si la tarea es viable controla los accionadores y motores de la plataforma para moverse al punto indicado. 4. Cuando se termine el movimiento envía la respuesta al cliente encapsulando el resultado de la acción.		
Fallos posibles	• Error de entrada. Debido a la petición con parámetros incorrectos. • Error de *alcanzabilidad*. La plataforma no ha podido alcanzar el objetivo propuesto porque no existe camino viable (o es excesivamente complejo) al punto requerido.		

Tabla 3 - Caso de uso CU-01

Código	CU-02	Nombre	Petición de teleoperación
Actores	Interfaz de Alto nivel		
Descripción	El alto nivel hace una petición al servidor de bajo nivel para teleoperar la plataforma mediante la llamada al cliente.		
Precondiciones	Arquitectura operativa. No hay tarea en ejecución. No existe una instancia del nodo de teleoperación ya activa.		
Efectos	El servidor ejecuta el nodo de teleoperación y se mantiene en segundo plano hasta que éste finalice.		
Escenario	1. El cliente establece conexión con servidor haciendo una llamada con el código de tarea *teleop* 2. El servidor recibe la tarea, procesa si es viable. 3. Si la tarea es viable inicia la ejecución del nodo de teleoperación por teclado. 4. Cuando se termine la ejecución del nodo se envía la respuesta al cliente encapsulando el resultado de la acción, si ha terminado satisfactoriamente o no.		
Fallos posibles	• Error de entrada. Debido a la petición con parámetros incorrectos. • Error de controladores: No hay teclado con el que teleoperar.		

Tabla 4 - Caso de uso CU-02

Código	CU-03	Nombre	Inicio servicio de mapeo
Actores	Interfaz de Alto nivel		
Descripción	El alto nivel hace una petición para iniciar el servicio de mapeo al servidor de bajo nivel mediante la llamada al cliente.		
Precondiciones	Arquitectura operativa. No hay tarea en ejecución. La Kinect está publicando las lecturas de la cámara.		
Efectos	El servicio de mapeo se inicia recibiendo los datos de las lecturas del sensor Kinect y generando un mapa de costes.		
Escenario	1. El cliente establece conexión con servidor haciendo una llamada con el código de tarea *startMapping*. 2. El servidor recibe la tarea, procesa si es viable. 3. Si la tarea es viable inicia el lanzamiento del sistema *Gmapping (OpenSlam)*. 4. Una vez se ha producido el lanzamiento, éste sigue en segundo plano y el servidor recupera		

	la imagen de memoria para seguir procesando tareas. 5. Se envía la respuesta al cliente encapsulando el resultado de la acción, si ha iniciado satisfactoriamente o no.
Fallos posibles	• Error de sensores: La Kinect no está publicando lecturas o lo está haciendo en tópicos distintos a los esperados. • Error de existencia del sistema de coordenadas */map*: El sistema *Gmapping* no está publicando bien el mapa de costes que se está creando. • Error de árbol no conectado: Árbol de coordenadas de mapeo y odometría no están conectados.

Tabla 5 - Caso de uso CU-03

Código	CU-04	Nombre	Fin del servicio de mapeo
Actores	Interfaz de Alto nivel		
Descripción	El alto nivel hace una petición para finalizar el servicio de mapeo al servidor de bajo nivel mediante la llamada al cliente.		
Precondiciones	Arquitectura operativa. No hay otra tarea en ejecución. El sistema *Gmapping* está activo.		
Efectos	Se finaliza la ejecución del sistema de mapeado y se genera el mapa de costes en base a la información recabada de los sensores.		
Escenario	1. El cliente establece conexión con servidor haciendo una llamada con el código de tarea *endMapping*. 2. El servidor recibe la tarea, procesa si es viable. 3. Si la tarea es viable lanza el generador de mapas de costes y guarda el mapa resultado en *root/maps*. 4. Finaliza la ejecución del sistema *Gmapping* (*OpenSlam*). 5. Una vez se ha generado el mapa y finalizado el servicio de mapeo, se envía la respuesta al cliente encapsulando el resultado de la acción, si ha finalizado satisfactoriamente o no.		
Fallos posibles	• Error EPOLL. La dirección para salvar el mapa no tiene los permisos necesarios. • Error de finalización. El sistema *Gmapping* no está activo por lo que no se puede finalizar.		

Tabla 6 - Caso de uso CU-04

Código	CU-05	Nombre	Petición de estado del robot
Actores	Interfaz de Alto nivel		
Descripción	El alto nivel hace una petición al servidor para saber el estado de bajo nivel del robot. El publicador comienza a enviar un mensaje que encapsula su estado.		
Precondiciones	Arquitectura operativa.		
Efectos	El publicador comienza a informar en el *topic /p3dx_sensors* el estado a bajo nivel del robot en el mundo.		
Escenario	1. El Cliente hace una petición para recibir información del estado. 2. El Bajo nivel comienza a publicar el estado a bajo nivel en el *topic /p3dx_sensor* 3. El Alto nivel inicia un subscriptor para leer la información del tópico generado.		
Fallos posibles	• Error de sincronización: Ros lanza este error si los mensajes entre máquinas no tienen el mismo reloj (el MD5Sum es diferente).		

Tabla 7 - Caso de uso CU-05

Código	CU-06	Nombre	Evaluación de tareas
Actores	Monitorización		
Descripción	El módulo de monitorización inicia Rviz y comienza a supervisar las tareas. En cualquier momento es capaz de leer los logs de Servidor y Cliente.		
Precondiciones	Arquitectura operativa. La monitorización tiene lanzado el modelo de robot para Rviz y los *Joint states*.		
Efectos	Simulación del mundo del robot y supervisión de las tareas realizadas.		
Escenario	1. El módulo de monitorización lanza el *Spawner* y los *Joint states*. 2. Muestra en Rviz el mundo del robot. 3. Hace unas lecturas de los archivos de log de Cliente y Servidor para identificar fallos. 4. Supervisa las acciones en el sistema de forma dinámica.		
Fallos posibles	• Error de conexión: La configuración del entorno bash no es correcta para la comunicación con la placa. • Error de lectura: No tiene permisos necesarios para leer los logs generados.		

	• Error de simulación: Rviz necesita establecer las variables de entorno de catkin para reconocer modelos y localizaciones de archivos.

Tabla 8 - Caso de uso CU-06

Código	CU-07	Nombre	Movimiento básico
Actores	Interfaz de Alto nivel		
Descripción	El alto nivel hace una petición al servidor para mover o girar la base del robot.		
Precondiciones	Arquitectura operativa. No hay tarea en ejecución.		
Efectos	Movimiento lineal o giro de la base dependiendo de los parámetros utilizados.		
Escenario	1. El cliente establece conexión con servidor haciendo una llamada con el código de tarea correspondiente (*moveForward o twist*) y los parámetros de Velocidad, Distancia y Dirección. 2. El servidor recibe la tarea, procesa si es viable. 3. Si la tarea es viable controla los accionadores y motores de la plataforma para moverse o girar. 4. Cuando se termine el movimiento envía la respuesta al cliente encapsulando el resultado de la acción.		
Fallos posibles	• Error de entrada. Debido a la petición con parámetros incorrectos. • Error de *alcanzabilidad*. El punto final no es alcanzable por la base.		

Tabla 9 - Caso de uso CU-07

3.2.5 Especificación de los requisitos del sistema

A continuación, se dispondrá el conjunto de requisitos del sistema. Estos requisitos emanan directamente de los casos de uso expuestos y se dividen en dos categorías principales:

- **Requisitos funcionales**: Definen funciones del sistema, de sus componentes.
- **Requisitos no funcionales**: Especifican criterios aplicados sobre la operación de un sistema. Tales como rendimiento, velocidad etc.

3.2.5.1 Descripción de los atributos de los requisitos

Para la correcta formalización de los requisitos se utilizarán una serie de atributos y una presentación tabular limpia y clara. Esta sección servirá para realizar una descripción de cada uno de estos atributos:

- **Código**: Identificador unívoco del requisito. La nomenclatura utilizada será RX-YY, donde las letras significan lo siguiente:
    - R: Identifica que es un requisito.
    - X: Determina el tipo de requisito. F para los Funcionales, que describen la operación a realizar y NF para los No Funcionales, que describen cómo debe realizarse la operación.
    - YY: Es el número del requisito dentro de la clasificación, empezando con 00.
- **Nombre**: Denominación del requisito tratado. Generalmente resumirá la descripción del requisito.
- **Descripción**: Especificación básica del requisito tratado.
- **Fuente**: Origen de donde emana el requisito, generalmente se corresponderá con uno o más casos de uso.
- **Prioridad**: Indicador del nivel de atención al requisito. Encontramos tres niveles:
    - Alta: El requisito debe ser satisfecho antes que otros requisitos de prioridad más baja.
    - Media: Necesita ser cumplido después de haber cumplido con los de nivel alto.
    - Baja: Requisito último en satisfacerse.
- **Estabilidad**: Describe si el requisito puede ser o no modificado durante el ciclo de vida del sistema. Puede ser:
    - Estable: El requisito no puede variar durante el ciclo de vida.
    - Inestable: El requisito puede variar a lo largo del ciclo de vida.
- **Verificabilidad**: Describe el grado de facilidad con el que se puede comprobar que se cumple el requisito. Se tienen tres niveles:
    - Alta: El requisito puede ser comprobado mediante todas las herramientas de verificación.
    - Media: El requisito no puede comprobarse mediante todas las herramientas de verificación aplicables.

- ○ Baja: Requisito que no puede verificarse con las herramientas que se tienen.

- **Requisitos relacionados**: Dispone el código del requisito o los requisitos relacionados. Si no los hay supondrá un campo vacío.

De esta manera se seguirá el siguiente formato tabular:

Código		Nombre	
Descripción			
Fuente			
Prioridad	□ Alta □ Media □ Baja		
Estabilidad	□ Estable □ Inestable		
Verificabilidad	□ Alta □ Media □ Baja		
Requisitos relacionados			

Tabla 10 - Formato de tabla para la especificación de los requisitos

3.2.5.2 Descripción textual de los requisitos

- Requisitos funcionales

Código	RF-00	Nombre	Script de entorno
Descripción	Los entornos de bash deben ser configurados mediante la ejecución del script personalizado de cada máquina.		
Fuente	CU-00, CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-01, RF-02, RF-03, RNF-00, RNF-01, RNF-02, RNF-05, RNF-08, RNF-09, RN-14, RNF-15		

Tabla 11 - Requisito funcional 00

Código	RF-01	Nombre	Lanzamiento al inicio
Descripción	Para levantar la arquitectura se debe lanzar el *launcher* que activa los nodos controladores, sensores, servidor y navegación.		
Fuente	CU-00, CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RF-02, RF-03, RNF-00, RNF-01, RNF-02, RNF-05, RNF-16		

Tabla 12 - Requisito funcional 01

Código	RF-02	Nombre	Monitorización
Descripción	El portátil deberá supervisar las tareas realizadas por el sistema.		
Fuente	CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RF-01, RF-03, RNF-00, RNF-01, RNF-02, RNF-05, RNF-6		

Tabla 13 - Requisito funcional 02

Código	RF-03	Nombre	Simulación
Descripción	El portátil lanzará todos los archivos necesarios para la supervisión del sistema.		
Fuente	CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RF-01, RF-02, RNF-00, RNF-01, RNF-02, RNF-05, RNF-6		

Tabla 14 - Requisito funcional 03

Código	RF-04	Nombre	Realizar tarea
Descripción	Las tareas se proponen por el alto nivel que llama al nodo cliente.		
Fuente	CU-01, CU-02, CU-03, CU-04, CU-05, CU-06, CU-7		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-05, RF-06, RF-07, RF-08, RF-09, RF-10, RF-11, RF-12, RF-13, RNF-00, RNF-01, RNF-02, RNF-13		

Tabla 15 - Requisito funcional 04

Código	RF-05	Nombre	Procesar tarea
Descripción	El Bajo nivel (servidor) debe comprobar que la tarea pedida es viable para poder ejecutarla.		
Fuente	CU-01, CU-02, CU-03, CU-04, CU-05, CU-06, CU-7		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	□ Alta ■ Media □ Baja		
Requisitos relacionados	RF-04, RF-06, RF-07, RF-08, RF-09, RF-10, RF-11, RF-12, RF-13, RNF-00, RNF-01, RNF-02, RNF-13		

Tabla 16 - Requisito funcional 05

Código	RF-06	Nombre	Resolución tarea
Descripción	El bajo nivel (servidor) será el único responsable de realizar la acción pedida.		
Fuente	CU-01, CU-02, CU-03, CU-04, CU-05, CU-06, CU-7		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-04, RF-05, RF-07, RF-08, RF-09, RF-10, RF-11, RF-12, RF-13, RNF-00, RNF-01, RNF-02, RNF-13		

Tabla 17 - Requisito funcional 06

Código	RF-07	Nombre	Inicio motores
Descripción	El Bajo nivel (servidor) será capaz de activar los motores de la plataforma siempre y cuando no estén ya activos.		
Fuente	CU-00		
Prioridad	□ Alta □ Media ■ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-04, RF-05, RF-06, RNF-10, RNF-12, RNF-17		

Tabla 18 - Requisito funcional 07

Código	RF-08	Nombre	Movimiento lineal
Descripción	El Bajo nivel (servidor) será capaz de mover la base del robot linealmente dadas la velocidad, la distancia y la dirección.		
Fuente	CU-07		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-04, RF-05, RF-06, RNF-10, RNF-12, RNF-18		

Tabla 19 - Requisito funcional 08

Código	RF-09	Nombre	Giro de la base
Descripción	El Bajo nivel (servidor) será capaz de girar la base del robot dadas la velocidad, los ángulos de giro y la dirección.		
Fuente	CU-07		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-04, RF-05, RF-06, RNF-10, RNF-12, RNF-19		

Tabla 20 - Requisito funcional 09

Código	RF-10	Nombre	Movimiento 2D
Descripción	El Bajo nivel (servidor) será capaz de ir a un punto en un mundo 2D dadas las coordenadas $[x, y, \theta]$.		
Fuente	CU-01		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-04, RF-05, RF-06, RNF-10, RNF-12, RNF-20		

Tabla 21 - Requisito funcional 10

Código	RF-11	Nombre	Teleoperación
Descripción	El Bajo nivel (servidor) será capaz de controlar el robot a través de un teclado.		
Fuente	CU-02		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-04, RF-05, RF-06, RNF-10, RNF-12, RNF-23		

Tabla 22 - Requisito funcional 11

Código	RF-12	Nombre	Inicio mapeo
Descripción	El Bajo nivel (servidor) deberá ser capaz de iniciar un servicio para mapear la zona una vez la tarea se ha registrado.		
Fuente	CU-03		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	□ Alta ■ Media □ Baja		
Requisitos relacionados	RF-04, RF-05, RF-06, RNF-10, RNF-12, RNF-21		

Tabla 23 - Requisito funcional 12

Código	RF-13	Nombre	Fin de mapeo
Descripción	El Bajo nivel (servidor) deberá ser capaz de finalizar el servicio de mapeo si lo requiere el Alto nivel (cliente).		
Fuente	CU-04		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	□ Alta ■ Media □ Baja		
Requisitos relacionados	RF-04, RF-05, RF-06, RNF-10, RNF-12, RNF-22		

Tabla 24 - Requisito funcional 13

Código	RF-14	Nombre	Publicar lecturas
Descripción	La arquitectura deberá ser capaz de comunicarse con la Kinect y publicar sus lecturas en el topic */camera*.		
Fuente	CU-03, CU-04		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-20, RF-21		

Tabla 25 - Requisito funcional 14

Código	RF-15	Nombre	Transformar lecturas
Descripción	Las lecturas de la Kinect deben ser transformadas a escaneos laser (*LaserScan*) y publicadas en el topic */scan_depth*.		
Fuente	CU-03, CU-04		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-11, RF-12, RF-20, RF-21		

Tabla 26 - Requisito funcional 15

Código	RF-16	Nombre	Log del servidor
Descripción	El nodo servidor deberá generar un log en el que se registre su actividad.		
Fuente	CU-01, CU-02, CU-03, CU-04, CU-05, CU-06, CU-7		
Prioridad	□ Alta □ Media ■ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-11, RF-12, RNF-00, RNF-01, RNF-02, RNF-04		

Tabla 27 - Requisito funcional 16

Código	RF-17	Nombre	Log del cliente
Descripción	El nodo cliente deberá generar un log en el que se registre su actividad.		
Fuente	CU-01, CU-02, CU-03, CU-04, CU-05, CU-06, CU-7		
Prioridad	□ Alta □ Media ■ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RNF-00, RNF-01, RNF-02, RNF-04		

Tabla 28 - Requisito funcional 17

Código	RF-18	Nombre	Recolección de datos
Descripción	El sistema deberá recuperar todos los datos de bajo nivel del robot en el mundo sobre el que actúa.		
Fuente	CU-05		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-19, RNF-00, RNF-01, RNF-02		

Tabla 29 - Requisito funcional 18

Código	RF-19	Nombre	Publicación de datos
Descripción	El sistema será capaz de publicar el estado de bajo nivel del robot en el topic */p3dx_sensor*		
Fuente	CU-05		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-18		

Tabla 30 - Requisito funcional 19

Código	RF-20	Nombre	Evasión de obstáculos
Descripción	El sistema de navegación 2D debe ser capaz de esquivar los obstáculos que se presenten en el camino.		
Fuente	CU-01		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	□ Alta ■ Media □ Baja		
Requisitos relacionados	RF-10, RF-14, RF-15, RNF-20		

Tabla 31 - Requisito funcional 20

Código	RF-21	Nombre	Generación de mapa
Descripción	El sistema de mapeo debe generar una imagen del mapa producido por el servicio de mapeo al finalizar éste.		
Fuente	CU-03, CU-04		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-14, RF-15, RNF-21, RNF-22		

Tabla 32 - Requisito funcional 21

- Requisitos no funcionales

Código	RNF-00	Nombre	Ejecución continua
Descripción	Los nodos base de la arquitectura (Servidor, sensor, navegación y controladores) deben estar ejecutándose siempre.		
Fuente	CU-00		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RF-01, RF-02, RF-03, RF-04, RF-05, RF-06, RF-016, RF-17, RF-18, RF-19		

Tabla 33 - Requisito no funcional 00

Código	RNF-01	Nombre	Conexión continua
Descripción	Portátil (Monitorización), Raspberry Pi 3 (Ejecución), Kinect y robot (P3-DX) deben estar continuamente conectados y en comunicación		
Fuente	CU-00, CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RF-01, RF-02, RF-03, RF-04, RF-05, RF-06, RF-016, RF-17, RF-18, RF-19, RNF-02, RNF-14, RNF-15		

Tabla 34 - Requisito no funcional 01

Código	RNF-02	Nombre	Red Wifi
Descripción	La comunicación entre máquinas se hará por medio de una red wifi configurada con las siguientes IP: • Portátil: 10.42.0.13 • Raspberry Pi: 10.42.0.1		
Fuente	CU-00, CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RF-01, RF-02, RF-03, RF-04, RF-05, RF-06, RF-016, RF-17, RF-18, RF-19, RNF-01, RNF-14, RNF-15		

Tabla 35 - Requisito no funcional 02

Código	RNF-03	Nombre	Teclado
Descripción	Para que se procese la tarea de teleoperación tiene que haber un teclado conectado a la Raspberry Pi.		
Fuente	CU-02		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-11, RNF-23		

Tabla 36 - Requisito no funcional 03

Código	RNF-04	Nombre	Permisos logs
Descripción	Para los dos ficheros log generados: La Raspberry Pi tendrá permisos de lectura y escritura, el Portátil solo de lectura.		
Fuente	CU-06		
Prioridad	□ Alta □ Media ■ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-16, RF-17, RNF-14, RNF-15		

Tabla 37 - Requisito no funcional 04

Código	RNF-05	Nombre	Simulación
Descripción	La monitorización en base a la simulación se realizará con Rviz.		
Fuente	CU-06		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RF-01, RF-02, RF-03, RNF-06, RNF-07		

Tabla 38 - Requisito no funcional 05

Código	RNF-06	Nombre	Permisos de Rviz
Descripción	Rviz deberá tener permisos sobre el entorno de ROS para poder simular y leer todos los topics publicados por la Raspberry Pi.		
Fuente	CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-02, RF-03, RNF-05, RNF-07		

Tabla 39 - Requisito no funcional 06

Código	RNF-07	Nombre	Lanzamientos para Rviz
Descripción	La simulación deberá hacer uso de los lanzadores que activan los *joint states* y el modelo 3D del robot (URDF, *Spawner*).		
Fuente	CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RNF-05, RNF-06		

Tabla 40 - Requisito no funcional 07

Código	RNF-08	Nombre	Sistema Operativo
Descripción	Será necesario que tanto Placa como Portátil posean el Sistema Operativo Ubuntu 16.04, no siendo necesario versiones específicas de cada uno.		
Fuente	CU-00, CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RNF-09		

Tabla 41 - Requisito no funcional 08

Código	RNF-09	Nombre	ROS
Descripción	El sistema se desarrollará con el *framework* de ROS.		
Fuente	CU-00		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RNF-08, RNF-11, RNF-12, RNF-13		

Tabla 42 - Requisito no funcional 09

Código	RNF-10	Nombre	Tiempo de expiración.
Descripción	Toda actividad que dure más de 60 segundos sin hacer nada en el servidor, entrará en expiración haciendo que la petición falle.		
Fuente	CU-01, CU-02, CU-03, CU-04, CU-05, CU-06, CU-7		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	□ Alta ■ Media □ Baja		
Requisitos relacionados	RF-07, RF-08, RF-09, RF-10, RF-11, RF-12, RF-13, RNF-12, RNF-13		

Tabla 43 - Requisito no funcional 10

Código	RNF-11	Nombre	Lenguaje de desarrollo
Descripción	El sistema deberá estar implementado en C++ bajo la ISO/IEC 9899:2011.		
Fuente	CU-00		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RNF-09		

Tabla 44 - Requisito no funcional 11

Código	RNF-12	Nombre	Mensaje personalizado
Descripción	La publicación del estado de bajo nivel se hará con este formato de mensaje personalizado: • Header: Conteniendo el *frame* de coordenadas y el tiempo en ROS. • Motor State: Entero conteniendo el estado del motor del Robot (1: Encendido ó 0: Apagado) • Battery State: Double conteniendo el estado de la batería en voltios. • Posición X: Posición X en metros del robot. • Posición Y: Posición Y en metros del robot. • Posición W: Ángulo de giro actual del robot en grados sexagesimales. • Front Bumpers: Booleano reflejando el estado actual de los Bumpers frontales (1 si hay al menos uno activo, 0 si no). • Rear Bumpers: Booleano reflejando el estado actual de los Bumpers traseros (1 si hay al menos uno activo, 0 si no).		
Fuente	CU-01, CU-02, CU-03, CU-04, CU-05, CU-06, CU-7		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-07, RF-08, RF-09, RF-10, RF-11, RF-12, RF-13, RNF-09, RNF-10, RNF-17, RNF-18, RNF-19, RNF-20, RNF-21, RNF-22, RNF-23		

Tabla 45 - Requisito no funcional 12

<table>
<tr><td>Código</td><td>RNF-13</td><td>Nombre</td><td>Srv personalizado</td></tr>
<tr><td>Descripción</td><td colspan="3">La comunicación entre Cliente y Servidor se hará mediante el tipo de mensaje especial de Ros srv con el siguiente formato:
Petición:
• Código de tarea: String representado la tarea que se quiere pedir.
• Vector_Movimiento: Vector de movimiento para la tarea de movimiento lineal. Es una tupla [Velocidad, Distancia, Dirección].
• Vector_Giro: Vector de giro para la tarea de giro de la base. Es una tupla [Velocidad, Grados, Dirección].
• Vector_x_y: Vector de movimiento para la tarea de movimiento en 2D. Es una tupla [Posición X, Posición Y, Ángulo de giro].
Respuesta:
• Feedback: String con la representación semántica de la resolución de la tarea.
• Eval: Entero representado si la tarea se ha finalizado correctamente o no.</td></tr>
<tr><td>Fuente</td><td colspan="3">CU-01, CU-02, CU-03, CU-04, CU-05, CU-06, CU-7</td></tr>
<tr><td>Prioridad</td><td colspan="3">■ Alta □ Media □ Baja</td></tr>
<tr><td>Estabilidad</td><td colspan="3">□ Estable ■ Inestable</td></tr>
<tr><td>Verificabilidad</td><td colspan="3">■ Alta □ Media □ Baja</td></tr>
<tr><td>Requisitos relacionados</td><td colspan="3">RF-04, RF-05, RF-06, RNF-09, RNF-10</td></tr>
</table>

Tabla 46 - Requisito no funcional 13

<table>
<tr><td>Código</td><td>RNF-14</td><td>Nombre</td><td>Configuración Portátil</td></tr>
<tr><td>Descripción</td><td colspan="3">El portátil debe poseer una interfaz wlan y estar conectado a la red wifi p3dx_net.</td></tr>
<tr><td>Fuente</td><td colspan="3">CU-00, CU-06</td></tr>
<tr><td>Prioridad</td><td colspan="3">■ Alta □ Media □ Baja</td></tr>
<tr><td>Estabilidad</td><td colspan="3">□ Estable ■ Inestable</td></tr>
<tr><td>Verificabilidad</td><td colspan="3">■ Alta □ Media □ Baja</td></tr>
<tr><td>Requisitos relacionados</td><td colspan="3">RF-00, RNF-01, RNF-02, RNF-04</td></tr>
</table>

Tabla 47 - Requisito no funcional 14

Código	RNF-15	Nombre	Configuración Raspberry
Descripción	La placa Raspberry debe tener configurada su interfaz *wlan* para poder conectarse a la red *p3dx_net*.		
Fuente	CU-00, CU-06		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-00, RNF-01, RNF-02, RNF-04		

Tabla 48 - Requisito no funcional 15

Código	RNF-16	Nombre	Conexión serie
Descripción	La conexión serie entre la Raspberry Pi y el robot debe tener permisos de administrador.		
Fuente	CU-00		
Prioridad	■ Alta □ Media □ Baja		
Estabilidad	■ Estable □ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-01		

Tabla 49 - Requisito no funcional 16

Código	RNF-17	Nombre	Parámetro inicio motores
Descripción	Para hacer la petición del inicio de motores el parámetro es único y debe ser 0.		
Fuente	CU-00		
Prioridad	□ Alta □ Media ■ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-07, RNF-12		

Tabla 50 - Requisito no funcional 17

Código	RNF-18	Nombre	Parámetros movimiento
Descripción	Para mover linealmente la base los parámetros introducidos en la llamada del cliente deben ser: • Código de tarea: 1. • Velocidad: Double en m/s. • Distancia: Double en m. • Dirección: 1 delante, 0 atrás.		
Fuente	CU-07		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-08, RNF-12		

Tabla 51 - Requisito no funcional 18

Código	RNF-19	Nombre	Parámetros giro
Descripción	Para girar la base los parámetros introducidos en la llamada del cliente deben ser: • Código de tarea: 2. • Velocidad: Double en grados/s. • Grados: Double en grados. • Dirección: 1 agujas del reloj, 0 al contrario.		
Fuente	CU-07		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-09, RNF-12		

Tabla 52 - Requisito no funcional 19

Código	RNF-20	Nombre	Parámetros movimiento 2D
Descripción	Para moverse a un punto de coordenadas en un plano 2D los parámetros introducidos en la llamada del cliente deben ser: • Código de tarea: 3. • Posición X: Double para definir la posición en el eje X (en metros). • Posición Y: Double para definir la posición en el eje Y (en metros). • Posición W: Double para definir los grados sexagesimales.		
Fuente	CU-01		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-10, RF-20, RNF-12		

Tabla 53 - Requisito no funcional 20

Código	RNF-21	Nombre	Parámetros inicio mapeo
Descripción	Para iniciar el servicio de mapeo el parámetro es único y debe ser 4.		
Fuente	CU-03		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-12, RF-21, RNF-12		

Tabla 54 - Requisito no funcional 21

Código	RNF-22	Nombre	Parámetros fin mapeo
Descripción	Para finalizar el servicio de mapeo el parámetro es único y debe ser 5.		
Fuente	CU-04		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-13, RF-21, RNF-12		

Tabla 55 - Requisito no funcional 22

Código	RNF-23	Nombre	Parámetros teleoperación
Descripción	Para ejecutar el nodo de teleoperación el parámetro es único y debe ser 6.		
Fuente	CU-02		
Prioridad	□ Alta ■ Media □ Baja		
Estabilidad	□ Estable ■ Inestable		
Verificabilidad	■ Alta □ Media □ Baja		
Requisitos relacionados	RF-11, RNF-03, RNF-12		

Tabla 56 - Requisito no funcional 23

3.2.5.3 Matriz de trazabilidad

Esta sección se dedicará al estudio y análisis para detectar inconsistencias y/o ambigüedad en el conjunto de casos de uso y requisitos propuesto en las secciones anteriores. Para abordar tal tarea, se definirán dos matrices de trazabilidad, siendo la primera entre los requisitos establecidos para poder identificar las interacciones internas de los mismos y asegurar la completitud y consistencia, y la segunda, entre los casos de uso y los requisitos, que nos servirá para justificar la existencia de cada requisito.

A continuación, se muestran las dos matrices:

	RF 00	RF 1	RF 2	RF 3	RF 4	RF 5	RF 6	RF 7	RF 8	RF 9	RF 10	RF 11	RF 12	RF 13	RF 14	RF 15	RF 16	RF 17	RF 18	RF 19	RF 20	RF 21
RF 00		X	X	X																		
RF 1	X		X	X																		
RF 2	X	X		X																		
RF 3	X	X	X																			
RF 4						X	X	X	X	X	X	X	X	X								
RF 5					X		X	X	X	X	X	X	X	X								
RF 6					X	X		X	X	X	X	X	X	X								
RF 7					X	X	X															
RF 8					X	X	X															
RF 9					X	X	X															
RF 10					X	X	X														X	
RF 11					X	X	X															
RF 12					X	X	X									X	X					
RF 13					X	X	X									X	X					
RF 14													X	X							X	X
RF 15													X	X							X	X
RF 16																						
RF 17																						
RF 18																				X		
RF 19																			X			
RF 20											X				X	X						
RF 21															X	X						
RNF 00	X	X	X	X	X	X	X										X	X	X			
RNF 1	X	X	X	X	X	X	X										X	X	X			
RNF 2	X	X	X	X	X	X	X										X	X	X			
RNF 3											X											
RNF 4																	X	X				
RNF 5	X	X	X	X																		
RNF 6			X	X																		
RNF 7																						
RNF 8	X																					
RNF 9	X																					
RNF 10								X	X	X	X	X	X	X								
RNF 11																						
RNF 12								X	X	X	X	X	X	X								
RNF 13					X	X	X															
RNF 14	X																					
RNF 15	X																					
RNF 16		X																				
RNF 17								X														
RNF 18									X													
RNF 19										X												
RNF 20											X										X	
RNF 21													X									X
RNF 22														X								X
RNF 23												X										

	RNF 00	RNF 1	RNF 2	RNF 3	RNF 4	RNF 5	RNF 6	RNF 7	RNF 8	RNF 9	RNF 10	RNF 11	RNF 12	RNF 13	RNF 14	RNF 15	RNF 16	RNF 17	RNF 18	RNF 19	RNF 20	RNF 21	RNF 22	RNF 23
RF 00	X	X	X			X			X	X					X	X								
RF 1	X	X	X			X											X							
RF 2	X	X	X			X	X																	
RF 3	X	X	X			X	X																	
RF 4	X	X	X											X										
RF 5	X	X	X											X										
RF 6	X	X	X											X										
RF 7											X		X					X						
RF 8											X		X						X					
RF 9											X		X							X				
RF 10											X		X								X			
RF 11				X							X		X											X
RF 12											X		X									X		
RF 13											X		X										X	
RF 14																								
RF 15																								
RF 16	X	X	X		X																			
RF 17	X	X	X		X																			
RF 18	X	X	X																					
RF 19	X	X	X																					
RF 20																					X			
RF 21																						X	X	
RNF 00																								
RNF 1			X												X	X								
RNF 2		X													X	X								
RNF 3																								X
RNF 4															X	X								
RNF 5							X	X																
RNF 6						X		X																
RNF 7						X	X																	
RNF 8										X														
RNF 9									X			X	X	X										
RNF 10													X	X										
RNF 11										X														
RNF 12										X	X							X	X	X	X	X	X	X
RNF 13										X	X													
RNF 14		X	X		X																			
RNF 15		X	X		X																			
RNF 16																								
RNF 17													X											
RNF 18													X											
RNF 19													X											
RNF 20													X											
RNF 21													X											
RNF 22													X											
RNF 23				X									X											

Tabla 57 - Matriz de trazabilidad Requisito-Requisito

	CU 00	CU 1	CU 2	CU 3	CU 4	CU 5	CU 6	CU 7
RF 00	X						X	
RF 1	X						X	
RF 2							X	
RF 3							X	
RF 4		X	X	X	X	X	X	X
RF 5		X	X	X	X	X	X	X
RF 6		X	X	X	X	X	X	X
RF 7	X							
RF 8								X
RF 9								X
RF 10		X						
RF 11			X					
RF 12				X				
RF 13					X			
RF 14				X	X			
RF 15				X	X			
RF 16		X	X	X	X	X	X	X
RF 17		X	X	X	X	X	X	X
RF 18						X		
RF 19						X		
RF 20		X						
RF 21				X	X			
RNF 00	X							
RNF 1	X						X	
RNF 2	X						X	
RNF 3			X					
RNF 4							X	
RNF 5							X	
RNF 6							X	
RNF 7							X	
RNF 8	X						X	
RNF 9	X							
RNF 10		X	X	X	X	X	X	X
RNF 11	X							
RNF 12		X	X	X	X	X	X	X
RNF 13		X	X	X	X	X	X	X
RNF 14	X						X	
RNF 15	X						X	
RNF 16	X							
RNF 17	X							
RNF 18								X
RNF 19								X
RNF 20		X						
RNF 21				X				
RNF 22					X			
RNF 23			X					

Tabla 58 - Matriz de trazabilidad Requisito-Caso de Uso

Como se puede observar los requisitos propuestos completan la funcionalidad que se pretende ofrecer según los casos de uso establecidos. Todo caso de uso tiene al menos un requisito asociado y viceversa.

3.3 Diseño del sistema

En esta sección se procederá a la explicación sobre el diseño del sistema propuesto. Por un lado, se establecerá el entorno tecnológico, que ha sido fundamental para sentar las bases de este diseño, por otro lado, y, secuencialmente, se dispondrá un análisis sobre los diferentes componentes que conforman el sistema y sus respectivas funciones dentro de éste.

3.3.1 Arquitectura del sistema

La arquitectura desarrollada se compone principalmente de dos componentes básicos: un nodo cliente y un nodo servidor. Adicionalmente este diseño hace las veces de interfaz para un alto nivel (simulado en nuestro caso), que sería capaz de acceder a los otros nodos implementados a través del cliente. Como no tenemos un alto nivel, primero vamos a fijarnos en el diseño del sistema de forma global presentado con la Ilustración 33.

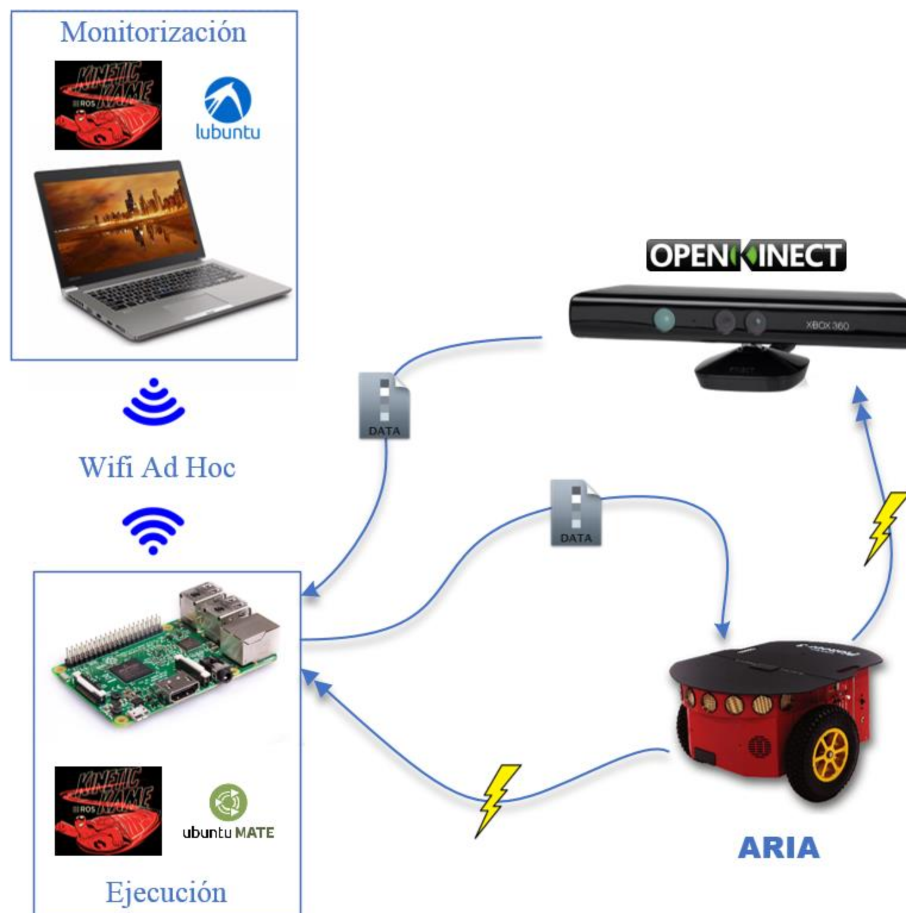


Ilustración 33 – Esquema del entorno operacional

Como se observa en dicha ilustración, la Raspberry está conectada al robot por el cable SERIE y a la Kinect con un adaptador que transforma la conexión propietaria a USB. La alimentación de ambos dispositivos es suministrada por el

robot (Esto se analizará en el Anexo A - Configuración de las conexiones). Mientras que el robot tiene su microcontrolador, la placa hace uso de Ubuntu Xenial y ROS (rosaria) para transmitir datos al microcontrolador con ayuda de ARIA. Luego vemos un portátil cuyo objetivo es monitorizar. Éste se comunica utilizando una conexión wifi adhoc con la Raspberry, y lanza y ejecuta los *spawners* y *joints* para monitorizar el trabajo de la placa en Rviz. Estos elementos sirven únicamente para crear una representación virtual del robot en Rviz y que éste sepa cuáles son las posiciones globales de las diferentes piezas del robot en el mundo, permitiendo una visualización y supervisión remota de la plataforma.

Una vez vista la arquitectura de forma global, nos dedicaremos a explicar la arquitectura que se encuentra en la Raspberry, que es la base del sistema desarrollado. Su descripción sería la siguiente:

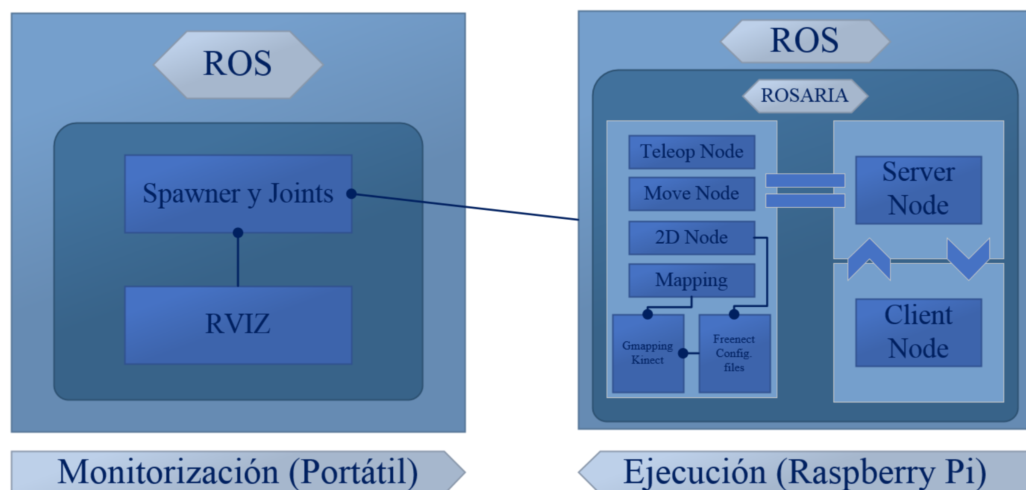


Ilustración 35 - Descripción de los componentes

3.3.2 Descripción general del sistema

El sistema, basado en ROS, se conforma con una serie de nodos. Al inicio del arranque se lanzarán los nodos necesarios para levantar la interfaz, el nodo *p3dx_sensors* comienza a publicar el estado a bajo nivel del robot, se lanza rosaria para activar los drivers y encoders del p3dx, posteriormente se activa la pila de navegación de ROS y, finalmente, se lanza el nodo servidor para que lea las peticiones del cliente. En este punto ya tenemos todo lo necesario para que el nodo cliente comience a generar tareas para el robot.

A continuación se muestra el archivo de lanzamiento (.launch) que se ejecuta al inicio del sistema para levantar todos los módulos de ROS, así como el nodo maestro de éste:

```
<launch>

<!-- Starting rosaria node for p3dx control-->
<include file="$(find p3dx_nav)/launch/p3dx_rosaria.launch"/>

<!-- Starting move base for navigation -->
<include file="$(find p3dx_mb)/navigation_p3dx.launch"/>

<!-- Starting sensors for p3dx messages -->
<node name="sensors" pkg="p3dx_nav" type="p3dx_sensors"/>

<!—- Start server for tasks -->
<node name="server" pkg="p3dx_nav" type="p3dx_server"/>

</launch>
```

Sección de código 1 - Lanzador de la arquitectura

El nodo publicador permite a cualquier interfaz de alto nivel descubrir las características del mundo del robot. Como ya se ha comentado, ROS implementa una serie de tópicos y mensajes que permiten la comunicación entre nodos [70]. Para permitir la posibilidad de comunicación entre el robot y ese alto nivel, se implementa el nodo *p3dx_sensors* que publica constantemente información en un mensaje propio. La definición del mensaje es la siguiente:

```
Header header          //Frame info
bool motors_state      //Motor state
bool frontBumpers      //Front bumpers state
bool rearBumpers       //Rear bumpers state
float64 battery_state  //Battery voltage
float64 posX           //Pos x on Odom frame
float64 posY           //Pos y on Odom frame
float64 posW           //Pos w (Angle in rad) on Odom frame
```

Sección de código 2 - Definición del mensaje p3dx_sensors

Básicamente el **publicador** se subscribe a los *topics* generados por RosAria a través de Aria, los procesa y encapsula en un mensaje que se publica en un topic nuevo (*/p3dx_sensor*) para que puedan ser leídos por el alto nivel. Teniendo así una definición del estado a bajo nivel para que cualquier interfaz de alto nivel sea capaz de convertirlo para utilizarlo en su planificador.

Por otro lado, la **pila de navegación** de ROS [71] nos permitirá implementar el movimiento del robot en un mapa 2D de una forma sencilla y eficaz. Además, nos dará la capacidad de desarrollar la evasión de obstáculos primitiva, ya que no incorpora AMCL (evasión y navegación global en un mapa estático proporcionado manualmente) debido a las restricciones e incompatibilidades con otros nodos como el de mapeo. Y es que resultaría imposible navegar globalmente en un mapa que se está generando dinámicamente, pues habrá ciertas partes del mismo que, lógicamente, no existirán porque no han sido mapeadas. Esta evasión de obstáculos se basa en la creación de una cola de puntos 2D, así, si los sensores descubren un

obstáculo en el camino, se generaría otra serie de puntos a alcanzar que evitarían el obstáculo, cuya distancia, posición y forma se calcula con anterioridad al movimiento haciendo uso de los datos proporcionados por los sensores; y generaría en última instancia, un nuevo conjunto de puntos a alcanzar. Es bastante importante añadir que la generación de los puntos mencionados se hace de forma dinámica, por lo que se destaca la necesidad de un margen de error moderado y una alta capacidad de cómputo. Los datos de los sensores se resumen en los puntos de nube de los sónares, publicados por el nodo rosaria (*/sonar_pointcloud2*) y los datos publicados por el sensor Kinect transformados a escaneo láser (*/scan_depth*) con el nodo *depth_image_to_laserscan*. Estos últimos, además, servirán para la tarea de mapeado pues, como ya se ha mencionado, existen dos posibilidades: correr la pila de navegación con el nodo de mapeo, que viene con sus propios algoritmos de localización (Montecarlo) o correrlo directamente con un mapa estático, al que tendríamos que añadir el nodo AMCL para la planificación de larga duración (global). Como nuestro objetivo es mapear y abordar el problema del SLAM, solo necesitaríamos el primero.

Ahora bien, la estructura de la pila de navegación de ROS y los archivos de configuración necesarios para tenerla completamente operativa se pueden ver definidos en la siguiente imagen:

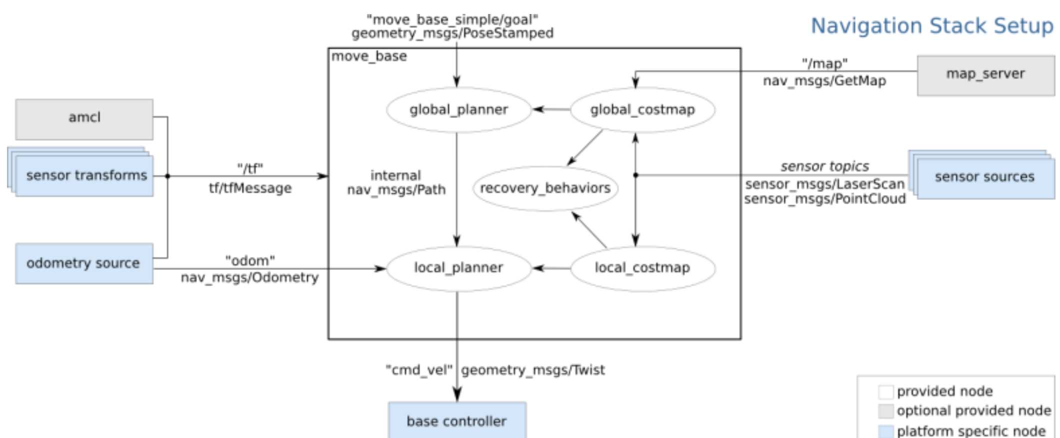


Ilustración 36 - Esquema general de la pila de navegación de ROS [71]

Como se puede observar, la pila necesita una serie de archivos de configuración, estos archivos son leídos al lanzamiento de esta para configurar aspectos importantes como la posición de los sensores, la forma y espacio que ocupa el robot, qué tipo de movimiento tiene, ciertos umbrales de error para éste, etc.: De la parte izquierda ya lo tenemos todo pues la fuente de odometría (posiciones relativas de la plataforma en el mundo) nos la da rosaria, mientras que AMCL (Algoritmos Montecarlo para la planificación de larga duración) ya hemos comentado que no lo necesitamos. Por otro lado, las transformadas de los sensores ya las tenemos, dadas por rosaria, a excepción del sensor Kinect; Para nuestro sistema necesitamos saber en todo momento de dónde vienen (posición exacta con respecto al *frame* en el que trabaja el robot) los datos de los sensores, esto será vital

para generar el mapa de costes con las distancias y formas relativas correctas, dichos datos nos ayudarán tanto en el mapeo como en la navegación. Así, necesitaremos lanzar una transformada estática cuando se lanza el driver del sensor. Esto le dirá a la pila de navegación cuál es el origen de las lecturas láser con respecto al *frame* de la odometría para que la plataforma robótica sepa exactamente dónde está posicionada en el mapa creado y cuál es la distancia y forma relativa de los obstáculos que puede encontrarse con las lecturas que proporciona la Kinect.

En la siguiente sección nos encontramos con el *base_controller*. Este es proporcionado por rosaria, pudiendo publicar mensajes *Twist* a través de este nodo. El *base_controller* se refiere básicamente al nodo de rosaria, pues es el que nos da la capacidad de mandar comandos de movimiento (*Twist*) al robot a través de ARIA, y nos permite mover o girar la base robótica. Por otro lado, a la derecha, ya tenemos los *topics* de los sensores, que en nuestro caso son los sonares y la Kinect transformada a escaneo láser con el nodo *depth_image_to_laserscan* [72] y, como ya se ha mencionado, son publicados con el lanzamiento de rosaria (sonares) y del nodo *freenect* (Kinect). Finalmente, tenemos el *map_server*. Este nodo nos proporciona varias funcionalidades, la primera, que en nuestro caso no es aplicable, nos da la oportunidad de leer un mapa de costes ya generado y proporcionado a la pila manualmente a través de los archivos de configuración para la navegación global; como nuestro objetivo no es este, nos vamos a centrar en la segunda funcionalidad: la implementación del servicio *map_server map_saver*, que nos va a permitir leer la bolsa de datos generada por los sensores durante el servicio de mapeo (*gmapping*) para generar un nuevo mapa (definido por una imagen y un archivo de configuración como en secciones anteriores se ha mencionado). Así, el resultado final de su ejecución sería la creación de un mapa de costes en 2D [73] abordando el problema del SLAM que nos daría la oportunidad de configurar la plataforma para la navegación global (aunque esto no es un objetivo del proyecto es una posibilidad). Es decir, sería posible utilizar un mapa generado y llevar a cabo una navegación global completamente autónoma, no obstante, esto no se ha llegado a desarrollar (debido a la gran extensión del trabajo resultante) y se ha decidido dejar para el desarrollo en un futuro. Independientemente de esto último, no tendría mayor complicación que configurar correctamente los archivos de configuración globales que se introducirán a continuación. Concretamente se debería establecer el parámetro `static_map` a *true* en la pila de navegación, modificar los archivos de configuración globales ajustando los planificadores preferidos y el `global_frame` que sería a */map*. Finalmente se añadiría al lanzamiento de la pila de navegación el nodo AMCL, nodo que implementa algoritmos de localización probabilísticos que nos permite tener completa autonomía de movimiento en el robot.

Así, como se puede observar, solo nos quedan los elementos centrales, que son los verdaderamente importantes pues son diferentes en cada plataforma robótica.

Los archivos *local_costmaps* y *global_costmaps* guardan información sobre el mundo, configuraciones del mapa de costes que servirían para navegar localmente, esquivando obstáculos y planificando a corto plazo, y globalmente, haciendo planes a largo, respectivamente. Además, en estos archivos vienen definidos cuales son los *topics* donde se deben leer los datos de los sensores, datos necesarios para construir el mapa de costes en un futuro. Sin embargo, el archivo *global_costmaps*, en nuestro caso, es menos importante pues, de la manera en la que está implementado en ROS, su objetivo es hacer una comparación de las lecturas con un mapa estático y hacer planes globales con respecto a los datos recabados; como no tenemos un mapa estático, sino que lo creamos, no hace falta configurarlo. Además, esto en específico sería tarea de la arquitectura por encima de esta interfaz.

De aquí nos vamos a los archivos *local_planer* y *global_planer* que servirán como configuración de los planificadores (local y global) tomando los datos de los archivos *local_costmaps* y *global_costmaps* para definir el mundo, las características del robot y publicar velocidades seguras con respecto a las lecturas de los sensores. Estos archivos nos ayudarán en la generación del mapa y el control del movimiento de la plataforma robótica en la navegación y evasión de obstáculos. Básicamente contienen información sobre los atributos físicos del robot, las fuentes de odometría, los *topics* donde escribir comandos de movimiento, etc. Finalmente nos quedan los *recovery_behaviours* que supondrán la definición de las acciones cuando el robot se haya perdido en el mundo o se haya quedado encajado en algún sitio. El objetivo de este archivo es que, si alguna vez el robot se pierde, sepa recuperar la fuente de odometría y su transformada con respecto al origen, es decir, sea capaz de localizarse en el mapa que está creando si está errante.

Así, todos estos archivos son leídos al lanzamiento por la pila de navegación para configurar las características específicas del robot que estamos utilizando. Debemos recordar que ROS pretende ser un *framework* para una gran variedad de plataformas robóticas; con todos estos archivos conseguimos configurar todas las funcionalidades de su pila de navegación ajustándonos a la nuestra. Inicialmente todas ellas están implementadas de forma genérica y debe ser el desarrollador el que aporte toda la información relevante a cerca de la plataforma con la que está trabajando.

Por otro lado, seguimos con el nodo **servidor**, que se ejecuta continuamente en la placa y está constantemente a la espera de que se produzca una petición del cliente leyendo el canal de información definido por un mensaje tipo srv. Cuando captura la petición y la descodifica, lee la tarea y la ejecuta. Tras su finalización

genera una respuesta dependiendo de la salida de esta y recupera la memoria para seguir procesando posibles peticiones.

Esto anterior nos lleva a presentar el nodo **cliente**, el cual realiza las peticiones al servidor, codificando un conjunto de tareas de una manera específica. Para ello ROS permite crear un tipo de mensaje especial denominado srv [74]. Así, se implementa un nuevo mensaje tipo *srv* cuya definición es la siguiente:

```
string codigo_tarea    //Código de la tarea a realizar
string[] vector_mover  //Vector de parámetros movimiento básico
string[] vector_girar  //Vector de parámetros de giro
string[] vector_x_y    //Vector de parámetros de movimiento 2D
---                    //Sección de respuesta
string feedback        //Respuesta en forma de cadena
int64  eval            //Respuesta en forma booleana
```

Sección de código 3 - Definición del srv

Este permitiría codificar la petición para poder ser interpretada en el *back-end* y añadiría una respuesta (que puede contener múltiples campos). En nuestro caso, esta respuesta es una tupla formada por un booleano y un mensaje de *feedback*. Nótese que en la misma definición encontramos tanto la codificación de las tareas (petición) como de la respuesta y que son totalmente extensibles y modificables.

Cabe destacar, no obstante, que, en todo momento, tanto servidor como cliente, generan archivos de logs que guardan tanto las peticiones realizadas, como la fecha en la que se hicieron. Esto serviría como un método para el depurado de un supuesto sistema completo, en la que dos interfaces (ya sea PELEA u otra arquitectura de alto nivel) y el bajo nivel vayan haciendo y recibiendo peticiones de tareas consecutivamente. Teniendo en nuestro caso una manera fácil de saber y entender qué ha hecho el sistema por debajo.

Así, ahora que ya está todo lo necesario activado se podrán realizar las peticiones de distintas tareas, que vamos a definir a continuación. Las tareas se resumen en nodos que pueden llegar a funcionar independientemente, siempre y cuando rosaria esté activo, y se formalizarán en la sección Descripción de los componentes.

3.3.3 Descripción de los componentes

En esta sección se van a formalizar las tareas implementadas en el sistema. Cada una de estas tareas se describirán con una terna: El código, la referencia, que es el nombre por el cual se registra en los archivos de log, y la función.

3.3.3.1 Inicio de motores

Código de tarea: initMotors

Referencia: Inicio motores.

Función: El nodo que implementa esta tarea se utiliza para arrancar el robot si este no se ha iniciado (aunque por defecto los motores se inician cuando rosaria se conecta al robot). La implementación se basa en conectarse con ARIA a través de rosaria y utilizar el servicio de inicio de motores para activarlos. Si se da el caso de que los motores ya están activos, el servidor procesa esta orden, pero la ignorará no realizándola.

3.3.3.2 Movimiento básico lineal

Código de tarea: moveForward

Referencia: Move linear *speed* m/s *distance* m *isForward* dirección.

Función: Si se da el caso de que el alto nivel quiera dar tareas específicas, el sistema cuenta con este nodo y el siguiente. Esta tarea define la posibilidad de asignar al robot recorrer una distancia determinada, con una velocidad y una dirección pedida. Haciendo uso del concepto de las transformadas, el nodo publica en el *topic* de rosaria la velocidad introducida y compara en un bucle la transformada origen con la actual para saber la distancia recorrida. Cuando se llegue al valor introducido de distancia la ejecución finaliza. Este concepto se utiliza aquí de forma básica por eso se prefiere explicar más detalladamente en la tarea de navegación en 2D, donde se hace un uso mucho más complejo.

3.3.3.3 Movimiento básico de giro

Código de tarea: twist

Referencia: Twist robot *speed* deg/s *degrees* sex. grades *cloakw* direction.

Función: Al igual que el anterior nodo, este implementa el movimiento angular de la base del robot. En este caso se dan como parámetros la velocidad, los grados de giro y la dirección (en sentido de las agujas del reloj o no). El desarrollo de este nodo guarda muchas similitudes con el anterior pues internamente hace uso de la comparación de transformadas para descubrir el ángulo de giro actual y parar cuando se ha llegado al pedido.

3.3.3.4 Navegación en 2D (Kinematic controller)

Código de tarea: moveTo

Referencia: Move to position *pos_x* X *pos_y* Y *pos_w* grades.

Función: El nodo desarrollado llamado *Nav_waypoints* implementa el movimiento del robot a través de un controlador capaz de realizar el cómputo de las velocidades lineal y angular. La idea detrás de este controlador subyace en el fundamento matemático de un movimiento en un plano 2D.

Para explicar el funcionamiento del nodo vamos a empezar introduciendo el flujo del sistema: el sistema toma una entrada, que será el estado objetivo y un estado actual. Con estos datos hace el cómputo de las variables de control (velocidades lineal y angular) y devuelve un *feedback* íntegro a la entrada para volver a repetir el ciclo hasta alcanzar la meta, ajustando los parámetros de movimiento a cada instante que toma nuevas posiciones. Nuestro estado en este caso viene definido por la tupla $\overrightarrow{X_t} = [x, y, \theta]$ que se identifica con la posición del robot en un tiempo t con respecto a un frame determinado. Si consideramos el problema de moverse hacia un punto cuyo estado objetivo es el siguiente: $\overrightarrow{X_t} = [x', y', \theta']$, tenemos dos cuestiones que discutir: el cálculo de la velocidad lineal (V') que será un valor estimado proporcional a la distancia entre el estado actual y el objetivo y, el cálculo de la velocidad angular (γ) que supondrá la decisión de girar la base del robot en dirección al objetivo marcado inicialmente. Así, matemáticamente [75] podríamos definirlo de la siguiente manera:

$$V' = K_v\sqrt{(x'-x)^2 + (y'-y)^2}$$

$$\theta' = \tan^{-1}\frac{y'-y}{x'-x}$$

$$\gamma = K_h(\theta' \ominus \theta),\ K_h > 0$$

Como se puede observar según x se acerca a x' (al igual que θ' a θ), las velocidades van disminuyendo proporcionalmente hasta llegar a 0, valor para el cual la base robótica ya ha llegado al punto requerido.

Ahora bien, sabiendo el fundamento teórico detrás del controlador, ¿cómo lo implementamos en ROS? Para empezar, debemos establecer las unidades en las que trabaja ROS, que es en metros para distancia, radianes para ángulos y segundos para el tiempo; esto hace que las velocidades tomen m/s y radianes/s, respectivamente. Por otro lado, debemos tener en cuenta los datos utilizados: una tupla que representa la posición actual y otra que representa el objetivo. Para sacar la posición del robot en un tiempo t, ROS tiene una implementación bastante útil de las transformadas entre *frames* de coordenadas. Una plataforma

robótica puede (y generalmente tiene) una gran cantidad de *frames* de coordenadas que pueden modificarse según pasa el tiempo, la librería tf nos permite almacenar en un buffer las posiciones en un tiempo t en las que el robot ha estado y, de forma algo más compleja, hacer una estimación de las transformadas futuras en caso de necesitarlo [76]. En el nodo se implementa la llamada al *listener* de tf, implementado en el *stack* de ROS, y se espera a que devuelva una transformada de la odometría desde el *frame* de la base (*base_link*). Una vez el *listener* es capaz de capturar la transformada se almacena como la actual. Con el objetivo de comparar la transformada actual con la objetivo, se instancia una nueva transformada en el instante inicial del movimiento con los datos introducidos por el cliente. Tanto para esta tarea como para la extracción de las posiciones y del ángulo de rotación (que serán utilizados posteriormente por el método que implementa el cálculo de las velocidades) se necesita saber cómo están estructurados los datos que porta el objeto, la transformada.

Vamos a partir del mensaje estandarizado de ROS para la odometría del robot: *geometry_msgs/Pose.msg*; este mensaje se estructura en dos partes importantes un *geometry_msgs/Point* que guarda la posición 2D del robot $[x, y]$ y un *geometry_msgs/Quaternion* que guarda la orientación del robot en forma de quaternion $[x, y, z, w]$. El *quaternion* es una representación de un giro en 3D que se puede reducir a lo que se conoce como sistema YPR (Yaw (ψ), Pitch (θ), Roll (ϕ)). Como esta representación no está ideada para hacer cálculos en un contexto de movimiento en 2D se debe realizar un procesado para sacar el ángulo verdadero de giro, que matemáticamente hablando acaba siendo el cómputo [65]:

$$Q = [q_0, q_1, q_2, q_3]^T$$

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} atan2(2(q_0q_1 + q_2q_3), 1 - 2(q_1^2 + q_2^2)) \\ \operatorname{asin}(2(q_0q_2 - q_3q_1)) \\ atan2(2(q_0q_3 + q_1q_2, 1 - 2(q_2^2 + q_3^2))) \end{bmatrix}$$

Aunque no es necesario hacer esos cálculos ya que la librería *tf* cuenta con una implementación de las conversiones y estructuras de datos (matrices y vectores) para realizar y almacenar estos cómputos.

Así pues, una vez obtenida la tupla actual $[x_t,\ y_t,\ \theta_t]$ se hace una comparación con la transformada objetivo que se creó inicialmente. La idea detrás de esta comparación está basada en la composición de transformadas (matrices), se calcula una transformada relativa realizando el producto entre la inversa de la transformada inicial y la actual y, con los métodos que implementa la librería *tf*, comprobamos la distancia en el vector de traslación de la transformada relativa (resultado), que es la implementación de la distancia euclídea entre dos

puntos: $\sqrt{(x' - x)^2 + (y' - y)^2}$, y el ángulo total girado, que será el escalar asociado a la matriz de giro: $(\theta' \ominus \theta)$. Si estos valores son muy pequeños, significa que, o hemos llegado al objetivo, o estamos muy cerca, por eso en cada iteración del bucle se va pasando el *feedback* al controlador de velocidad para que haga el cómputo con los datos actuales del robot hasta que estas distancias sean mínimas (es un parámetro que se puede ajustar), lo que ocasiona la parada del bucle de control y la posterior parada del robot. Así, el robot va ajustando las velocidades con la odometría nueva hasta llegar al punto objetivo.

No obstante, como cabe la posibilidad de que el robot nunca llegue a su destino (algo que se comunicará al planificador de alto nivel para establecer un posible obstáculo en el camino), se ha implementado una comprobación en el nodo que no permite ejecutar el controlador si ha pasado más de un tiempo determinado, asumiendo que tras este límite el robot se halla atascado en un mismo punto. Se podrían comprobar las posiciones del robot en una serie temporal y parar la ejecución cuando se hayan repetido X veces, pero la primera opción es la más asequible y sencilla de implementar.

Nótese que esta misma implementación la ofrece el módulo *move_base* de la pila de navegación y que se podría utilizar de manera alternativa.

3.3.3.5 Inicio servicio de mapeo

Código de tarea: startMapping

Referencia: Init mapping service.

Función: Es posible que el alto nivel requiera la generación de un mapa ya que la idea es que éste comience con ninguna o muy poca información del mundo. Es por esto por lo que se incluye el servicio para iniciar el mapeo de una zona. Esta tarea implementa el lanzamiento del nodo *gmapping* (Sección de código 4) [77], que crea una bolsa de datos (lecturas de los sensores) para definir en un futuro el mundo con sus obstáculos en un mapa de costes en 2D.

Nótese que es de vital importancia establecer un número de partículas moderado pues para valores mayores de 30-40, el sistema se ve muy ralentizado por el procesamiento de los puntos escaneados. Por otro lado, el valor `map_update_interval` nos permite modificar cada cuanto tiempo se debe esperar para ver el mapa actualizado. Finalmente, hay que destacar que los valores `*_min` permiten decir al algoritmo de SLAM el tamaño inicial del mapa para que se ajuste a nuestras condiciones, incrementando la eficacia del proceso.

```
<launch>
      <node pkg="gmapping" type="slam_gmapping"
name="slam_gmapping" output="screen" >
             <param name="xmin" value="-10"/>
             <param name="xmax" value="10" />
             <param name="ymin" value="-10" />
             <param name="ymax" value="10" />
             <param name="particles" value="30" />
             <param name="map_update_interval" value="2.0" />

             <param name="delta" value="0.02" />
        <remap from="scan" to="/camera/scan_depth"/>
      </node>
</launch>
```

Sección de código 4 - Launcher para el mapeo

3.3.3.6 Fin servicio de mapeo

Código de tarea: endMapping

Referencia: End mapping service.

Función: Cuando el alto nivel haya decidido que tiene información suficiente del mundo, la tarea de mapeo se puede finalizar con esta petición. Esta tarea implementa la finalización del servicio de mapeo con SIGTERM, terminando la ejecución y creando el mapa resultante de las lecturas de la tarea anterior.

Para crear el mapa se utiliza el nodo *map_server* anteriormente visto en la pila de navegación. A este nodo se le hace una llamada a la función *map_saver* que recoge los datos de los sensores recabados hasta este momento y los convierte en un mapa del mundo. El mapa viene definido por un archivo .yaml y una imagen en formato .pgm. El archivo .yaml define el origen de movimiento, la resolución de la imagen y entre otras cosas menos relevantes, la dirección en la que se encuentra ésta, que es un archivo de extensión .pgm donde se representan los datos de ocupación recabados por los sensores.

Así, tras la finalización del servicio de mapeo, el sistema guardará dos archivos en la siguiente dirección: *root/maps*. Estos archivos se podrán utilizar para la navegación autónoma y global (implementando algoritmos AMCL) a través de la pila de navegación de ROS.

Nota: El uso del comando `rosrun image_view image_view image:= /camera/rgb/image_color` puede ser de gran ayuda para observar lo que el sensor Kinect está viendo.

A continuación, se muestra un ejemplo del uso del servicio de mapeo:

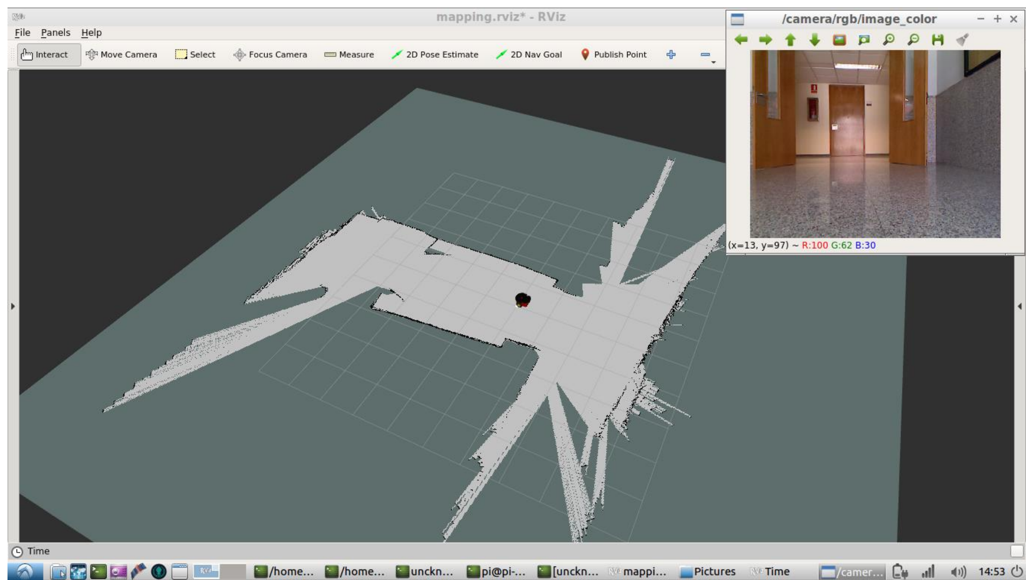


Ilustración 37 - Visualización del servicio de mapeo

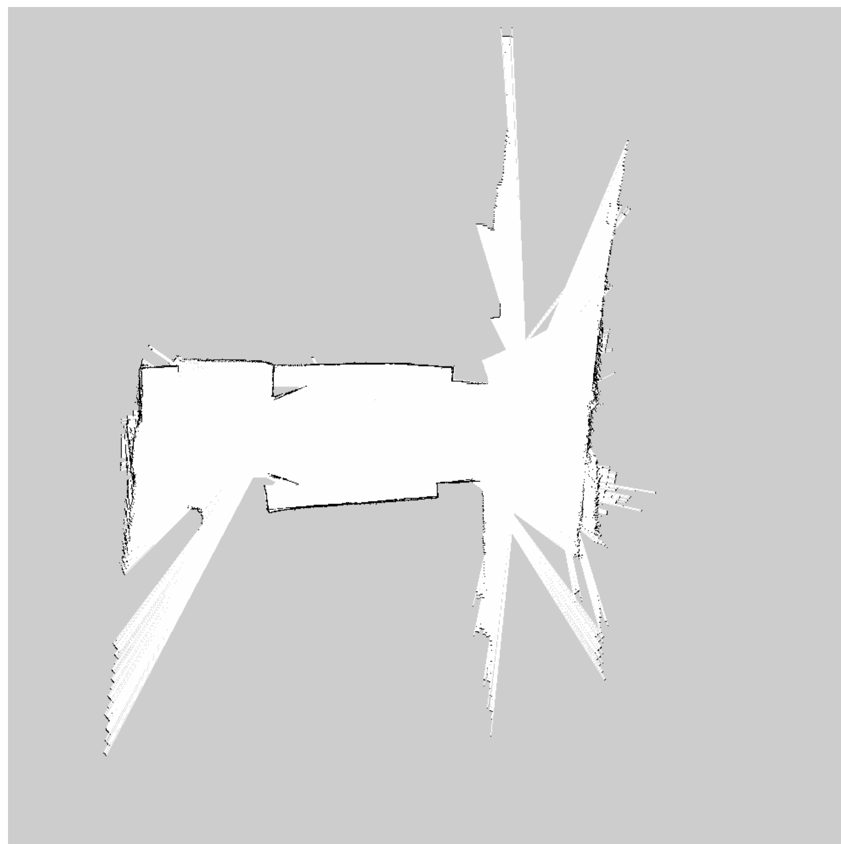


Ilustración 38 - Mapa generado tras finalizar el servicio

3.3.3.7 Teleoperación

Código de tarea: teleop

Referencia: Teleop node.

Función: La teleoperación podríamos definirla como la capacidad de controlar la base del robot desde la distancia con algún dispositivo como un joystick o un teclado. En este caso se ha desarrollado un nodo que se conecta al robot a través de rosaria y publica mensajes de velocidad (*Twist*) para controlar el robot por teclado. Así, el nodo implementa un bucle infinito que va leyendo los eventos del teclado, y dependiendo de las teclas pulsadas, ejecuta acciones diferentes. Las teclas se describen como:

- Flechas de dirección → movimiento lineal o angular.
- i/k → Aumentar/Disminuir velocidad lineal.
- u/j → Aumentar/Disminuir velocidad angular.
- q → Finalizar ejecución.

Este nodo permite controlar el robot de manera supervisada. Puede ser muy útil, por ejemplo, en el caso de que se quiera supervisar el mapeo de una zona compleja.

3.3.4 Topics generados y utilizados por el Sistema

Debido a que ROS trabaja con una transferencia de datos a través de *topics* se debe dedicar una sección para explicar cuáles son los que se generan en el sistema propuesto y cómo se podrían utilizar en el caso de tener un componente adicional que quiera hacer uso de éste. A continuación, se podrá encontrar una tabla que resumirá cada topic con su utilidad y descripción:

Topic	Tipo	Descripción	Tiempo
/p3dx_sensors	Lectura	Encapsula el estado a bajo nivel del robot. El alto nivel puede leerlo para saber el estado actual del robot.	10Hz
/cmd_vel	Escritura	*Topic* en el que se publican comandos de velocidad a la plataforma a través de RosAria. Útil para el movimiento.	10Hz
/tf	Lectura	Contiene información sobre los diversos *frames* de coordenadas en los que se ejecuta el sistema.	20Hz
/scan_depth	Lectura	Contiene una descripción con los datos de lectura de la Kinect transformados a	10Hz

		escaneo laser. Muy útil en el caso de querer implementar un sistema SLAM.	
/map	Lectura/ Escritura	*Topic* en el que se publica el mapa de costes creado con el servicio de mapeo a través de las lecturas de la Kinect.	20Hz
/move_base/ *	Lectura	*Topics* generados por la pila de navegación de ROS para la evasión de obstáculos y la gestión de fuentes de odometría y sensores. (* significa que son múltiples, pero no se describen individualmente por la gran cantidad que se generan). Realmente son *topics* autogestionados por ROS que no deberían tocarse.	20Hz
/~entropy	Lectura	Entropía asociada a la posición del robot en el mapa creado. Es importante tener esto encuentra pues existe diferencias entre las tranformadas entre distintos *frames* de coordenadas.	20Hz

Tabla 59 - Topics en el sistema

Como se puede observar, utilizando los *topics* anteriormente descritos es posible complementar la arquitectura propuesta de distintas maneras. Por un lado, sería posible implementar un algoritmo propio de SLAM, teniendo en cuenta que las lecturas de la Kinect (*/~scan_depth*) son escaneos laser, y que siempre sería posible mover la base del robot escribiendo en el topic */cmd_vel* estableciendo la velocidad lineal en X y angular en Z. Además, siempre seríamos capaces de leer el estado del robot y saber su posición con el *topic /p3dx_sensors*. Por otro lado, sería posible realizar un SLAM en 3D evitando lanzar el nodo *Depth_image_to_laserscan* y haciendo uso del *topic /camera/pointclouds2* que ofrece los puntos de nube para construir las lecturas de la cámara en formas en 3D, es decir, aquí solo se describen los *topics* utilizados en el sistema; se tiene que tener en cuenta que los diferentes nodos utilizados publican más mensajes que son utilizables en otros aspectos.

3.3.5 Diagramas de secuencia

A continuación, se van a presentar los diagramas de secuencia que definen el flujo de ejecución. Se proporcionan dos: el primero pretende establecer la ejecución del inicio del sistema, el segundo supone la especificación de la ejecución de las tareas. Este último es extrapolable a todas las tareas disponibles, es por esto que no se proporcionan diagramas de secuencia independientes.

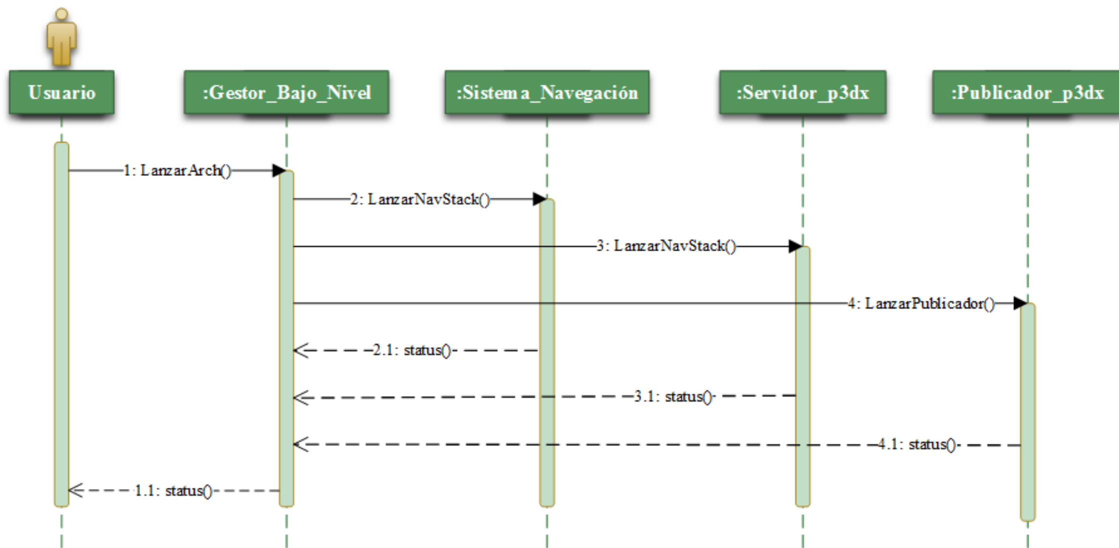


Ilustración 39 - Diagrama de secuencia Inicio de ejecución

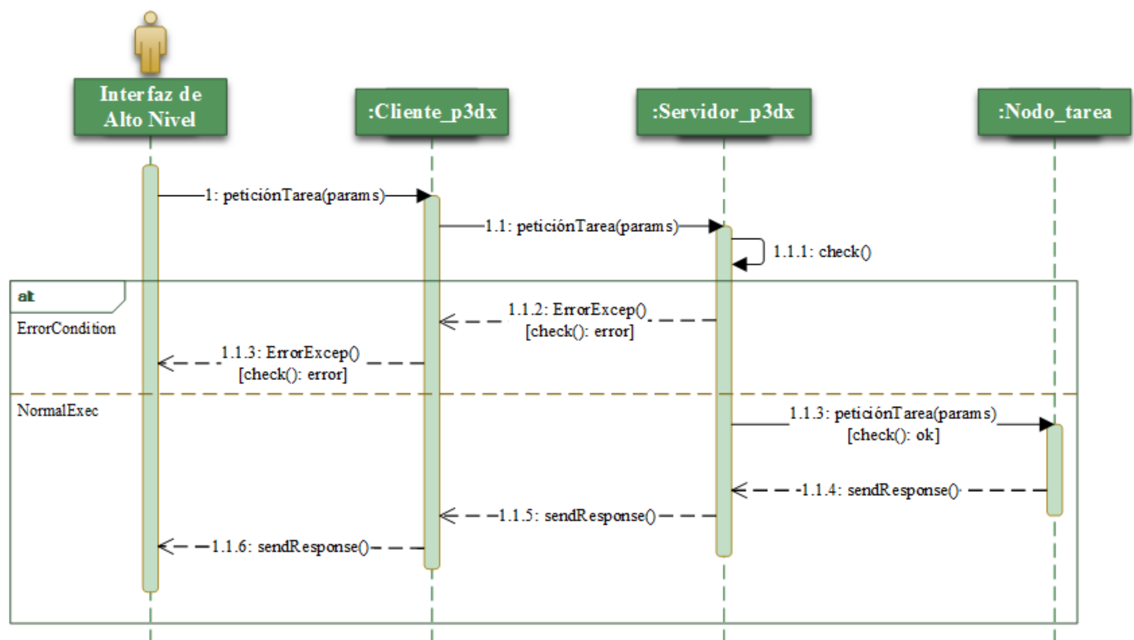


Ilustración 40 - Diagrama de secuencia general del sistema de peticiones de tareas

Capítulo 4: Experimentación

4.1 Introducción

En esta sección se procederá a realizar una descripción detallada de las pruebas realizadas para comprobar el correcto funcionamiento del sistema y el entorno de éstas. Así, para la evaluación que se propone, se tendrán en cuenta diferentes fuentes como pueden ser casos de uso, requisitos del sistema, etc. ya especificados.

De esta manera, nos encontramos con dos tipos de pruebas fundamentales: Pruebas unitarias, de caja negra; que se encargarán de probar los diferentes módulos y componentes del sistema por separado, y Pruebas del sistema, mucho más completas, pues se basan en comprobar el correcto funcionamiento del sistema completo bajo el entorno definido. Debido a que en este tipo de sistemas es muy importante la efectividad y la eficacia, se van a realizar diferentes modificaciones del entorno de pruebas para verificar distintos puntos relevantes en el desempeño del sistema. Es importante añadir que el hardware y el software se mantendrán estáticos durante las pruebas, pero éstas se realizarán y contrastarán con variaciones sobre dónde se ejecuta el software. Esto nos permitirá estudiar y ahondar en las posibilidades del sistema y su eficiencia cuando la carga de trabajo de los módulos de la arquitectura se reparte entre distintas máquinas.

4.2 Entorno de pruebas

Este apartado se utilizará para describir los elementos hardware y software del entorno sobre el que se va a realizar el conjunto de pruebas.

4.2.1 Hardware

- Placa Raspberry Pi 3 Modelo B.
- Portátil de monitorización.
- Robot P3DX.
- Sensor Kinect.
- Cables de alimentación y datos para la placa y el sensor.

4.2.2 Software

- Sistema Operativo Ubuntu Xenial (16.04) en placa y portátil.
- *Framework* de ROS en su distribución Kinetic.
- Ambas máquinas poseen los nodos ya explicados y servidor/cliente SSH.
- El portátil posee una implementación de Rviz.

4.3 Especificación de pruebas

4.3.1 Pruebas unitarias

Este apartado se utilizará para el testeo de los componentes y módulos que conforman el sistema. Estos componentes se reducen a las tareas que el sistema puede realizar y los diferentes nodos implementados que gestionan otros sistemas y son fundamentales para el correcto funcionamiento. Para la formalización de estas pruebas se utilizará el siguiente formato tabular:

ID		Título	
Objetivo			
Fuente/s			
Procedimiento			
Resultado esperado			

Tabla 60 - Formato tabular para las Pruebas unitarias

y sus campos son:

- **ID:** Código identificativo único de cada prueba unitaria. La nomenclatura utilizada será TU-XX siendo XX cualquier número entero desde el 00.
- **Título:** Breve descripción de la prueba en cuestión. Generalmente una palabra que describe el nodo o módulo que se prueba.
- **Objetivo:** Aquello que se pretende conseguir mediante la realización de la prueba.
- **Fuente/s:** Elementos de los que emana la prueba en cuestión. Generalmente estos serán Casos de uso o Requisitos del sistema.
- **Procedimiento:** Parámetros introducidos y procesos que se llevan a cabo para la realización de la prueba.
- **Resultado esperado:** Estado final del sistema, salidas resultantes tras la ejecución de la prueba.

4.3.1.1 Descripción de las pruebas unitarias

ID	TU-00	Título	Publicador
Objetivo	Comprobación de las publicaciones del *topic /p3dx_sensor*, elemento base para la comunicación con el módulo de monitorización y la arquitectura de alto nivel.		
Fuente/s	CU-05, RF-18, RF-19, RNF-12		
Procedimiento	1. Levantar sistema (Roscore). a. Lanzar nodo servidor. b. Lanzar nodo cliente. c. Lanzar nodo rosaria. d. Lanzar publicador.		
Resultado esperado	El comando `rostopic echo /p3dx_sensor` muestra el estado a bajo nivel del robot en consola en base al mensaje personalizado creado.		

Tabla 61 - Prueba Unitaria TU-00

ID	TU-01	Título	Movimiento básico
Objetivo	El sistema es capaz de mover linealmente y girar la plataforma robótica.		
Fuente/s	CU-07, RF-04, RF-05, RF-6, RF-08, RF-09, RNF-10, RNF-13, RNF-18, RNF-19		
Procedimiento	1. Levantar sistema. 2. Realizar petición desde el cliente con el comando: `rosrun p3dx_nav p3dx_client moveForward [Velocidad] [Distancia] [Dirección]` para moverse linealmente. 3. Realizar petición desde el cliente con el comando: `rosrun p3dx_nav p3dx_client twist [Velocidad] [Grados] [Dirección]` para girar la base. 4. Esperar respuesta del servidor (finalización de la tarea).		
Resultado esperado	El cliente recibe respuesta y la odometría de Rosaria (`rostopic echo /RosAria/odom`) muestra la posición del robot en X=Distancia (m) Y=0 (m) Z=0 (m) y el cuaternión correspondiente al grado introducido: X Y Z W.		

Tabla 62 - Prueba Unitaria TU-01

ID	TU-02	Título	Movimiento 2D
Objetivo	El sistema es capaz de conducir el robot hasta una posición X Y θ dada, del plano 2D en el que se mueve.		
Fuente/s	CU-01, RF-04, RF-04, RF-06, RF-10, RNF-10, RNF-13, RNF-20		
Procedimiento	1. Levantar el sistema. 2. Realizar la petición desde el cliente con el comando: `rosrun p3dx_nav p3dx_client moveTo [P_X] [P_Y] [P_W]`. 3. Esperar la respuesta del servidor.		
Resultado esperado	El cliente recibe respuesta y la odometría de Rosaria (`rostopic echo /RosAria/odom`) muestra la posición del robot en X=P_X (m) Y=P_Y (m) Z=0 (m) y el cuaternión correspondiente al grado introducido en P_W: X Y Z W, cuando el movimiento ha finalizado.		

Tabla 63 - Prueba Unitaria TU-02

ID	TU-03	Título	Teleoperación
Objetivo	El sistema es capaz de controlar el robot con la ayuda de un teclado.		
Fuente/s	CU-02, RF-04, RF-05, RF-06, RF-11, RNF-03, RNF-13, RNF-23		
Procedimiento	1. Levantar el sistema. 2. Realizar petición desde el cliente con el comando: `rosrun p3dx_nav p3dx_client teleop`. 3. Esperar el lanzamiento del nodo de teleoperación. 4. Mover la plataforma con el teclado: Teclas de dirección y modificar velocidades: (i/k) (u/j). 5. Parar ejecución del nodo: q. 6. Esperar respuesta del servidor.		
Resultado esperado	Una vez lanzado el sistema y el nodo de teleoperación, las entradas por teclado en consola permiten controlar la base robótica, la tecla q finaliza el nodo y el cliente recibe respuesta.		

Tabla 64 - Prueba Unitaria TU-03

ID	TU-04	Título	Mapeo
Objetivo	El sistema es capaz de generar un mapa de la zona con las lecturas de los sensores.		
Fuente/s	CU-03, CU-04, RF-04, RF-05, RF-06, RF-12, RF-13, RF-14, RF-15, RF-21, RNF-13, RNF-21, RNF-22		
Procedimiento	1. Levantar el sistema. 2. Realizar petición desde cliente con el comando: `rosrun p3dx_nav p3dx_client startMapping` (inicio del servicio). 3. Mover el robot por el entorno para que generar las lecturas de la Kinect. 4. Realizar petición desde cliente con el comando: `rosrun p3dx_nav p3dx_client endMapping` (fin del servicio). 5. Esperar respuesta del servidor y generación del mapa resultante de las lecturas realizadas.		
Resultado esperado	Se inicia el servicio de mapeo y el comando `rostopic list` muestra el topic de */map* donde se publica el mapa de costes generado por las lecturas de la Kinect. Una vez finalizado el servicio de mapeo este topic desaparece (el nodo *Gmapping* finaliza), el cliente recibe respuesta y el mapa se genera en la carpeta *maps* del proyecto.		

Tabla 65 - Prueba Unitaria TU-04

4.3.2 Pruebas del sistema

A continuación, se introducen las pruebas del sistema que tienen como objetivo la verificación del correcto funcionamiento del conjunto de elementos desarrollados. Para ello se hará una simulación de una arquitectura de alto nivel que se conecta al sistema y comienza la petición de tareas de forma paulatina como si de un plan se tratase.

Para la formalización de este conjunto de pruebas se utilizará el siguiente formato tabular:

ID		Título	
Objetivo			
Procedimiento			
Resultado esperado			

Tabla 66 - Formato tabular para las Pruebas del sistema

siendo cada uno de sus campos:

- **ID:** código identificativo único de cada prueba del sistema. La nomenclatura utilizada será TS-XX donde XX es cualquier número entero empezando desde 00.
- **Título:** Breve descripción de la prueba.
- **Objetivo**: Aquello que se busca conseguir con la ejecución de la prueba.
- **Procedimiento:** Parámetros introducidos y procesos que se llevan a cabo para la realización de la prueba.
- **Resultado esperado:** Estado final del sistema, salidas resultantes tras la ejecución de la prueba.

4.3.2.1 Descripción de las pruebas del sistema

ID	TS-00	Título	Simulación, comunicación
Objetivo	El portátil puede visualizar y monitorizar con Rviz las acciones del robot real.		
Procedimiento	1. Levantar sistema en Raspberry Pi. 2. Lanzar *Spawner* y Rviz en portátil. 3. Modificar el conjunto displays para mostrar el robot, posición y las lecturas de los sensores.		
Resultado esperado	En el portátil se muestra el modelo 3D del robot, su posición y las lecturas de los sensores en la interfaz de Rviz.		

Tabla 67 - Prueba del Sistema TS-00

ID	TS-01	Título	Evasión de obstáculos
Objetivo	El sistema de navegación es capaz de evadir los obstáculos que hay en el camino hacia el objetivo impuesto por el alto nivel.		
Procedimiento	1. Levantar el sistema en Raspberry Pi. 2. Realizar tarea de movimiento en 2D estableciendo un objetivo cuyo camino está obstruido por un objeto (una caja). 3. Monitorizar las acciones de la plataforma robótica y esperar a la respuesta del servidor.		
Resultado esperado	El robot acaba en la posición requerida y es capaz de ajustar los parámetros de velocidad (linear y angular) para evitar colisionar con el obstáculo.		

Tabla 68 - Prueba del Sistema TS-01

ID	TS-02	Título	Mapeo y movimiento 2D
Objetivo	El alto nivel comienza a dar tareas de movimiento 2D de forma secuencial mientras el servicio de mapeo está activo. Simulando el alto nivel, este debería replanificar con respecto a las lecturas que el robot publica. Tras realizar los planes, se finaliza el servicio de mapeo y se genera el mapa.		
Procedimiento	1. Levantar el sistema en Raspberry 2. Realizar petición desde cliente con el comando: `rosrun p3dx_nav p3dx_client startMapping` (inicio servicio). 3. Ejecutar plan: llamadas secuenciales a la tarea de movimiento en 2D. 4. Realizar petición desde cliente con el comando: `rosrun p3dx_nav p3dx_client endMapping` (fin de servicio). 5. Esperar respuesta del servidor y generación del mapa resultante de las lecturas realizadas.		
Resultado esperado	El sistema es capaz de procesar las tareas de movimiento en 2D mientras mapea la zona. Una vez el servicio finaliza genera el mapa de costes resultante de la actividad.		

Tabla 69 - Prueba del Sistema TS-02

ID	TS-03	Título	Mapeo teleoperado
Objetivo	El alto nivel comienza a controlar la base robótica manualmente mientras el servicio de mapeo está activo. Así, se movería el robot por la zona y cuando termine el servicio se genera el mapa.		
Procedimiento	1. Levantar el sistema en Raspberry 2. Realizar petición desde cliente: `rosrun p3dx_nav p3dx_client startMapping` (inicio servicio). 3. Realizar petición desde cliente: `rosrun p3dx_nav p3dx_client teleop` y mover la plataforma con el teclado. 4. Realizar petición desde cliente: `rosrun p3dx_nav p3dx_client endMapping` (fin de servicio). 5. Esperar respuesta del servidor y generación del mapa resultante de las lecturas realizadas.		
Resultado esperado	El sistema es capaz de gestionar los comandos de teclado para mover el robot mientras mapea la zona. Tras la finalización del servicio se detiene el nodo de teleoperación y se genera el mapa.		

Tabla 70 - Prueba del Sistema TS-03

4.4 Metodología

La metodología seguida para evaluar el desempeño del sistema y su funcionalidad será la siguiente: Se van a establecer un total de tres configuraciones distintas, a saber:

En primer lugar, el sistema completo será ejecutado por la placa (Configuración 2). De esta manera, evaluaremos, por un lado, el correcto funcionamiento de los sistemas implementados y, por el otro, la capacidad de la propia placa por si se da el caso de ser incapaz de mover el sistema completo.

En segundo lugar, se procederá a relegar la ejecución del nodo maestro de ROS al portátil de monitorización para aliviar la capacidad de proceso necesaria en la Raspberry (Configuración 1); así, mientras que ésta se encarga exclusivamente de la ejecución de tareas (servidor, publicador, mapeo y navegación), el portátil cargará con la ejecución del sistema operativo de ROS.

En último lugar, se aliviará casi por completo la carga de procesos en la Raspberry. Para ello el sistema operativo de ROS y el procesamiento de imágenes en vivo de la Kinect se pasarán a ejecutar en el portátil (Configuración 3). Esta configuración es, en última instancia, la única posible que permite llevar el conjunto más grande de procesos al portátil, ya que tanto servidor como sistema de navegación tienen la restricción de tener que ejecutarse en la Raspberry.

De esta manera tenemos tres configuraciones con las que podremos realizar una comparación del impacto que producen las fluctuaciones de nodos entres máquinas y nos permitirá valorar la mejor situación en la que utilizar el sistema, exponiendo tanto aspectos positivos como negativos de cada una. Así, para cada prueba, se realizará una evaluación del tiempo de respuesta del sistema, es decir, el tiempo (en segundos) que pasa desde que el cliente solicita la ejecución de la tarea hasta que ésta se termina de realizar. Cabe remarcar, sin embargo, que en las Pruebas del sistema TS-02 y TS-03, que son las que evalúan el mapeo de una zona, se tendrá en cuenta el tiempo efectivo en el que el sistema es capaz de mapear una misma zona. Nótese que esta última evaluación no solo comprenderá tiempos sino también, la calidad del mapa creado.

4.5 Análisis de consistencia

En esta sección se proporciona un método de visualización para comprobar cómo las pruebas realizadas cubren correctamente los requisitos definidos en el apartado.

	TU 00	TU 1	TU 2	TU 3	TU 4	TS 00	TS 1	TS 2	TS 3
RF 00	X	X	X	X	X	X	X	X	X
RF 1	X	X	X	X	X	X	X	X	X
RF 2						X			
RF 3						X			
RF 4		X	X	X	X			X	X
RF 5		X	X	X	X			X	X
RF 6		X	X	X	X			X	X
RF 7		X	X	X					
RF 8		X							
RF 9		X							
RF 10			X						
RF 11				X					X
RF 12					X			X	X
RF 13					X			X	X
RF 14	X				X				
RF 15	X				X				
RF 16	X	X	X	X	X	X	X	X	X
RF 17	X	X	X	X	X	X	X	X	X
RF 18	X								
RF 19	X								
RF 20			X				X		
RF 21					X			X	X
RNF 00	X	X	X	X	X	X	X	X	X
RNF 1	X	X	X	X	X	X	X	X	X
RNF 2	X	X	X	X	X	X	X	X	X
RNF 3				X					X
RNF 4						X			
RNF 5						X			
RNF 6						X			
RNF 7						X			
RNF 8	X	X	X	X	X	X	X	X	X
RNF 9	X	X	X	X	X	X	X	X	X
RNF 10		X	X	X	X			X	X
RNF 11	X	X	X	X	X	X	X	X	X
RNF 12	X	X	X	X	X	X	X	X	X
RNF 13	X	X	X	X	X	X	X	X	X
RNF 14	X	X	X	X	X	X	X	X	X
RNF 15	X	X	X	X	X	X	X	X	X
RNF 16	X	X	X	X	X	X	X	X	X
RNF 17		X	X	X					
RNF 18		X							
RNF 19		X							
RNF 20			X						
RNF 21				X				X	X
RNF 22				X				X	X
RNF 23				X					X

Tabla 71 - Tabla de consistencias Requisito/Prueba

4.6 Análisis de resultados

Tras la realización de las pruebas se comprobará el funcionamiento del sistema. La valoración se realizará con respecto a varios aspectos básicos que se suponen como los más importantes en este tipo de arquitecturas como robustez, desempeño, etc.

La primera prueba del sistema es la base de este plan de pruebas, pues supone la correcta comunicación de todos los módulos que lo componen. La configuración del entorno es vital para la arquitectura y a través de los scripts introducidos es fácilmente comprobable la correcta comunicación entre el portátil y la Raspberry, así como la integración de ROS, la Raspberry y el Robot. Esta parte de comunicación también está relacionada con la capacidad del sistema para publicar el estado a bajo nivel del robot, funcionalidad comprobada con la prueba unitaria TU-00 y que se testea rápidamente con el comando `rostopic list` y `rostopic echo /p3dx_sensors` (leer el *topic*).

Los servicios de teleoperación y movimiento parecen comportarse correctamente, no obstante el sistema de movimiento 2D y la evasión de obstáculos no es muy robusta y eficiente; por un lado existe un retraso entre la ejecución de la tarea y su inicio y por el otro, la evasión de obstáculos es primitiva (como ya se ha comentado) y solo es capaz de esquivar a corto plazo (no existiría un escenario en el que hagamos una petición al servidor de una tarea con un punto 2D demasiado alejado de la posición actual). Sin embargo, ya se ha dejado claro que el objetivo del sistema no era este y, eventualmente, la base robótica logra completar la tarea de movimiento. Recordemos pues, que la ejecución se lanza a través de la Raspberry y que la potencia de la placa demuestra ser poca para lo que requiere el sistema.

Algo similar ocurre con el servicio de mapeo, cuyo desempeño es bastante bajo. Esto es comprensible ya que la placa tiene que lidiar con ROS, la ejecución del servicio en sí y el procesamiento dinámico de imágenes. No obstante, el servicio logra funcionar relativamente bien siendo capaz de generar mapas de zona, aun así, tendremos que tener en cuenta que, para mapear una zona, debemos realizar movimientos lentos, atendiendo al problema del “loop clousure”.

Finalmente realizaremos un análisis de los tiempos obtenidos. En la Ilustración 41 se puede encontrar la gráfica de tiempo resultado de la medición de las Pruebas unitarias, mientras que en la Ilustración 42 podremos ver los resultados obtenidos de la realización de las Pruebas del sistema. Cabe destacar que la medición de los tiempos es en segundos.

A continuación, se procede a realizar la comparativa para evaluar el desempeño del sistema y el mejor escenario posible:

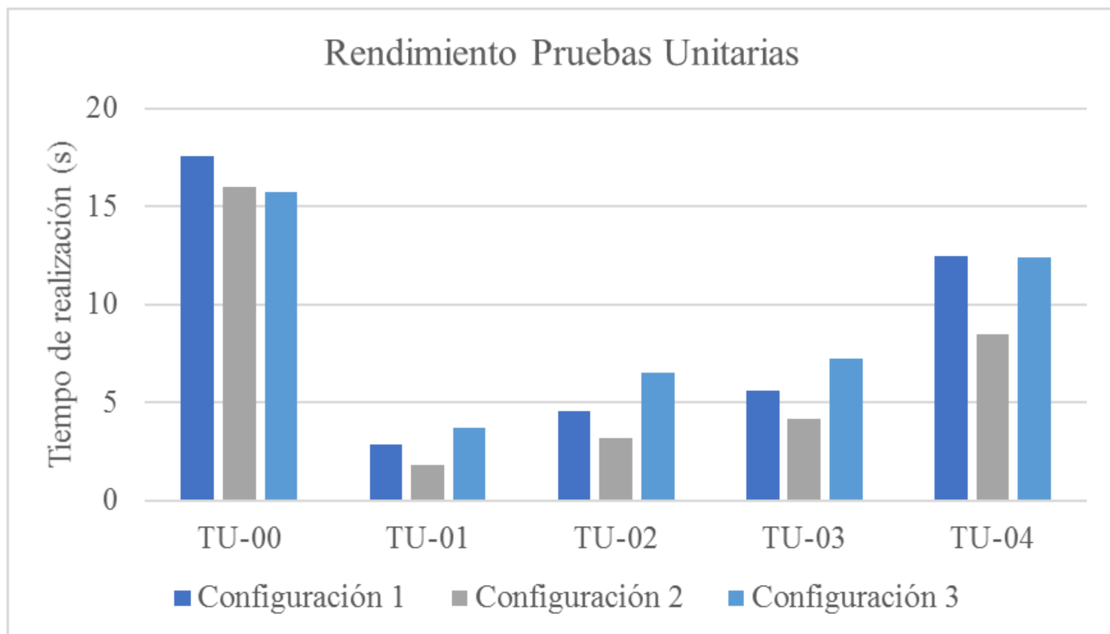


Ilustración 41 - Gráfica desempeño de las Pruebas Unitarias

Como se puede observar en la Ilustración 41, la configuración que parece tener un mejor desempeño es la Configuración 2. Para argumentar estos resultados debemos comentar cómo se da la comunicación entre la Raspberry, que es la que ejecuta, y el portátil. Y es que, resulta que el retraso en la comunicación wifi desde la placa al portátil es bastante alto, y esto afecta enormemente al tiempo de realización de las diferentes tareas. Por otro lado, debemos destacar los resultados de la Prueba TU-00, con un poco más de eficiencia en la Configuración 3. Este dato resulta ser engañoso pues se mide el tiempo en levantar el sistema, y como siempre se mide en la placa, son normales estos resultados dado que en la Configuración 3 hay menos nodos para lanzar en la Raspberry.

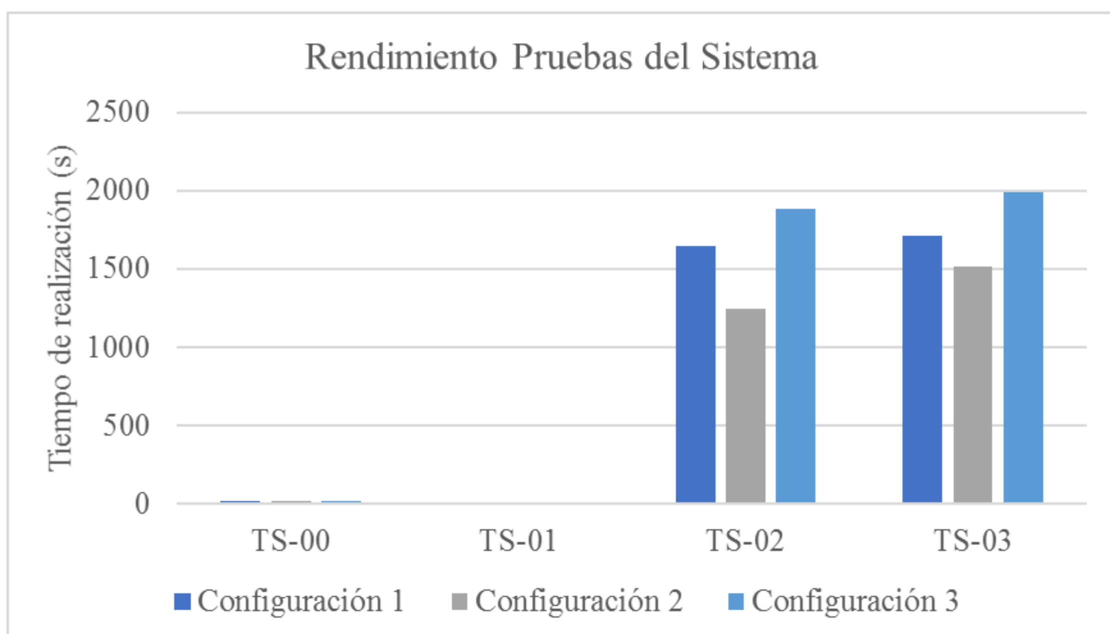


Ilustración 42 - Gráfica desempeño de las Pruebas del Sistema

Por otro lado, observando la Ilustración 42, vemos que las dos primeras pruebas del sistema son poco relevantes en la evaluación del sistema global. Los resultados obtenidos en éstas son significativamente menores que los obtenidos en la evaluación del mapeo. Como ocurre con el análisis anterior, observamos un mejor desempeño en la Configuración 2, probablemente debido al mismo motivo, el retraso en la comunicación de información, que en el caso del procesamiento de imágenes (Configuración 3) resulta ser muy costoso y tardío. Los mapas resultantes de todas las configuraciones son idénticos, la única diferencia en las pruebas realizadas ha sido la necesidad de pasar en un mayor número de veces por un mismo sitio (en la Configuración 2 siempre ha sido menor el número de pasadas) para que las publicaciones de los obstáculos en el mapa de costes se actualizasen correctamente sin pérdidas de información. Estas pasadas son necesarias en algunas ocasiones puntuales en las que los registros de la cámara no han sido procesados y por tanto no se han utilizado para crear el mapa de costes. De esta manera, se realiza una segunda pasada por la misma sección que asegura la generación de las nuevas lecturas y la actualización y/o creación de la nueva parte del mapa.

Ahora bien, unido a esto último comentado es necesario destacar ciertas limitaciones encontradas al realizar las distintas pruebas y demos. En primer lugar, ya se ha destacado el problema general derivado de la potencia de la placa. La Raspberry Pi necesita de un tiempo en la ejecución para hacer una lectura de la posición actual y estimar los diferentes puntos a alcanzar hasta la meta durante la ejecución de los diferentes nodos para el movimiento y la teleoperación. No obstante, una vez iniciada la ejecución, esta parece comportarse de forma eficaz realizando las tareas correctamente a corto plazo. Sin embargo, el sistema acaba encontrando ciertas dificultades en objetivos a largo plazo por el retraso que puede conllevar la ejecución de todo el sistema. Esto último se traduce en ligeros desajustes en la frecuencia con la que se ejecuta ROS por debajo de la arquitectura, siendo necesario, en casos puntuales, dejarle un tiempo al sistema para que procese todos los datos de un ciclo para pasar al siguiente. Esto influye directamente en la necesidad de realizar varias pasadas como se comentaba en el párrafo anterior.

Por otro lado, debemos evaluar el proceso de SLAM. El mapa generado en el entorno de pruebas tiene unas dimensiones finales de 2144×2048 píxeles y se corresponde a una sección del edificio Sabatini de la Universidad Carlos III de Madrid (vestíbulo y dos pasillos: $\sim 100m^2$). El mapa, como se puede observar en la Ilustración 43, posee similitudes con el espacio real que se ha mapeado, sin embargo, debemos comentar que no se trata de un mapa extremadamente preciso. Existen ciertas limitaciones en el proceso de mapeo que disminuyen indudablemente la calidad del mapa. La primera es evidente y se trata de la capacidad de procesar imágenes de manera dinámica por parte de la Raspberry Pi; esto se traduce en posibles pérdidas de datos y la necesidad de ralentizar el sistema para que todos los datos se procesen adecuadamente en cada ciclo en el que se registra la bolsa de datos del sensor. Por otro lado, nos

encontramos con las restricciones impuestas en el sensor Kinect, que no deja de ser una cámara 3D de calidad media con un alcance bastante pobre. Existen ciertas zonas que se mapean eficientemente, pero la Kinect muestra ciertas complicaciones al mapear a distancias largas y, por supuesto, en superficies transparentes como cristaleras (es por esto que hay ciertos huecos en las partes internas de los pasillos). Esto es debido principalmente al sensor de profundidad de la Kinect, incapaz de detectar estas superficies al no producirse una reflexión del infrarrojo. De esta manera, el sensor no puede estimar la distancia relativa de la superficie al sensor y no se tienen datos suficientes para construir el escaneo láser para generar el mapa. Por otro lado, también se observa la problemática de las escaleras, que poseen estructuras de metal con huecos con las que el sensor se acaba confundiendo (Parte inferior derecha).

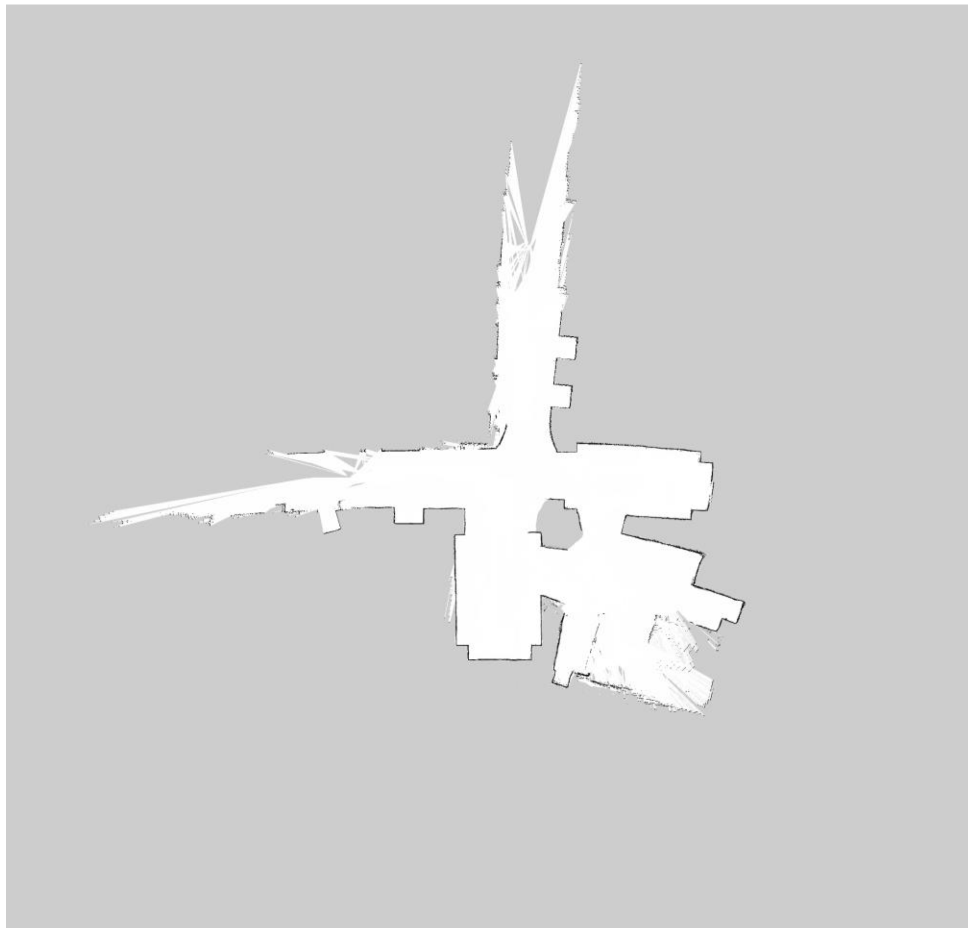


Ilustración 43 - Mapa generado con el Servicio de mapeo

Además, póngase especial énfasis en el error que se arrastra al mapear. Como se puede ver existe un desajuste en el mapa creado; este es el resultado del error del algoritmo de SLAM al calcular la posición relativa del robot durante el proceso de mapeo. Por ese motivo se destaca el problema del *"Loop Clousure"*. Un problema que se antoja complejo de resolver, aún más cuando hay una capacidad de cómputo limitada. Cada vez que se realiza una actualización del mapa, se procede a leer el mismo (internamente representado como nodos), se actualizan los nodos ya mapeados y se

generan los nuevos. Este proceso conlleva gran cantidad de cálculos, cálculos no exentos de errores que si se arrastran pueden hacer que el mapa generado sea inservible al no representar la realidad de forma adecuada. Por otro lado, se tiene que tener en cuenta que el algoritmo siempre utilizará el *frame* del mapa (*/map*) para calcular posiciones relativas, si falla éste, por cualquier motivo, acabará utilizando por defecto los datos de odometría proporcionados por el controlador (RosAria), algo que puede ocasionar que haya un poco más de error. Por suerte, este problema, aunque a día de hoy no hay una solución perfecta, se solventa con la introducción de pequeños ajustes que permiten actuar antes de que el error se haga muy grande y que son completamente parametrizables en el algoritmo de SLAM.

Ilustración 44 - Vestíbulo Ala B Sabatini 1º planta

Ilustración 45 - Vestíbulo Ala C Sabatini 1º planta

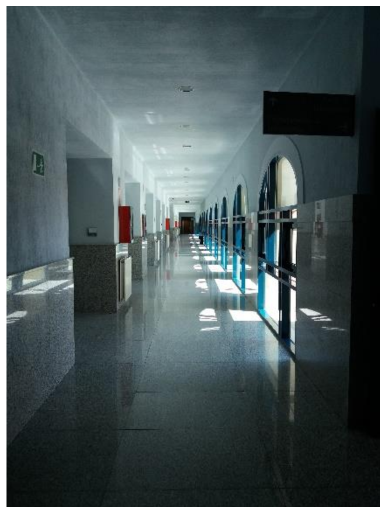


Ilustración 46 - Pasillo Ala C Edificio Sabatini

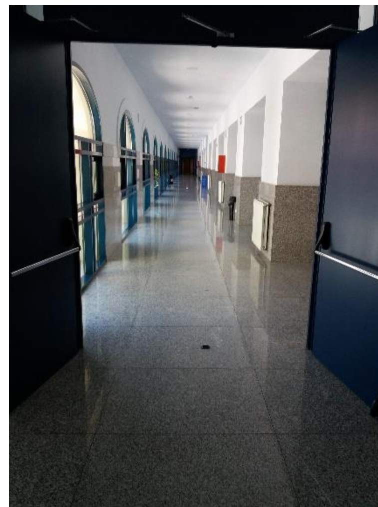


Ilustración 47 - Pasillo Ala B Edificio Sabatini

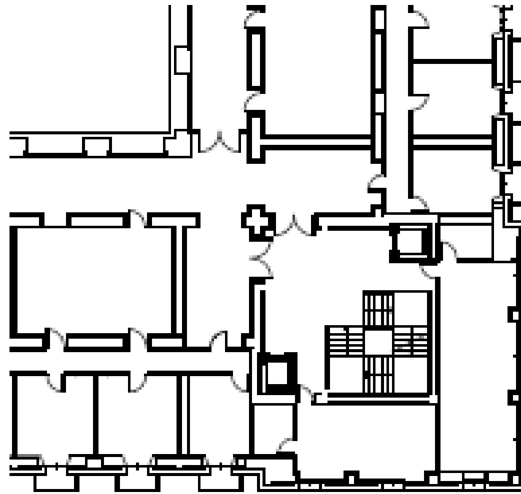


Ilustración 48 - Plano oficial de la sección del edifico Sabatini

Como conclusión parcial a este plan de pruebas podemos comentar varias cosas:

- Los resultados son satisfactorios en cuanto a robustez pues cada módulo ha cumplido con los requisitos establecidos y ha sido capaz realizar las tareas propuestas.
- El análisis de la eficiencia de las pruebas ha demostrado que es de vital importancia el desempeño de los sistemas de comunicación. Es por esto que la Configuración 2 propuesta ha sido la más eficiente según las pruebas realizadas. Éstas han desmentido una primera hipótesis que argumentaba una mayor eficiencia en la división de procesos entre máquinas.
- Como cabe esperar el sistema se ha visto mermado en eficacia debido a las limitaciones de la arquitectura de la Raspberry. La potencia necesaria ha resultado ser más de la que se estimaba. No obstante, eventualmente el sistema es totalmente funcional, siendo un éxito agridulce por el hecho de tener que forzarlo a una ejecución más lenta y pausada de lo que se esperaba.

Un video resumen que comprende distintas pruebas del sistema realizadas (Demos) puede encontrarse en el siguiente enlace:

Capítulo 5: Gestión del proyecto

En la siguiente sección se introducirá el proceso de gestión del proyecto y se realizará una valoración de los tiempos y plazos estimados con respecto a los que finalmente se han tenido. Por otra parte, se dispondrá un resumen y análisis del presupuesto y el coste asociado a la realización del proyecto en su totalidad.

5.1 Planificación temporal

En este apartado se expondrá la planificación realizada inicialmente y se comparará con el desarrollo del proyecto de forma realista, remarcando posibles retrasos no esperados y/o cualquier incidencia que pudiera modificar el plan inicial propuesto.

5.1.1 Planificación inicial

La fecha de comienzo del proyecto se establece el día 1 de septiembre de 2017, más o menos un mes después de la formalización del Trabajo de Fin de Grado. El final estimado se ha establecido el día 1 de septiembre de 2018. Se trata de un periodo de unos 12 meses debido a la complejidad relativamente alta de algunas tareas y la compaginación con un trabajo a media jornada, clases y exámenes.

Generalmente, las horas de dedicación media serán un total de 3 al día, de lunes a viernes, excluyendo fines de semana y festivos. No obstante, aunque en el diagrama que se expone a continuación se observen todos los días de este periodo de 12 meses, se deberá tener en cuenta que únicamente los días laborales son los efectivos en el desarrollo del proyecto.

A continuación, se puede observar la planificación inicial del proyecto en base a un Diagrama Gantt:

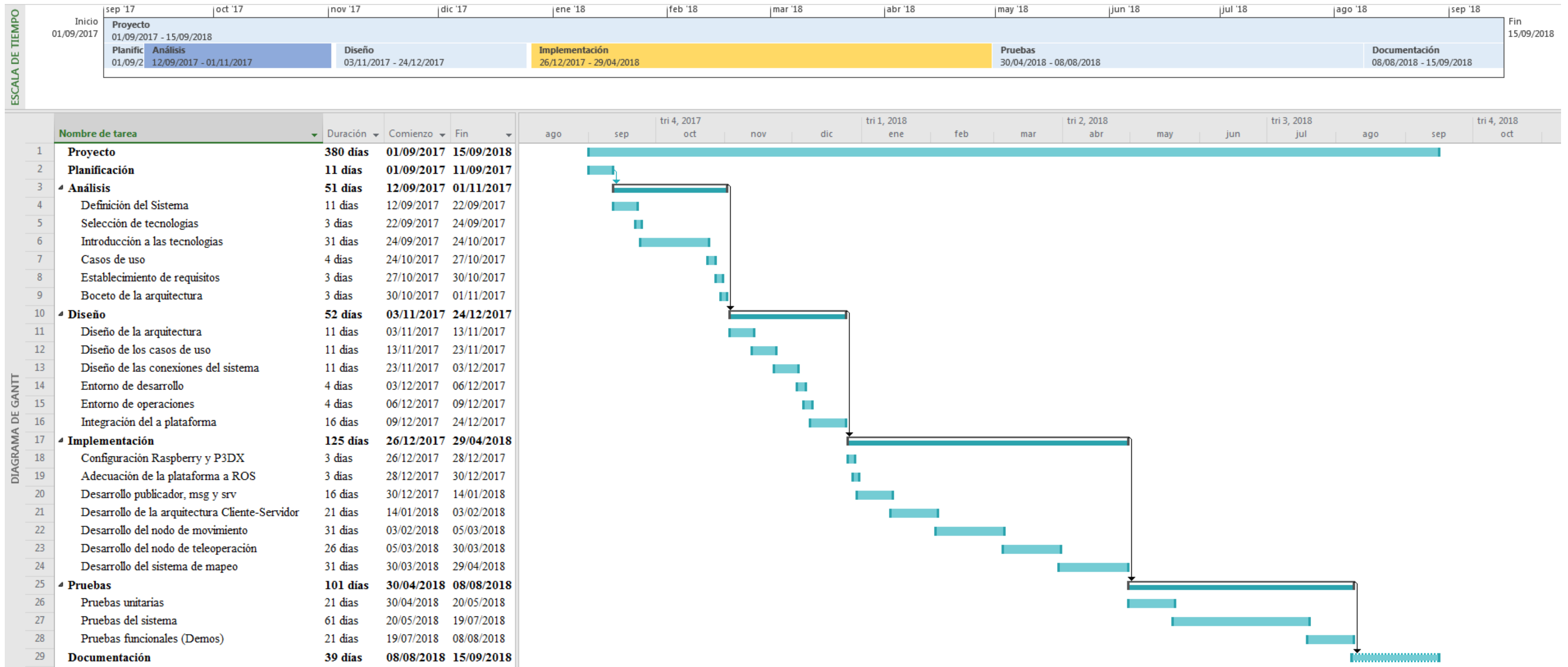
	Nombre de tarea	Duración	Comienzo	Fin
1	**Proyecto**	**366 días**	**01/09/2017**	**01/09/2018**
2	**Planificación**	**11 días**	**01/09/2017**	**11/09/2017**
3	**Análisis**	**41 días**	**12/09/2017**	**22/10/2017**
4	Definición del Sistema	11 días	12/09/2017	22/09/2017
5	Selección de tecnologías	3 días	22/09/2017	24/09/2017
6	Introducción a las tecnologías	21 días	24/09/2017	14/10/2017
7	Casos de uso	4 días	14/10/2017	17/10/2017
8	Establecimiento de requisitos	4 días	17/10/2017	20/10/2017
9	Boceto de la arquitectura	3 días	20/10/2017	22/10/2017
10	**Diseño**	**56 días**	**25/10/2017**	**19/12/2017**
11	Diseño de la arquitectura	11 días	25/10/2017	04/11/2017
12	Diseño de los casos de uso	11 días	04/11/2017	14/11/2017
13	Diseño de las conexiones del sistema	11 días	14/11/2017	24/11/2017
14	Entorno de desarrollo	6 días	24/11/2017	29/11/2017
15	Entorno de operaciones	6 días	29/11/2017	04/12/2017
16	Integración del a plataforma	16 días	04/12/2017	19/12/2017
17	**Implementación**	**111 días**	**20/12/2017**	**09/04/2018**
18	Configuración Raspberry y P3DX	11 días	20/12/2017	30/12/2017
19	Adecuación de la plataforma a ROS	11 días	30/12/2017	09/01/2018
20	Desarrollo publicador, msg y srv	16 días	09/01/2018	24/01/2018
21	Desarrollo de la arquitectura Cliente-Servidor	11 días	24/01/2018	03/02/2018
22	Desarrollo del nodo de movimiento	21 días	03/02/2018	23/02/2018
23	Desarrollo del nodo de teleoperación	16 días	23/02/2018	10/03/2018
24	Desarrollo del sistema de mapeo	31 días	10/03/2018	09/04/2018
25	**Pruebas**	**101 días**	**09/04/2018**	**18/07/2018**
26	Pruebas unitarias	21 días	09/04/2018	29/04/2018
27	Pruebas del sistema	61 días	29/04/2018	28/06/2018
28	Pruebas funcionales (Demos)	21 días	28/06/2018	18/07/2018
29	**Documentación**	**46 días**	**18/07/2018**	**01/09/2018**

Ilustración 49 - Diagrama de Planificación estimada

5.1.2 Desarrollo real del proyecto

No obstante, la planificación mencionada inicialmente no deja de ser en base a una estimación de las tareas propuestas excluyendo asuntos personales, eventualidades, exámenes, asuntos del trabajo etc. Es por esto por lo que, en realidad, el proyecto ha durado un poco más de lo planeado.

Así el Diagrama de planificación queda de la siguiente forma:



	Nombre de tarea	Duración	Comienzo	Fin
1	**Proyecto**	**380 días**	**01/09/2017**	**15/09/2018**
2	**Planificación**	**11 días**	**01/09/2017**	**11/09/2017**
3	**Análisis**	**51 días**	**12/09/2017**	**01/11/2017**
4	Definición del Sistema	11 días	12/09/2017	22/09/2017
5	Selección de tecnologías	3 días	22/09/2017	24/09/2017
6	Introducción a las tecnologías	31 días	24/09/2017	24/10/2017
7	Casos de uso	4 días	24/10/2017	27/10/2017
8	Establecimiento de requisitos	3 días	27/10/2017	30/10/2017
9	Boceto de la arquitectura	3 días	30/10/2017	01/11/2017
10	**Diseño**	**52 días**	**03/11/2017**	**24/12/2017**
11	Diseño de la arquitectura	11 días	03/11/2017	13/11/2017
12	Diseño de los casos de uso	11 días	13/11/2017	23/11/2017
13	Diseño de las conexiones del sistema	11 días	23/11/2017	03/12/2017
14	Entorno de desarrollo	4 días	03/12/2017	06/12/2017
15	Entorno de operaciones	4 días	06/12/2017	09/12/2017
16	Integración del a plataforma	16 días	09/12/2017	24/12/2017
17	**Implementación**	**125 días**	**26/12/2017**	**29/04/2018**
18	Configuración Raspberry y P3DX	3 días	26/12/2017	28/12/2017
19	Adecuación de la plataforma a ROS	3 días	28/12/2017	30/12/2017
20	Desarrollo publicador, msg y srv	16 días	30/12/2017	14/01/2018
21	Desarrollo de la arquitectura Cliente-Servidor	21 días	14/01/2018	03/02/2018
22	Desarrollo del nodo de movimiento	31 días	03/02/2018	05/03/2018
23	Desarrollo del nodo de teleoperación	26 días	05/03/2018	30/03/2018
24	Desarrollo del sistema de mapeo	31 días	30/03/2018	29/04/2018
25	**Pruebas**	**101 días**	**30/04/2018**	**08/08/2018**
26	Pruebas unitarias	21 días	30/04/2018	20/05/2018
27	Pruebas del sistema	61 días	20/05/2018	19/07/2018
28	Pruebas funcionales (Demos)	21 días	19/07/2018	08/08/2018
29	**Documentación**	**39 días**	**08/08/2018**	**15/09/2018**

Ilustración 50 - Diagrama de Planificación Real

Concretamente, la introducción a las tecnologías utilizadas, sobre todo a ROS, ha resultado ser un poco más compleja de lo esperado; debido a esto, la duración de esta actividad es mucho mayor que la estimada inicialmente. Por otro lado, para el Diseño del sistema no se ha necesitado tanto tiempo, al igual que en la implementación, para la que se ha reducido su duración total, afectando en mayor medida a la adecuación del sistema y la configuración de la placa, y en menor medida a la implementación de nodos como el de teleoperación. Sin embargo, la implementación del movimiento en 2D y el servicio de mapeo se mantienen y la estimación ha sido correcta, destacando que han sido las tareas más costosas de esta sección.

Por último, se debe comentar que la documentación se ha ido esbozando durante todo el proyecto, pero su generación se inició una vez terminadas las pruebas, durando un poco menos de lo estimado, y finalizando su formalización el 15 de septiembre de 2018.

5.2 Presupuesto

Esta sección supondrá un análisis del impacto económico del proyecto. De forma estimada se presentará el coste total del proyecto, resultado de desglosar posteriormente los diferentes costes necesarios para la culminación del sistema.

5.2.1 Presupuesto total

El proyecto con título: **Control de un Robot con Raspberry Pi**. Con una duración de 12 meses y 14 días (incluyendo días no laborables).

Tiene un presupuesto total para su realización de **34.860,10 € (TREINTA Y CUATRO MIL OCHOCIENTOS SESENTA CON DIEZ EUROS)** (I.V.A incluido).

En las siguientes secciones se procederá al desglose en el que se justificarán estos costes.

5.2.2 Desglose presupuestario

Este apartado servirá para remarcar cada uno de los gastos que se han tenido en cuenta para el cómputo del coste total del proyecto.

5.2.2.1 Costes directos

- Costes de personal

  Dentro de cualquier proyecto de desarrollo es necesaria la coexistencia de diversos roles, indispensables para la correcta implementación del sistema.

Así, durante el proyecto, el alumno ha tenido que asumir una serie de roles que se definen a continuación:

- **Jefe de proyecto:** Se trata del supervisor del trabajo que están realizando los demás integrantes del grupo. Además, se asegurará que el proyecto cumple con los objetivos que se han marcado inicialmente.
- **Analista:** Perfil encargado de examinar las necesidades de aquellos usuarios del sistema y valorar su entorno de actuación a través de la extracción de casos de uso, requisitos…
- **Diseñador:** Integrante cuyo trabajo se basa en esbozar el sistema en función de los resultados recabados por el analista.
- **Programador:** Perfil en cuya responsabilidad recae la implementación del sistema propuesto por el diseñador.
- **Gestor de pruebas:** Elemento encargado de comprobar que todo aquello implementado funciona correctamente, siguiendo los requisitos impuestos y las salidas esperadas.

Cada uno de estos perfiles realizan un trabajo distinto, que incluye diversas tareas, es por eso por lo que cada uno tendrá un salario diferente acorde con sus responsabilidades. Extrayendo los costes de la *Guía de mercado laboral del año 2018* con una experiencia en un rango 0-2 años:

Perfil	Horas dedicadas	Coste (€/h)	Coste (€)	Coste con S.S.*
Jefe de proyecto	140	35,00	4.900,00	6.639,50
Analista	140	26,00	3.640,00	4.932,20
Diseñador	140	25,00	3.500,00	4.742,50
Programador	300	23,00	6.900,00	9.349,50
Gestor de pruebas	300	18,00	5.400,00	7.317,00
TOTAL	1020	-	-	**32.980,70 €**

Tabla 72 - Costes de personal

**Al salario base de cada puesto se le suma el coste de seguridad social que supone un 35,5% del salario bruto.*

- Costes de material

La realización del proyecto ha conllevado también un conjunto de gastos de material informático, tanto en Hardware como en Software. Además, se añadirá también los costes de material fungible utilizado. Nótese que se supone una depreciación de 4 años (48 meses).

Unidades	Elementos	Costes (€)	Meses de uso	Coste aplicable*
1	Robot P3DX	2885	12	721,25
1	Raspberry Pi 3 Model B Kit	57	12	14,25
1	Cable USB-MicroUSB	8	12	2,00
1	Cable SERIE y adaptador USB	6	12	1,50
1	Sensor Kinect	50	3	3,13
1	Adaptador para Kinect	12	3	0,75
1	Portátil Toshiba SATELLITE Pro	420	12	105,00
1	Herramientas de circuitería	21	3	1,31
1	Juego de llaves Allen	16	2	0,67
1	Licencia Windows 10	250	6	31,25
1	Microsoft Office 365	150	6	18,75
TOTAL	-	-	-	**899,85 €**

Tabla 73 - Costes material informático

**Amortización = Precio × (Dedicación/Depreciación)*

Unidades	Artículo	Costes (€/ud.)	Coste Total
3	Bolígrafo Pilot	1,2	3,6
1	Paquete Folios Din-A4	1,43	1,43
1	Paquete Post-It 90 Sheets	8,53	8,53
TOTAL	-	-	**13,56 €**

Tabla 74 - Costes de material fungible

5.2.2.2 Costes indirectos

Los costes indirectos que ha conllevado el proyecto se calculan a partir de una estimación porcentual sobre los costes directos ya analizados. Concretamente un 10%, incluyendo los gastos de luz, agua, conexión a internet, teléfono…

Costes indirectos	
10% sobre gastos directos	1.694,71 €

Tabla 75 - Costes indirectos

Ahora bien, el beneficio estimado, basándonos en los datos obtenidos anteriormente, se concreta en un 20% sobre los costes directos del proyecto.

Beneficios	
20% sobre gastos directos	6.778,82 €

Tabla 76 - Beneficios

5.2.3 Resumen de costes

Así, finalmente, se muestra una tabla resumen en la que se proporciona el coste total del proyecto con inclusión del I.V.A.

Resumen de costes	
Costes directos	
Salarios	32.980,70 €
Material tecnológico	899,85 €
Material fungible	13,56 €
Costes indirectos	1.694,71 €
Beneficios	6.778,82 €
Total sin I.V.A	28.810,00 €
I.V.A (21%)	6.050,10 €
TOTAL con I.V.A	**34.860,10 €**

Tabla 77 - Resumen de costes

Capítulo 6: Marco regulador y ético

Esta sección se dedicará al análisis de la legislación aplicable en todo lo referente al desarrollo de este proyecto. Así, se comentará acerca de licencias, estándares y prácticas aplicadas sobre las diferentes tecnologías utilizadas durante el desarrollo.

6.1 Responsabilidades profesionales y éticas

Nótese que el sistema creado impone una serie de responsabilidades que se discuten a continuación.

6.1.1 Movimiento, Navegación y Mapas

El sistema puede crear mapas de zonas, pero se debe tener en cuenta que pueden existir zonas prohibidas o que, simplemente, no se deberían mapear al ser lugares cuyos propietarios o responsables no lo permiten. De esta manera se debería contar con los permisos necesarios si se da el caso de una zona conflictiva.

Por otro lado, el sistema permite el control manual de la plataforma. Esto impacta directamente en la responsabilidad de aquellos que la manejan con el objetivo de no infligir daño alguno a personas o atentar contra la integridad de cualquier infraestructura, objetos, maquinaria etc.

Finalmente, hay que tener en cuenta que también hay cierta parte de navegación autónoma. Esto, aunque es en gran parte gestionado por el alto nivel, se tiene que tener en cuenta al valorar los procesos de planificación y las acciones realizadas finalmente por el robot.

6.2 Privacidad y seguridad

Actualmente la arquitectura propuesta no hace uso de datos sensibles, lo que sí que es cierto es que, con respecto a la seguridad, debemos tener en cuenta los puntos antes vistos: asegurarnos de que la plataforma está haciendo las tareas correctamente y que no hay ningún peligro de terceros durante la ejecución.

Por otro lado, destacamos que, a día de hoy, la conexión entre la monitorización y la ejecución no tiene procesos de seguridad asociados ya que es una red sin protección. En un futuro, la comunicación con una arquitectura de alto nivel supondría un esfuerzo extra para establecer una conexión segura, para así asegurarnos de que nadie pueda asignar planes y tareas sin verificar primero su procedencia.

6.3 Estándares técnicos: lenguajes y herramientas

En el proyecto se han utilizado una serie de herramientas que pueden estar en mayor o menor medida afectadas por políticas de privacidad, de propiedad intelectual, restricciones de uso, etc. A continuación, se discuten las más importantes.

6.3.1 C++ y XML

El lenguaje de programación utilizado no está sujeto a ningún derecho de autor pues es un lenguaje libre que se puede utilizar sin necesidad de licencias. Por otro lado, el compilador utilizado g++ es un conjunto libre de compiladores que están acogidos por la GNU General Public License haciendo que este paquete sea gratuito y libre para cualquier desarrollador [78]. Además, se ha seguido el estándar fijado por c11.

Al igual ocurre con XML, el propio lenguaje y el analizador son de formato abierto, no sujeto a restricciones legales y/o económicas de uso.

6.3.2 ROS y su estándar

ROS tiene sus propios estándares y el *framework* completo está recogido bajo la licencia Creative Commons Attribution 3.0 [79] que permite compartir y adaptar el material procedente de éste siempre y cuando se respeten los términos de uso.

Por otro lado, ROS sigue una serie de estándares utilizados en el proyecto. El primero es el de las unidades, donde metros es para distancia, radianes para giros y segundos para tiempo [80]; esto contrasta directamente con las unidades utilizadas por ARIA, metros, grados sexagesimales y segundos, respectivamente [81]. Además de esto, tenemos una estructura fija de los proyectos, con una jerarquía de carpetas específica y unos procedimientos más o menos estandarizados para la compilación (Catkin), el lanzamiento de nodos (*rosrun*) y la ejecución de *launchers* (*roslanch*) [82].

6.3.3 Ubuntu (Linux)

Linux es un sistema operativo libre de tipo Unix recogido bajo la licencia GNU. Al ser un sistema operativo libre, es gratuito y de fácil acceso, al igual que la mayoría de software procedente de sus repositorios oficiales. El código fuente está recogido bajo la licencia GPL, pero como no se ha llegado a modificar nada de este, no hace falta entrar en detalle.

No obstante, Ubuntu Mate está recogido bajo la licencia Creative Commons Attribution-ShareAlike 4.0 International License [83], que sigue los mismos preceptos que la propuesta por ROS.

6.3.4 VMware

Para la virtualización de máquinas se ha utilizado VMware. Este software propiedad de DELL EMC es de pago, sin embargo, existen versiones gratuitas que se pueden conseguir sin necesidad de licencia para uso personal y doméstico. El componente software en cuestión es VMware Workstation Player [84].

6.3.5 Librerías utilizadas

A continuación, se van a tratas los aspectos relacionados con todas aquellas librerías utilizadas. Éstas se reducen a servicios externos, controladores como el de la plataforma robótica o los de la Kinect, etc.

6.3.5.1 Aria, MobileSim y Mapper

Todo este software producido por el fabricante está recogido bajo la licencia de GNU [85] [86] [87] Public License por lo que no habría problemas legales y/o económicos en cuanto a su utilización en el proyecto.

6.3.5.2 Libfreenect

Libfreenect es la librería que permite el desarrollo con el sensor Kinect. Resulta ser una librería de código abierto recogida bajo la licencia GPL v2 [88]. El código es de libre acceso y se puede encontrar en su repositorio oficial de GitHub [39], no obstante, para el proyecto solo se ha requerido su uso, sin modificaciones.

6.4 Propiedad intelectual

ROS tiene la peculiaridad de aunar mucho código de millones de usuarios diferentes repartidos por el mundo. Durante el proyecto se han reutilizado pequeñas implementaciones, ideas y diferentes paradigmas extraídos de la página oficial de ROS [89], de algunos investigadores que trabajan a día de hoy con este tipo de sistemas o de los propios nodos que podemos descargar de los diferentes módulos que ofrece ROS a través de los repositorios de Linux. Es por esto que se ha utilizado cierto contenido que no es de la total autoría del alumno, sin embargo, todo este contenido resulta ser de libre distribución bajo la licencia GPL como ya se ha comentado. Todo código de autoría externa vendrá reflejado con su respectiva sección de comentarios en el código a cerca de su procedencia.

De esta manera, en el Anexo D, se podrá encontrar una tabla informativa sobre la procedencia de todos los recursos de ROS utilizados.

Capítulo 7: Impacto Socioeconómico

Esta sección se va a dedicar a la realización de un análisis y estudio del impacto social y económico del proyecto. Esto nos servirá para hacer una medición de la repercusión de éste en diferentes contextos, proporcionando información detallada sobre la huella en producción, empleo, medioambiente…

Así, formalizaremos este análisis haciendo un estudio de los diferentes tipos de impacto:

- Impacto directo: Basado en la producción y empleo generado en los sectores que reciben directamente los bienes del proyecto.
- Impacto indirecto: Basado en la producción y empleo generado en aquellos sectores que suministran servicios necesarios para el desarrollo del proyecto.
- Impacto inducido: Basado en la producción y empleo generado a través del consumo de los servicios proporcionados por el proyecto.
- Impacto medioambiental: Basado en la huella ecológica que deja el proceso en los múltiples aspectos en el desarrollo del proyecto.

7.1 Impacto directo

El impacto directo del proyecto está muy relacionado con la gestión del proyecto en sí y el análisis del presupuesto proporcionado en secciones anteriores. Concretamente, si realizamos un análisis de este presupuesto, nos encontraremos que es necesario destacar la generación de empleo y fuerza de trabajo: Desarrollador, Analista …

Eminentemente esto supone algo muy positivo, se trata de un impacto importante en el sector TIC y la gestión y administración de empresas o proyectos, basado principalmente en la generación de puestos de trabajo; algo que evidentemente es necesario dado el estado actual del paro [90] y la economía española.

7.2 Impacto indirecto

Por otro lado, el impacto indirecto radica fundamentalmente en la producción de los bienes y servicio necesarios para el proyecto. Fundamentalmente, debemos destacar la aparición del sector de la automatización y robótica, debido a la necesidad de generar y crear plataformas robóticas. Esto supone en primera instancia una estimulación del sector industrial [91] que, desgraciadamente, tiene poca presencia en España pero que es de vital importancia para el crecimiento tecnológico.

También debemos destacar en este análisis la aparición del sector TIC, siendo importante la adecuación del hardware con el software y la aparición de proveedores de

servicios (microcontroladores, sistemas de gestión…) necesarios para la culminación de cualquier proyecto robótico, ya sean accionadores, sensores etc.

7.3 Impacto inducido

El proyecto desarrollado deja una marca mucho mayor si analizamos el impacto inducido. Es evidente que la motivación principal que cualquier empresa pueda tener para financiar este tipo de proyecto es la obtención de beneficios a medio-largo plazo.

La particularidad de estos sistemas es que tienen muchas aplicaciones que se pueden adecuar a infinidad de contextos. El indiscutible cambio de mentalidad de esta sociedad, tecnológica y consumista, pone de manifiesto el crecimiento exuberante de la demanda de servicios cada vez más robustos, especializados y competentes; elementos tecnológicos que sean capaces de pasar desapercibidos a la vez que mejoran nuestras vidas en diferentes aspectos, ya sea facilitando tareas o aportando nuevas metas.

Es importante, por todo esto, destacar la huella en un usuario definido por la tecnología, cuya mentalidad cambia a medida que los diferentes sistemas robóticos se introducen poco a poco en su vida. Usuarios que se ven beneficiados, al igual que las empresas que ofrecen el servicio final, por el desarrollo tecnológico que suponen este tipo de proyectos robóticos. Y es que no solo tenemos el ejemplo del polémico asunto de los coches autónomos, que aúnan sistemas de SLAM y navegación autónoma, sino también de la mejora en servicios básicos como el trasporte, la gestión documental (almacenamiento físico y ordenación), en las tareas del hogar (robots aspiradores) … Como ya se ha mencionado, elementos que pasan desapercibido pero que la sociedad considera necesarios hoy en día.

7.4 Impacto medioambiental

Finalmente, estudiamos el impacto medioambiental del proyecto, tanto al nivel de desarrollo como el de utilización del usuario final.

Por un lado, tenemos el impacto del desarrollo de los servicios, pues es conocida la alta huella ecológica dejada en los procesos de producción de hardware, así como el alto nivel de demanda de electricidad, que, si bien puede producirse respetando el medio ambiente, esta puede venir directamente de campos como la energía nuclear.

Por otro lado, tenemos el impacto del usuario final, que básicamente se limita al ingente consumo eléctrico, aunque este es en menor medida por cada usuario, ciertamente acaba siendo el más elevado si evaluamos el uso tecnológico de cada usuario final.

Capítulo 8: Conclusiones, resultados y trabajos futuros

8.1 Resultados

Tras la completa realización del proyecto y la implementación de la arquitectura propuesta podemos hacer alusión a los distintos puntos de la sección de objetivos y realizar una valoración de aquellos que se han logrado cubrir. Esta sección será la culminación del trabajo desarrollado donde se argumenta la consecución de estos objetivos.

- Aunque ya se tenía cierta experiencia con la Raspberry Pi, se ha llegado a conseguir un nivel más alto de experiencia con la plataforma y su arquitectura. No solo en cuanto a sus posibilidades, sino también teniendo en cuenta sus limitaciones. Instalación, manejo y gestión de la placa han sido partes fundamentales del proyecto. De esta manera, se ha conseguido una completa integración de la misma con la plataforma robótica, logrando, en última instancia, verificar la viabilidad de esta para el control robótico.
- ROS supuso un reto mucho más complejo. Esta gran comunidad está repleta de aportaciones, y el propio *framework* cuenta con infinidad de posibilidades. El planteamiento del reto y su abordaje han sido gratificante, pues se ha conseguido un nivel de conocimiento bastante alto sobre la arquitectura de éste y sus distintas aplicaciones, aun no habiendo ahondado en ellas en este proyecto por motivos de extensión. La consecución de una arquitectura que posibilita la ejecución de tareas a bajo nivel ha sido mucho más ágil gracias a su uso, ayudando a simplificar tareas que, de otra manera, se antojan mucho más complicadas. Son tantas las posibilidades y oportunidades que es necesario muchas veces centrarse y enfocar los esfuerzos en algo en concreto, no intentar abarcarlo todo.
- El movimiento de una base robótica puede parecer trivial, pero supone un esfuerzo que no lo es tanto. Tras la integración de todos los sistemas, se necesita una arquitectura robusta, la implementación correcta de controladores y publicadores y, en general, sistemas que controlen y gestionen adecuadamente los datos y mensajes generados para asegurar comportamientos robóticos adecuados. Así, se ha conseguido aportar un conjunto de comandos simples e implementar una interfaz de fácil uso que permite utilizar todas estas herramientas que nos ofrece ROS: mensajes, topics, etc. de manera rápida y sencilla.
- El problema del SLAM es un problema recurrente en el campo de la robótica, con el que se ha aprendido cómo abordar cuestiones muy complejas de forma eficiente, intentando maximizar los recursos disponibles. Si bien es cierto que los mapas creados no son excesivamente

detallistas y complejos, podemos concluir que eso es parte de los sensores: la arquitectura está ahí, solo queda mejorar el hardware; por ejemplo, no son comparables las lecturas de un láser que supondría un mapeado más preciso. De esta manera, se ha conseguido desarrollar una primera versión de un sistema SLAM. Cierto es que este necesita ser pulido, pero debemos tener en cuenta las limitaciones de nuestros sensores. No obstante, en los mapas realizados, se atisba la similitud que se quería buscar entre la realidad y los mapas generados, y eso es un paso muy importante para el desarrollo de un sistema SLAM más complejo si cabe.

- Finalmente, debemos discutir la integración de ROS y la Raspberry, algo que ha sido relativamente complejo debido a las limitaciones de la placa. Si bien es cierto que la Raspberry es bastante potente, tenemos que recordar que los sistemas robóticos y sobre todo los algoritmos de SLAM necesitan bastante capacidad de cómputo. Si a eso le sumamos la capacidad de procesamiento para correr ROS y sus módulos tenemos una combinación que requiere unas especificaciones altas. Por eso es de vital importancia una implementación basada en la eficiencia, algo que se ha conseguido con lenguajes como C++, la ejecución cruzada o la implementación de hilos que maneja ROS internamente.

8.2 Conclusiones

Es evidente que el problema del SLAM y la navegación en plataformas robóticas sigue teniendo ciertos aspectos en los que encontramos dificultades. Abordar este tipo de tareas es, en muchas ocasiones, algo complicado por la cantidad de variables y situaciones que se deben tener en cuenta. La calibración de sensores y los márgenes de error son realmente relevantes, y esta prueba de concepto muestra que con pocos recursos se pueden hacer cosas realmente complejas. No obstante, queda demostrada la relevancia de la calidad de los elementos integrantes de este tipo de arquitecturas, pues la minimización de errores y el aumento del desempeño de un sistema dinámico, resultan de vital importancia cuando hablamos de entornos poco controlados y supervisados. Entornos en los que el ser humano también forma parte, y que se puede ver comprometido de distintas maneras por sistemas que no logren cumplir con una serie de requisitos básicos que establezcan un umbral de seguridad y actuación.

El sistema propuesto pretende ser una vía para abordar este problema, en el que no solo encontramos poca presencia de elementos de supervisión, sino que nos ayudamos de formalizaciones en alto nivel que serán las encargadas de la asignación de tareas de forma autónoma y dinámica. En cierta manera, el campo de estudio de la robótica ha avanzado enormemente en estos años, consiguiendo hitos realmente relevantes e importantes. No obstante, siempre cabe la posibilidad de pensar que los modelos actuales no son los mejores, que las implementaciones pueden ser más eficaces y que los sistemas robustos pueden ser más seguros y mejores al abordar fallos contingentes.

Esto, en verdad, es la motivación que mueve al ser humano en el camino del desarrollo, es lo que nos hace avanzar y mejorar en muchos y diferentes aspectos.

Son pequeños pasos que nos van acercando a sistemas mejores, que van proporcionando distintos puntos de vista y que suponen pilares sobre los que se puede edificar el desarrollo a posteriori. El sistema implementado permite la integración con módulos de alto nivel, permite el comportamiento autónomo y el mapeado de una zona, aspectos que no hace muchos años se creían pertenecientes al campo de la ciencia ficción. ¿Por qué no podríamos permitirnos soñar con un robot capaz de transportar personas invidentes o un exoesqueleto que permita andar a personas discapacitadas autónomamente? Esto se conseguiría con este tipo de sistemas, donde la acción y la reacción son compañeras y necesitan de un alto grado de robustez en el tratamiento de posibles fallos.

8.3 Conclusiones generales

Si hacemos un ejercicio honesto de valoración del proyecto global podremos concluir distintos aspectos que parecen ser importantes.

En primer lugar, debemos hablar sobre ROS y las funcionalidades que nos aporta. La comunicación de una plataforma robótica con una Rapsberry Pi se torna relativamente sencilla con el desarrollo de arquitecturas básicas como la de Cliente-Servidor, muy utilizada hoy en día. Estas arquitecturas nos brindan la oportunidad de crear interfaces de fácil uso que posibilitan la capacidad para gestionar distintas tareas a bajo nivel. Y, aunque la placa parece ofrecer ciertas complicaciones al estar limitada en capacidad de cómputo, con un poco de paciencia logra resolver tareas bastante complejas.

Por otro lado, la implementación de sistemas de planificación, SLAM… son relativamente complejas tanto en desarrollo como en disponibilidad de recursos. Este hecho, se ha ido haciendo más evidente con la aparición de comunidades enteras y plataformas colaborativas que, aun estando integradas por miles de personas diseminadas por el mundo, permiten aunar grandes bases de conocimiento y edifican el desarrollo sobre los distintos campos de estudio que motivan a cualquier científico o desarrollador. En este caso es ROS y la robótica, en otro puede ser la física con la Teoría del todo o la computación con la igualdad entre P y NP.

Además, con los desafortunados eventos que hemos visto recientemente con vehículos autónomos, estudiando desde distintas perspectivas estos sistemas, podemos concluir que la implementación de la arquitectura propuesta supone una valoración de cómo abordar el problema, de cómo hacer eficiente algo que resulta tener una elevada complejidad. Pone de manifiesto la necesidad de crear sistemas robustos, capaces de actuar en diferentes situaciones bajo un criterio objetivo, encaminado a aportar no a destruir. Al igual que los ojos de cualquier animal necesitan estar sanos para la

percepción espacial del entorno, se ha descubierto que los sensores de una plataforma robótica necesitan estar igual o más "sanos" (calibrados); necesitan tener un catálogo de limitaciones muy reducido, asegurando una minimización de errores lo más eficiente posible.

Tras la evaluación del sistema, por otro lado, resulta evidente pensar que el mejor escenario donde se ejecuta el sistema y el propio ROS (quizás debido a su carácter modular) es la Configuración 2 mencionada anteriormente, donde todo el sistema se gestiona desde la placa. Cabe destacar, sin embargo, la importancia de la compartición de datos y el desempeño de la comunicación de éstos en un sistema robótico dinámico. Los mapas resultantes, que comparten similitudes con la realidad, han sido semejantes en todas las configuraciones propuestas. Estos mapas, de dimensión variable, han sido un primer acercamiento a un sistema de SLAM que ha finalizado con la culminación del mapa más grande creado (Edificio Sabatini – Universidad Carlos III), poniendo especial relevancia en la calidad de los sensores utilizados y el procesamiento de sus datos. De esta manera, es muy importante evaluar el desempeño del propio sistema global que es lo que al fin y al cabo sirve de evidencia para establecer un sistema robusto y eficiente. Así, en su conjunto, dicho sistema ha conseguido cumplir con los objetivos propuestos en un inicio, abordando todos los problemas que puedan derivar de arquitecturas como la propuesta.

En definitiva, un pequeño paso más para el desarrollo robótico, con el que algunos problemas se solucionan, pero surgen otros aún más interesantes si cabe. En cierta manera podríamos pensar que tras esto ya habríamos acabado, pero la mejora puede ser infinita y la motivación, una de las causas más importantes del conocimiento.

8.4 Trabajos futuros

Resulta indiscutible pensar que en este campo es necesario avanzar, seguir creando y disfrutando del desarrollo de este tipo de sistemas. Tras la finalización de este proyecto se han conseguido dos hitos de vital importancia para establecer diferentes rutas a seguir a partir de este punto.

En primer lugar, se ha llegado a implementar una arquitectura capaz de conectarse a un sistema de alto nivel y gestionar las acciones y los sistemas a bajo nivel de una plataforma robótica. En este punto, debemos comentar que arquitecturas como PELEA pueden ser elementos con los que podríamos comenzar un proyecto de integración que nos permitan crear un navegador completo. La generación de dominios en alto nivel y el uso del paradigma de la planificación automática con conceptos de *Machine Learning* (Aprendizaje por Refuerzo) son trabajos que en un futuro podríamos implementar dada la arquitectura actual propuesta.

En segundo lugar, se ha abordado el problema del SLAM, y de forma eficaz se ha conseguido la creación de mapas a partir de las lecturas de sensores. Algo que puede

usarse con una arquitectura de alto nivel como anteriormente se ha mencionado. La comparación con mapas de zona es la base sobre la que trabaja la navegación autónoma, así, podríamos establecer un nuevo punto en nuestra hoja de ruta para seguir desarrollando este sistema y preparar la pila de navegación de ROS de forma completa, adjuntando algoritmos de navegación (AMCL) que permitan la autonomía de la base robótica con la evasión de obstáculos a largo plazo.

Finalmente, si quisiéramos mejorar la arquitectura propuesta de manera independiente, deberíamos proponer un cambio en el hardware utilizado. Quizás crear un clúster de Raspberry Pi para tener más potencia, incluir sensores de mayor precisión como LIDAR o incluso hacer uso de más cámaras RGBd que amplíen el rango de visión para el mapeo. No es raro en este sentido encontrar sistemas con tres o cuatro Kinect, algo que aportaría una visión completa de la zona, a cambio de la necesidad de una capacidad de procesamiento mucho más elevada. Valdría la pena barajar si el clúster propuesto sería suficiente para mantener estos nuevos requisitos.

Referencias

[1] «RoboCup,» 2016. [En línea]. Available: http://www.robocup.org/. [Último acceso: 20 septiembre 2017].

[2] «RoboCup Soccer,» 2016. [En línea]. Available: http://www.robocup.org/domains/1. [Último acceso: 20 septiembre 2017].

[3] «ROS,» 2018. [En línea]. Available: http://www.ros.org/. [Último acceso: 20 septiembre 2017].

[4] O. MobileRobots, «MobileRobots P3-DX,» 2011. [En línea]. Available: http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx. [Último acceso: 10 octubre 2017].

[5] R. P. Foundation, «Raspberry Pi 3 Model B,» 2017. [En línea]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/. [Último acceso: 10 octubre 2017].

[6] G. Lucas, Dirección, *Star Wars.* [Película]. Estados Unidos: Lucasfilm, 1977.

[7] I. Asimov, Yo, robot, Estados Unidos: Gnome Press, 1950.

[8] Wikipedia, «Automatic Parking,» 14 junio 2018. [En línea]. Available: https://en.wikipedia.org/wiki/Automatic_parking. [Último acceso: 15 octubre 2017].

[9] Wikipedia, «Vehículo Aéreo No Tripulado,» 19 junio 2018. [En línea]. Available: https://es.wikipedia.org/wiki/Veh%C3%ADculo_a%C3%A9reo_no_tripulado. [Último acceso: 15 octubre 2017].

[10] SITA, «Leo, robot para equipajes de SITA,» 21 junio 2017. [En línea]. Available: https://es.wikipedia.org/wiki/Veh%C3%ADculo_a%C3%A9reo_no_tripulado. [Último acceso: 15 octubre 2017].

[11] G. d. Canarias, «Isaac Castellano da la bienvenida a la primera humanoide española en el Museo Elder,» 30 agosto 2017. [En línea]. Available: http://www.gobiernodecanarias.org/noticias/tcd/86452/isaac-castellano-da-bienvenida-primera-humanoide-espanola-museo-elder. [Último acceso: 15 octubre 2017].

[12] C. Guzmán, V. Alcázar, D. Prior, E. Onaindía, D. Borrajo, J. Fdez-Olivares y E. y Quintero, «"PELEA: a Domain-Independent Architecture for Planning, Execution and Learning",» de *ICAPS'12 Scheduling and Planning Applications woRKshop (SPARK)*, Londres, 2012.

[13] L. Zi, El libro de la perfecta vacuidad, España: Kairos, 1987.

[14] H. d. Alejandría, Neumática, Reino Unido: University College London, 2016.

[15] H. d. Alejandría, Autómata.

[16] K. Čapek, Artist, *Rossum's Universal Robots.* [Art]. Teatro Nacional de Praga, 1921.

[17] I. Asimov, «Círculo vicioso,» *Astounding Science-Fiction,* vol. marzo, 1942.

[18] I. Asimov, El robot completo, Estados Unidos: Doubleday, 1982.

[19] HONDA, «ASIMO de HONDA,» 2018. [En línea]. Available: http://asimo.honda.com/. [Último acceso: 24 octubre 2017].

[20] Sony, «Aibo de Sony,» 2018. [En línea]. Available: https://aibo.sony.jp/en/. [Último acceso: 24 octubre 2017].

[21] H. R. C. Ltd, «SOPHIA de Hanson Robotics,» 2018. [En línea]. Available: http://www.hansonrobotics.com/robot/sophia/. [Último acceso: 24 octubre 2017].

[22] Wikipedia, «Valle inquietante,» 15 junio 2018. [En línea]. Available: https://es.wikipedia.org/wiki/Valle_inquietante. [Último acceso: 27 octubre 2017].

[23] O. MobileRobots, «Omron Home,» 2018. [En línea]. Available: https://industrial.omron.us/en/home. [Último acceso: 30 octubre 2017].

[24] O. MobileRobots, «Manual de operaciones P3-DX,» 7 abril 2017. [En línea]. Available: http://robots.mobilerobots.com/docs/all_docs/P3OpMan6_5.pdf. [Último acceso: 5 noviembre 2017].

[25] O. MobileRobots, «ARCOS MobileRobots firmware,» 15 septiembre 2016. [En línea]. Available: http://robots.mobilerobots.com/wiki/ARCOS. [Último acceso: 28 octubre 2017].

[26] Arduino, «Arduino Home,» 2018. [En línea]. Available: https://www.arduino.cc/. [Último acceso: 28 octubre 2017].

[27] BeagleBone, «BeagleBone board Home,» 7 mayo 2018. [En línea]. Available: https://www.arduino.cc/. [Último acceso: 28 octubre 2017].

[28] ROS, «ROSberry Pi Wiki,» 31 mayo 2012. [En línea]. Available: http://wiki.ros.org/ROSberryPi. [Último acceso: 28 octubre 2017].

[29] R. P. Foundation, «Manual Raspberry Pi 3 Model B,» 2017. [En línea]. Available: https://www.raspberrypi.org/documentation/. [Último acceso: 9 octubre 2017].

[30] Procount, «PINN Sofware at Github,» 1 junio 2018. [En línea]. Available: https://github.com/procount/pinn. [Último acceso: 29 octubre 2017].

[31] R. P. Foundation, «NOOBS Software,» 1 marzo 2018. [En línea]. Available: https://www.raspberrypi.org/documentation/installation/noobs.md. [Último acceso: 29 octubre 2017].

[32] Microsoft, «Kinect para XBox,» [En línea]. Available: https://www.xbox.com/es-ES/xbox-one/accessories/kinect. [Último acceso: 29 octubre 2017].

[33] R. L. H. W. y. H. X. M. Li, «An efficient SLAM system only using RGBD sensors,» de *2013 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, Shenzhen, 2013.

[34] S. K. B. C. W. y. B. M. Ayrton Oliver, «Using the Kinect as a navigation sensor for mobile robotics,» de *27th Conference on Image and Vision Computing New Zealand*, Dunedin, 2012.

[35] Wikipedia, «Kinect 2.0,» 31 marzo 2018. [En línea]. Available: https://es.wikipedia.org/wiki/Kinect#Kinect_2.0. [Último acceso: 30 octubre 2017].

[36] Wikipedia, «Cámara PTZ,» 24 mayo 2018. [En línea]. Available: https://es.wikipedia.org/wiki/C%C3%A1mara_PTZ. [Último acceso: 29 octubre 2017].

[37] ASUS, «ASUS Xtion PRO,» [En línea]. Available: https://www.asus.com/es/3D-Sensor/Xtion_PRO/. [Último acceso: 29 octubre 2017].

[38] Microsoft, «Manuales y especificaciones de los dispositivos Kinect,» 2018. [En línea]. Available: https://support.xbox.com/es-ES/xbox-360/console/manual-specs. [Último acceso: 29 octubre 2017].

[39] OpenKinect, «Libfreenect at Github,» 10 enero 2018. [En línea]. Available: https://github.com/OpenKinect/libfreenect. [Último acceso: 30 octubre 2017].

[40] Wikipedia, «Localización y modelado simultáneos,» 17 marzo 2018. [En línea]. Available: https://es.wikipedia.org/wiki/Localizaci%C3%B3n_y_modelado_simult%C3%A1neos. [Último acceso: 30 octubre 2017].

[41] Wikipedia, «Método de Montecarlo,» 20 julio 2017. [En línea]. Available: https://es.wikipedia.org/wiki/M%C3%A9todo_de_Montecarlo. [Último acceso: 30 octubre 2017].

[42] Y. P. J. S. y. X. L. Josep Aulinas, «"The SLAM Problem: A Survey",» de *2008 Conference on Artificial Intelligence Research & Development: 363–71*, Amsterdam, 2008.

[43] E. B. y. A. McAfee, The Second Machine Age, New York: W. W. Norton & Company, 2014.

[44] Github, «Características de Github,» 2018. [En línea]. Available: https://github.com/features. [Último acceso: 2 noviembre 2017].

[45] GitKraken, «Cliente GitKraken,» 2018. [En línea]. Available: https://www.gitkraken.com/git-client. [Último acceso: 2 noviembre 2017].

[46] VMWare, «VMWare Home,» 2018. [En línea]. Available: https://www.vmware.com/es.html. [Último acceso: 2 noviembre 2017].

[47] Hostinger, «¿Cómo funciona el SSH?,» [En línea]. Available: https://www.hostinger.es/tutoriales/que-es-ssh. [Último acceso: 2 noviembre 2017].

[48] FileZilla, «FileZilla Home,» 2018. [En línea]. Available: https://filezilla-project.org/. [Último acceso: 2 noviembre 2017].

[49] O. MobileRobots, «ARIA Software,» 25 enero 2018. [En línea]. Available: http://robots.mobilerobots.com/wiki/ARIA. [Último acceso: 4 noviembre 2017].

[50] W. S. y. G. S. S. Zaman, «"ROS-based mapping, localization and autonomous navigation using a Pioneer 3-DX robot and their relevant issues",» de *2011 Saudi International Electronics, Communications and Photonics Conference (SIECPC)*, Riyadh, 2011.

[51] ROS, «About ROS,» 2018. [En línea]. Available: http://www.ros.org/about-ros/. [Último acceso: 6 noviembre 2017].

[52] ROS, «Nodos de ROS,» 21 febrero 2012. [En línea]. Available: http://wiki.ros.org/Nodes. [Último acceso: 6 noviembre 2017].

[53] ROS, «Topics en ROS,» 1 junio 2014. [En línea]. Available: http://wiki.ros.org/Topics. [Último acceso: 6 Noviembre 2017].

[54] ROS, «Documentación de Roslaunch,» 10 febrero 2018. [En línea]. Available: http://wiki.ros.org/roslaunch. [Último acceso: 8 noviembre 2017].

[55] ROS, «Documentación del gestor Catkin,» 26 mayo 2017. [En línea]. Available: http://wiki.ros.org/catkin. [Último acceso: 6 Noviembre 2017].

[56] VisualCode, «VisualCode Home,» 2018. [En línea]. Available: https://code.visualstudio.com/. [Último acceso: 6 noviembre 2017].

[57] G. Project, «Emac Official,» 2018. [En línea]. Available: https://www.gnu.org/software/emacs/. [Último acceso: 6 noviembre 2017].

[58] O. MobileRobots, «Simulador MobileSim,» 21 abril 2016. [En línea]. Available: http://robots.mobilerobots.com/wiki/MobileSim. [Último acceso: 7 noviembre 2017].

[59] Gazebo, «Simulador Gazebo,» 2018. [En línea]. Available: http://gazebosim.org/. [Último acceso: 7 noviembre 2017].

[60] O. MobileRobots, «Mapper Software,» 20 junio 2017. [En línea]. Available: http://robots.mobilerobots.com/wiki/Mapper3. [Último acceso: 7 noviembre 2017].

[61] ROS, «Documentación de Rviz,» 16 mayo 2018. [En línea]. Available: http://wiki.ros.org/rviz. [Último acceso: 8 noviembre 2017].

[62] ROS, «Especificaciones del URDF,» 12 octubre 2014. [En línea]. Available: http://wiki.ros.org/urdf. [Último acceso: 8 noviembre 2017].

[63] ROS, «Documentación del formato Xacro,» 24 mayo 2018. [En línea]. Available: http://wiki.ros.org/xacro. [Último acceso: 8 noviembre 2017].

[64] O. MobileRobots, «Wrappers para ARIA,» 25 enero 2018. [En línea]. Available: http://robots.mobilerobots.com/wiki/ARIA#Additional_Ubuntu_and_Debian_Packages_for_Java_and_Python_Wrappers. [Último acceso: 10 noviembre 2017].

[65] Wikipedia, «Conversión entre quaternion y ángulos euler,» 19 junio 2018. [En línea]. Available: https://en.wikipedia.org/wiki/Conversion_between_quaternions_and_Euler_angles. [Último acceso: 15 febrero 2018].

[66] IBM, «Lenguajes compilados e interpretados,» [En línea]. Available: https://www.ibm.com/support/knowledgecenter/zosbasics/com.ibm.zos.zappldev/zappldev_85.htm. [Último acceso: 11 noviembre 2017].

[67] Invensis, «Ventajas de C++,» 13 marzo 2015. [En línea]. Available: https://www.invensis.net/blog/it/benefits-of-c-c-plus-plus-over-other-programming-languages/. [Último acceso: 11 noviembre 2017].

[68] W. W. W. Consortium, «Documentación de XML,» 11 octubre 2016. [En línea]. Available: https://www.w3.org/XML/. [Último acceso: 11 noviembre 2017].

[69] IBM, «Ventajas de XML,» [En línea]. Available: https://www.ibm.com/support/knowledgecenter/en/ssw_ibm_i_73/rzamj/rzamjintroadvantages.htm. [Último acceso: 11 noviembre 2017].

[70] ROS, «Documentación ROS msg,» 22 agosto 2017. [En línea]. Available: http://wiki.ros.org/msg. [Último acceso: 15 enero 2018].

[71] ROS, «Pila de navegación de ROS,» 10 marzo 2017. [En línea]. Available: http://wiki.ros.org/navigation. [Último acceso: 20 enero 2018].

[72] ROS, «Nodo de ROS depth_image_to_laserscan,» 16 enero 2018. [En línea]. Available: http://wiki.ros.org/depthimage_to_laserscan. [Último acceso: 20 enero 2018].

[73] ROS, «Documentación de Costmap_2D de ROS,» 10 enero 2018. [En línea]. Available: http://wiki.ros.org/costmap_2d. [Último acceso: 12 marzo 2018].

[74] ROS, «Documentación ROS srv,» 7 enero 2017. [En línea]. Available: http://wiki.ros.org/srv. [Último acceso: 24 enero 2018].

[75] M. Eatemadi, «Mathematical Dynamics, Kinematics Modeling and PID Equation Controller of QuadCopter,» de *International Journal of Applied Operational Research*, Tehran, 2016.

[76] ROS, «Documentación ROS tf,» 2 octubre 2017. [En línea]. Available: http://wiki.ros.org/tf. [Último acceso: 15 febrero 2018].

[77] ROS, «Nodo ROS Gmapping,» 14 febrero 2018. [En línea]. Available: http://wiki.ros.org/gmapping. [Último acceso: 10 marzo 2018].

[78] P. GNU, «Derechos de las librerías C++,» [En línea]. Available: https://gcc.gnu.org/onlinedocs/libstdc++/manual/license.html. [Último acceso: 12 abril 2018].

[79] ROS, «Licencias de ROS,» [En línea]. Available: https://creativecommons.org/licenses/by/3.0/us/. [Último acceso: 15 abril 2018].

[80] ROS, «Estándar de ROS,» 31 diciembre 2014. [En línea]. Available: http://www.ros.org/reps/rep-0103.html. [Último acceso: 15 abril 2018].

[81] O. MobileRobots, «Class reference de ARIA,» 25 enero 2018. [En línea]. Available: http://robots.mobilerobots.com/docs/api/ARIA/2.9.1/docs/index.html. [Último acceso: 10 enero 2018].

[82] ROS, «Guía del desarrollador de ROS,» 1 febrero 2017. [En línea]. Available: http://wiki.ros.org/DevelopersGuide. [Último acceso: 24 septiembre 2017].

[83] U. Mate, «Licencia Ubuntu,» [En línea]. Available: https://creativecommons.org/licenses/by-sa/4.0/. [Último acceso: 15 abril 2018].

[84] VMWare, «Licencia VMWare Player,» 2018. [En línea]. Available: https://www.vmware.com/es/products/workstation-player/workstation-player-evaluation.html. [Último acceso: 14 diciembre 2017].

[85] O. MobileRobots, «Licencia ARIA,» 25 enero 2018. [En línea]. Available: http://robots.mobilerobots.com/ARIA/download/current/README.txt. [Último acceso: 15 abril 2018].

[86] O. MobileRobots, «Licencia MobileSim,» 25 enero 2018. [En línea]. Available: http://robots.mobilerobots.com/ARIA/download/current/README.txt. [Último acceso: 15 abril 2018].

[87] O. MobileRobots, «Licencia Mapper3,» 25 enero 2018. [En línea]. Available: http://robots.mobilerobots.com/Mapper3Basic/README.txt. [Último acceso: 15 abril 2018].

[88] OpenKinect, «Licencia Libfreenect,» 11 febrero 2011. [En línea]. Available: https://openkinect.org/wiki/Policies. [Último acceso: 15 abril 2018].

[89] ROS, «Wiki de ROS,» 9 junio 2018. [En línea]. Available: http://wiki.ros.org/. [Último acceso: 10 enero 2018].

[90] Datosmacro, «Paro en España,» 1 abril 2018. [En línea]. Available: https://www.datosmacro.com/paro/espana?sc=LAB-. [Último acceso: 15 abril 2018].

[91] Industria.CCOO, «Situación del sector industrial en España,» [En línea]. Available: http://www.industria.ccoo.es/d6101308b06dd803b697939051ac1574000060.pdf. [Último acceso: 15 abril 2018].

[92] ROS, «Instalación de ROS Kinetic en Linux,» 25 julio 2017. [En línea]. Available: http://wiki.ros.org/kinetic/Installation/Ubuntu. [Último acceso: 20 diciembre 2017].

[93] ROS, «Instalación y configuración de ROSARIA,» 24 mayo 2017. [En línea]. Available: http://wiki.ros.org/ROSARIA/Tutorials/How%20to%20use%20ROSARIA. [Último acceso: 20 diciembre 2017].

[94] ROS, «Implementación de Servidor-Cliente en ROS,» 7 enero 2017. [En línea]. Available: http://wiki.ros.org/srv.

[95] Microsoft, «Requisitos Kinect v2,» 2018. [En línea]. Available: https://support.xbox.com/es-ES/xbox-on-windows/accessories/kinect-for-windows-v2-setup#bb15b3afa3134653bd3f523a74523ba4.

Anexo A – Instalación y configuración del sistema

1. Configuración de la Raspberry Pi 3

Para poder tener la placa totalmente operativa se necesitan una serie de elementos. En primer lugar, hay que tener en cuenta que se necesita, además de la placa, una fuente de alimentación que aporte al menos 5.1 V y 2.5 A, que, en este caso, será proporcionada por el puerto auxiliar del robot (Esto se explicará más en detalle en la sección Configuración de conexiones).

Además, necesitaremos una tarjeta microSD de clase 10 y el programa *Win32DiskImager* (o *GParted* en su versión Linux). Este programa nos va a permitir dar formato a la memoria para poder acceder a ella desde un sistema operativo (Adicionalmente se puede utilizar la herramienta *SD Card Formatter*, que permite no solo dar formato, sino también crear una imagen ISO de la tarjeta para poder transportar y gestionar la instalación en otra distinta). Una vez hemos conseguido acceder a la tarjeta lo único que tenemos que hacer es descargar los archivos de PINN y guardarlos en ella. Cuando la Raspberry se encienda, PINN detectará si existe una instalación en la tarjeta, si no es así nos proporcionará un sistema de gestión para la instalación de sistemas operativos. Tras configurar la red, ya sea wifi o física, elegiremos Ubuntu Mate como SO de entre todos los proporcionados y procederemos a instalarlo como cualquier sistema operativo en una máquina convencional.

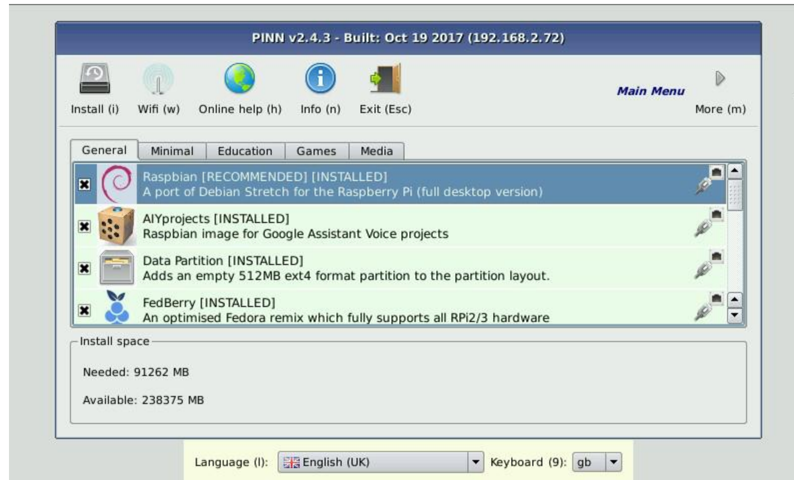


Ilustración 51 - Gestor de SSOO de PINN [30]

Una vez instalado el SO debemos configurar el entorno de trabajo. Para empezar, deberemos instalar el servidor SSH. Para ello abrimos una terminal e introducimos `sudo apt-get install ssh`, finalizada la instalación podremos iniciar un demonio para arrancar el servidor con `sudo /etc/init.d/ssh start`, no obstante como lo ideal sería iniciar un servidor SSH cada vez que la Raspberry arranque, debemos abrir una terminal y acceder a la configuración de la Raspberry con el siguiente comando: `sudo raspi-config`. Nos aparecerá una ventana de sistema en la que deberemos ir a la sección

Interfacing Options y establecer en *Enable* la fila correspondiente a SSH. Tras esto introduciremos el comando `sudo update-rc.d ssh defaults` para establecer el servidor SHH al arranque y `sudo shutdown -r now` para reiniciar la placa. Se utiliza este comando y no `sudo reboot` para evitar posibles fallos de corrupción de datos en la memoria.

Una vez podemos acceder por SSH a la placa debemos instalar y configurar el entorno de trabajo para ROS. Primero debemos instalar ARIA. Como no existe una versión precompilada para la arquitectura armhf debemos construir y compilar el código fuente. Para ello descargamos el paquete *tar* de ARIA en la página oficial de MobileRobots. Tras esto hacemos en una terminal:

```
cd Descargas (Por defecto se guardará en esta carpeta)
tar -xvf ARIA-src-2.9.4.tar.gz
mv Aria-scr-2.9.4 Aria
sudo mv Aria /usr/local
cd /usr/local/Aria
make
```

Esto instalará Aria en la placa; tras ello debemos instalar y configurar ROS. Como vamos a utilizar la versión Kinetic de ROS (la única soportada por la distribución Xenial de Ubuntu) debemos abrir una terminal e introducir los siguientes comandos:

Debemos primero configurar el fichero source.list para que acepte software de ROS:

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

Y configurar adecuadamente las claves de los repositorios:

```
sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 421C365BD9FF1F717815A3895523BAEEB01FA116
```

Primero nos aseguramos de que el índice de paquetes esté actualizado con `sudo apt-get update` y luego introducimos `sudo apt-get install ros-kinetic-desktop-full` para instalar ROS al completo. Esta acción tomará varios minutos y tras su finalización sólo nos quedan unos últimos pasos:

Inicializar rosdep, un servicio necesario para algunos componentes de ROS que se utiliza para gestionar dependencias.

```
sudo rosdep init
rosdep update
```

Tras esto ya tendremos ROS instalado y completamente operativo, sin embargo, podemos realizar unas acciones más para facilitar el trabajo en el *framework*:

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

Con este comando las variables de entorno de ROS se añadirán automáticamente a la sesión del bash cada vez que un nuevo terminal se lanza.

Y para instalar algunas herramientas de ROS para la gestión de instalaciones de proyectos, paquetes y entornos de trabajo introduciremos:

```
sudo apt-get install python-rosinstall python-rosinstall-generator python-wstool build-essential
```

Tras esto habremos instalado ROS, pero no configurado el entorno de trabajo, para ello, en una terminal introducimos:

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

Esto creará e inicializará el entorno de trabajo. Tras esto podremos instalar o crear nuevos proyectos en *~/catkin_ws/src* y construir con `catkin_make` (nótese que para construir cualquier paquete nuevo se deben establecer en la sesión las variables de entorno de ROS con `. devel/setup.bash` [92] [93]).

De esta manera solo nos queda instalar las dependencias del proyecto:

`sudo apt-get install ros-kinetic-mapping` para el mapeo.

`sudo apt-get install ros-kinetic-navigation` para la navegación.

Para configurar la Kinect:

```
sudo apt-get install ros-kinetic-freenect-launch
sudo apt-get install ros-kinetic-depthimage-to-laserscan
```

Una vez ya hemos configurado el entorno de trabajo, hace falta configurar el entorno de comunicación con el portátil de monitorización. Cada vez que una terminal nueva se inicia es necesario establecer ciertas variables de entorno que le digan a ROS dónde está el nodo maestro y dónde se están ejecutando los nodos locales. Para ello se ha desarrollado un script que se ejecuta con la inicialización de cada nueva bash.

Así, el código mostrado a continuación muestra los pasos necesarios para la configuración del entorno. Estas directrices se incluyen en el archivo .bashrc del sistema operativo para ejecutarlo cada vez que se inicia una nueva terminal (`source start_p3dx.sh`).

```
#!/bin/bash
echo "Creating enviroment..."
echo "Setting perimissions on port: /dev/ttyUSB0" #Comentar en el
portátil de monitorización

sudo chmod 777 /dev/ttyUSB0 #Comentar en el portátil

#Current dir
DIR="$( cd "$( dirname "${BASH_SOURCE[0]}" )" && pwd )"
#Target dir
SPER="$DIR/catkin_ws"

if [ "$SPER" != "$DIR" ]; then
 cd "$SPER"
fi

echo "Setting ROS Environment variables: ROS_IP and ROS_MASTER_URI"
export ROS_IP=10.42.0.1 #En el portátil es: 10.42.0.13
export ROS_MASTER_URI=http://10.42.0.1:11311 #El master es la placa
. devel/setup.bash #Entorno de ROS
```

Sección de código 5 - Script para configurar el entorno bash

Finalmente, como consejo para que la compilación en la placa sea más eficiente debemos hacer algunos cambios. Cuando se instala Ubuntu en la Raspberry, el instalador no crea por defecto una partición *swap*, esto significa que en procesos de alto coste puede que la placa se quede sin memoria virtual suficiente (como es el caso de la compilación), por lo que solo nos queda crear un espacio *swap* para tener suficiente capacidad para compilar los proyectos de ROS. Esto lo podemos hacer de la siguiente manera:

Creamos el swap:

```
sudo fallocate -l 1G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
```

Una vez creado, cada vez que queramos activarlo ejecutamos:

```
sudo swapon /swapfile
```

Si queremos tener la partición permanentemente:

```
echo '/swapfile none swap sw 0 0' | sudo tee -a /etc/fstab
```

Y con esto ya tendremos la placa totalmente operativa para utilizarla con el sistema desarrollado en el proyecto.

2. Configuración del portátil de monitorización

El portátil que monitoriza el sistema debe correr la misma distribución de ROS que la placa, pero no es absolutamente necesario que ejecute la misma distribución de Ubuntu; solo es necesario que sea la versión 16.04 que es la compatible con ROS Kinetic. Por ejemplo, en el caso de este proyecto, mientras que las Raspberry Pi tiene un Ubuntu Mate

(esto es porque es la única distribución de Ubuntu disponible para instalarse en la placa), el portátil ha ejecutado ROS Kinetic sobre Lubuntu.

De esta manera el proceso de configuración en el portátil es exactamente el mismo que en la Raspberry Pi.

Por otro lado, como ya se ha comentado anteriormente, para que ROS sea consciente de donde corre cada nodo, hay que realizar la configuración de las IP. Para ello se ejecuta el script introducido ya en la configuración de la placa, con el único cambio de mantener la IP de la máquina que levanta el nodo maestro de ROS y cambiar la variable de entorno ROS_IP a la IP del portátil para que ROS reconozca todos los elementos que toman parte en el sistema de comunicación.

3. Configuración de la Kinect

La Kinect es un sensor bastante fácil de utilizar. Una vez se le ha dado potencia empieza a transmitir datos sin ningún calibrado especial; de hecho, éste solo sería necesario en casos en lo que claramente vemos que las lecturas de la cámara no se corresponden con la realidad. Afortunadamente esto no ha sido necesario.

Lo que sí que es interesante desde el punto de vista de este proyecto, es la integración con ROS. Así, será necesario el uso de dos paquetes: *freenect-launch* que será el driver del sensor y *depthimage-to-laserscan*, utilidad que nos servirá para convertir lecturas RGBd en *laserScans*. Esto nos permitirá dos cosas: poder mapear la zona con los escaneos láser y mejorar la eficiencia general del sistema de visión para aligerar la carga de trabajo de la Raspberry.

De esta manera, la configuración e integración con ROS se basa en una serie de *launchers* que lancen los drivers y la conversión a *laserScan* y, por otro lado, un último *launcher* que publique de manera periódica una transformada estática para que ROS sepa en todo momento dónde está el sensor con respecto el *frame* de navegación (*/odom*).

```
<launch>

<!-- Start sensor driver -->
<include file="$(find freenect_launch)/launch/freenect.launch"/>

<!-- Launch kinect and depthimage_to_laser node -->
 <include file="$(find p3dx_mb)/sensors/kinect_to_laser.launch"/>

<node pkg="tf" type="static_transform_publisher"
name="base_to_camera_broadcaster" args="0.030 0 0.425 0 0 0
base_link camera_link 1" />

</launch>
```

Sección de código 6 - Lanzador para los drivers, la transformada estática y el depth_image_to_laserscan

```
<launch>
    <node pkg="depthimage_to_laserscan"
type="depthimage_to_laserscan" name="depthimage_to_laserscan">
        <remap from="image" to="/camera/depth/image_raw"/>
        <remap from="camera_info" to="/camera/depth/camera_info"/>
        <remap from="scan" to="camera/scan_depth"/>

        <rosparam>
            scan_height: 100
            scan_time: 0.167
        </rosparam>
    </node>
</launch>
```

Sección de código 7 - Lanzador para el nodo depth_image_to_laserscan

4. Configuración de las conexiones

El sistema desarrollado posee una serie de conexiones que tienen que estar configuradas de una determinada manera para que toda la arquitectura funcione sin fallos. Esta sección se va a dedicar a la explicación detallada sobre las conexiones de los diferentes componentes hardware y su relevancia en el sistema global. Así, nos vamos a encontrar con tres tipos de conexiones que vamos a tratar a continuación, conexión de datos, alimentación de dispositivos y conexión de red wifi entre máquinas:

4.1. Raspberry-Kinect

La Kinect debe transmitir los datos de las lecturas de la cámara 3D a la placa. Esta conexión se hace por medio de un cable USB con un adaptador que transforma la conexión propietaria de Microsoft en un USB convencional que se puede conectar a la Raspberry.

4.2. Raspberry-Robot

La Raspberry no tiene alimentación por sí misma, es por esto por lo que es necesaria una fuente de alimentación para que esté operativa. De esta manera se ha decidido utilizar uno de los puertos auxiliares de potencia del P3-DX; concretamente el que da 5V. Para ello se ha utilizado un cable cuyo extremo es un micro USB compatible con la placa, el otro extremo se ha tenido que modificar para poder conectarlo a la alimentación del robot.

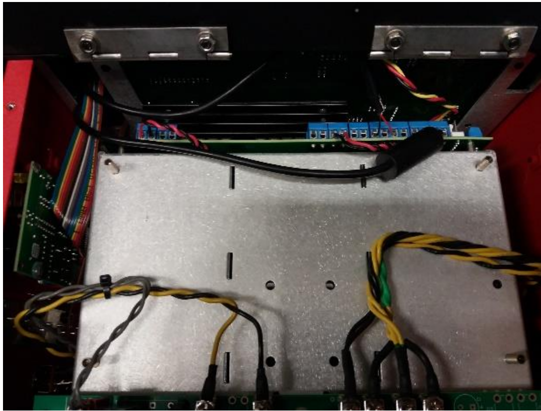


Ilustración 52 - Puertos y conexiones del p3dx

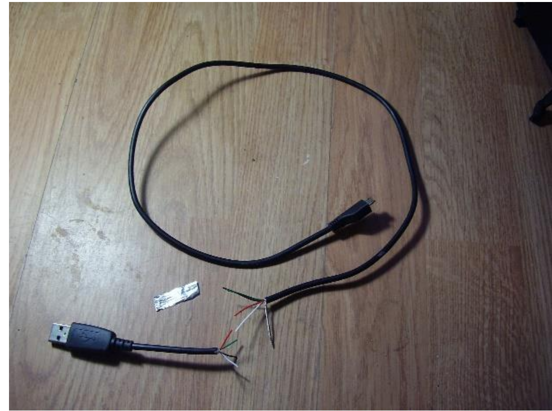


Ilustración 53 - Cable para la Raspberry

Aplicando sencillos conceptos de circuitería se ha pelado el extremo macho del cable y se han apartado los cables blanco y verde (que son los que transportan datos), preparando por otro lado el rojo y negro que son los de alimentación. Así, se utiliza un soldador para reforzar las dos conexiones negativa y positiva, y se introduce en la clema del puerto del robot.

Ilustración 54 - Set up de las conexiones

Por otro lado, la placa necesita intercambiar datos con el robot por lo que para que ambas máquinas se comuniquen se hace uso de un cable serie conectado por un extremo a un adaptador USB para la Raspberry y al otro al puerto serie del P3-DX.

4.3. Kinect-Robot

Al igual que la placa, el sensor Kinect tampoco tiene alimentación propia por lo que es necesario alimentarlo de alguna manera. Procediendo de la misma manera, se

prepara un cable que sea capaz de suministrar los 12v que requiere el sensor. Recordemos que este cable está unido al adaptador ya que la conexión propietaria contiene tanto datos como alimentación y debemos dividirla entre la conexión a la placa y la conexión al puerto auxiliar del robot.

4.4. Raspberry-Portátil (Monitorización)

La placa posee una interfaz wifi configurable, esto nos servirá para poder crear una conexión wifi Ad-Hoc entre ambas máquinas. Para ello solo necesitamos crear una nueva red de tipo Ad-Hoc con la utilidad de redes de Ubuntu y configurar el SSID y las ips de las dos partes. Una vez creada la red, ambas se podrán conectar a ella y comunicarse.

Así, mientras el portátil se encarga de la monitorización del sistema, la Raspberry se dedicará exclusivamente a la ejecución. Recordemos que la placa es potente, pero tiene sus limitaciones, y además no tenemos una manera de visualizar la actuación del sistema (sobre todo muy importante en la generación de mapas). Esto nos da la oportunidad de saber qué está haciendo el robot de manera remota en el caso de no verlo físicamente. Es importante remarcar que la placa, no obstante, es totalmente capaz de mover todo el sistema por sí sola, siendo la sección de monitorización opcional. Lo que nos permite la sección de monitorización es una respuesta dinámica del mapeo y la realización de tareas; esto nos puede ayudar en gran medida para valorar la ejecución de tareas o establecer parámetros y umbrales en el mapeo de una zona.

Anexo B – Estructura del proyecto

En esta sección se discutirá la estructura del proyecto y la gestión de código de este. El proyecto se ha gestionado con el gestor Git, está alojado en la forja Github y se divide en varios elementos que se pueden ver en la ilustración 55.

Estos componentes conforman el núcleo del proyecto, es decir, todo lo desarrollado para implementar el sistema propuesto. No obstante, ROS tiene la peculiaridad de ser un *framework* modular, y es posible que fallen algunos componentes si no se ha seguido correctamente la configuración de la placa anteriormente expuesta. Más concretamente, la arquitectura tiene una dependencia con el paquete de navegación de ROS y el de mapeo que implementa las librerías de algoritmos SLAM a través del proyecto *OpenSlam*. Por otro lado, también tenemos dependencias de los drivers de la Kinect: *freeneect_launch* y el nodo *depth_image_to_laser_scan*. Con el objetivo de clarificar este aspecto, en el anexo D se puede encontrar una tabla con las dependencias del proyecto (y los nodos de ROS necesarios utilizados durante el proyecto).

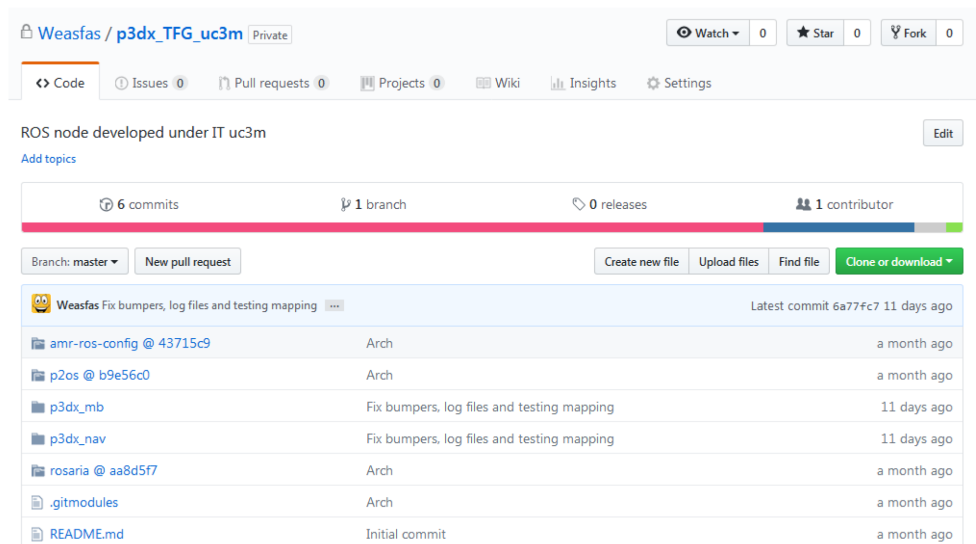


Ilustración 55 - Estructura del proyecto en Github

- **amr-ros-config** → Paquete ROS para la simulación en Rviz del p3dx.
- **p2os** → Paquete ROS con controladores, modelos y archivos de configuración para el p3dx.
- **p3dx_mb** → Paquete desarrollado que implementa la configuración y los launchers necesarios para el sistema de navegación (*navigation_stack* y Kinect) de ROS en el p3dx.
- **p3dx_nav** → Paquete desarrollado que implementa las tareas vistas durante este proyecto (Navegación 2D, teleoperación, Movimiento básico…).
- **rosaria** → Nodo puente entre ROS y ARIA, el controlador del p3dx.

Enlace al repositorio: https://github.com/Weasfas/p3dx_TFG_uc3m

Anexo C – Manual

En esta sección se expondrá el manual del sistema desarrollado. Con él, cualquiera que posea los elementos propuestos en el entorno operacional del proyecto podrá comprobar el funcionamiento del sistema.

Lo primero que necesitamos son los archivos que implementan la arquitectura, para ello podemos clonar el repositorio de Github en el *workspace* de ROS. Esto lo tendremos que hacer tanto para el portátil (en el caso de utilizarlo) como para la Raspberry Pi. Copiados los archivos tendremos que configurar el entorno de ROS: En la placa ejecutamos `source start_pi_p3dx.sh` y en el portátil `source start_desk_p3dx.sh`. Una vez hecho esto ejecutamos en cada uno `catkin_make` para construir el proyecto. Ya tendríamos instalado todo lo necesario para utilizar el sistema (es importante seguir los pasos del Anexo A para tener todo configurado correctamente).

Ahora necesitamos al menos tres terminales en el portátil de monitorización, como son nuevos entornos bash debemos ejecutar de nuevo el script: `source start_desk_p3dx.sh` en cada una. Hecho esto lanzamos los archivos para la visualización: `roslaunch p3dx_nav p3dx_rviz.launch` y en la segunda RVIZ: `rviz`. Dentro de RVIZ podremos abrir los displays de la carpeta del proyecto */p3dx_nav/rviz_displays* para visualizar el robot y su entorno (*LaserScans*, *PointClouds*, etc.). Con esto habremos terminado en el portátil.

En la tercera terminal nos tendremos que conectar a la Raspberry Pi por SSH: `ssh pi@10.42.0.1`, dentro de la Raspberry, con el entorno configurado y el proyecto construido, podremos utilizar el sistema de la siguiente manera: primero lanzamos la arquitectura con `roslaunch p3dx_nav p3dx_arch_real.launch`, esto lanzará el núcleo de la arquitectura: Sistema de navegación, publicadores, rosaria y servidor. Una vez lanzado este archivo podemos asignar tareas a través del cliente haciendo peticiones al servidor (el nodo cliente puede ejecutarse desde el portátil o la Raspberry):

- **Inicio motores:** `rosrun p3dx_nav p3dx_client initMotors`
- **Movimiento lineal:** `rosrun p3dx_nav p3dx_client moveForward [Velocidad] [Distancia] [Dirección]`
- **Movimiento angular:** `rosrun p3dx_nav p3dx_client twist [Velocidad] [Grados] [Dirección]`
- **Movimiento 2D:** `rosrun p3dx_nav p3dx_client moveTo [PosX] [PosY] [θ]`
- **Mapeo:** El mapeo se divide en varias secciones:
  - Si queremos iniciar el servicio de mapeo tendremos que ejecutar: `rosrun p3dx_nav p3dx_client startMapping`
  - Si queremos finalizar el servicio de mapeo tendremos que ejecutar: `rosrun p3dx_nav p3dx_client endMapping`
  - Si queremos crear el mapa con las lecturas de la Kinect (procurar navegar la zona lentamente para que las lecturas se registren correctamente, en RVIZ se podrá encontrar un display que permite ver a tiempo real el mapa que se está generando), tendremos que lanzar, antes de finalizar el servicio: `rosrun map_server map_saver -f [Nombre del archivo]` (el nombre no es obligatorio). Esto salvará el mapa en la carpeta raíz del proyecto, o si has proporcionando un nombre para el modificador -f que contenga una dirección, se encontrará en la dirección proporcionada.
- **Teleoperación:** `rosrun p3dx_nav p3dx_client teleop` (utilizar las teclas de dirección: ← ↑ → ↓ para controlar la base robótica y u/j, i/k para disminuir o aumentar la velocidad lineal y angular, respectivamente y q para finalizar la ejecución del nodo).

En cualquier momento una interfaz de alto nivel puede leer el topic */p3dx_sensor* para obtener el estado de bajo nivel del robot. Por otro lado, podemos finalizar la ejecución de la arquitectura con Control-C. Es importante destacar, finalmente, que, tras la ejecución, podemos comprobar los archivos *server_log.txt* y *client_log.txt* para supervisar lo que ha ocurrido en cada módulo.

Anexo D – Tablas Informativas

1. Especificaciones de Hardware

1.1. Raspberry Pi 3

En la siguiente tabla se puede encontrar toda la información oficial de las especificaciones de la placa Raspberry Pi 3 Modelo B.

Procesador	Broadcom BCM2837B0, Cortex-A53 64-bit SoC @ 1.4GHz
Memoria	1GB LPDDR2 SDRAM
Conectividad	• 2.4GHz and 5GHz IEEE 802.11.b/g/n/ac wireless LAN, Bluetooth 4.2, BLE. • Gigabit Ethernet over USB 2.0 (maximum throughput 300Mbps) • 4 × USB 2.0 ports
Acceso	Extended 40-pin GPIO header
Video y Sonido	• 1 × full size HDMI • MIPI DSI display port • MIPI CSI camera port • 4 pole stereo output and composite video port
Multimedia	H.264, MPEG-4 decode (1080p30); H.264 encode (1080p30); OpenGL ES 1.1, 2.0 graphics
Soporte SD	Micro SD format for loading operating system and data storage
Potencia de entrada	• 5V/2.5A DC via micro USB connector • 5V DC via GPIO header • Power over Ethernet (PoE)–enabled (requires separate PoE HAT)
Entorno	Operating temperature, 0–50°C
Periodo de producción	El modelo se mantendrá en producción al menos hasta enero de 2023.

Tabla 78 - Especificaciones de hardware Raspberry Pi 3 [5]

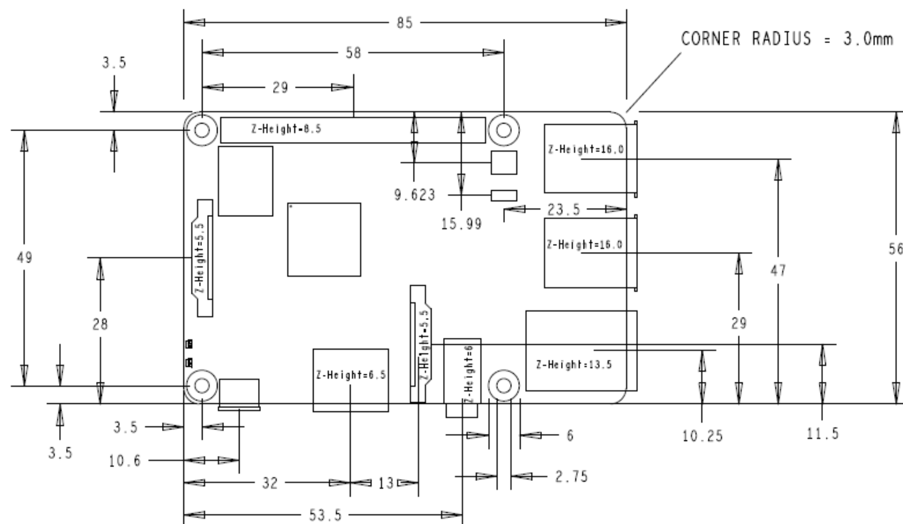


Ilustración 56 - Especificaciones físicas Raspberry Pi [3]

1.2. Plataforma robótica p3dx (Pioneer 3DX)

Construcción	• Body: 1.6 mm aluminum (powder-coated) • Tires: Foam-filled rubber
Manejo	• Robot Weight: 9 kg • Operating Payload: 17 kg
Movimiento	• Turn Radius: 0 cm • Swing Radius: 26.7 cm • Max. Forward/Backward Speed: 1.2 m/s • Rotation Speed: 300°/s • Max. Traversable Step: 2.5 cm • Max. Traversable Gap: 5 cm • Max. Traversable Grade: 25% • Traversable Terrain: Indoor, wheelchair accessible
Potencia	• Run Time: 8-10 hours w/3 batteries • Charge Time: 12 hours (standard) or 2.4 hrs (optional high-capacity charger) • Available Power Supplies: o 5 V @ 1.5 A switched o 12 V @ 2.5 A switched
Batería	• Supports up to 3 at a time • Voltage: 12 V • Capacity: 7.2 Ah (each) • Chemistry: lead acid • Hot-swappable Batteries: Yes
Microcontrolador (I/O)	• System Serial • 32 digital inputs • 8 digital outputs

	• 7 analog inputs • 3 serial expansion ports
Panel de Control	• MIDI programmable piezo buzzer • Main power indicator • Battery charge indicator • 2 AUX power switches • System reset • Motor enable pushbutton

Tabla 79 - Especificaciones de hardware p3dx

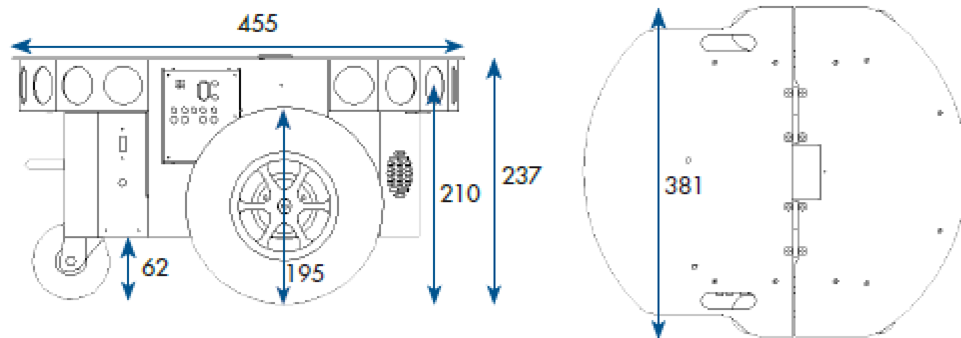


Ilustración 57 - Dimensiones p3dx [24]

1.3. Sensor Kinect

Ángulo de visión	43° vertical by 57° horizontal field of view
Rango de inclinación	±27°
Frame rate	30 frames per second (FPS)
Formato de audio	16-kHz, 24-bit mono pulse code modulation (PCM)
Entrada de audio	A four-microphone array with 24-bit analog-to-digital converter (ADC) and Kinect-resident signal processing including acoustic echo cancellation and noise suppression
Acelerómetros	A 2G/4G/8G accelerometer configured for the 2G range, with a 1° accuracy upper limit.

Tabla 80 - Especificaciones de hardware Kinect [38]

2. Especificaciones de Software

2.1. Nodos de ROS

Nodo	Uso	Referencia
Navigation Stack	Navegación y generación de mapas	wiki.ros.org/navigation
Depth_image_to_laserscan	Transformar lecturas de la Kinect	wiki.ros.org/depthimage_to_laserscan
Freenect_launch	API de ROS para los drivers de la Kinect	wiki.ros.org/freenect_launch
Gmapping	Servicios de mapeo	wiki.ros.org/gmapping
RosAria	Puente entre ROS y Aria	wiki.ros.org/ROSARIA
Rviz	Visualización	wiki.ros.org/rviz

Tabla 81 - Paquetes de ROS utilizados

2.2. Librerías utilizadas

Librería	Uso	Referencia/s
ARIA	Drivers del p3dx	mobilerobots.com/wiki/ARIA
MobileSim	Simulación del robot	mobilerobots.com/wiki/MobileSim
Mapper3	Generación de mapas de prueba	mobilerobots.com/wiki/Mapper3
OpenKinect	Drivers de la Kinect	openkinect.org/wiki/Main_Page github.com/OpenKinect/libfreenect

Tabla 82 - Librerías utilizadas

Anexo E – Acrónimos

- **ROS**: **R**obot **O**perating **S**ystem
- **PTZ**: **P**an-**T**ilt-**Z**oom
- **SLAM**: **S**imultaneous **L**ocalization **A**nd **M**apping
- **LIDAR**: **L**aser **I**maging **D**etection **A**nd **R**anging
- **AMCL**: **A**daptative **M**onte **C**arlo **L**ocalization
- **YPR**: **Y**aw-**P**itch-**R**oll
- **ARIA**: **A**dvanced **R**obots **I**nterface for **A**pplications

- **P3DX**: **P**ioneer **3-DX**
- **XML**: **E**xtensible **M**arkup **L**anguage
- **SSID**: **S**ervice **S**et **ID**entifier
- **FTP**: **F**ile **T**ransfer **P**rotocol
- **SSH**: **S**ecure **Sh**ell
- **TCP**: **T**ransmission **C**ontrol **P**rotocol
- **GNU**: **G**nu is **N**ot **U**nix
- **GPL**: **G**eneral **P**ublic **L**icense
- **RGB**: **R**ed **G**reen **B**lue
- **RGBd**: **R**ed **G**reen **B**lue and **D**epth
- **USB**: **U**niversal **S**erial **B**us
- **SO**: **S**istema **O**perativo
- **URDF**: **U**nified **R**obot **D**escription **F**ormat
- **S.S**: **S**eguridad **S**ocial

Anexo F – Glosario

- **Spawner**: Programa que implementa la generación de un modelo robótico en un entorno simulado.
- **Joint States**: En un robot, es la integración de dos o más enlaces (la conexión de las diferentes articulaciones de un manipulador).
- **Coordinate frame**: Sistema que utiliza uno o más números para la determinación de la posición de un punto o de otro objeto geométrico.
- **Nodo**: Se trata de un proceso de ROS que realiza tareas y/o cómputos.
- **Topic**: Canales de datos con nombre específico sobre los cuales diferentes nodos intercambian mensajes.
- **Launcher**: Fichero en formato XML utilizado para iniciar simultáneamente varios nodos a través de la herramienta de ROS roslaunch.
- **Transformada (transform)**: Resultado de convertir las coordenadas específicas de un sistema de coordenadas a otro, generalmente a través de la rotación y la escalado.
- **Catkin:** Sistema de construcción (build) oficial de ROS y el sucesor del sistema de construcción original rosbuild. Combina macros CMake y scripts de Python para proporcionar alguna funcionalidad sobre el flujo de trabajo normal de CMake.

- **Sistema de teleoperación**: Sistema que permite gobernar un robot ubicado en una zona remota a través del manejo de un conjunto de accionadores y/o mandos que modifican velocidad, fuerza, etc. de la plataforma.
- **Escaneo láser (LaserScan)**: Datos generados por un sensor láser al encontrarse con obstáculos en el camino de lanzamiento de la señal.
- **Punto de nube (Pointcloud)**: Colección de puntos N-dimensionales que contienen información acerca de diferentes aspectos (intensidad, coste, normales). Resultado de las lecturas de un sensor.
- **Problema de la *Alcanzabilidad***: Traducido literalmente de *"Reachability Problem"*. Se trata de un problema de decisión basado en dirimir si es posible llegar a un estado final desde un estado inicial dado con un conjunto finito de transiciones entre estados.
- **Framework**: Es un conjunto estandarizado de conceptos, prácticas y criterios que tienen el objetivo de afrontar un tipo de problemática en particular tratada como referente o problema base, para abordar y resolver nuevos problemas de índole similar.
- **Unix**: Se trata de un sistema operativo que, siendo totalmente portable, permite múltiples tareas y usuarios.
- **GNU/Linux (Ubuntu)**: Término para referirse a la combinación de varios proyectos, entre los cuales destacan GNU y el núcleo Linux. Particularmente Ubuntu es una distribución de Linux basada en la arquitectura de Debian.
- **Swap**: Zona del disco (un fichero o partición) utilizada para almacenar las imágenes de los procesos que no se vayan a mantener en memoria física.
- **PELEA**: Arquitectura de planificación formada por distintos componentes que permiten la integración de la monitorización, planificación, replanificación y técnicas de aprendizaje en una plataforma robótica.
- **NOOBS**: Aplicación que facilita la instalación de diversas distribuciones Linux en la placa Raspberry Pi.
- **PINN**: Inicialmente un fork de NOOBS, se trata de una versión más avanzada de éste que da soporte a más sistemas operativos y distribuciones Linux.
- **Git**: Software de control de versiones diseñado por Linus Torvalds.
- **Github**: Forja para alojar proyectos que utilizan el sistema Git.
- **GitKraken**: Cliente gráfico para gestionar los proyectos en sistemas de control de versiones.

Anexo G – Summary

1. Introduction

The human being always had a skill to came up with new destructive ways of thinking, little time has passed since human inventiveness presented those stories of madness and science fiction in which robots subdued the human being under the yoke of slavery and tyranny, even of those creations with a general purpose, capable of performing any task as a human or felt and behave like us.

It is true that we are far from these concepts (although we saw many and varied talks and discussions about the artificial intelligence we are currently developing and discovering), we are not so far from a robotics that opens an infinite number of doors to fields of study and research as diverse as useful. Home automation is a clear example, it has gradually taught us many of its applications in an effective way, each one more imaginative.

On the other hand, large factories have developed new manufacturing methods supported by robotics. These systems, integrated in the production chains are capable of handling objects faster than a human, improving the productivity and efficiency of the industry in general. Thus, robotics begins to displace the human being from jobs that involve monotonous and boring tasks, providing the opportunity to focus on other fields of study for the development of knowledge. We all remember that cute couple of C3PO and R2-D2 from the StarWars film series [6], with total movement autonomy, complete perception of their environment and logical reasoning that, even if discussed by some, would be considered totally human; or Sonny [7] that agile and autonomous robot, capable of questioning, reasoning and bordering the three famous laws of robotics.

Unfortunately, the robotics we are discussing is still far from this, but we are not so distant from finding an autonomous motion system that offers a dynamic and more or less "intelligent" response in a controlled environment. The aerospace industry is a clear example of the latter and of how important the development and study of mobile robotics will be in the future, both in the abysmal space and in the earth domain; we are immersed in the "space race" of the 21st century. Many milestones and lines of research have been and will be opened. While much remains to be discovered, we can venture to say we are on the path to technological progress and development that has not been seen so far in this field.

If we stop and look closely, we will notice there are many types of mobile robots. Most of which offer common features such as teleoperated movement or target planning, others, however, offer much more specific aspects such as path finding or the use of clamps or hooks to manipulate objects (Who does not know the Blocks World Domain?). Specifically for this TFG, a P3-DX mobile robot has been used. This platform is capable of navigating through a controlled environment using pre-set or dynamically set targets.

This is one of many robots and series of robots that seek to open new research lines and applications in different fields in which, in the past, there was no presence of robotics or artificial intelligence, and that little by little we are realizing they are really useful and practical in them.

Anywise, mobile robots have meant (and still do) a drastic change in the mindset of programmers and builders, as they have to take into account not only the mere statics of the machine, but also its dynamics and the way it interacts with an environment that is intended to simulate the real world. And who has not seen a car driving or parking alone [8] or a drone flying over the city [9]?

To illustrate all this, this year, we had news of Leo [10], a robot designed to manage and take care of the luggage of passengers at the Comodoro Arturo Merino Benítez airport in Santiago, or Ada [11], the first humanoid robot to arrive in Spain to serve as a guide for visitors to the Elder Museum of Science and Technology in Las Palmas de Gran Canaria. A few examples of the extension of these two fields: robotics and artificial intelligence, which, in most cases, go hand in hand to offer us services as useful as those mentioned above.

1.1. Motivation

Many times, with a set of scarce resources you can come up with really useful and effective creations. The P3-DX platform does not have a navigation system nor a powerful computer, thus low-cost resources such as the Raspberry Pi or sensors like the Kinect can offer us cheap and capable alternatives to carry out complex and elaborate tasks. The implementation of these systems is often only dependent on the robotic platform, since what may probably differ will be the addition of more quality sensors, which, if we look closely, do not really pose a problem, since the base implementation is there and you just have to modify the set of signals, noise, etc. that is received. The basics of the system will be the same, adapted to the new conditions.

However, it is interesting to delve into certain problems that appear in this field and turn out to be recurrent. Not all of us have the capacity of large companies to develop this type of architecture, and this is where the community development of ROS or the use of low-cost components comes in handy. Although they do not guarantee the best version of the systems, they are able to be surprisingly competent to culminate in the robust and effective behavior that we always look for in any robotic behavior. We all have witnessed the problems that arise from autonomous cars; certainly, its operation is much better than you would expect, since not only do they have real-time data from sensors, but maps that have already been built allowing more secure and controlled set of actions. Even if we look a little further from home, the Rovers launched on Mars also pose a series of challenges, robotic platforms whose set of objectives contains the tasks of recognition and area mapping. And this is where the SLAM (Simultaneous Localization And Mapping) has emerged as one of the most

recurrent problems in robotics, and it would be clear to think that the more resources we allocate to these complex problems, the more quickly and efficiently we will devise ways to solve them or, ultimately, to avoid them.

Thus, anecdotally we will say we do not only find this problem in complex fields such as space navigation. I recently had the opportunity to discuss how those cleaning robots are able to navigate the environment. To my surprise, it turns out that some of these devices address the SLAM problem, and if you are a little curious you can see with the manufacturer's applications how they are able to build their own map and navigate in a reactive way. A little curiosity turned into motivation, a practice that extends beyond the academic and delves into the most mundane, our homes; And who wants a system like these, that navigates alone and does not work properly?

1.2. Goals

The initial goal of the project was to be able to control a robotic platform with a Raspberry Pi Model 3 board. The robotic platform soon took shape and the MobileRobots P3DX was chosen. However, as this seemed a bit limited and scarce, an extension, based on the original main idea was proposed, to which some layers of complexity were added.

In this context, it was decided to address the problem of navigation and mapping on the robotic platform. A navigation system is usually divided into two main layers, the first one is a control system that manages the assignment of tasks at the high level (initially developed with automated planning and, thus, is not part of the project goals), the second one, implemented in this project, is the one in charge of managing the resources at a low level, in the robot; basic movements, sensors, encoders, etc. For this purpose, the ROS framework was chosen and a series of nodes were developed to allow not only the integration of the platform with PELEA[12] (or any architecture that involves the implementation of a high-level interface) with ROS itself, but also the communication between the platform microcontroller and all other management systems, generally provided by ROS.

Therefore, it is proposed the develop of a low-level architecture capable of providing a simple interface that would allow autonomous mapping and navigation in the P3-DX through the execution of simple commands and the publication of ROS topics. The latter would be given to a high-level system to provide the ability to manage and monitor the platform.

The objectives proposed in the implementation of this project are set out below. Note that they are presented in order of relevance; however, they are all treated with equal importance.

1. Demonstrate the possibility of controlling a mobile robotic platform with a Raspberry Pi using ROS.

2. Having checked the previous point, provide a simple interface consisting of a series of low-level tasks that can be used for a high-level external control, either naturally or through paradigms such as automatic planning.
3. Abstract ROS architecture and propose simple commands and topics that can be used quickly by a high level system.
4. Develop a first version of SLAM, based on the functionalities provided by ROS and using the Kinect sensor. Assume the limitations for this extra behaviour and demonstrate the feasibility of mapping with the Raspberry Pi.

In this manner, the final system will provide an easy-to-access interface to any high-level application that intends to generate a plan composed of tasks that must be transformed to low level. This modular system will be implemented based on a series of nodes that will execute the tasks for the movement of the robotic base and the treatment of the SLAM problem (mapping with the Kinect and localization of the robotic base). Thus, to address the communication problem a Client (High level) Server (Low level) architecture will be used. To address the movement of the robotic base we will use the microcontroller of the P3DX, ARIA, and the node to connect to ROS called RosAria, that will allow us to send speed commands to the robot. The ROS navigation stack will help us with the obstacle avoidance using the Kinect readings, which will also be used with the *Gmapping* node and *Depth_image_to_laser_scan* for mapping with the laser scans provided by the Kinect readings transformation by the latter node. Finally, the system must incorporate tools for the platform management, as well as for its monitoring, which will ultimately make possible to evaluate and verify that its performance is safe and consistent with the expectations proposed for this type of architecture. This is why we will use the generation of log files in each part and Rviz tool, a program with which we can visualize the behaviour of the platform.

1.3. State of the art

1.3.1. Mobile robots

Many robots have been used in the industry since the 60s with the aim of facilitating different tasks to human beings. Mobile robots in particular can be used for the exploration and transport of different goods and objects; however, until the moment of its appearance, there were not so many complications that emanated from the very place that the robot in question occupied in the world. Although for the human being the fact of moving is something trivial, for the mobile robot it is a hard issue to address. A mobile robot needs to know all the time its location and the space it occupies in a world that is often quite difficult to define with its logic. The ultimate goal of these platforms is, generally, to consistently follow a series of rational actions to achieve a task that can range

from the simplicity of reaching a specific goal or position to moving and placing a series of boxes or pallets. This is, in a certain way, one of the biggest problems intended to be solved in the field of mobile robots and where most of the other problems lie.

Since 1995, Omron Adept MobileRobots [23] (formerly Adept MobileRobots) has been the preferred source for research and education with mobile robots. This company is dedicated to the billing of mobile Robots, with all its hardware, software and accessories to guarantee autonomous cartography and navigation, as well as a flexible and fluid control. In this particular work we will make use of one of its most recent models, the P3-DX, with a huge set of possibilities and characteristics that will be discussed within the pages of this report.

1.3.2. Raspberry Pi 3

The raspberry pi 3 is the third generation of Raspberry Pi, a low-cost board computer developed in the United Kingdom by the Raspberry Pi Foundation with the aim of stimulating the teaching of computer science in schools. It is a board with high specifications that comes with a long list of applications. Its main feature is none other than the price. Under the premise of offering the power of a computer at low prices, the Raspberry Pi has become one of the most intelligent and affordable ways to have a particularly efficient and dynamic hardware accessible to anyone. This particular model was brought to light in 2016, and with it, the processor is renewed once again by the Broadcom company and new features such as Wi-Fi and Bluetooth are added without the need for adapters.

1.3.3. Kinect

Kinect for Xbox 360 [32] is a device originally intended for the entertainment industry, more specifically, it is an RGB camera accompanied by a depth sensor, developed under the codename "Project Natal" and based on the reference design of the PrimeSense company. Created by Alex Kipman and developed within the Microsoft company, it would allow interaction with the Xbox 360 console without the need of physical contact, nor the use of traditional controllers such as gamepads. The Kinect is a clear example of a really cheap but very powerful device, which did not reach deep in the world of entertainment but used by developers and researchers with low budgets to have very accurate sensor readings.

Since, in our case, it is vital to incorporate sensors with much more advanced readings than sonars (since these have not really been intended for mapping) it is very easy to incorporate this type of 3D cameras that guarantee a high quality of readings and, therefore, the generation of maps closer to reality. Hence, in

academic articles that address the problem of SLAM, efficient systems based on RGBD sensors are proposed [34], and since the Kinect sensor is quite cheap and effective [33], it turns out to be a very good choice to address the goals stated in this project.

1.3.4. SLAM

SLAM stands for **S**imultaneous **L**ocalization **A**nd **M**apping. It can be defined as the problem, within the area of computing, that involves building a map of an unknown environment while knowing at all times and simultaneously the location of the agent within it.

The problem is formalized mathematically based on statistics, on the estimated calculation of the location of the agent by means of various data sources such as sensors or the relative odometry of the robot. Therefore, after all, the final goal is computing. The data used can be data collected by sensors such as a laser or a 3D camera such as the Kinect sensor.

To really explain what SLAM is, we could divide it into several sections: the first one is the mapping, where the algorithms used to compute the relative location of the agent and the scanned objects are discussed (Algorithms based on the Monte Carlo method [41] are quite useful or approaches such those used by autonomous cars, which already have a significant database of maps already built: cities, streets, intersections etc). Another element are the sensors, with which the sources of the data received in order to generate maps are managed. Generally, 3D cameras like the Kinect are very useful, lasers or LIDAR sensors, which have a higher precision ... On the other hand, the problem of movement is discussed, in which the relative speeds and actions of the agent are addressed on the environment that is being mapped. These commands are usually simplified, being linear and angular speeds to reach a goal point. And finally, the problem of complexity and the "Loop clousure" is tackled.

Like any computational problem, there is a certain degree of complexity due to the computational capacity needed when relative position calculations are carried out, especially in the context of 3D SLAM [42]. We have an example in robots with embedded systems, unable to perform an efficient SLAM due to the need for that computational capacity they do not have. In addition to this, many researchers and experts continue to claim these days that SLAM is still one of the fundamental challenges in robotics [43].

Furthermore, and equally important, is the problem of "Loop clousure", very common in SLAM. It is the problem that arises when recognizing points that have already been visited previously and updating the constructed map accordingly. The correction of this issue is a fundamental task in the development of robust

and effective SLAM systems, because its resolution is quite useful to greatly increase the accuracy over the calculated odometry of the robot; since it is necessary to take into account the calculations of the positions are estimated and subject to errors that generally creep over time, which means they worsen the quality of the final map if not treated with care. The approach to this type of problems implements point comparison algorithms that would ultimately allow the recognition of the points already detected and adjust odometry and speed more accurately.

2. Results

After the complete development of the project and the implementation of the architecture, we can perform an assessment of the achieved goals. This section will be the culmination of the work developed where the achievement of these objectives is argued.

- Although I already had experience with the Raspberry Pi, I have developed a higher level of experience with the platform and its architecture. Not only in terms of its possibilities, but also regarding its limitations. Installation, operation and management of the board have been fundamental parts of the project. In this way, the board has been completely integrated with the robotic platform, ultimately verifying its viability for robotic control.
- ROS was a much more complex challenge. This great community is full of contributions, and the framework itself has countless possibilities. The solutions proposed to the problems stated and the approach have been satisfactory. A high knowledge level about the framework architecture and its different applications have been achieved, even though a part of them have not been included in this project for extension reasons. The implementation of an architecture that enables the execution of tasks at a low level has been much more agile thanks to its use, helping to simplify tasks that, otherwise, would have been much more complicated. There are so many possibilities and opportunities that it is often necessary to focus the efforts on something in particular, not try to cover everything.
- The movement of a robotic base may seem trivial, but it is an effort that is not so easy to tackle. After the integration of all the systems, a robust architecture is needed, the correct implementation of controllers and publishers and, in general, systems that adequately control and manage the data and messages generated to ensure adequate robotic behavior. Thus, we have managed to provide a set of simple commands and implemented an easy-to-use interface that allows us to use all these tools that ROS offers: messages, topics, etc. in a quick and easy way.
- The problem of SLAM is a recurring problem in the field of robotics, used in this project to learn how to deal with very complex issues efficiently, trying to

maximize the available resources. Although the maps generated are not excessively detailed and complex, we can conclude that this is part of the sensors: the architecture is there, all that remains is to improve the hardware; for example, the readings of a laser are not comparable, which would mean a more precise mapping. In that manner, a first version of a SLAM system has been developed. True, it needs to be polished, but we must take into account the limitations of our sensors. However, with the maps generated, we can see the similarity between them and the reality, something we stated as a goal in our early steps, and that suppose a very important milestone for the development of a more complex SLAM system.

- Finally, we must discuss the integration of ROS and Raspberry, which has been relatively complex due to the board constraints. It is true the Raspberry is quite powerful, however we have to remember that robotic systems and especially SLAM algorithms need a lot of computing power. If we add to that the processing capacity to run ROS and its modules we have a combination which requires high specifications. That is why an efficiency-based implementation is of vital importance, something achieved with languages such as C++, cross execution or the threads implementation that handle ROS internally.

3. Conclusions

3.1. Main conclusions

It is evident the SLAM and navigation problems in robotic platforms continue to have certain aspects in which we can find some difficulties. Addressing this type of task is, in many cases, somewhat complicated due to the number of variables and situations that must be taken into account. Sensor calibration and error margins are really important, and this proof of concept shows that with low amount of resources, greats things can be achieved. However, the relevance of the quality of the elements that includes these types of architectures is demonstrated, since the optimization and error minimization of a dynamic system are of vital importance when we speak of fairly controlled and supervised environments. Environments in which the human being can also be a part of and can be affected by systems that may not comprise with a series of basic requirements that establish a threshold of safety and performance.

The proposed system is a tool to tackle this problem, in which there is not only a small presence of supervisory elements, but also the help of high level formalizations that address the assignments of autonomous and dynamic tasks. In a certain way, the field of study of robotics has improved enormously in these years, achieving really relevant and important milestones. However, we will have always the chance to think the current models are not the best, some implementations can be more effective and robust systems can be safer and better when addressing contingent failures. This, in

truth, is the motivation that moves the human being on the path of development, is what really makes us improve in many different aspects.

These are small steps that bring us closer to better systems, which provide different points of view and represent pillars on which development can be built a posteriori. The implemented system allows the integration with high level modules, allows the autonomous behavior and mapping of an area, aspects that not many years ago were believed to belong to science fiction. Why could not we allow ourselves to dream with a robot capable of transporting blind people or an exoskeleton that allows disabled persons to walk independently? This would be achieved with this type of systems, where the action and reaction are partners and require a high degree of robustness tackling possible failures.

3.2. General conclusions

If we conduct an honest assessment of the global project, we can conclude different aspects that seem to be important.

First of all, we should talk about ROS and the functionalities it provides. The communication in a robotic platform with a Rapsberry Pi becomes relatively simple with the development of basic approaches such as the Client-Server architecture, widely used today. These architectures give us the opportunity to develop easy-to-use interfaces that enable the ability to manage different tasks at a low level. And, although the Raspberry Pi seems to have certain problems being limited in computing capacity, with a little patience, it manages to solve quite complex tasks.

On the other hand, the implementation of planning systems, SLAM... are relatively complex both in development and availability of resources. This fact has become more evident with the appearance of entire communities and collaborative platforms that, even being integrated by thousands of people scattered around the world, allow us to unite large knowledge bases and construct the development on many different fields of study which may motivate any scientist or developer. In this case it is ROS and robotics, in another, it can be the physics with the Theory of everything or the computation science with the equality between P and NP.

In addition, with the unfortunate events we have seen recently with autonomous vehicles, studying from different perspectives these kind of systems, we can conclude the implementation of the proposed architecture involves an assessment on how to approach the problem, how to make efficient something that turns out to be really hard and complex. It highlights the need to create robust systems, capable of acting in different situations under an unbiased criterion, aimed at contributing not to destroy. Just as the eyes of any animal need to be healthy for the spatial perception of the environment, it has been discovered the sensors of a robotic platform need to be

equally or "healthier" (calibrated); They need to have a very limited catalog of limitations, ensuring a minimization of errors as efficiently as possible.

After the system evaluation, on the other hand, we can conclude that the best scenario in which the system and ROS itself are executed (perhaps due to its modular character) is in Configuration 2, where the whole system is managed with the board. However, the importance of data sharing and the performance of data communication in a dynamic robotic system should be stressed. The resulting maps, which share similarities with reality, have been similar in all the proposed configurations. These maps, of variable dimensions, entail a first approach to a SLAM system that has ended with the generation of the largest map created (Sabatini Building – University Carlos III), putting special emphasis on the quality of sensors used and the processing of their data. In this way, it is very important to evaluate the performance of the global system itself, which is what ultimately serves as an evidence to establish a robust and efficient system.

All in all, a small step for robotic development, with which some problems are solved, but others even more interesting, if possible, arise. Somehow, we could think that after this we would have finished, but the improvement can be infinite and motivation, one of the most important causes of knowledge.

3.3. Future work

It is easy to think that in this field it is necessary to make progress, to continue creating and going further with this type of systems. After the completion of this project two milestones of vital importance have been achieved to establish different ways to follow from this point.

In the first place, an architecture capable of connecting to a high-level module and managing the actions and systems at the low level layer of a robotic platform has been implemented. At this point, we must comment that architectures like PELEA can be elements with which we could start an integration project that will allow us to create a complete navigation system. The generation of high-level domains and the use of paradigms of planning with Machine Learning concepts (Learning by Reinforcement) are tasks that in the future we could implement given the proposed architecture.

Secondly, the SLAM problem has been addressed, and the creation of maps based on the sensor readings has been achieved. Something that can be used with a high-level architecture as we mentioned earlier. The comparison with zone maps is the basis on which autonomous navigation works, thus, we could establish a new point in our route sheet to further develop this system and prepare the ROS navigation stack completely, attaching navigation algorithms (AMCL) that would allow the autonomy of the robotic base with obstacle evasion and long-term navigation.

Finally, if we want to improve the proposed architecture independently, we could propose a change in hardware. Maybe we can develop a Raspberry Pi cluster to have more computational power, include more accurate sensors such as LIDAR or even make use of more RGBd cameras to expand the vision range for mapping. It is not uncommon in this sense to find systems with three or four Kinect, something that would provide a complete view of the area, in exchange for the need of a much higher processing capacity. It would be worth considering if the proposed cluster would be sufficient to maintain these new requirements.

Anexo H – Documentos y enlaces de interés

- Comparativas de Robots Móviles y más especificaciones
- Manual completo p3dx (Link a Drive debido a que es necesario hacer una petición al distribuidor).
- Wiki de ROS
- ROS by example Vol. 1
- Class reference de ROS tf
- Class reference de ARIA