eman ta zabal zazu
Universidad del País Vasco
Euskal Herriko Unibertsitatea
BILBOKO INGENIARITZA ESKOLA
ESCUELA DE INGENIERÍA DE BILBAO

MÁSTER UNIVERSITARIO EN INGENIERÍA DE CONTROL, AUTOMATIZACIÓN Y ROBÓTICA

TRABAJO FIN DE MÁSTER

INTEGRACIÓN DE AGV (AUTOMATED GUIDED VEHICLES) EN SISTEMAS DE FABRICACIÓN FLEXIBLES EN EL CONTEXTO DE LA INDUSTRIA 4.0

Estudiante	*Díez, Artime, Borja Guillermo*
Director	*Casquero, Oyarzabal, Oskar*
Codirectora	*Armentia, Díaz de Tuesta, Aintzane*
Departamento	*Ingeniería de Sistemas y Automática*
Curso académico	*2021/2022*

Bilbao, 18 de mayo de 2022

AGRADECIMIENTOS

Este documento contiene mi Trabajo Fin de Máster llevado a cabo dentro del Máster en Ingeniería de Control, Automatización y Robótica, de la Escuela de Ingeniería de Bilbao. Ha sido desarrollado desde inicios de octubre de 2021 hasta mediados de mayo de 2022. En su realización, así como en la finalización de mis estudios, ha sido vital toda la ayuda obtenida por parte de mis supervisores, compañeros, familia y amigos, a los que me gustaría agradecer en las siguientes líneas.

Primeramente, agradecer al Doctor Oskar Casquero Oyarzabal y al Doctorando Alejandro López García por la oportunidad que me ofrecieron de llevar a cabo mi trabajo fin de máster dentro del grupo de investigación GCIS. También, por toda la confianza que depositaron en mí y por todo el tiempo que dedicaron a formarme y ayudarme con los problemas que surgieron por el camino.

Por otro lado, agradecer a mis compañeros de departamento y aula, los cuales siempre se mostraron dispuestos a ayudar o complementar el contenido del presente trabajo.

Agradecer también a mi familia, pareja y amigos, por ser una fuente de inspiración y aliciente para ofrecer siempre lo mejor de mi persona.

Finalmente, agradecer al Doctor Asier Zubizarreta Pico, quien me animó y aconsejó a realizar el máster en el que se recoge este proyecto.

RESUMEN

Resumen:

Los sistemas de fabricación convencionales se encuentran inmersos en una transición hacia un modelo de producción más flexible, capaz de hacer frente a las demandas actuales del mercado. En este nuevo modelo de producción, los Vehículos Autónomos de Transporte (AGV) juegan un papel fundamental para dotar a la industria de dos características clave para lograr la plena flexibilidad: *conectividad* y *descentralización*.

La investigación en el área del transporte autónomo se ha centrado, principalmente, en el diseño de algoritmos y en el desarrollo de herramientas de ingeniería y librerías para la integración de componentes. Sin embargo, no se ha prestado tanta atención a las necesidades de interacción (*conectividad*) que surgen entre los AGV y otros activos de fabricación en entornos distribuidos y cambiantes como los que presentan los Sistemas de Fabricación Avanzada o Sistemas de Fabricación Flexible (FMS). En este sentido, la revisión de la literatura refleja que dichos requisitos de conectividad entre activos robóticos y no robóticos carecen de soporte por parte de los principales frameworks robóticos. A esta problemática, se le suma la necesidad de dotar de inteligencia distribuida (*descentralización*) a los activos de fabricación para que puedan realizar tareas de forma autónoma (e.g., la navegación) y resolver conflictos de forma colaborativa (e.g., negociación y toma distribuida de decisiones), aspecto, este último, que es objeto de estudio actualmente.

Los sistemas multiagente han demostrado ser adecuados para satisfacer los requisitos de conectividad y descentralización de los activos de fabricación en diferentes entornos industriales, hasta el punto de que el concepto de Agente Industrial se concibe como una implementación del concepto de Componente I4.0 (una particularización del concepto de Sistema Ciberfísico alineado con la arquitectura de referencia RAMI 4.0) mediante agentes. Así, existe una línea de investigación dedicada a la aplicación de la tecnología de agentes en frameworks robóticos.

Este este contexto, este trabajo contribuye a la integración de AGV como Agentes Industriales dentro FlexManSys, una plataforma de gestión de aplicaciones de fabricación desarrollada por el grupo de investigación GCIS de la UPV/EHU. Esta integración consta de la definición de una interfaz, a nivel de operación, en ROS para el AGV; la provisión de mecanismos para el intercambio de datos entre el activo y su agente asociado; y la definición de una interfaz, a nivel de servicio, en el agente para la gestión virtual del AGV (concepto de Asset Administration Shell de RAMI 4.0). Además, como parte de las pruebas de concepto, se describe la puesta en marcha un sistema de navegación autónomo basado en técnicas SLAM en un robot Turtlebot2.

Palabras Clave: AGV; FMS; MAS; SLAM; ROS; JADE; FoF; LiDAR; MES; Fábrica Inteligente; Navegación Autónoma; Robotic Cloud; Robotic Framework; Fog.

Laburpena:

Ohiko fabrikazio sistemak ekoizpen eredu malguago baterako trantsizioan murgilduta daude, merkatuaren egungo eskaerei aurre egiteko gai dena. Ekoizpen eredu berri horretan, garraio ibilgailu autonomoek (AGV) funtsezko zeregina dute industrian erabateko malgutasuna lortzeko funtsezko diren bi ezaugarri lortzeko: konektibitatea eta deszentralizazioa.

Garraio autonomoaren arloko ikerketa, batez ere, algoritmoen diseinuan eta osagaiak integratzeko ingeniaritzako tresnen eta liburutegien garapenean oinarritu da. Hala ere, Fabrikazio Sistema Aurreratuek bereizten dituzten ingurune banatu eta aldakorretan, AGVen eta beste fabrikazio aktibo batzuren artean sortzen diren elkarrekintza-premiei (konektibitatea) ez zaie arreta handirik jarri. Alde horretatik, literaturaren berrikuspenak aktibo robotikoen eta ez-robotikoen arteko konektibitate-baldintza horiek framework robotiko nagusien aldetik euskarririk ez dutela erakusten du. Arazo horri, fabrikazio aktiboei adimen banatua (deszentralizazioa) emateko beharra gehitu behar zaio, zereginak modu autonomoan egin ahal izateko (adibidez, nabigazioa) eta gatazkak elkarlanean konpondu ahal izateko (adibidez, negoziazioa eta erabakiak modu banatuan hartzea). Azken alderdi hori gaur egun aztergai da.

Agente anitzeko sistemek fabrikazio aktiboen konektibitate eta deszentralizazio baldintzak hainbat industria ingurunetan betetzeko egokiak direla erakutsi dute; izan ere, Industria Agentearen kontzeptua I4.0 Osagaia kontzeptuaren (RAMI 4.0 erreferentziazko arkitekturarekin lerrokatuta dagoen Sistema Ziberfisikoaren kontzeptua, alegia) agente bidezko inplementazio gisa ulertzen da. Horrela, framework robotikoetan agenteen teknologia aplikatzeari buruzko ikerketa lerro bat dago.

Testuinguru honetan, lan honek AGVak industria agente gisa FlexManSys-en (UPV/EHUko GCIS ikerketa taldeak fabrikazio-aplikazioak kudeatzeko garatu duen plataforma) integratu daitezala laguntzen du. Integrazio honek honako alderdi hauek biltzen ditu: AGVaren eragiketa mailako interfaze baten definizioa ROSen; aktiboaren eta hari lotutako agentearen artean datuak trukatzeko mekanismoen hornidura; eta agentean zerbitzu mailako interfaze baten definizioa AGVaren kudeaketa birtualerako (RAMI 4.0ren Asset Administration Shell kontzeptua). Gainera, kontzeptu-proben barruan, TurtleBot2 robot batean SLAM tekniketan oinarritutako nabigazio sistema autonomoaren martxan jartzea deskribatzen da.

Gako-Hitzak: *AGV; FMS; MAS; SLAM; ROS; JADE; FoF; LiDAR; MES; Fabrikazio Adimendun; Nabigazio Autonomoa; Robotic Cloud; Robotic Framework; Fog.*

Abstract:

Conventional manufacturing systems are transitioning towards a more flexible production, capable of heading-on current market demands. Within this new model of production and manufacturing, Automated Guided Vehicles (AGV) play a crucial role in giving actual industry two fundamental characteristics in flexible manufacturing systems: Connectivity and Decentralization.

Research in the autonomous transport subject has focused mainly on the designing of algorithms and development of tools and software libraries for the integration of multiple components. Nevertheless, researchers have omitted the interaction necessities (connectivity) that come up among AGVs and other fabrication assets in distributed and volatile workspaces such as Advanced Fabrication Systems or Flexible Manufacturing Systems (FMS). On this matter, the state of the art reflects that those requirements of connectivity between robotic and non-robotic assets lack support from principal robotic frameworks.

Multi-agent systems have proved to be suitable for the requirements of connectivity and decentralization of the manufacturing assets in different industrial environments. Hence, the concept of "Industrial Agent" is known as an implementation of the concept of "I4.0 Component" (a particularization of a Cypher-physical System in accordance with the RAMI 4.0 reference architecture.) among software agents. Thus, there is a line of research focused on the development of agent technologies in robotic frameworks.

All in all, this project contributes to the integration of AGVs as Industrial Agents within FlexManSys, a management platform of manufacturing applications developed by the research group GCIS from the UPV/EHU. This integration is formed by the definition of an interface, at an operative level, carried out with ROS for the AGV; the designing of communication protocols and mechanisms for the exchange of data and information between the asset and its associated agent; and the definition of an interface, in a service level, within the agent for the virtual management of the AGV (concept of Asset Administration Shell from RAMI 4.0 architecture). Besides, among the concept validation checks, the launch and start-up of an autonomous navigation system based on SLAM techniques in a Turtlebot2 robot are described.

Key Words: *AGV; FMS; MAS; SLAM; ROS; JADE; FoF; LiDAR; MES; Smart Factory; Autonomous Navigation; Robotic Cloud; Robotic Framework; Fog.*

ÍNDICE DE CONTENIDOS

A. ÍNDICE DE CONTENIDOS

1 INTRODUCCIÓN 2
1.1 Integración de AGV como Agentes Industriales 3

2 OBJETIVOS Y ALCANCE 6
2.1 Objetivos 6
2.2 Alcance 7
2.3 Beneficios 8
2.4 Requisitos Funcionales 9
2.5 Requisitos no Funcionales 10

3 ESTADO DEL ARTE 14
3.1 Vehículos Autónomos de Transporte en Sistemas de Fabricación Flexible 14
3.1.1 Sistemas de Fabricación Flexible 14
3.1.2 Framework Robóticos 17
3.1.3 Limitaciones Robóticas de Hardware 18
3.1.4 Sistemas Multi-Agente 19
3.1.5 Sistemas de Navegación 21
3.2 Antecedentes 24
3.2.1 Puesta en Marcha del Control de Robots de Transporte 24
3.2.2 Robots Colaborativos en Procesos de Fabricación Flexibles: Integración ROS-JADE 25
3.2.3 Autonomous Ground Vehicle Management for Flexible Manufacturing Systems 27
3.3 Motivación 29

4 ANÁLISIS DE LAS HERRAMIENTAS EMPLEADAS 32
4.1 Herramientas Software 32
4.1.1 Análisis de Framework Robóticos 32
4.1.2 Framework ROS 34
4.1.3 Plataforma JADE 67
4.1.4 Sistema Operativo 69
4.1.5 IDE 69
4.2 Herramientas Hardware 70
4.2.1 Kobuki iCleBot Turtlebot2 70
4.2.2 RPLiDAR A2 72
4.2.3 Cámara Kinect XBOX 360 74
4.2.4 Odroid XU4 76
4.2.5 Router TL-WR802N 77
4.2.6 Estación AirPort Extreme A1354 78
4.2.7 Brazo Robótico WidowX 79
4.2.8 Concentrador USB HUB UH720 82

4.2.9 Convertidor Buck LM-2596 82
4.2.10 MacBook Air A1369 84

5 DATOS DE PARTIDA 86
5.1 Entorno de Operación 86
5.2 Material Hardware 90
5.3 Material Software 90
5.4 Comparativa Unidades Robóticas 91

6 DESARROLLO DE LA SOLUCIÓN 94
6.1 Estructura General de la Solución 94
6.2 Distribución de las Unidades de Transporte 96
6.3 Distribución y Configuración de Redes 99
6.4 Sistema de Navegación 102
6.4.1 Estructura General del Sistema de Navegación 102
6.4.2 Estructura Software del Sistema de Navegación 104
6.4.3 Scripts del Sistema de Navegación 108
6.4.4 Máquina de Estados del Sistema de Navegación 109
6.4.5 Sistema de Detección de Obstáculos 119
6.4.6 Sistema de Percepción de Imagen 119
6.5 Sistema Gateway 121
6.6 Sistema de Agentes de Transporte 126
6.6.1 Funcionamiento General 127
6.6.2 Lógica de Negociación 131
6.7 Distribución y Operación en Planta 134
6.7.1 Operación en Planta 134
6.7.2 Comandos Definidos en el Sistema 138

7 RESULTADOS Y ANÁLISIS 142
7.1 Navegación 142
7.1.1 Prueba 1: Creación de Mapas del Entorno 142
7.1.2 Prueba 2: Navegación Simple 144
7.1.3 Prueba 3: Navegación con Obstáculos y Percepción del Entorno 148
7.2 Negociación 152
7.2.1 Prueba 4: Integración del Sistema de Navegación en el MAS 152
7.2.2 Prueba 5: Negociación entre Agentes de Transporte 156
7.3 Prueba de Concepto General 157
7.3.1 Prueba 6: Navegación Simple a Petición de Agente Máquina 158
7.4 Sistemas de Seguridad 161
7.4.1 Prueba 7: Parada de Emergencia 161

8 PLANIFICACIÓN Y PRESUPUESTO 164

8.1 Línea Temporal 164

8.2 Presupuesto 173

8.2.1 Cuadro de Precios Unitarios 173

8.2.2 Cuadro de Precios Totales 176

9 CONCLUSIONES Y TRABAJOS FUTUROS 178

9.1 Mejoras y Trabajos Futuros 178

10 REFERENCIAS BIBLIOGRÁFICAS 182

11 ANEXO I: GLOSARIO 188

12 ANEXO II: MANUAL DE USO DEL SISTEMA DE NAVEGACIÓN 192

12.1 Puesta en Marcha del Sistema de Navegación 192

12.2 Puesta en Marcha del Sistema de Agentes de Transporte y Sistema Gateway 198

12.3 Puesta en Marcha General 201

13 ANEXO II: ESQUEMAS Y GRÁFICOS 204

ÍNDICE DE FIGURAS

B. ÍNDICE DE FIGURAS

Figura 3.1 Planta Basada en un Sistema de Fabricación Convencional 15
Figura 3.2 Planta Basada en un Sistema de Fabricación Flexible 16
Figura 3.3 Planta Basada en un Sistema de Fabricación Flexible con Presencia de AGV 16
Figura 3.4 Entorno Generado por ROS 17
Figura 3.5 Distribución del Consumo de Batería en AGV 18
Figura 3.6 Ejemplo de MAS 19
Figura 3.7 Ejemplo de Interoperabilidad de un MAS 20
Figura 3.8 Ejemplo de Cloud Computing y Fog Computing 20
Figura 3.9 Ejemplo de AGV Equipado con Sensor LiDAR 22
Figura 3.10 Planta Propuesta por el Primer Antecedente 25
Figura 3.11 Intefaz Desarrollada por el Primer Antecedente 25
Figura 3.12 Arquitectura Multi-capa de Integración JADE-ROS Propuesta por el Antecedente Segundo 26
Figura 3.13 Arquitectura de Integración ROS-JADE Propuesta por el Tercer Antecedente 28
Figura 3.14 Esquema Ejemplo de Solución a Desarrollar 29
Figura 4.1 Detalle de Entorno de Trabajo Generado por ROS entre Distintos Elementos 33
Figura 4.2 Ejemplo de Estructura Básica de un Entorno ROS 35
Figura 4.3 Ejemplo de Distribución de los Elementos Básicos de ROS 38
Figura 4.4 Estructura de una Comunicación Mediante Tópicos 39
Figura 4.5 Estructura de una Comunicación por Servicio 39
Figura 4.6 Estructura de una Comunicación por Acción 40
Figura 4.7 Comunicación Nodos Mediante Mensaje Tipo std_msgs/String 40
Figura 4.8 Comunicación Nodos Mediante Servicio Tipo std_srvs/SetBool 41
Figura 4.9 Comunicación Nodos Mediante Acción tipo action/Move() 42
Figura 4.10 Distribución de varios Entornos de Trabajo dentro de una misma Red ROS 43
Figura 4.11 Distribución de un Entorno de Trabajo ROS 44
Figura 4.12 Distribución de un Paquete ROS 45
Figura 4.13 Ejemplo de Árbol de Transformadas en AGV 46
Figura 4.14 Posicón relativa de camera_link respecto a base_link 48
Figura 4.15 Relación base_footprint y odom inicial 49
Figura 4.16 Relación base_footprint y odom tras un desplazamiento 49
Figura 4.17 Distribución general de los nodos del ROS Navigation Stack 50
Figura 4.18 Mapa Convencional Generado por Gmapping 52
Figura 4.19 Diagrama de bloques de la técnica SLAM de Gmapping 53
Figura 4.20 Mapa de Costes Generado por Gmapping 54
Figura 4.21 Archivos generales relativos a Gmapping 55
Figura 4.22 AMCL descalibrado (izquierda) y AMCL Calibrado (derecha) 56
Figura 4.23 Ejemplo de Visualización de Trayectoria de Desplazamiento 58
Figura 4.24 Ejemplo de Integración de Nodo en ROS y MAS 59
Figura 4.25 Estructura de un Entorno ROSJava 60
Figura 4.26 Interfaz de Usuario de Gazebo 61
Figura 4.27 Mapa del Entorno de Gazebo (Izquierda) Entorno de Gazebo (Derecha) 62
Figura 4.28 Ejemplo de parte de la definición de un Sensor LiDAR Hokuyo en Gazebo mediante Xacro 62
Figura 4.29 Distribución de los archivos y paquetes ROS empleados en Gazebo 63
Figura 4.30 Interfaz de usuario de RViz 64
Figura 4.31 Comparativa Entorno en Gazebo (Izquierda) y Gemelo Visualizado en RViz (Derecha) 65
Figura 4.32 Comparativa Entorno Real (izquierda) y Gemelo Visualizado en RViz (Derecha) 65
Figura 4.33 Logo Oficial de ROS Kinetic Kame 66
Figura 4.34 GUI de la Plataforma JADE 67
Figura 4.35 Sniffer de la GUI de la Plataforma JADE 68
Figura 4.36 Logo de Ubuntu 16.04 69
Figura 4.37 Icono de Acceso PyCharm (Izquierda) e InteliJ Idea (Derecha) 69
Figura 4.38 Kobuki iCleBot Turtlebot2 70

Figura 4.39 Interfaz Hardware del Kobuki iCleBot Turtlebot2 71
Figura 4.40 Diseño Estructural Turtlebot2 72
Figura 4.41 Sensor RPLiDAR A2 73
Figura 4.42 Detalle del Rango de Lectura de un Sensor LiDAR 2D vs 3D 73
Figura 4.43 Cámara Kinect XBOX 360 74
Figura 4.44 Distribución de los Elementos de la Cámara Kinect XBOX 360 75
Figura 4.45 Odroid XU4 76
Figura 4.46 Router TL-WR802N 77
Figura 4.47 Estación AirPort Extreme A1354 78
Figura 4.48 Brazo Robótico WidowX 79
Figura 4.49 Comunicación ROS con Arbotix y Brazo Robótico 80
Figura 4.50 Espacio de Trabajo Singularidades 80
Figura 4.51 Espacio de Trabajo Planta 81
Figura 4.52 Hub UH720 82
Figura 4.53 Módulo Convertidor Buck LM2596 83
Figura 4.54 MacBook Air A1369 84
Figura 5.1 Plano Simplificado del Entorno de Operación de las Unidades de Transporte 87
Figura 5.2 Secciones C y K 88
Figura 5.3 Secciones A (Izquierda), B (Centro) y D (Derecha) 88
Figura 5.4 Secciones X y U (Izquierda) y E (Derecha) 89
Figura 5.5 Células Robóticas UR (Izquierda) y KUKA (Derecha) 89
Figura 6.1 Esquema de Comunicación de los Sistemas de la Solución 94
Figura 6.2 Esquema de Comunicación de los Sistemas a través de Tópicos 95
Figura 6.3 Esquema de Conexiones de Elementos de un AGV Completo Vista Trasera 97
Figura 6.4 Distribución de Periféricos en AGV Completo 98
Figura 6.5 Esquema de Comunicación MAS-AGV 99
Figura 6.6 Distribución y Configuración General de los Elementos en la Red 101
Figura 6.7 Estructura General Sistema Navegación 102
Figura 6.8 Estructura General Software del Sistema de Navegación 103
Figura 6.9 Estructura de la Comunicación ROS-JADE 107
Figura 6.10 Diagrama de la Máquina de Estados del Sistema de Navegación 109
Figura 6.11 Diagrama de Flujo del Estado Idle 110
Figura 6.12 Diagrama de Flujo del Estado Localization 111
Figura 6.13 Diagrama de Flujo del Estado Active 112
Figura 6.14 Diagrama de Flujo del Estado Operative 113
Figura 6.15 Diagrama de Flujo del Estado Stop 114
Figura 6.16 Diagrama de Flujo del Estado Error 115
Figura 6.17 Diagrama de Flujo del Estado Recovery 116
Figura 6.18 Indicadores de Estado LED 117
Figura 6.19 Áreas de Detección de Obstáculos 119
Figura 6.20 Áreas de Detección de Obstáculos con Cámara 120
Figura 6.21 Estructura General Software Sistema Gateway 121
Figura 6.22 Esquema de Funcionamiento en Ejecución del Sistema Gateway 123
Figura 6.23 Ejemplo Escritura de Tareas desde Agente Transporte 125
Figura 6.24 Esquema Simplificado de la Estructura Formada por los Agentes Transporte y los AGV 126
Figura 6.25 Esquema de Herencias de Clases Relativas al Agente Transporte 127
Figura 6.26 Esquema de Funcionamiento del Agente Transporte 129
Figura 6.27 Curva de Descarga de la Batería del Turtlebot2 132
Figura 6.28 Etapa Inicial 134
Figura 6.29 Etapa de Calibración 135
Figura 6.30 Etapa Tercera 136
Figura 6.31 Etapa Cuarta 137
Figura 6.32 Etapa Quinta 137
Figura 6.33 Etapa Sexta 138
Figura 7.1 Mapeado durante la Prueba 1 143
Figura 7.2 Mapa Obtenido de la Prueba 1 144

Figura 7.3 Transporte en Estación de Carga en la Prueba 2 145
Figura 7.4 Desplazamiento del Transporte desde la Estación de Carga hasta el Punto A en la Prueba 2 145
Figura 7.5 Desplazamiento del Transporte desde el Punto A hasta el punto B en la Prueba 2 146
Figura 7.6 Transporte en Punto A (Izquierda). Transporte en Punto B (Derecha) 146
Figura 7.7 Comparativa Realidad con RViz: Trayectoria desde la Estación de Carga al Punto A 147
Figura 7.8Comparativa Realidad con RViz: Trayectoria desde el Punto A al Punto B 147
Figura 7.9 Percepción de Código AR en la Prueba 3 149
Figura 7.10 Obstáculos Trayectoria Dock al Punto A (Izquierda) y del Punto A al Punto B (Derecha) .. 149
Figura 7.11 Trayectoria desde la Estación de Carga al Punto A con Obstáculos 150
Figura 7.12 Trayectoria desde el Punto A al Punto B con Obstáculos 150
Figura 7.13 Fin de Trayectoria desde el Punto A al Punto B con Obstáculos 151
Figura 7.14 Interfaz Mostrada por el Sistema de Navegación 151
Figura 7.15 Mensaje de Estado de Unidad de Transporte Replicado 153
Figura 7.16 Interfaz Mostrada por el Agente Transporte 153
Figura 7.17 Datos Publicados en el Tópico de Coordenadas 154
Figura 7.18 Datos Recibidos por el Agente Gateway tanto desde el MAS como desde el Sistema de Navegación 154
Figura 7.19 GUI de JADE con el Sistema de Agentes al Completo 155
Figura 7.20 GUI de JADE con el Sniffer de Mensajes entre el Agente Gateway y el Agente Transporte 155
Figura 7.21 Mensaje de Solicitud de Negociación 157
Figura 7.22 Mensajes de la Negociación del Agente Transporte 157
Figura 7.23 Equipos Empleados para la Prueba 159
Figura 7.24 Célula Virtual Robot KUKA para Agente Máquina 159
Figura 7.25 Interacción Transporte con Agente Máquina y Gemelo Virtual 160
Figura 7.26 Ancho de Banda Consumido por el Sistema de Navegación 160
Figura 7.27 Unidad de Transporte en Estado de Error o Parada de Emergencia 162
Figura 7.28 Mensaje de Terminal tras la Solicitud de Parada de Emergencia 162
Figura 8.1 Planificación Etapas 1-3 de la Fase de Viabilidad 169
Figura 8.2 Planificación Etapas 4-6 de la Fase de Diseño 170
Figura 8.3 Planificación Etapa 7 de la Fase de Desarrollo 171
Figura 8.4 Planificación Etapas 8-10 de la Fase de Validación 172
Figura 12.1 Cofiguración RViz Parte 1 194
Figura 12.2 Configuración RViz Parte 2 195
Figura 12.3 Información Mostrada por Sistema navegación 196
Figura 12.4 Información Mostrada por Tópico de Estado 196
Figura 12.5 Configuración RViz para la Visualización de Cámara Kinect 197
Figura 12.6 Mensaje de Estado Autocompletado 199
Figura 13.1 Herencias y Extensiones de los Distintos Tipos de Agentes Recuso en el MAS 205
Figura 13.2 Distribución Nodos del Sistema de Navegación según ROSGraph 206
Figura 13.3 Gráfico UML de la Interacción Agente Transporte e Interfaz del Sistema de Navegación .. 207

ÍNDICE DE TABLAS

C. ÍNDICE DE TABLAS

Tabla 4.1 Características de los Principales RSF 34
Tabla 4.2 Campos del Tipo de Mensaje del Nodo static_transform_publisher/Tf() 47
Tabla 4.3 Transformada de camera_link respecto a base_link 47
Tabla 4.4 Comparativa entre las Distros de ROS 66
Tabla 4.5 Especificaciones Funcionales Turtlebot2 70
Tabla 4.6 Especificaciones RPLiDAR A2 74
Tabla 4.7 Especificaciones Cámara Kinect XBOX 360 75
Tabla 4.8 Especificaciones de la Odroid XU4 76
Tabla 4.9 Especificaciones Router TL-WR802N 77
Tabla 4.10 Especificaciones Estación AirPort Extreme A1354 78
Tabla 4.11 Especificaciones Brazo Robótico WidowX 81
Tabla 4.12 Especificaciones del Hub UH720 82
Tabla 4.13 Especificaciones Módulo Convertidor Buck LM2596 83
Tabla 4.14 Especificaciones MacBook Air 84
Tabla 5.1 Material Hardware Inicial 90
Tabla 5.2 Comparativa Turtlebot2 con Turtlebot3 91
Tabla 6.1 Equipamiento Unidades de Transporte 96
Tabla 6.2 Direcciones IP Transportes 100
Tabla 6.3 Direcciones IP Sistema de Navegación 100
Tabla 6.4Campos de turtlebot_transport_flexmansys/TransportUnitState 118
Tabla 6.5 Contenido de las Variables de TransportUnitState 118
Tabla 6.6 Estructura Mensaje Solicitante del Servicio 131
Tabla 6.7 Regiones Linealizadas de la Curva de Descarga 133
Tabla 6.8 Contenido Mensaje Servicio Transporte a Agente Transporte de AGV T_01 136
Tabla 6.9 Comandos Definidos en el Sistema 139
Tabla 7.1 Tabla de Verificación de Prueba 1 142
Tabla 7.2 Tabla de Verificación de Prueba 2 144
Tabla 7.3 Tabla de Verificación de Prueba 3 148
Tabla 7.4 Tabla de Verificación de Prueba 4 152
Tabla 7.5 Tabla de Verificación de Prueba 5 156
Tabla 7.6 Tabla de Verificación de Prueba 6 158
Tabla 7.7 Tabla de Verificación de Prueba 7 161
Tabla 8.1 Presupuesto de Componentes y Consumibles 174
Tabla 8.2 Presupuesto de Amortizaciones 175
Tabla 8.3 Presupuesto de Recursos Humanos 175
Tabla 8.4 Resumen de Precios Totales 176

CAPITULO 1

INTRODUCCIÓN

1 Introducción

La globalización de la industria, el aumento de la oferta, la alta competitividad del mercado, la especialización del cliente y las nuevas aplicaciones de la tecnología, conforman una serie de desafíos que están cambiando significativamente la forma en que operan las empresas manufactureras. La necesidad de hacer frente a estos desafíos está transformando la manera en que se diseñan y despliegan los sistemas de producción. En este sentido, se necesitan sistemas de producción inteligentes y personalizables, capaces de asegurar una gestión eficiente de los recursos de fabricación, y de adaptar su producción a la demanda de forma dinámica, respondiendo a cualquier cambio derivado de incidencias en planta o modificaciones en el plan de fabricación.

El paradigma de la Industria 4.0 propone el uso de tecnologías como el big data, los gemelos digitales, los Sistemas Ciber-Físicos (Cyber-Physical Systems, CPS) o el Internet de las Cosas para permitir una fabricación inteligente y descentralizada. En este sentido, se han establecido varias iniciativas y programas de apoyo a la Industria 4.0 en diferentes países: Industrie 4.0 en Alemania, Industrial Internet en EE.UU., Made in China 2025 en China e Industrial Value Chain en Japón. Su denominador común es un esfuerzo coordinado entre el gobierno, la industria y academia para dar soporte a la innovación en los procesos de fabricación basados en la interconexión total entre los diferentes activos de fabricación (físicos o lógicos) en un contexto tecnológico de captura y procesamiento de grandes volúmenes de datos. En este sentido, todas las iniciativas están trabajando en la estandarización de sus arquitecturas de referencia (RAMI 4.0 en el caso de Industrie 4.0, IIRA en el caso de Industrial Ethernet e IMSA en el caso de Made in China 2025) y en el análisis de su confluencia.

Sin embargo, la estandarización y la correlación de las arquitecturas de referencia suele ser un proceso lento. Mientras se avanza en esta dirección, es importante dotar a la industria de elementos de apoyo que le permitan ir avanzando en la transformación digital. El concepto de fábrica inteligente puede cimentarse sobre una red de CPS en la que la comunicación tiene lugar tanto horizontal como verticalmente. La abstracción o virtualización de los activos de fabricación en cuanto a sus funcionalidades, proporcionadas como servicios de red, es una condición necesaria para materializar el concepto de CPS. Éste es el enfoque seguido por RAMI 4.0, que define el concepto de Componente I4.0 como un participante de un Sistema I4.0 que comprende un activo y su interfaz de gestión virtual (AAS, Asset Administration Shell). A su vez, el AAS se define como una representación virtual del Componente I4.0 en el sistema, representación que contiene submodelos de información a los que se puede acceder a través de un Gestor de Componentes.

Los Componentes I4.0 deben cumplir ciertos requisitos, a saber: proporcionar capacidades de comunicación para garantizar la interoperabilidad entre los Componentes I4.0 del sistema (R1); identificar de forma exclusiva tanto el AAS como el activo (R2); representar el estado y los servicios del activo en submodelos (R3); gestionar el acceso de otros componentes a los submodelos del Componente I4.0 (R4); integrar los AAS y los activos (R5); y, por último, gestionar la automatización y ejecución de los servicios realizados por el activo (R6). Sin embargo, a pesar de los numerosos trabajos en torno al concepto de Componente I4.0, todavía no existe una implementación de referencia para el AAS. Dos son las posibles causas atribuibles a este problema: por una parte, la falta de consolidación de soluciones que faciliten la integración de un AAS con su correspondiente activo de fabricación; por otra parte, la falta de una

infraestructura de apoyo que asegure el flujo de información entre Componentes i4.0. Este trabajo se centra en la problemática de la integración AAS-activo.

Los agentes software son entidades autónomas capaces de competir o colaborar con otros agentes del sistema para alcanzar sus objetivos. Los agentes representan una alternativa adecuada para la implementación del AAS, ya que cumplen con los requisitos de *interoperabilidad* (R1), *identificación* (R2), *representación* (R3) y *gestión de submodelos* (R4), y proporcionan características adicionales, como la capacidad de negociación y toma de decisiones distribuida. Sin embargo, la tecnología de agentes no contempla los requisitos de *integración* (R5) y *gestión de activos* (R6). El paradigma de agentes industriales surge para hacer frente a esta limitación, proponiendo marcos, como la norma IEEE 2660.1-2020 (sobre prácticas recomendadas para agentes industriales), para la integración de un agente con un activo físico. No obstante, la escalabilidad y reutilización de soluciones de basadas en agentes industriales siguen constituyendo un reto debido a diversidad de activos físicos presentes en una fábrica (en cuanto a la heterogeneidad de los activos y sus interfaces de comunicación) y la ubicación de los agentes.

En este contexto, este trabajo parte de los resultados previos del grupo de investigación "Control e Integración de Sistemas" GCIS del Dpto. de Ingeniería de Sistemas y Automática, UPV/EHU, para la implementación de agentes industriales mediante una arquitectura de cuatro capas compatible con la norma IEEE 2660.1-2020: una capa superior o de inteligencia, que aglutina las capacidades de comunicación e interoperabilidad, además de otras capacidades asociadas a los agentes, como la negociación y la toma de decisiones; una capa intermedia o de gestión, que transforma los datos recibidos del activo en información útil para los Componentes I4.0 en el sistema; y dos capas inferiores, operativa y funcional, que gestionan la llamada, la ejecución y el estado de los servicios del activo [1] .

1.1 Integración de AGV como Agentes Industriales

Dentro de la gran diversidad de activos físicos presentes en una fábrica, los Vehículos Autónomos de Transporte (AGV) juegan un papel fundamental para dotar a la industria de dos características clave para lograr la plena flexibilidad: *conectividad* y *descentralización*.

La investigación en el área del transporte autónomo se ha centrado, principalmente, en el diseño de algoritmos y en el desarrollo de herramientas de ingeniería y librerías para la integración de componentes. Sin embargo, no se ha prestado tanta atención a las necesidades de interacción (*conectividad*) que surgen entre los AGV y otros activos de fabricación en entornos distribuidos y cambiantes como los que presentan los Sistemas de Fabricación Flexible (FMS). En este sentido, la revisión de la literatura refleja que dichos requisitos de conectividad entre activos robóticos y no robóticos carecen de soporte por parte de los principales frameworks robóticos. A esta problemática, se le suma la necesidad de dotar de inteligencia distribuida (*descentralización*) a los activos de fabricación para que puedan realizar tareas de forma autónoma (e.g., la navegación) y resolver conflictos de forma colaborativa (e.g., negociación y toma distribuida de decisiones), aspecto, este último, que es objeto de estudio actualmente.

Los resultados previos de GCIS han demostrado que los agentes pueden ser adecuados para satisfacer los requisitos de conectividad y descentralización de los activos de fabricación en diferentes entornos industriales. Así, existe una línea de investigación dedicada a la aplicación de la tecnología de agentes en frameworks robóticos. Concretamente, en [2] se propuso un enfoque inicial de arquitectura de cuatro capas para la integración de agentes JADE y AGV gestionados con el framework robótico ROS. Sin embargo, este enfoque inicial adolece de ciertas limitaciones

que no permiten desacoplar la interfaz de comunicación definida en el activo del agente, lo cual impide, a su vez, abstraer la funcionalidad de gestión de agente (capa de inteligencia) del activo.

Bajo este contexto, este trabajo contribuye a la integración de un AGV como Agente Industrial dentro FlexManSys, una plataforma de gestión de aplicaciones de fabricación desarrollada por GCIS y basada en un sistema multi agente o MAS (Multi-Agent System). Esta integración consta de la configuración y programación del sistema de navegación del AGV (capa funcional); la definición de una interfaz, a nivel de servicio, mediante tópicos ROS para la estación (capa operativa); la provisión de mecanismos para el intercambio de datos entre el activo y su agente asociado (capa de gestión); y la definición de una interfaz, a nivel de sistema, para la gestión virtual de la estación (capa de inteligencia). Las pruebas de concepto se realizan para haciendo uso de un AGV modelo Turtlebot 2 de amplio uso en entornos académicos.

CAPITULO 2

OBJETIVOS Y ALCANCE

2 Objetivos y Alcance

En el presente capítulo se establecerán los principales objetivos y alcance del presente trabajo, así como los beneficios que aportará el desarrollo de este al cliente. Finalmente, se enumerarán los diferentes requisitos y especificaciones establecidas por y para el desarrollo de la solución.

2.1 Objetivos

El objetivo principal de este proyecto es el diseño e implementación de una solución software capaz de integrar unidades autónomas de transporte o AGV que empleen el middleware ROS dentro de una plataforma de gestión de aplicaciones de fabricación basada en un sistema multi agente o MAS previamente desarrollada haciendo uso de la arquitectura de capas definida en [1]. De tal modo, se identifican los siguientes tres objetivos principales:

- **Desarrollo de un Sistema de Navegación Autónomo (P1).**
    - Se refiere al diseño de una solución que permita navegar de manera autónoma y sin la necesidad de intervención humana, a una unidad de transporte desde un punto inicial hasta un punto requerido por un agente o persona usuaria. En el marco de la integración del AGV en FlexManSys, este objetivo se corresponde con la configuración y programación del sistema de navegación del AGV (capa funcional) y la definición de una interfaz, a nivel de servicio, mediante tópicos ROS para la estación (capa operativa). Para la realización de las pruebas de concepto se tomará como referencia un AGV modelo Turtlebot 2 de amplio uso en entornos académicos.
- **Integración del Sistema de Navegación en el MAS (P2).**
    - Se establece el desarrollo y diseño de una serie de agentes del tipo transporte que se encarguen de gestionar y representar a las unidades de transporte dentro del MAS siguiendo la estructura de clases previamente establecida. En el marco de la integración del AGV en FlexManSys, esta integración consta de la definición de una interfaz, a nivel de sistema, para la gestión virtual de la estación (capa de inteligencia).
- **Integración de la Comunicación entre Ambos Sistemas (P3).**
    - Se requiere del desarrollo de un sistema traductor o Gateway que permita la conexión entre el Sistema de Navegación y el MAS, de manera que pueda llevarse a cabo una comunicación bidireccional a modo de cliente y servidor. En el marco de la integración del AGV en FlexManSys, este objetivo se corresponde con la provisión de mecanismos para el intercambio de datos entre el activo y su agente asociado (capa de gestión).

Del mismo modo, existen una serie de objetivos secundarios que han de considerarse a la hora de desarrollar el presente proyecto y que deben de cumplirse para considerarse satisfactoria la solución propuesta:

- **Detección de Obstáculos y Percepción del Entorno (S1).**

  - Se establece que el Sistema de Navegación ha de ser capaz de detectar obstáculos en su entorno, además de modificar su trayectoria desde el punto inicial hasta el requerido de manera dinámica en plena operación y según lo que perciba del entorno.

- **Desarrollo de un Sistema Funcional y Escalable (S2).**
  - La solución propuesta ha de ser no solo lo suficientemente cerrada y fiable como para realizar todas las tareas necesarias, sino que deberá de permitir la posibilidad de ser aplicable para diferentes sistemas robóticos y migrable a otras plataformas.

- **Desarrollo de una Documentación y Material Suficiente (S3).**
  - Dado que se pretende que el presente trabajo sea mejorado y adaptado para necesidades futuras, no puede darse la posibilidad de que el conocimiento adquirido o "Know-How" pueda perderse en dicha transición. Es por ello, que la documentación y material relativo al presente proyecto debe de ser suficiente como para que una persona no familiarizada con el entorno encuentre todos los aspectos necesarios para replicar el trabajo a partir de ella y obtenga una visión generalista del funcionamiento a bajo nivel.

2.2 Alcance

El presente trabajo se desarrolla dentro del grupo de investigación GCIS, concretamente en el marco de la tesis doctoral de Alejandro López García que se desarrolla, a su vez, en el marco del proyecto RTI2018-096116-B-I00 financiado por MCIU/AEI/FEDER, UE. En este sentido, los resultados del presente trabajo forman parte de los resultados esperados de las siguientes tareas del proyecto:

- Dentro del paquete de trabajo "WP3: Distributed Intelligence at Production Level", tarea "T3.3 MAS-RECON customization for Production systems", donde se personalizan las plantillas de agentes de dominio de fabricación (agentes recurso -máquinas y transportes, y agentes de aplicación -lotes, órdenes y planes-) y se definen las interacciones entre ellos y los agentes de la plataforma. En el caso del presente trabajo, su contribución consiste en la definición de la plantilla del agente transporte y sus correspondientes interacciones con el resto de los elementos del sistema.
- Dentro del paquete de trabajo "WP5: Proof of concept Demonstrators", tarea "T5.3 Demonstrator Assessment", donde se desarrolla un prototipo de aplicación industrial y se una prueba de concepto para analizar el cumplimiento de requisitos de la plataforma. En el caso del presente trabajo, su contribución consiste se describe la puesta en marcha e integración de un Turtlebot2 como AGV dentro de FlexManSys.

Por otra parte, hay que tener en cuenta que este trabajo se desarrolla dentro de un marco académico, junto con todas sus limitaciones económicas y temporales. Es por ello, que, algunos de los desarrollos llevados a cabo para el diseño de la solución final podrían no ser suficiente como para integrarse en un ámbito plenamente industrial sin realizarse modificaciones previas, al no haber sometido la solución a los exámenes propios de dichas normativas y estándares internacionales. Estas limitaciones, estarían demarcadas principalmente por el empleo de unidades robóticas no preparadas para entornos industriales muy exigentes en los que se requiera un grado de protección IP muy alto o el uso íntegro de comunicaciones industriales que requieran de tiempo real, entre otras.

2.3 Beneficios

En lo referente a los beneficios que aporta el presente trabajo, se identifican dos tipos: Los beneficios generales, los cuales están relacionados con los objetivos a obtener por proyectos desarrollados en la misma línea de investigación; y los beneficios específicos, que se tratan de aquellas mejoras obtenidas con respecto a los antecedentes recogidos dentro del GCIS. De tal modo, se establecen como beneficios generales los siguientes puntos:

- **Aumento de la Flexibilidad y Adaptabilidad de la Planta (B1)**.
    - Al poder interconectar los distintos procesos productivos y células de una manera no lineal, se abre la posibilidad de modificar la secuencia de producción de un producto cualquiera según las necesidades del cliente y del propio fabricante.
- **Diversificación de la Oferta (B2).**
    - De manera que puedan realizarse mayor número de productos al permitirse un mayor tipo de combinaciones entre procesos productivos y células.
- **Disminución de los Riesgos en Tareas Peligrosas (B3)**.
    - La adaptación a fallos del proceso productivo permite que los robots puedan realizar tareas que podrían llegar a suponer un riesgo para las personas operarias de planta, de manera que estos se ocupen de las tareas más peligrosas.
- **Aumento de la Competitividad de las Empresas (B4)**.
    - De forma que puedan obtener una fabricación más fiable, precisa y personalizada, disminuyendo los tiempos de parada y aumentando su capacidad productiva; beneficio primordial para permitir a las PYMEs el competir con organizaciones mucho mayores.
- **Reducción de los Tiempos Productivos y de Entrega (B5)**.
    - Puesto que la capacidad de flexibilizar la producción según las necesidades del cliente permite contar con una puesta en marcha mucho más corta a la de las empresas convencionales y abre la posibilidad a sustituir las células no operativas o en estado de fallo por otras dentro de la misma planta.

En lo que concierne a los beneficios específicos, es decir, aquellos que influyen directamente sobre el producto desarrollado para el cliente, se recogen los siguientes:

- **Desarrollo de la Integración Completa de un AGV (B6)**.
    - Los antecedentes del presente trabajo nunca llegaron a completar toda la integración necesaria del trabajo, es decir, únicamente se centraron o bien en el desarrollo de un Sistema de Navegación, en el desarrollo de agentes tipo transporte o en la interconexión entre el entorno de ROS y el de JADE. Sin embargo, nunca se llegaron a desarrollar todos los módulos necesarios para la integración completa.
- **Desarrollo de un Sistema de Navegación Flexible y Distribuible (B7).**
    - La solución diseñada se basa en el ROS Navigation Stack, el cual permite no solo la integración de AGV de la gama Kobuki iCleBot Turtlebot2, sino que habilita la integración de múltiples unidades de transporte de diferentes gamas siempre y cuando estas puedan soportar ROS.

- **Desarrollo de una Solución Escalable (B8).**
  - Diseño de una solución que abre la posibilidad de migrar el proyecto desde una distro de ROS y robot ya obsoletos, como es ROS Kinetic Kame y el Turtlebot2, a una distro y robots de última generación como lo son ROS2 y los Turtlebot3 y Turtlebot4. Por otro lado, también permite la posibilidad de integrar sistemas de manipulación y percepción sin la necesidad de adquirir nuevo hardware.
- **Desarrollo de un Sistema de Seguridad (B9).**
  - Desarrollo de un sistema de seguridad que no solo reacciona por debajo de los 100 ms, sino que es independiente al Sistema de Navegación y al MAS imitando el comportamiento de un "watchdog", de manera que siempre estará activo ante cualquier imprevisto surgido o fallo en la ejecución.
- **Reducción del 18.9 % del Coste Total (B10).**
  - En referencia al precio total del presupuesto establecido por antecedentes del presente proyecto.

2.4 Requisitos Funcionales

En cuanto a los requisitos del trabajo, se han seguido principalmente las indicaciones aportadas por el cliente, es decir, el doctorando del GCIS. Por otro lado, también se han seguido los requisitos propios de la normativa FIPA (Foundation for Intelligent Physical Agents) para el desarrollo y diseño de los agentes respectivos al objetivo P2. Finalmente, se han seguido las recomendaciones aportadas por la página oficial de la web de ROS. De tal modo, en relación con los objetivos previamente mencionados, se han recogido los siguientes requisitos:

- **Sistema de Navegación (R1).**
  - Las unidades de transporte empleadas como AGV deberán de ser de la gama Kobuki iClebot Turtlebot2, preferiblemente. (R1.1).
  - El Sistema de Navegación deberá de ser desarrollado en ROS, preferiblemente en la distro o versión de ROS Kinetic Kame, cuyo sistema operativo viene previamente definido por Ubuntu LTS 16.04. (R1.2).
  - El Sistema de Navegación desarrollado debe de ser capaz de trabajar en un entorno cerrado, de manera que pueda trasladar a una unidad de transporte o AGV desde un punto inicial a otro requerido por la persona usuaria. (R1.3).
  - El Sistema de Navegación deberá de integrar las funcionalidades suficientes y necesarias como para llevar a cabo el mapeo de diferentes entornos, la localización de la unidad autónoma de transporte y la ejecución de trayectorias. (R1.4).
  - El Sistema de Navegación desarrollado debe de ser capaz de modificar su trayectoria de manera autónoma en caso de detectar un obstáculo de por medio, por lo que debe de tener integrada la percepción dentro de su controlador de navegación. (R1.5).
  - El Sistema de Navegación autónoma ha de ser capaz de indicar el estado en el que se encuentra la unidad de transporte o AGV en todo momento, de manera que el agente gestor pueda tomar decisiones de seguridad u operativas según los datos que reciba por parte del estado del Sistema de Navegación. (R1.6).

  - El Sistema de Navegación deberá de ser capaz de entrar en una parada de emergencia o estado seguro de manera inmediata siempre que la persona usuaria o agente lo requiera. (R1.7).
  - El Sistema de Navegación debe de ser distribuible y escalable, de manera que varios AGV puedan emplear el mismo sistema sin la necesidad de realizar cambios a bajo nivel, es decir, en la codificación. (R1.8).
  - El Sistema de Navegación deberá de abstraer la máxima funcionalidad posible de las unidades de transporte, de manera que pueda efectuarse concurrencia. (R1.9).
  - El Sistema de Navegación ha de poder integrarse en un sistema-multi agente previamente desarrollado, concretamente en la plataforma JADE. Para ello, han de codificarse los agentes encargados de la gestión del transporte según la estructura propia de la arquitectura desarrollada. (R1.10).

- **Integración en el Sistema Multi-Agente (R2).**
  - El sistema de negociación propio de los agentes encargados de la gestión de las unidades de transporte deberá de implementar un sistema de negociación para decidir el agente ganador, sistema basado tanto en la distancia desde el punto inicial al objetivo como en el nivel de batería restante de la unidad de transporte. (R2.1).
  - La comunicación e intercambio de mensajes entre los agentes dentro del sistema-multi agente debe de realizarse a través de mensajes ACL. Además de respetarse la estructura de datos correspondiente para dicha comunicación, el contenido de dichos mensajes debe de ser siempre en formato String. (R2.2).
- **Integración de la Comunicación entre Ambos Sistemas (R3).**
  - El Gateway desarrollado para integrar la comunicación entre el entorno de ROS y la plataforma JADE ha de ser realizada mediante el paquete de ROSJava. (R3.1)
- **Características Secundarias de la Solución (R4).**
  - La solución desarrollada, en su conjunto, deberá de ser suficientemente escalable como para integrar cambios y futuras mejoras, como lo necesariamente abstracta para que la interacción con ella sea simple y sin requerimiento de un conocimiento amplio de esta. (R4.1).
  - La solución desarrollada, en su conjunto, ha de tener una documentación lo suficientemente amplia y extensa como para comprender todas las decisiones de diseño tomadas, así como integrar un manual de usuario que indique el como ponerla en marcha y pequeñas pautas de actuación ante emergencias. (R4.2).

2.5 Requisitos no Funcionales

En lo relativo a los requisitos no funcionales s a cumplir por la solución desarrollada, se han seguido únicamente las indicaciones y necesidades aportadas por el doctorando responsable de la tesis en la que se sitúa el presente trabajo, recogiéndose los siguientes puntos:

- **Un AGV Funcional (E1).**

  - El Sistema de Navegación desarrollado deberá de integrar al menos a una unidad de transporte plenamente funcional que cumpla con todo el conjunto de requisitos y especificaciones. El número máximo de transportes será cuatro.

- **Tasa de Transmisión Inferior a 300 Mbps (E2).**
  - El Sistema de Navegación en pleno rendimiento, es decir, con todos sus periféricos activos, no deberá de alcanzar los 300 Mbps de transmisión de datos, máximo de banda ancha establecido por el dispositivo rúter dónde se ejecutará el sistema.

- **Tiempo de Respuesta Inferior a 100 ms para Parada de Emergencia (E3).**
  - El Sistema de Navegación ha de reaccionar en un plazo menor a 100 ms a los requerimientos de parada de emergencia.

- **Tiempo de Refresco del Estado Inferior a 4 segundos (E4).**
  - El Sistema de Navegación ha de notificar a los agentes recogidos en el sistema multi-agente en un plazo menor de 4 segundos de su estado de manera periódica. En caso de no cumplirse, se considerará que la unidad de transporte ha dejado de estar operativa.

- **Posibilidad de Requerir hasta 30 Coordenadas de Desplazamiento (E5).**
  - El Sistema de Navegación ha de ser capaz de gestionar varias coordenadas o peticiones de desplazamiento en una sola comunicación, estableciendo en 30 coordenadas en una sola petición como máximo.

- **Autonomía de las Unidades de Transporte no Inferior a las 2 horas (E6).**
  - Las unidades de transporte deben de contar con una autonomía superior a las 2 horas de operación, de manera que la carga computacional que supone la ejecución del Sistema de Navegación no debe de suponer una reducción de dicha autonomía, la cual se establece en 2 horas y media como en condiciones normales y nominales según fabricante.

- **Las Unidades de Transporte no Pueden Superar el 20 % de Energía Mínima (E7).**
  - Las unidades de transporte no deberán de ser capaces de operar o recibir nuevas encomiendas si cuentan con un nivel de batería inferior a los 12.8 VDC o cabe la posibilidad de que lleguen a dicho nivel mientras operan, lo que se traduce en el 20 % de energía o SOC (State of Charge). En dicho caso, deberán de dirigirse a la estación de carga de manera inmediata

CAPITULO 3

ESTADO DEL ARTE

3 Estado del Arte

En el presente capítulo, se desarrollará el estado del arte relacionado con la temática del presente trabajo. Por otro lado, también se hará un breve desglose de los antecedentes y trabajos previos realizados dentro del grupo de investigación en el que se ha desarrollado este trabajo.

3.1 Vehículos Autónomos de Transporte en Sistemas de Fabricación Flexible

En este apartado, se llevará a cabo la contextualización del presente proyecto dentro del marco industrial y académico en el que se sitúa. Para ello, primeramente, se llevará a cabo un breve desarrollo sobre los Sistemas de Fabricación Flexible o FMS (Flexible Manufacturing Systems) por sus siglas en inglés, y la importancia de los AGV (Automated Guided Vehicles) en dichos sistemas. Posteriormente, se expondrá la principal problemática existente de las diferentes herramientas y entornos de trabajo robóticos para la integración de unidades robóticas en dichos sistemas, así como las limitaciones hardware existentes. Tras ello, se hablará de los Sistemas Multi-Agente o MAS, los cuales, se tratan de una alternativa a la integración de dichas herramientas robóticas en Sistemas de Fabricación Flexible. Finalmente, se hablará acerca de los diferentes Sistemas de Navegación existentes en las aplicaciones con AGV.

3.1.1 Sistemas de Fabricación Flexible

En un mundo cada vez más globalizado y digitalizado, la industria se ve obligada a afrontar una demanda cada día más personalizada e individualizada. Bajo este pretexto y la recientemente surgida Industria 4.0, nacen los conceptos de la Smart Factory y la Flexible Factory, los cuales replantean los métodos de fabricación y producción más tradicionales. Este replanteamiento, surge de la necesidad de adaptar las empresas para que sean capaces de hacer frente a este nuevo tipo de demanda. Una necesidad que tal y como apuntan numerosos estudios, las principales competencias que las empresas deben adquirir para afrontar los retos de la nueva revolución industrial son: la denominada producción "on-demand", altos requisitos de calidad, reducción de tiempos de entrega, flexibilidad y adaptabilidad de los medios de producción y rentabilidad en la producción de pequeños lotes [2], [3].

Sin embargo, los medios productivos tradicionales sobre los que se basan la mayoría de las empresas y compañías no cumplen, en general, con dichos requisitos. El ejemplo más simple se muestra en la *Figura 3.1*, donde se puede observar una planta formada por dos cadenas de montaje y cuatro tipos de máquinas o procesos diferentes: Las de tipo A, B, C y D. Al interconectar dichos procesos con cintas de transporte convencionales, podrían generarse dos combinaciones diferentes, o lo que es lo mismo, podrían fabricarse dos productos diferentes. A pesar de ello, esta configuración carece tanto de flexibilidad como de adaptabilidad, puesto que en caso de quedar inoperativa la máquina B de la segunda cadena de montaje, la mitad de la producción de la planta cesaría sin la posibilidad de emplear como sustituta la máquina B de la primera cadena de montaje, al no existir ningún otro medio de transporte de materia entre las células de la planta. Además, la planta no contaría con la posibilidad de explotar todo su potencial de generar combinaciones distintas, siendo extremadamente costoso tanto en dinero como en tiempo el tener que adaptar los medios de transporte para cada pedido que pueda requerirse.

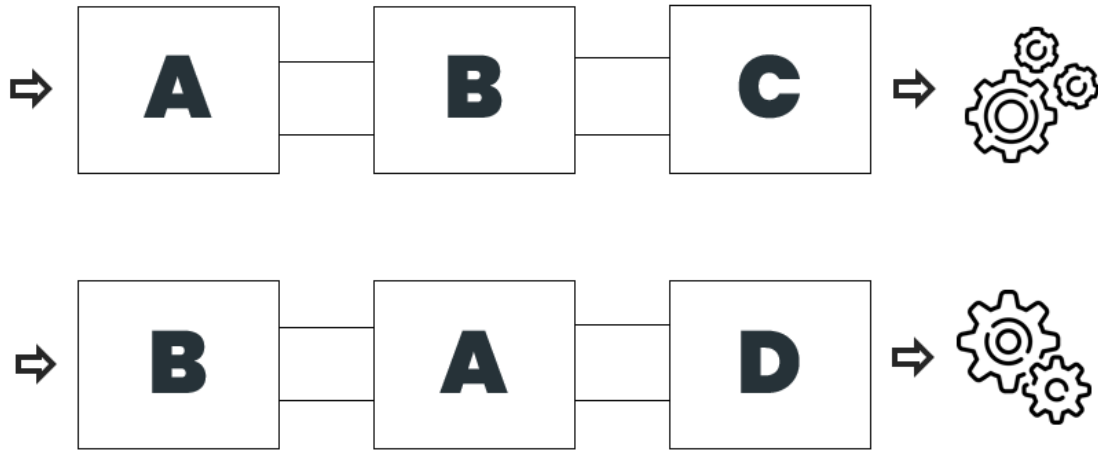


Figura 3.1 Planta Basada en un Sistema de Fabricación Convencional

Siendo esta necesidad de adaptar los medios de producción uno de los mayores retos de la industria actual y uno de los precursores de la Industria 4.0, surge el concepto de la FoF (Factory of the Future), estrechamente ligado al FMS. Ambos entienden a las plantas industriales como un conjunto de células o estaciones de trabajo interconectadas entre ellas de manera no lineal. Es decir, sin seguir estrictamente un método de fabricación y secuencia única, como podría ser el ya conocido FPS (Ford Production System), basado en cadenas de montaje unidas mediante cintas transportadoras. Tanto la FoF, como el FMS, cuentan con dos características principales: la conectividad, que consiste en la habilidad y capacidad de comunicación entre diferentes entidades de fabricación (robots, operadores, máquinas, células de montaje, etc.) en un mismo entorno; y la descentralización, que dota a las diferentes entidades de autonomía en las decisiones y la capacidad de negociar entre ellas para resolver conflictos [3], [4].

Es debido a estas dos características de conectividad y descentralización, que varios autores indican que cada entidad que forme parte de una FoF puede ser representada a través de un CPS (Cyber Physical System), es decir, una serie de elementos distribuidos en red trabajando de manera paralela y cooperativa por un objetivo común. De esta manera, las principales soluciones y desarrollos de industriales para FMS estarían evolucionando hacia soluciones basadas en Sistemas de Control Distribuido. En esta clase de sistemas, el control a bajo nivel y la gestión de cada célula o elemento no residirá única e íntegramente en su controladora local, sino que necesitaría de la cooperación del resto de unidades del sistema para poder lograr su objetivo de control. De esta manera, este concepto de modularización y distribución podría trasladarse hasta el desarrollo de sistemas interconectados de manera no lineal y secuencial, es decir, sin cintas transportadoras [5].

Volviendo al ejemplo de la planta de la *Figura 3.1* y eliminando las cintas transportadoras se genera la planta de la *Figura 3.2* . De esta forma, las seis máquinas o procesos de la planta quedan interconectados con todas las demás células de la planta, sin la necesidad de seguir una secuencia u orden único. Este cambio, dota a la planta de las dos características principales de la fabricación flexible, lo que, a su vez, permite el poder adaptar la secuencia de producción según las necesidades requeridas, pudiendo explotar todo el potencial de la planta y generando mayor gama de productos. Tal es así, que esa misma planta podría pasar de generar únicamente dos tipos de engranajes, a cuatro modelos diferentes, ofreciendo una gama más amplia que le permita ser competente, flexible y adaptable dentro de la nueva demanda generada por los cambios que trae consigo la nueva revolución industrial.

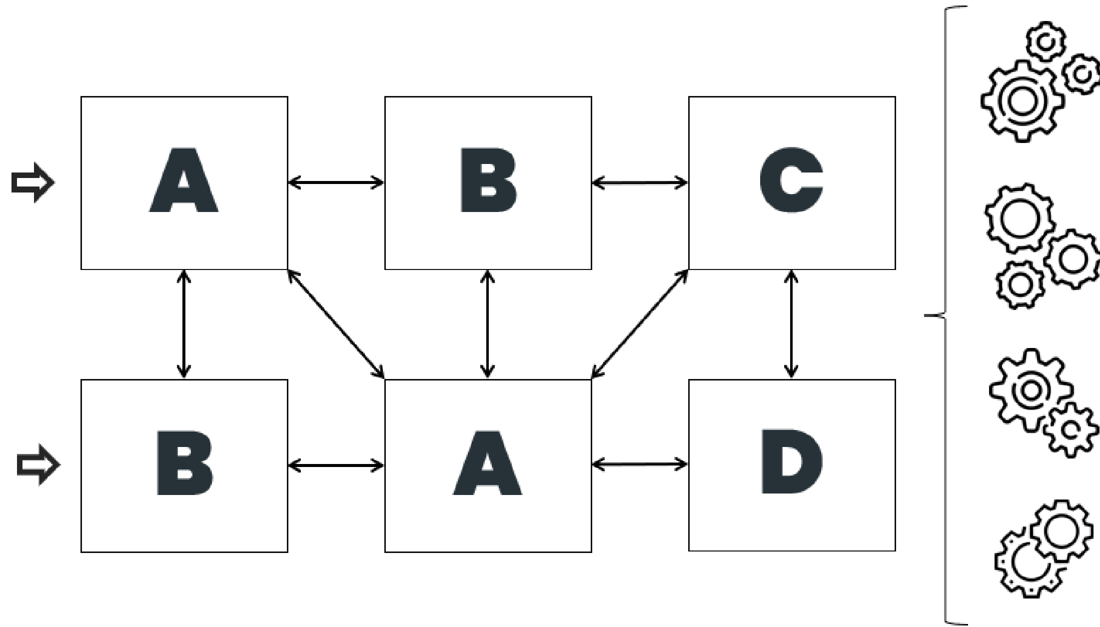


Figura 3.2 Planta Basada en un Sistema de Fabricación Flexible

Sin la presencia de cintas transportadoras y con la idea de proporcionar conectividad a todas las células de la planta, así como flexibilidad y adaptabilidad en los procesos y métodos de fabricación, surgen los AGVs (Automated Guied Vehicles). Estos son unidades móviles automatizadas de navegación autónoma cuyo objetivo principal es el del transporte, bien sea para el reabastecimiento de células o para interconectar las estaciones entre sí. De esta forma, se habilita la posibilidad de realizar distintos procesos de fabricación según el orden de las estaciones por las que deba pasar el producto, maximizando la eficiencia de las plantas al poder darse más combinaciones entre procesos y reduciendo los tiempos de producción [6], como puede observarse en la *Figura 3.3*.

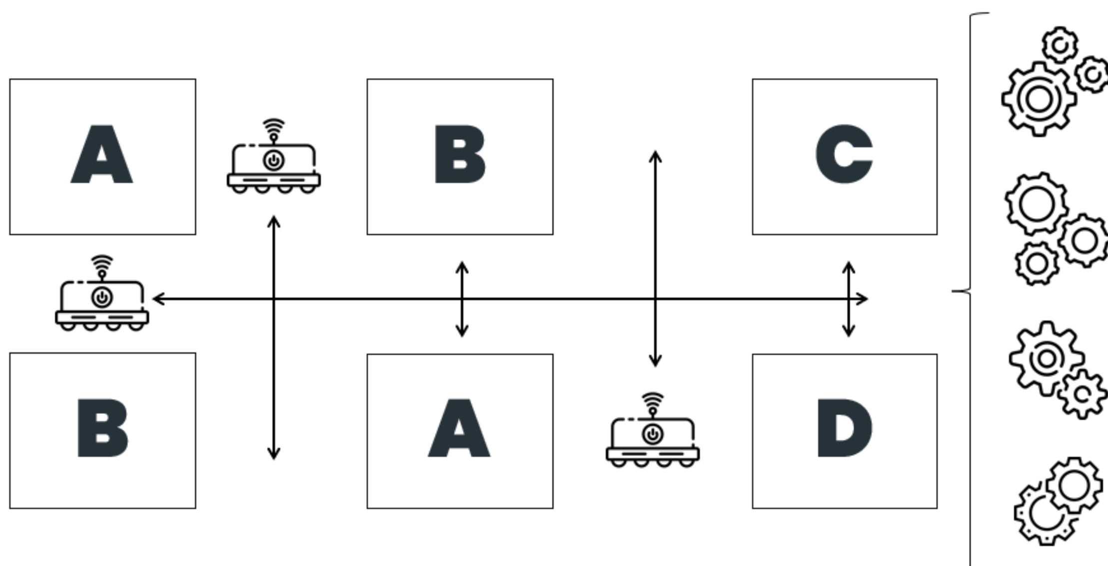


Figura 3.3 Planta Basada en un Sistema de Fabricación Flexible con Presencia de AGV

3.1.2 Framework Robóticos

Para poder satisfacer los tiempos de fabricación requeridos por la demanda actual, es necesaria la presencia de varios AGV trabajando de forma colaborativa, formando lo que varios autores conocen como "Robotic Cloud" o MRS (Multi Robot System) [7]. Para ello, además de la monitorización del entorno por parte de los AGV para que puedan operar dentro de un entorno dinámico y cambiante, también es indispensable la presencia de un Framework que contemple la cooperación entre varias unidades robóticas, en este caso, AGVs [2].

Sin embargo, la industria de la robótica y la automatización cuenta con un problema considerable en este sentido, y es que no existe un marco de referencia común o estándar parala interconexión de elementos robóticos, o no al menos una norma a la que todos los fabricantes hagan referencia. Esto ha hecho que hayan surgido diversas plataformas y Framework Robóticos de código abierto u "Open-Source", tales como: ROS (Robotic Operative System), ORCOS (Open Robot Control Software) o YARP (Yet Another Robot Platform) [8], [9].

Estas plataformas son en su mayoría Middleware, o derivados de CPS, las cuales también cuentan con funcionalidades propias de una herramienta de desarrollo. Los Middleware, son sistemas que dotan de conectividad a diversos elementos dentro de un mismo entorno, independientemente del Sistema Operativo (SO) sobre el que trabajen o el fabricante que los haya desarrollado. Lo hacen a través de servicios, habitualmente servicios comunes como pueden ser la suscripción y publicación o arquitecturas del tipo cliente y servidor. En cierto modo, podrían considerarse como herramientas que proporcionan conectividad acoplándose al Sistema Operativo de un dispositivo generando una de interfaz basada en APIs (Application Programming Interface) a modo de Driver que permite comunicarse con las interfaces de otros dispositivos, como muestra la *Figura 3.4*. De esta manera, dentro de una misma planta se puede llevar a cabo el intercambio de datos y variables sin la necesidad de integrar Hubs, Switches o Gateways para integrar las diferentes distribuciones y aplicaciones sobre las que funciona cada elemento. A pesar de ello, el emplear estas plataformas o middleware suele implicar el renunciar a las comunicaciones en tiempo real [2], [10].

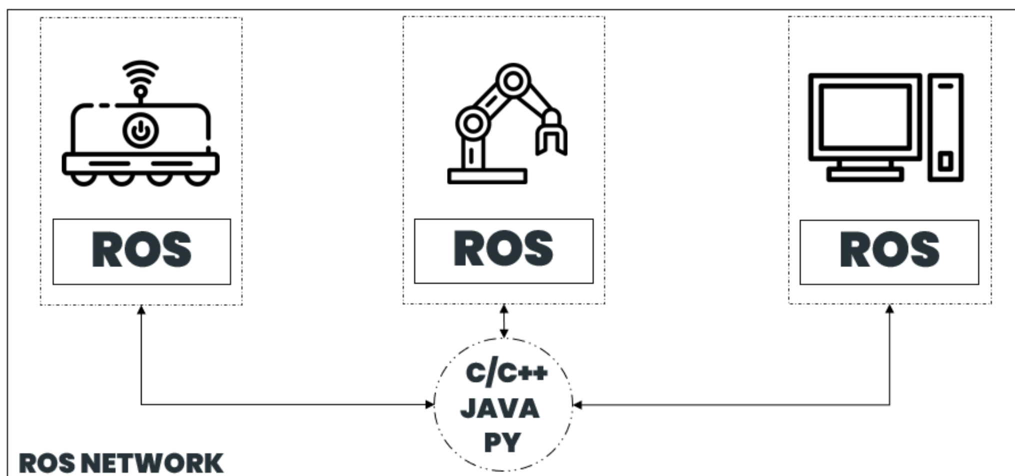


Figura 3.4 Entorno Generado por ROS

De todas estas plataformas, ROS ha sido aquella que ha alcanzado el mayor grado de popularidad para toda clase de aplicaciones de propósito general, tanto en el mundo industrial

como académico. Pese a ello, ni ROS ni ningún otro framework robótico contempla las funcionalidades sociales necesarias para implementar una Robotic Cloud, característica fundamental para poder trabajar con un grupo de AGV. Es por ello, que es necesario el desarrollar aplicaciones capaces de ocupar dicha necesidad [11].

3.1.3 Limitaciones Robóticas de Hardware

A todo esto, se le suman las limitaciones económicas y hardware de los AGV. Por un lado, para implementar una navegación autónoma eficiente y capaz de responder a los estímulos dinámicos provenientes del entorno, es necesario equipar a los AGV con periféricos costosos como sensores LiDAR (Light Detection and Ranging) o cámaras RGB (Red Green Blue). Por el otro, la descentralización elimina la presencia de una unidad gestora principal, obligando a que cada uno de los AGV tenga que tomar sus propias decisiones en base a lo que perciba de su entorno y el estado del resto de células de la planta. Esto, hace que sea necesario que los AGV cuenten con un hardware capaz de soportar elevadas cargas computacionales, incrementando su coste económico y reduciendo la eficiencia de la batería al consumirse más energía, como se muestra en la *Figura 3.5* [12], [13], [14].

Esta reducción de autonomía no se traduce únicamente en la limitación del tiempo eficiente de operación de los AGV, sino que incrementa considerablemente la dificultad de desarrollo de soluciones basadas en AGV al requerir de personal altamente cualificado, generando aplicaciones difícilmente escalables y con un mantenimiento costoso tanto en tiempo como en dinero. Es por ello, que surge la necesidad de abstraer las funcionalidades con mayor carga computacional fuera de las unidades de transporte, aplicando concurrencia o balanceo de carga de computación, de manera que dentro de los transportes únicamente residan las funcionalidades básicas para su operación normal.

En este sentido, y al igual que con otros activos de planta que no cuentan, en general, con los recursos computacionales necesarios para realizar ciertos procesos, se plantea el despliegue de estas funcionalidades en el fog, donde no existen las altas latencias y problemas de ciberseguridad asociadas a la nube. Así, la jerarquía de los sistemas de transporte ve transformada en un modelo de interconectividad global que se vertebra a través de tres niveles: planta, fog y cloud .

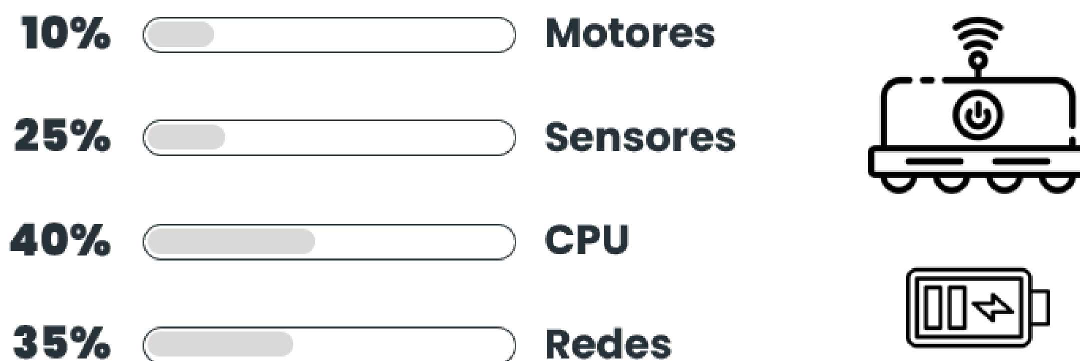


Figura 3.5 Distribución del Consumo de Batería en AGV

3.1.4 Sistemas Multi-Agente

Para solventar tanto los problemas de hardware, como los de gestión y cooperación entre varias unidades robóticas, han surgido diversas soluciones, principalmente basadas en arquitecturas de agentes en los denominados MAS . Todas ellas, tratan de añadir un órgano gestor en el que cada proceso, célula o elemento está representado a través de uno o varios agentes. Cada agente es una unidad software gestora que negocia en nombre del elemento que representa con el resto de los agentes. Su labor es la de indicar el resultado de la negociación a su representado en formato de órdenes o peticiones, a modo de cliente y servidor. Un MAS el cual gestiona múltiples unidades robóticas, suele denominarse también como MARS (Multiple-Agent Robotic System) [2], [8], [13].

El ganador o ganadores de dicha negociación se determinan en función del elemento mejor situado para llevar a cabo la solicitud desde las máquinas o desde el MES (Manufacturing Execution System). De esta manera, cada vez que llegase una nueva petición de tarea los agentes involucrados intercambiarían las variables o datos recibidos por parte de sus representados, y entre ellos, decidirían qué agente es el que cuenta con mejores condiciones para llevar a cabo la tarea. Por ejemplo, en un sistema formado por tres agentes como el de la *Figura 3.6,* a la petición de transporte de material evidentemente el elemento mejor posicionado para llevar a cabo la tarea se trataría de una unidad de transporte o AGV, mientras que para manipular dicho material sería necesaria la acción del brazo robótico antropomórfico.

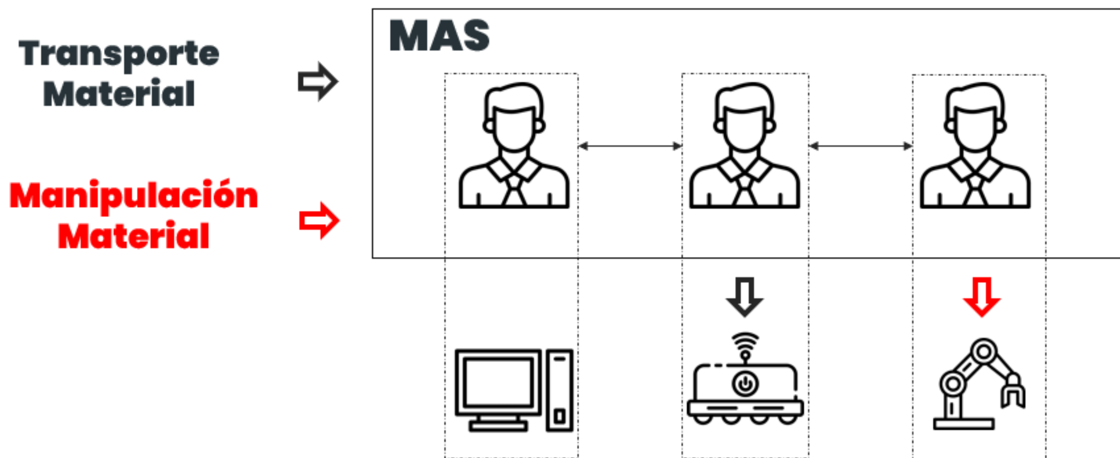


Figura 3.6 Ejemplo de MAS

Al tratarse de los MAS de arquitecturas plenamente software, cuentan con la principal ventaja de que, para llevar a cabo la integración de elementos de diversos fabricantes, únicamente sería necesario el desarrollar un Gateway o pasarela software desde dicho dispositivo al MAS, de forma que diversas plataformas, entornos y RSF puedan trabajar dentro de una misma planta gestionada por un único MAS, como se muestra en la *Figura 3.7.*

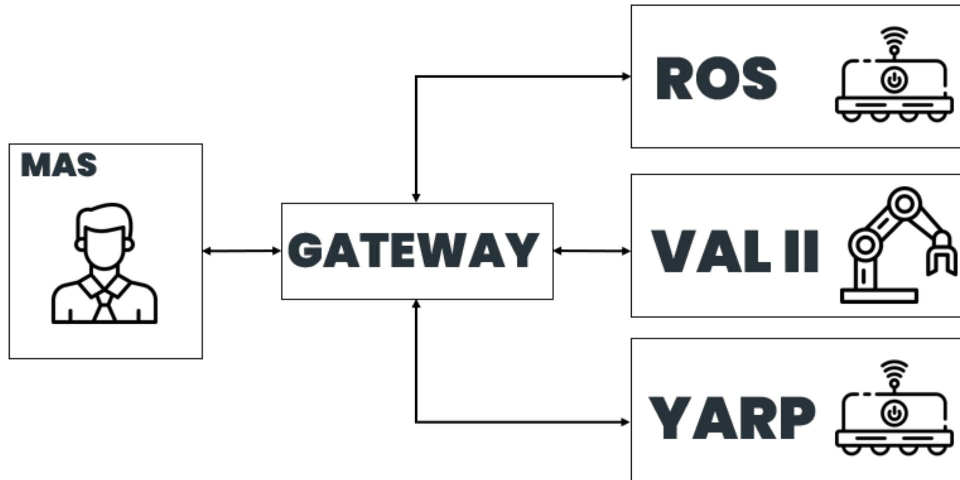


Figura 3.7 Ejemplo de Interoperabilidad de un MAS

Del mismo modo, este MAS permitiría abstraer y jerarquizar todas las funcionalidades necesarias para la gestión de una Robotic Cloud a la nube. Por tanto, en los AGV únicamente residirían las funciones básicas de navegación y seguridad, simplificando considerablemente su hardware y aumentando su autonomía, solucionando la principal problemática de limitaciones tanto económicas como hardware en su integración a los FMS [15].

Sin embargo, las aplicaciones propias de Cloud Computing cuentan con dos desventajas: La latencia en los tiempos de respuesta, que ya de por sí es crítica al tener que trabajar en tiempo real; y la ciberseguridad. A esta problemática, la mayoría de los autores proponen un elemento intermedio entre la nube y la planta: el Fog Computing o niebla. Este, cuenta con las mismas funcionalidades que el cloud, con la salvedad de que se gestiona a nivel local o nivel de planta, es decir, en una LAN (Local Area Network). De este modo, al tratarse de una nube local gestionada dentro de la propia planta, se reducirían los tiempos de respuesta y se incrementaría la seguridad, siendo inaccesible desde el exterior [3], [16], [17].

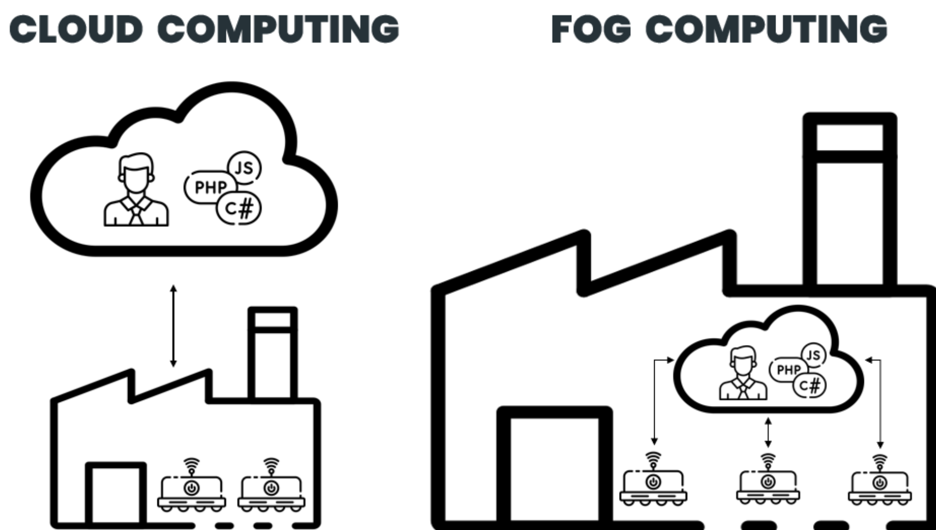


Figura 3.8 Ejemplo de Cloud Computing y Fog Computing

3.1.5 Sistemas de Navegación

Tal y como se ha comentado anteriormente, para dotar a los AGV la capacidad de interactuar con su entorno es necesario que cuenten con diversos elementos y periféricos orientados a la percepción. Desde la aparición del primer AGV en el año 1973, los AGV se han empleado principalmente para la gestión de almacenes. Por otro lado, los AMR (Autonomous Guided Robots) se han centrado en la manipulación y transporte simple de material desde un punto A hasta otro punto B dentro de una trayectoria fija y cerrada. Sin embargo, la aparición de nuevas tecnologías en el ámbito del control y la automatización han hecho que sea posible el desarrollo de diversos Sistemas de Navegación que les han permitido integrarse en todo tipo de procesos industriales repartidos por todos los ámbitos de una planta industrial . Habitualmente, los Sistemas de Navegación empleados para los desarrollos industriales suelen ser una combinación de varios sistemas, siendo los más comunes los siguientes [3], [18]:

- **Sistemas de navegación ópticos.** La navegación se basa en la detección y seguimiento de líneas de colores pintadas en el suelo. Son muy empleados en soluciones en los que tengan que desplazarse AGV con elevadas cargas o gran tamaño. Cuentan con el principal inconveniente de que el Sistema de Navegación quedaría inservible en caso de situarse una mancha o despegarse parte de ella.

- **Sistemas de navegación basados en inducción.** Derivados de los Sistemas de Navegación Ópticos, en vez de emplearse una línea pintada sobre el suelo se emplean railes magnéticos, de manera que la unidad de transporte tenga como referencia el campo magnético generado por el rail. Igualmente, se emplean para desplazar AGV de gran tamaño, como los empleados en refinerías para transportar materia prima. Sin embargo, cuentan con el inconveniente de que su instalación es realmente costosa tanto en tiempo como en dinero, al ser necesario levantar el suelo para ello. Además, son muy poco flexibles, puesto que una vez realizada la obra e instalado el carril magnético bajo suelo la trayectoria no se puede modificar hasta realizar una nueva obra.

- **Sistemas de navegación basados en banda ancha.** Se tratan de sistemas muy empleados en entornos en los que el espacio de navegación es simétrico, poligonal y sin una gran cantidad de obstáculos de por medio, como podría ser un puerto de descarga de material de un almacén. El concepto de funcionamiento se inspira en los GPS (Global Positioning System), salvo que en este caso en vez de satélites se emplean antenas que generan señales de alta frecuencia. Estas antenas se situarían en los extremos del entorno de navegación, de manera que cada AGV contaría con un receptor capaz de detectar dichas señales. En función del retraso y de la distorsión de la señal recibida, el AGV sería capaz de situarse en el entorno aplicando conceptos básicos de trigonometría y geometría. Sin embargo, se tratan de sistemas que únicamente pueden emplearse en entornos libres de obstáculos, siendo considerablemente difícil el implementarlos en entornos industriales convencionales donde pueden darse toda clase de obstáculos y geometrías.

- **Sistemas de navegación basados en ultrasonidos.** Esta clase de Sistemas de Navegación son muy empleados en entornos académicos o no industriales, como pueden ser los dedicados al ocio o servicios, como cafeterías o estaciones. Se basan en la emisión y recepción de sonidos de alta frecuencia imperceptibles al oído humano pero que el AGV es capaz tanto de generar como de recibir. De esta manera, basándose en los rebotes del ultrasonido generado por la unidad de transporte, puede detectar los obstáculos en su camino. No son precisamente una solución viable en entornos industriales, al ser el ruido una de las características principales de estas plantas.

- **Sistemas de navegación basados en inercia.** Esta clase de sistemas emplean giroscopios o IMU (Inertia Measurement Unit) para estimar la posición relativa de una unidad de transporte con respecto una posición inicial de partida. A través de estos elementos, es posible calcular cuánto se ha movido una unidad de transporte y en qué dirección o direcciones. La gran mayoría de vehículos o unidades autónomas dedicadas a la navegación cuentan con una funcionalidad que se encarga de calcular dicho desplazamiento, o lo que se le conoce como "odometría". El principal problema de las soluciones basadas en esta clase de sistemas es que las unidades no son capaces de detectar cambios en un entorno dinámico y mucho menos obstáculos.

- **Sistemas de navegación basados en reconocimiento de códigos y etiquetas**: Esta clase de Sistemas de Navegación se basan en la percepción y detección de códigos, bien sean estandarizados como los códigos QR o de barras, o bien etiquetas creadas por la aplicación en concreto. Habitualmente, esta clase de sistemas se emplean para complementar otra clase de Sistemas de Navegación, de manera que pueda complementarse las funciones de localización y manipulación propias de las unidades de transporte. Esta clase de códigos, suelen situarse o bien en el suelo para verificar posiciones en el plano o bien en posiciones clave que puedan influir en la navegación. Por otro lado, también son empleados para verificar el tipo de objeto que va a manipularse. A pesar de ello, no hay muchas aplicaciones en las que se emplee únicamente la percepción y reconocimiento de códigos para llevar a cabo una navegación autónoma.

Sin embargo, aquel Sistema de Navegación que más prestaciones ha demostrado tener ha sido el Sistema de Navegación Basado en LiDAR y Técnicas SLAM (Simultaneous Localization and Mapping). Esta clase de Sistemas de Navegación combinan tanto la capacidad del sensor LiDAR para percibir su entorno como algoritmos capaces de estimar la posición en la que se encuentra la unidad de transporte dentro de un mapa del entorno previamente creado, como se muestra en la *Figura 3.9* [19].

Figura 3.9 Ejemplo de AGV Equipado con Sensor LiDAR

Las Técnicas SLAM, o simplemente SLAM, son un método utilizado principalmente en vehículos o unidades de transporte autónomas, el cual permite el crear un mapa del entorno y localizar el vehículo en ese mismo mapa de manera dinámica. Combinan algoritmos propios tanto de localización como el método Monte Carlo o algoritmos relativos a la ejecución de trayectorias como A-Star (A*), D-Star (D*) o Dijkstra [13], [20]. Asimismo, también existen diversos algoritmos para la creación de mapas, siendo los más comunes Gmapping, LOAM, Cartographer o HectorSLAM.

Sus altas prestaciones y adaptabilidad al entorno han permitido a esta clase de Sistemas de Navegación el convertirse en los más empleados para llevar a cabo soluciones que impliquen navegación autónoma, tanto en ámbito industrial como en el académico. Es por ello que Framework Robóticos como ROS ya integran varios paquetes software que encapsulan diversos algoritmos enfocados a la implementación del Sistema de Navegación mediante SLAM

Al igual que el resto de los sistemas de navegación mencionados, esta clase de sistemas basados en SLAM y LiDAR cuentan con diversas debilidades, como la detección de cierta clase de obstáculos, la deriva en la localización del vehículo con el tiempo o la exactitud en el posicionamiento. Es por ello, que la gran mayoría de los desarrollos y soluciones de este ámbito, suelen ser una combinación de las características proporcionadas por cada Sistema de Navegación, de manera que los puntos débiles de cada uno de ellos puedan ser cubierto por uno de los puntos fuertes del resto. Habitualmente, es común el encontrar soluciones que combinen los Sistemas Ópticos, Sistemas Basados en Inercia, Sistemas Basados en Reconocimiento de Códigos y Etiquetas, y los Basados en LiDAR y Técnicas SLAM [21].

3.2 Antecedentes

Dado que el presente trabajo se ha realizado dentro del GCIS de la UPV/EHU, en la realización del presente estado del arte se ha tenido en cuenta los antecedentes y trabajos previos en este ámbito desarrollados dentro de dicho grupo de investigación. Concretamente, se han identificado tres, que se presentarán en orden cronológico y de los cuales se identificarán sus características más importantes y su relación este trabajo.

3.2.1 Puesta en Marcha del Control de Robots de Transporte

Este desarrollo llevado a cabo por Gorka Alfonso Aberasturi en 2018 bajo la dirección de la profesora Aintzane Armentia Díaz de Tuesta, se llevó a cabo como un trabajo fin de máster [22]. El objetivo principal de este trabajo era el de realizar la implementación de un software de control que permitiese realizar las labores de transporte dentro de una planta de producción flexible. Los puntos que se marcaron para cumplir dicho objetivo fueron el de desarrollar mecanismos que asegurasen la recuperación del sistema de transporte ante fallos, cubrir las necesidades de transporte más básicas y el desarrollar una interfaz para que una persona operaria pudiese interactuar con las unidades de transporte.

Este proyecto se realizó dentro del mismo entorno de navegación en el cual va a realizarse este trabajo, es decir, el laboratorio del GCIS. Se emplearon para ello cuatro unidades robóticas basadas en el modelo Kobuki iCleBot Turtlebot2 las cuales trabajan todas dentro del Framework Robótico ROS. El Sistema de Navegación empleado, estaba basado en tecnología LiDAR y técnicas SLAM. Sólamente uno de los cuatro AGV llegó a emplearse como unidad de transporte, siendo los otros tres restantes elementos empleados para verificar el funcionamiento de las tareas de rescate de material por parte del primer AGV, el cual era el único que contaba con los periféricos suficientes como para identificar cada transporte. Dicha identificación, se realizaba mediante la percepción de códigos AR a través de una cámara Kinect XBOX 360.

En lo relativo a su integración en un sistema multi-agente, el trabajo únicamente llegó a proponer una arquitectura de interacción entre el sistema de navegación desarrollado y los agentes de transporte de la plataforma, pero sin llegar a integrar ambas partes. Por su parte, sí que se definieron distintos estados de funcionamiento para el Sistema de Navegación desarrollado, como eran los estados de: Delivery, para la entrega de material; Exploration, para la búsqueda de unidades de transporte durante sus tareas de rescate; y Rescue, como operación de rescate. No se llegó a desarrollar ni plantear tampoco tareas de manipulación ni de percepción de obstáculos.

En su conjunto, los objetivos principales del trabajo se alcanzaron, llegando a diseñar un layout de planta como el mostrado en la *Figura 3.10* y su correspondiente interfaz para el operario *Figura 3.11*. Este desarrollo, pese a no conservarse en los repositorios de código del grupo de investigación, ha permitido a sentar las bases para el Sistema de Navegación desarrollado en el presente trabajo, además de proponer una distribución y solución hardware para las distintas unidades de transporte. Por último, y con el objetivo de contextualizar este trabajo dentro de la arquitectura de capas planteada en [1], hay que destacar que este trabajo significó una primera aproximación a las capas operativa y funcional.

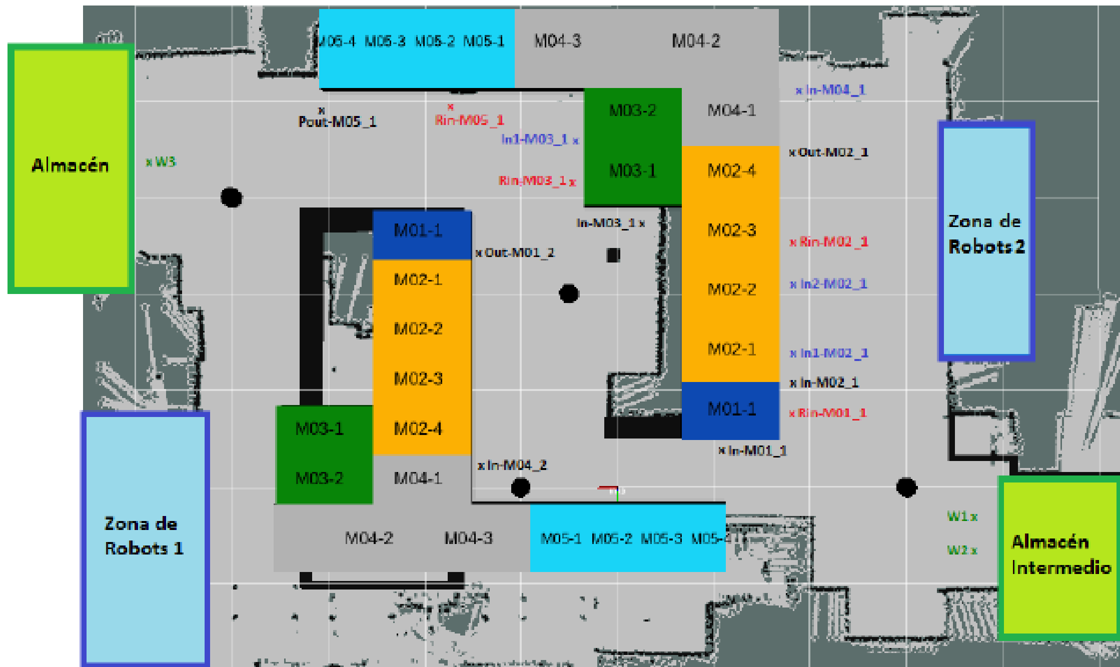


Figura 3.10 Planta Propuesta por el Primer Antecedente

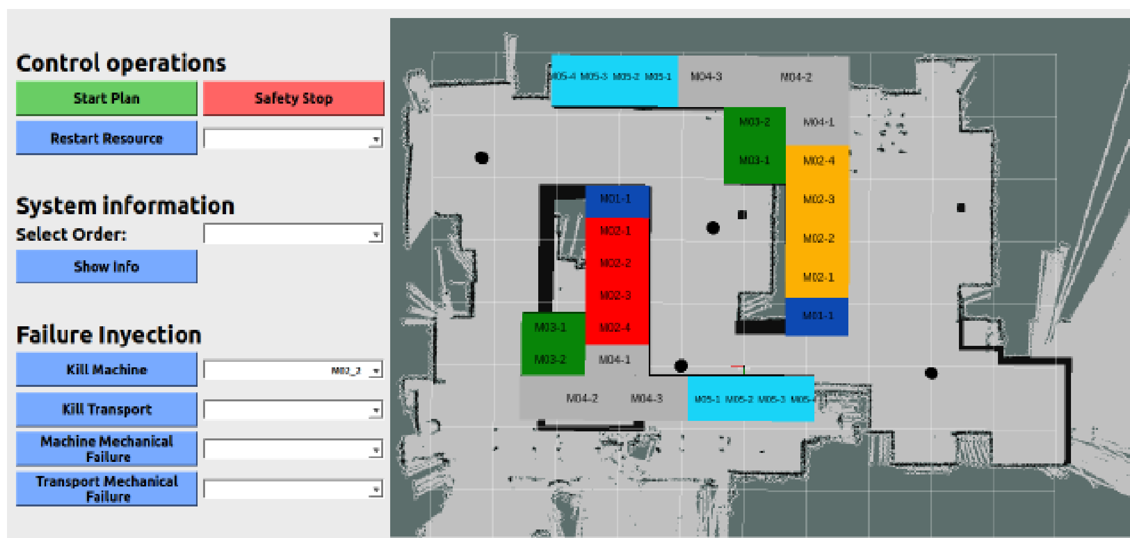


Figura 3.11 Intefaz Desarrollada por el Primer Antecedente

3.2.2 Robots Colaborativos en Procesos de Fabricación Flexibles: Integración ROS-JADE

Este desarrollo llevado a cabo por Brais Fortes Novoa en 2018 junto bajo la dirección de las profesoras Marga Marcos Muñoz y Aintzane Armentia Díaz de Tuesta se realiza nuevamente dentro del marco de un trabajo fin de máster [23]. El objetivo principal de este trabajo era el de realizar la implementación de una solución software capaz de integrar robots móviles que empleen ROS como framework robótico dentro de una industria descentralizada y flexible basada en sistemas multi-agente. Los puntos que se marcaron para cumplir dicho objetivo fueron el de dotar a los vehículos autónomos de toma de decisiones distribuida y el crear una serie de elementos que permitiesen la integración del entorno ROS con una plataforma de gestión de aplicaciones de

fabricación basada en agentes y desarrollada sobre la plataforma JADE (Java Agent Development Network).

Este trabajo en cuestión se desarrolló de manera paralela al trabajo descrito en el apartado 3.2.1. Concretamente, mientras que dicho abstrajo todos los aspectos relativos a las tareas de más bajo nivel enfocadas al transporte de material y navegación mediante unidades autónomas de transporte, este trabajo aglutinó todos los aspectos relativos a la arquitectura necesaria para su integración la plataforma MAS. Por otro lado, ha de destacarse que esta integración de los sistemas de transporte se realizó dentro de una arquitectura software previamente definida en el GCIS, concretamente, la Arquitectura Multi-Agente de FLEXMANSYS. El resultado de dicha integración JADE-ROS, se muestra en la *Figura 3.12* .

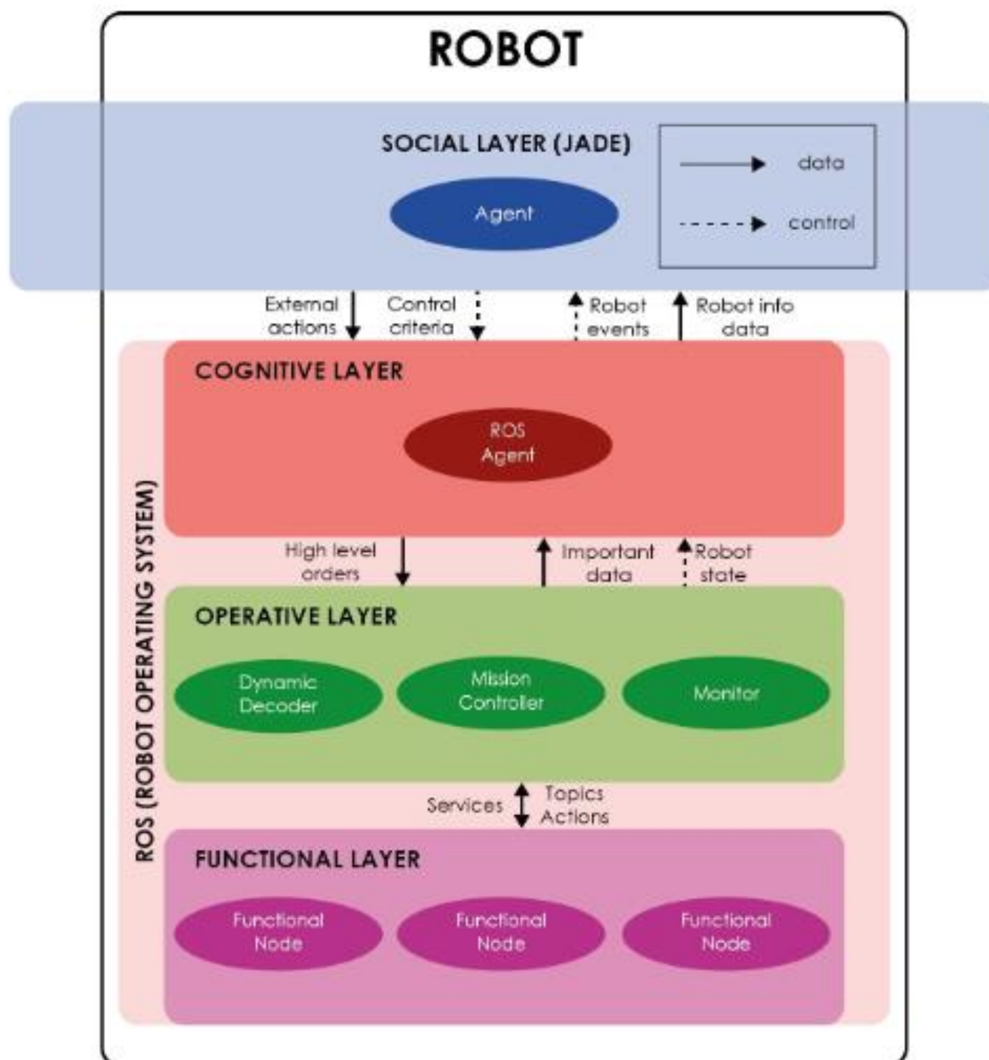


Figura 3.12 Arquitectura Multi-capa de Integración JADE-ROS Propuesta por el Antecedente Segundo

Tal y como ha podido observarse en la figura anterior, la arquitectura de la solución desarrollada cuenta con cuatro capas, las cuales concuerdan con la arquitectura multi-agente propuesta por la FLEXMANSYS original. Las principales funciones de cada una de las capas se resumen en las siguientes:

- **Social Layer**: La capa social se trata del nivel más alto de la arquitectura, la cual está formada por tantos agentes como contenga la plataforma. Se encarga de planificar y decidir las tareas de cada uno de los representados por los agentes según los objetivos generales del MAS, bien estén estos definidos por un MES o por necesidades internas del sistema multi-agente.
- **Cognitive Layer**: La capa cognitiva se trata de aquella que hace de enlace entre los entornos de JADE y ROS. Los componentes que forman parte de esta capa serían los denominados como "ROS Agents", o nodos ROS que cuentan con funcionalidad de agente. En pocas palabras, estos ROS Agents realizan el intercambio de información vertical y bidireccionalmente entre la capa operativa y la social. Por otro lado, también almacenan toda la información que sea relevante para sus agentes asociados.
- **Operative Layer**: La capa operativa es la encargada de traducir las órdenes de más alto nivel en órdenes entendibles por los nodos funcionales de bajo nivel. Lo hace a través de tres tipos de nodos:
- **Functional Layer**: La capa funcional se trata del nivel más bajo de la arquitectura y aquella que está en relación directa con los sensores y actuadores. Es decir, son los nodos que se dedican a gestionar y comunicar órdenes directas sobre el hardware.

Con todo, los objetivos principales del trabajo se alcanzaron, llegando a realizar varias pruebas de concepto que mostraron que la arquitectura e integración diseñada funcionaban correctamente. Sin embargo, el trabajo no llegó a incorporar la solución llevada a cabo en el primer antecedente , por lo que se realizaron todas las pruebas de concepto a través de suposiciones y sin llegar a acoplar un Sistema de Navegación real formado por AGV físicos, aunque tampoco era el objetivo principal del proyecto al abstraer esos niveles bajos de la arquitectura a un sistema formado por una caja negra.

Por lo tanto, y con el objetivo de contextualizar este trabajo dentro de la arquitectura de capas planteada en [1], hay que destacar que este trabajo significó una primera aproximación a las capas de gestión y de inteligencia. Entre las limitaciones de esta primera aproximación, se encuentra el fuerte acoplamiento entre ambas capas y su diseño dependiente de las tecnologías subyacentes. Estos aspectos constituyen áreas de mejora en el presente trabajo.

3.2.3 Autonomous Ground Vehicle Management for Flexible Manufacturing Systems

Este trabajo realizado por Jon Peñalver Bravo bajo la dirección del profesor Oskar Casquero Oyarzabal en 2020 se lleva a cabo dentro del marco de un trabajo fin de máster [24]. El objetivo principal de este trabajo es doble: por una parte, identificar todos los elementos de arquitectura propuestos en los dos trabajos previos y documentar el proceso de generación de código mediante rosjava (la librería que permite programar nodos ROS en Java y, por tanto, permite su integración con agente JADE, también programados en Java); por otra parte, realizar una prueba de concepto que muestre el funcionamiento de la arquitectura de integración ROS-JADE y su interacción con las máquinas durante la ejecución de un plan de fabricación

A diferencia de la solución propuesta en el segundo antecedente, este trabajo desarrollado por Jon Peñalver Bravo no solo readaptó la arquitectura para ajustarla de la manera óptima posible a las necesidades de la aplicación, sino que llevó a cabo la codificación y diseño de las unidades software funcionales para poder integrar la arquitectura. Concretamente, llevó a cabo el diseño de los nodos funcionales ROS en el lenguaje de programación C/C++, mientras que propuso y empleó el paquete ROSJava para llevar a cabo la integración del entorno JADE con ROS

empleando el SO Ubuntu 16.04. La solución final junto con las unidades software desarrolladas y su lugar en la arquitectura de MINECO, se muestra en la siguiente *Figura 3.13*.

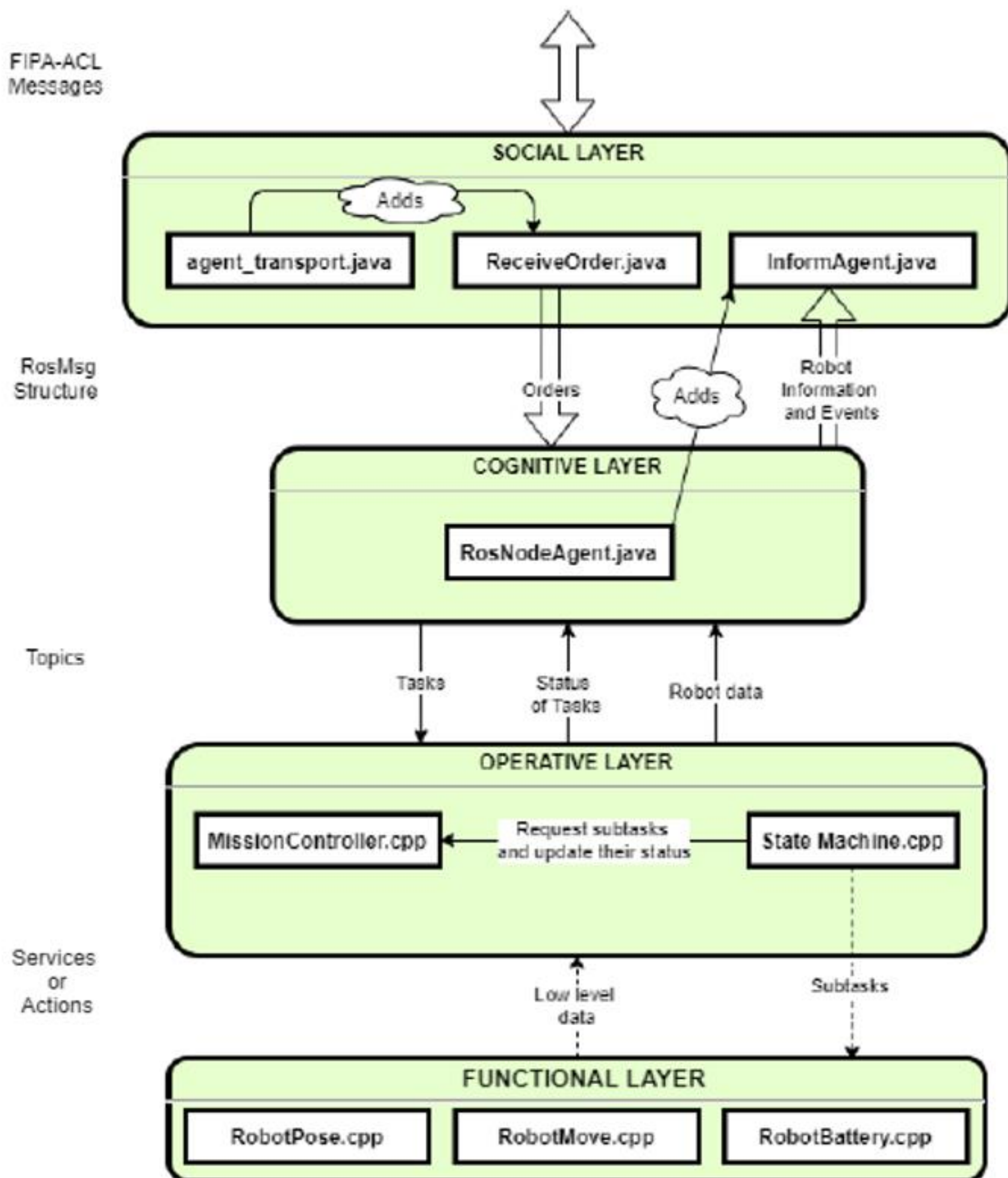


Figura 3.13 Arquitectura de Integración ROS-JADE Propuesta por el Tercer Antecedente

En términos generales, los objetivos de este antecedente se complementaron de manera satisfactoria, codificando gran parte del repositorio y arquitectura donde se recoge el sistema multi-agente sobre el que se desarrolla el presente trabajo y el cual está siendo empleado para la realización de la tesis doctoral en la que se sitúa el proyecto. Sin embargo, al igual que el antecedente segundo, carece de un sistema de navegación funcional.

3.3 Motivación

Bajo todo este contexto, este trabajo contribuye al desarrollo de un sistema que integre varios AGVs de la gama Turtlebot dentro del marco del FMS y la FoF empleando ROS. Estos, estarán gestionados a través de una plataforma MAS implementada en JADE, la cual se integra en una arquitectura software multicapa previamente desarrollada basada en agentes software. Por otro lado, se llevará a cabo el desarrollo e implementación de navegación autónoma mediante un Sistema de Navegación Basado en LiDAR y técnicas SLAM, además de la percepción de obstáculos en el entorno.

También, se llevará a cabo tanto la integración del Sistema de Navegación y diseño de la negociación entre agentes de transporte dentro de la plataforma MAS, como la comunicación entre los agentes desarrollados en el entorno JADE con las unidades de transporte. Dado que trabajan sobre ROS, la comunicación se hará a través de un Gateway implementado mediante ROSJava. Toda esta solución, se ejecutará sobre una plataforma del tipo Fog Computing o niebla, donde se abstraerá toda funcionalidad no vital de las unidades de transporte, generando así un sistema concurrente.

En resumen, este trabajo tratará de unificar el esqueleto de todas las soluciones desarrolladas por antecedentes para diseñar una solución única con toda la arquitectura multicapas integrada, además de realizar nuevas integraciones. Esta solución, se llevará a cabo considerando y teniendo en cuenta el estado del arte actual existente en lo relativo a los MAS y los AGV en sistemas de fabricación flexible. Por otro lado, se llevarán a cabo diversos cambios de concepto que mejoren las soluciones previas y que se integren dentro de la tesis doctoral en la que se sitúa el presente proyecto.

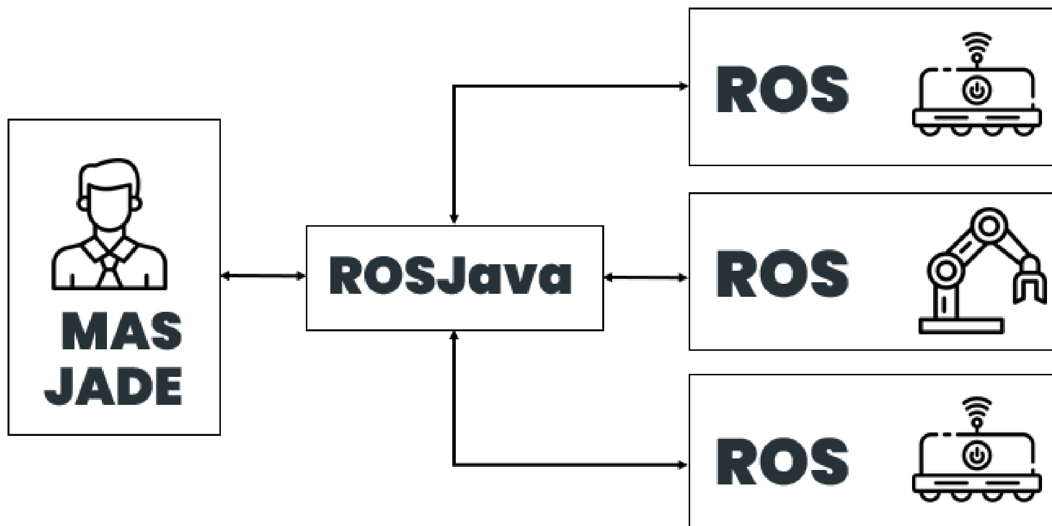


Figura 3.14 Esquema Ejemplo de Solución a Desarrollar

CAPITULO 4

ANÁLISIS DE LAS HERRAMIENTAS EMPLEADAS

.

4 Análisis de las Herramientas Empleadas

En el presente capítulo, se detallan todas las herramientas empleadas para el desarrollo de la solución, así como la justificación de su elección.

4.1 **Herramientas Software**

En este apartado, se describirán todas las herramientas software que se han empleado para el presente trabajo, así como conceptos básicos de las mismas.

4.1.1 Análisis de Framework Robóticos

El mundo de la robótica siempre se ha caracterizado por la falta de un estándar global y bien definido para el desarrollo de soluciones robóticas, tal y como otros sectores de la automatización ya cuentan con los estándares IEC-61131 o IEC-61499. Ante este vacío, han surgido múltiples plataformas y lenguajes de programación que han ido creciendo en popularidad hasta convertirse en referencias para el desarrollo de soluciones robóticas, como lo son el lenguaje de programación VAL (Variable Assembly Language) o el Framework de Software Robótico ROS, pero sin llegar nunca a asentarse como la herramienta de desarrollo de referencia. Sin embargo, tal y como se ha comentado en el capítulo anterior, dada su accesibilidad, su gran comunidad de desarrolladores y todas las funcionalidades que integra, ROS se ha convertido en el Framework Robótico más popular, tanto en el mundo académico como en el industrial [8], [9].

Como su propio nombre indica, un RSF presenta una solución para facilitar el desarrollo de software específico para robots, proporcionando las herramientas necesarias para integrar y controlar un robot o conjunto de ellos [25]. En lo referente a ROS, se trata de un middleware, es decir, un software que brinda servicios y funciones comunes a las aplicaciones, además de las que ya ofrece el sistema operativo; incluyendo la gestión de datos y conectividad con otros equipos.

De esta forma, tal y como muestra el esquema de la *Figura 4.1* ROS es capaz de integrar dentro de un mismo entorno a múltiples elementos, sean robóticos o no, sin importar si cuentan con sistemas operativos distintos; siempre y cuando estos cumplan con los requisitos y configuración mínima y necesaria de ROS. Esta característica de ROS da una solución al problema de la falta de estandarización entre los distintos fabricantes de robots, facilitando la comunicación, cooperación y control entre los distintos equipos [26] .

A todo esto, se le suma la capacidad de poder desarrollar código sin importar el lenguaje de programación que se emplee, bien sea en Python, C/C++ o Java; hablando de aquellos que se emplearán en este trabajo. Las principales características con las que cuenta ROS y de las cuales se requieren de un RSF, se resumen en las siguientes:

- **Modularidad**. Para garantizar una arquitectura flexible, fácilmente reutilizable y con un grado alto de cohesión, es necesario que la herramienta empleada cuente con un grado bajo de acoplamiento. Para ello, se requiere de un alto grado de modularidad, de manera que puedan integrarse funcionalidades aisladas de manera sencilla sin que haya que reconfigurar parámetros del sistema al que se quiere integrar dicha nueva funcionalidad.

- **Robustez y tolerancia a fallos**. Toda solución robótica industrial no debe de permitir que el fallo de un único componente comprometa al funcionamiento global de la célula. Es por ello, que la herramienta de desarrollo empleada debe de ser capaz de continuar con sus labores restantes independientemente del fallo ocurrido.

- **Arquitectura distribuida y concurrente**. En sistemas en los que se requiera un alto nivel de complejidad, el gasto computacional puede ser muy elevado. Es por ello, que es necesario poder emplear todas las unidades de procesamiento disponibles, de manera que varios equipos puedan procesar el código de un único robot en varias unidades de procesamiento de manera concurrente y distribuida.

- **Eficiencia y tiempo real**. Al trabajar con sistemas dinámicos en los que las excitaciones y perturbaciones externas no son arbitrarias, sino más bien aleatorias, es necesario contar con un entorno capaz de trabajar con restricciones de tiempo muy bajas, capaces de cumplir con requerimientos de seguridad muy altos.

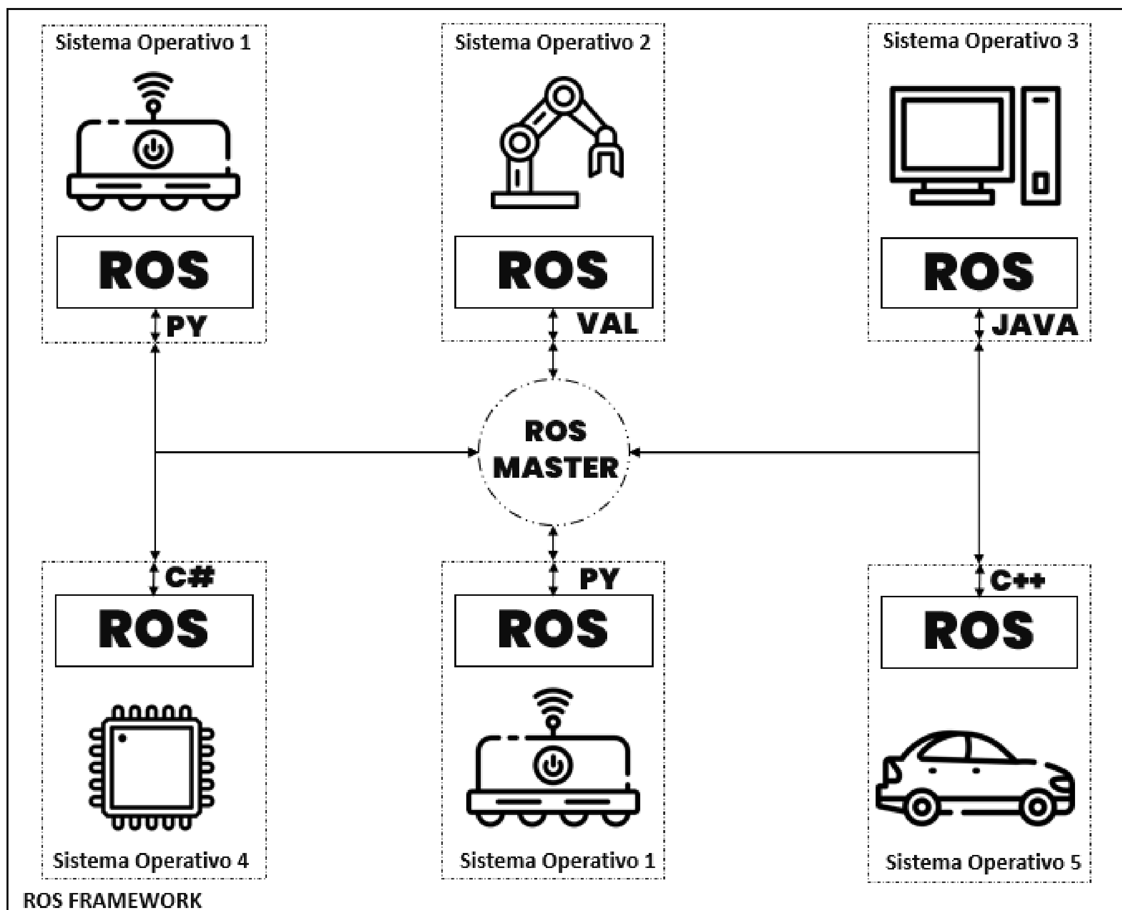


Figura 4.1 Detalle de Entorno de Trabajo Generado por ROS entre Distintos Elementos

Al ser ROS uno de los RSF más populares, se ha considerado como la herramienta software ideal para el desarrollo del presente trabajo. Sin embargo, no únicamente se ha tenido en cuenta su considerable grado de popularidad, sino que se ha tenido en cuenta otras características como la escalabilidad, reusabilidad, simuladores o herramientas de modelado. Para ello, se

analizaron diversos RSF tales como: OROCOS (Open Robot Control Software), Player, YARP (Yet Another Robot Platform), OpenRTM (Open Real Time Middleware) u OpenRAVE (Open Robotics Automation Virtual Environment). Sus principales características, se recogen en la siguiente *Tabla 4.1*. [25].

RSF	Arquitectura Distribuida	Herramientas Administración	Interfaces Hardware	Algoritmos Robóticos	Simuladores	Tiempo Real
ROS	Sí	Sí	Sí	Sí	Sí	No
ORCOS	Sí	No	Sí	Sí	No	Sí
Player	No	No	Sí	Sí	Sí	No
YARP	Sí	Sí	Sí	No	Sí	No
Op.RTM	Sí	Sí	Sí	Sí	No	No
Op.Rave	No	No	No	Sí	Sí	No

Tabla 4.1 Características de los Principales RSF

Por otro lado, ha de considerarse que este proyecto se ha desarrollado dentro de un entorno universitario, luego el hecho de que ROS sea una herramienta de código abierto completamente gratuita ha influido notablemente en la decisión; además de tratarse de la herramienta software empleada dentro del grupo de investigación en el que se ha desarrollado el presente trabajo. Tal es así, que también se ha tenido en cuenta la existencia del paquete software de ROSJava, el cual permite integrar el entorno de ROS con la plataforma JADE basada en Java, en la cual residen los agentes de la arquitectura en la cual se va a integrar este trabajo.

Cabe destacar, también, las múltiples soluciones y paquetes software que ROS ofrece para la navegación autónoma, implementando dentro de sus funcionalidades varios algoritmos enfocados a la realización de dicha tarea. En lo relativo a navegación autónoma, ROS se trata del RSF más empleado, siendo incluso la herramienta empleada para el desarrollo de la navegación de coches autónomos. Los paquetes más populares de ROS empleados para el control y diseño de robots que requieran de navegación autónoma, son: ROS Navigation Stack, el cual se desglosará más adelante; Cartographer ROS y Hector Slam. Es por ello, que, dado que el presente trabajo requiere del desarrollo de un sistema de navegación formado por varios AGV, ROS se trata del RSF ideal para llevar a cabo la solución del presente documento.

4.1.2 Framework ROS

En este apartado, se desarrollarán los conceptos básicos de funcionamiento y estructura de ROS. Se detallarán tanto el funcionamiento general, como los tipos de comunicación y elementos más empleados.

4.1.2.1 Conceptos Básicos de ROS

Antes de entrar con los conceptos más técnicos de ROS, hay que entender que ROS se basa en una arquitectura de reticular en los que los nodos del sistema se comunican entre sí, ya

sea a través de comunicaciones tipo cliente-servidor o bien mediante tópicos conducidos por enlaces del tipo publicista-suscriptor. Tal es así, que la plataforma está basada en un comportamiento similar al P2P (Peer-to-Peer), por lo que los diferentes nodos que conforman el sistema se comunican entre sí mediante mensajes y servicios sin tener que recurrir a que toda la información intercambiada pase por un elemento central como sería un elemento maestro en otras arquitecturas Es más, tal y como se ha comentado anteriormente, ROS permite la integración de múltiples equipos dentro de una misma red Wi-Fi, por lo que no necesariamente la comunicación entre los nodos debe de hacerse entre nodos de un mismo sistema [27], [28].

Para comprender mejor los conceptos generales de ROS, se aporta el esquema de la *Figura 4.2* . En él, se muestra el ejemplo de una aplicación en la que desde una cámara situada en un equipo AGV, se toma una foto del entorno y se publica en la nube. Para ello, toman parte dos unidades de procesamiento distintas, ambas integradas dentro del entorno generado por ROS. La primera unidad de procesamiento gestiona el AGV y la cámara, mientras que la segunda, el ordenador personal que coordina el AGV. Como puede apreciarse, los datos provenientes de la cámara pasan previamente por tres nodos: uno dedicado a la lectura explícita de la cámara, otro para el procesamiento de los datos leídos, convirtiendo la estructura de datos del mensaje que proporciona la cámara de camera_msg/CameraData a camera_msg/CameraImage; y uno final que se dedica a subir los datos de la imagen a la nube. La comunicación entre estos nodos se hará a través de tópicos, de los cuales serán capaces de leer los datos que el resto de los nodos han procesado y publicado, pudiendo actuar todos ellos tanto como publicistas o suscriptores.

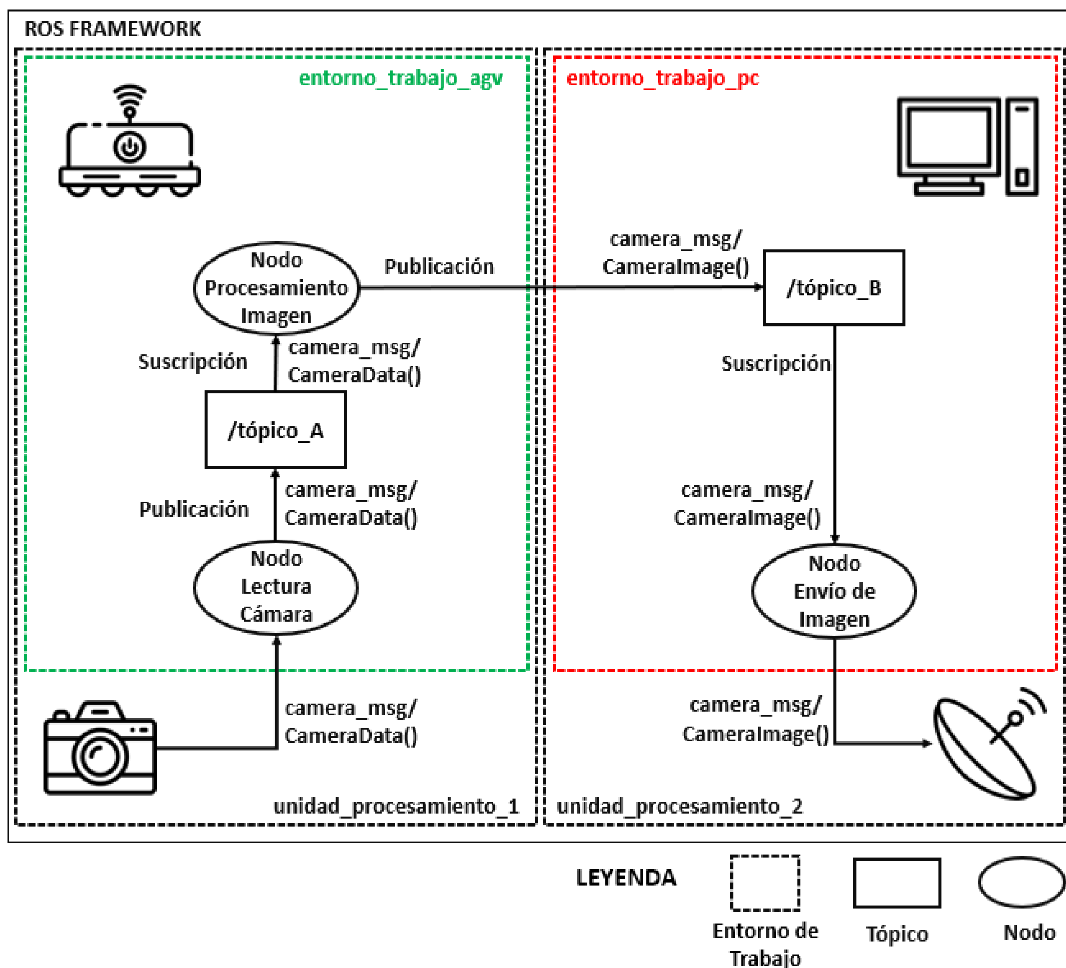


Figura 4.2 Ejemplo de Estructura Básica de un Entorno ROS

Del ejemplo mostrado, hay que entender que los nodos no son más que unidades de computación, los cuales efectúan una tarea, y que la comunicación entre estos nodos o unidades de computación se efectúa mediante un comportamiento Peer-to-Peer. Dicha comunicación, puede efectuarse entre diferentes entornos de trabajo y equipos, siempre y cuando estos estén integrados dentro del mismo entorno ROS y respeten las estructuras de datos de los mensajes intercambiados entre los nodos. Es gracias a esta característica de ROS, que pueden realizarse soluciones de elevada modularidad, robustez y concurrencia.

4.1.2.1.1 Elementos Principales

Sin embargo, ROS no cuenta únicamente con nodos y tópicos. Los elementos principales que conforman el Framework Robótico ROS pueden resumirse en los siguientes: Nodos, Maestro, Tópicos, Paquetes, Servicios, Acciones y Bolsas; entre otros que se irán desglosando a lo largo de este capítulo [29], [30].Los nodos, tal y como se ha visto anteriormente en el ejemplo de la *Figura 4.2* no son más que meras unidades computacionales. Estos, tienen como objetivo la ejecución de uno o varios procesos, según el archivo de código ejecutable al que se asocien. Estos nodos pueden comunicarse entre ellos mediante elementos intermediarios como son los tópicos, o bien directamente entre ellos generando servicios. Estos nodos, pueden visualizarse como funciones de un programa, los cuales aportan una serie de argumentos de salida en función de los argumentos de entrada introducidos.

Con respecto al maestro, en ROS existe la figura del nodo maestro. Pese a su nombre, el nodo maestro no interviene en la comunicación directa de los nodos . Su funcionalidad, reside en el nombramiento y registro de los nodos que entren dentro del entorno y de proporcionar las direcciones y puertos en los que cada nodo vaya a alojarse. En cierto modo, se trata de una especie de servidor DNS (Domain Name System) de una red Wi-Fi convencional. Se encargará también de recoger las direcciones de los servicios y tópicos del entorno. De esta forma, los nodos que quieran realizar la comunicación con otro nodo o suscribirse a un tópico, preguntarán al maestro por la dirección y puerto al cual deben de dirigirse, como si se tratara de una LUT (Lookup Table).

Los tópicos, por su parte, ya se ha mostrado su funcionamiento en el ejemplo de la *Figura 4.2.* Estos tópicos, no dejan de ser elementos intermedios en los cuales pueden guardarse datos, de manera que cualquier integrante del sistema interesado en dichos datos pueda acceder a ellos cuando lo requiera. Sin embargo, todo nodo bien sea publicista o suscriptor, deberá de acceder al tópico siguiendo la estructura de los datos que dicho tópico contiene. Es decir, que, si el contenido de un tópico es de tipo numérico string, no aceptará datos publicados de otro tipo numérico, como enteros o booleanos. Asimismo, tal y como se verá en el siguiente apartado de comunicaciones, no únicamente se emplean los tópicos para realizar el intercambio de información entre nodos, puesto que también existen los servicios y las acciones [31].

En lo que refiere a los paquetes, todo software que se integra en ROS está organizado en paquetes. Un paquete contiene todo lo necesario para realizar una funcionalidad definida, es decir, que, si un paquete en cuestión tiene como objetivo la lectura de datos de un sensor, este tendrá incluido en su interior todo lo necesario para poder llevar a cabo dicha actividad de lectura de datos. Un paquete ROS puede contener tanto nodos, librerías, datos de configuración, archivos de configuración como otros elementos que no necesariamente tengan relación directa con la funcionalidad del código integrado en el paquete. En esencia, se tratan de meras “carpetas” las cuales pueden ser integradas dentro de ROS y de las cuales se puede construir software. En

resumen, todos los recursos necesarios para definir la funcionalidad de cada uno de los nodos del sistema se recogen dentro de estos paquetes [32].

Las bolsas, mejor conocidas dentro del entorno de ROS como "bagfiles", son mecanismos de recopilación de datos los cuales nos permiten tanto guardar mensajes de ROS, como comportamientos. En lo referente a su primera funcionalidad, el guardar mensajes de ROS, se refiere a que son elementos que permiten registrar todos los datos publicados en los tópicos del sistema dentro de un intervalo de tiempo definido por el sistema, como una especie de "Data Logger". Por otro lado, en cuanto al registro de comportamientos, las bolsas habilitan la posibilidad de emular el comportamiento de un nodo, conjunto de nodos o un entorno de trabajo al completo en un intervalo de tiempo definido. Es decir, que una bolsa sería, en cierto modo, capaz de actuar como una especie de gemelo virtual de un elemento físico del sistema al completo, e incluso en rangos más pequeños, ser el gemelo de un nodo en concreto. Es por ello, que, las bolsas son elementos muy útiles tanto para la depuración de código como para la simulación y validación de sistemas [33].

Para comprender mejor la funcionalidad que cada uno de los elementos principales de ROS realizan dentro del entorno, se aporta el ejemplo de la *Figura 4.3,* la cual es una ampliación del ejemplo mostrado en la *Figura 4.2* del apartado anterior. Al igual que en el anterior ejemplo, se muestra una aplicación en la que, desde una cámara situada en un AGV, se busca procesar los datos de esta y de subir la imagen obtenida a la nube. Sin embargo, se cuenta con nuevos elementos que conforman el sistema, como lo son: el nodo maestro, cuya labor es la de simplemente actuar como registro de los nodos y tópicos del sistema; tres paquetes ROS, en los cuales se integran y empaquetan las distintas funcionalidades del sistema y ejemplo mostrado de la anterior *Figura 4.2;* y dos carpetas de configuración, que integran todo lo necesario para poder permitir la lectura y escritura de los periféricos cámara y antena.

Al igual que en el ejemplo anterior, se cuenta con tres nodos dedicados a la lectura, procesamiento y envío de la imagen, respectivamente. Cada uno de estos nodos, están integrados dentro de un paquete cada uno, como lo podrían estar dentro de un único paquete los tres. Cada uno de estos paquetes, tiene como objetivo la realización de cada uno de los procedimientos que se necesitan para la publicación de la imagen en la nube: lectura, procesamiento de imagen y envío de imagen. A su vez, se aprecia como uno de los tópicos se integra dentro del paquete B, mientras que el otro no está necesariamente definido dentro de ningún paquete. Sin embargo, pese a que se trata de unidades y empaquetamientos software independientes y distribuidos, el sistema sigue siendo completamente funcional, permitiendo la comunicación entre sus distintos elementos.

La integración de estos paquetes en el sistema permite proporcionar al sistema de la modularidad y distribución requerida por parte de las aplicaciones robóticas industriales. Tal es así, que los paquetes A y C son unidades software que, perfectamente, pueden emplearse en otros sistemas en los que intervengan los mismos periféricos o similares, sin la necesidad de reprogramar ningún parámetro interno. Esto, se debe a que cuentan en su interior con todo lo necesario para llevar a cabo el procesamiento de imagen y comunicación con la nube, representado en el ejemplo mediante las carpetas "Configuración Cámara" y "Configuración Antena".

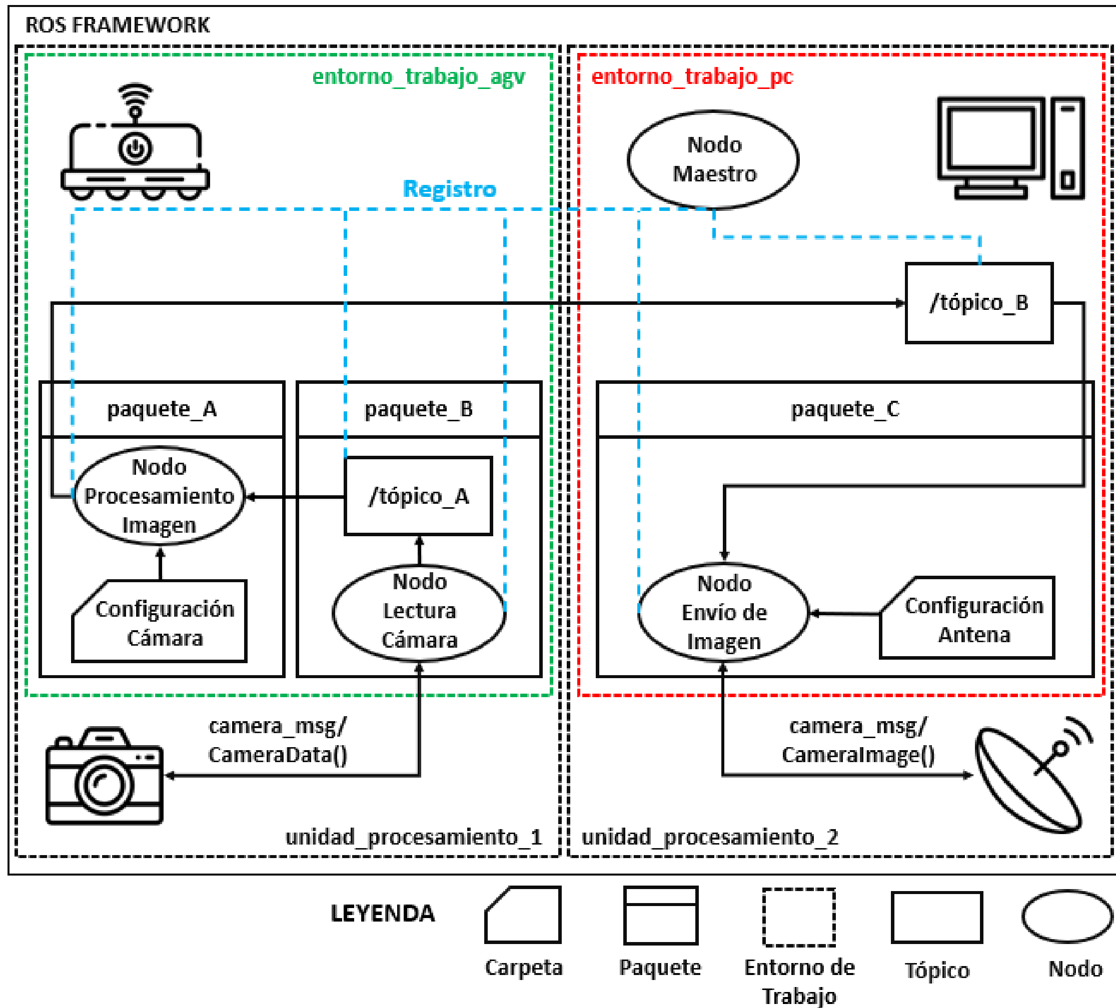


Figura 4.3 Ejemplo de Distribución de los Elementos Básicos de ROS

Del mismo modo, el ejemplo podría complicarse mucho más e integrar el resto de los elementos de ROS que de un modo u otro, toman parte en el sistema ROS. Elementos tales como: Manifiestos, Transformadas o Estructuras de Mensajes, Launchfiles o CMakelists. Sin embargo, estos elementos y los conceptos relacionados con ellos se tratarán a lo largo de este capítulo con otra clase de ejemplos e ilustraciones que faciliten su comprensión y su papel dentro del entorno de ROS.

4.1.2.1.2 Comunicaciones

Los nodos ROS pueden comunicarse entre ellos mediante tres métodos diferentes: A través de tópicos emulando el paradigma publicista-suscriptor; mediante servicios, emulando el paradigma cliente-servidor; o a través de acciones. Todos los nodos pueden emplear cualquiera de los tres métodos para comunicarse con el resto de los elementos del entorno generado por ROS. De hecho, un nodo puede tener los cuatro roles que se dan en la comunicación: Cliente, servidor, publicista y suscriptor [34]. A continuación, se muestran las principales diferencias entre los tres métodos:

En lo referente a los tópicos, tal y como se ha comentado anteriormente, se trata de elementos intermedios donde los nodos del entorno ROS pueden, o bien publicar datos, o bien suscribirse para leer los datos publicados en el tópico. De esta manera, se genera una estructura tipo publicista-suscriptor, como se aprecia en la *Figura 4.4.*

Figura 4.4 Estructura de una Comunicación Mediante Tópicos

Por otro lado, los servicios no necesitan de un elemento intermedio como lo es un tópico, puesto que la comunicación se da directamente entre dos nodos: uno cliente, y otro servidor, como se ilustra en la *Figura 4.5* A diferencia de los tópicos, dónde cualquier nodo del sistema publicar o suscribirse a un tópico, en la comunicación por servicio, únicamente los dos nodos involucrados pueden tomar parte de la comunicación. Cuenta con la característica de que, una vez iniciado el servicio, el cliente únicamente tendrá que esperar la respuesta del servidor para seguir con sus actividades. Es decir, que el cliente se quedará bloqueado hasta recibir la respuesta del servidor.

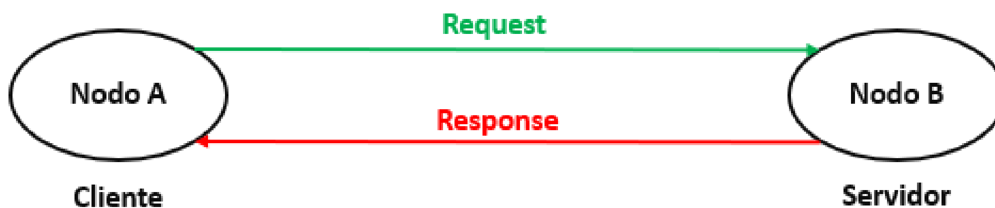


Figura 4.5 Estructura de una Comunicación por Servicio

Finalmente, las acciones, al igual que los servicios, se dan entre dos nodos: uno cliente y otro servidor. Cuenta con dos principales diferencias con respecto a los servicios: la primera, que el nodo cliente no espera a que el nodo servidor termine de hacer la actividad que le ha requerido, sino que es capaz de seguir ejecutando sus actividades de manera paralela al nodo servidor; y la segunda, que emplea tópicos para la comunicación entre el cliente y el servidor, concretamente cinco. Estos cinco tópicos, son: Goal, depende del tipo de mensaje de la acción, pero se refiere al objetivo que el cliente le pide al servidor que realice; Cancel, empleado principalmente para detener la realización de la acción mientras esta está en curso; Status, suele emplearse para devolver un “estado” en el que se encuentra el servidor mientras ejecuta la acción requerida; Result, flanco que indica que la acción ha finalizado; y Feedback, que simplemente son los argumentos de salida de la acción. La estructura típica de una comunicación basada en acción se muestra en la *Figura 4.6.*

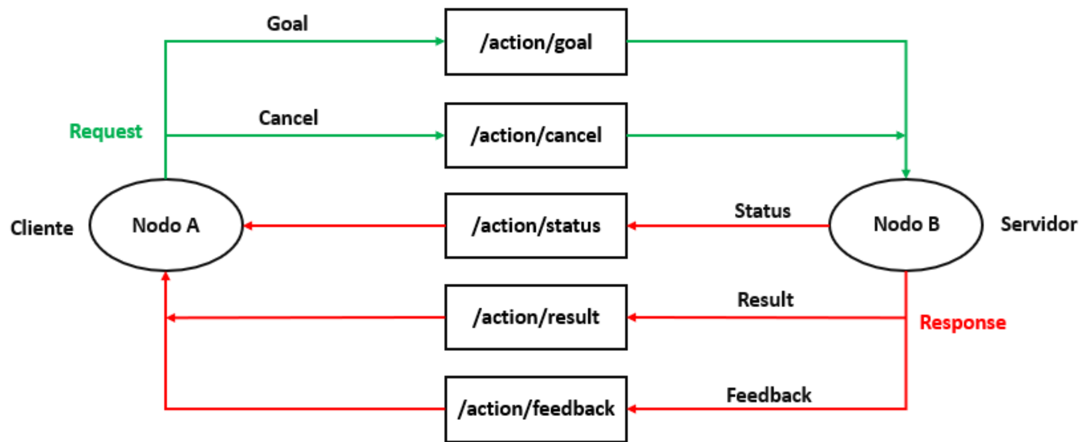


Figura 4.6 Estructura de una Comunicación por Acción

Cabe destacar, que ROS emplea un derivado del protocolo de comunicaciones TCP/IP para llevar a cabo el correcto transporte de la información, concretamente, TCPROS.

4.1.2.1.3 Tipos de Mensajes

Al igual que en otros entornos, ROS cuenta con la posibilidad de definir diferentes estructuras de datos para la comunicación [35]. Todos los tópicos de ROS cuentan con un tipo de mensaje asociado, el equivalente a un tipo da dato en cualquier lenguaje de programación. Es decir, en el caso en el que un nodo quiera publicar en un tópico, deberá hacerlo siguiendo la misma estructura de datos asociada a éste. Por ejemplo, en el caso de que dos nodos quieran suscribirse y publicar en un tópico asociado a una estructura de datos "std_msgs/String", ambos nodos deberán leer y escribir siguiendo el orden de los campos asociados a dicha estructura, tal y como se muestra en la *Figura 4.7*.

Figura 4.7 Comunicación Nodos Mediante Mensaje Tipo std_msgs/String

En la figura anterior, se ilustra la comunicación de dos nodos a través del tipo de mensaje "std_msgs/String", que, en esencia, se trata de un intercambio de datos de tipo string. Como puede apreciarse, el contenido del mensaje que el nodo A publica en el tópico es "hi". De este modo, el nodo B puede leer el contenido del mensaje publicado por el nodo A en el tópico, empleando la misma estructura de datos que ha empleado el nodo A para publicar.

Sin embargo, existen cientos de tipos de mensajes dentro del entorno que proporciona ROS de manera estándar, más allá de los tipos sencillos como std_msgs/String. Muchos de ellos están declarados dentro del entorno de forma predefinida, siendo únicamente necesario el referenciar el tipo de mensaje que quiera emplearse dentro de la codificación del nodo para poder hacer uso de dichas estructuras de datos. Para ello, ha de saberse que el primer campo de std_msgs/String hace referencia al paquete dónde reside el tipo de mensaje, mientras que el segundo campo se refiere al tipo de dato que se quiera emplear. En pocas palabras, std_msgs/String reside dentro del paquete std_msgs, el paquete que proporciona ROS para las estructuras de datos estándar, mientras que el tipo de dato que se empleará será un string. Es el nodo que primero publica en un tópico el que define el tipo de datos que dicho tópico tendrá, no aquel que se suscribe.

Por otro lado, al igual que se emplean distintas estructuras de datos para la comunicación tipo publicista-suscritor, también se emplean diversas estructuras para las comunicaciones basadas en cliente-servidor, tanto en servicios como en acciones. Los servicios deben de definir la estructura de los argumentos de entrada del servicio, es decir, los datos que el cliente le manda al servidor, y los argumentos de salida del servicio, es decir, los datos que el servidor le devuelve al cliente. Dentro de ROS, los argumentos de entrada en un servicio se denominan como request, mientras que los argumentos de salida se definen como response. Ambos, se definen dentro de una misma estructura de datos, de manera que si se emplea un servicio del tipo "std_srvs/SetBool", los campos de ambos argumentos de entrada y de salida, estarán definidos dentro del paquete std_srvs y recogidos dentro del tipo de dato o tipo de servicio "SetBool". La comunicación entre nodos por servicio se muestra en la siguiente *Figura 4.8.* En este ejemplo, se puede ver cómo la estructura std_srvs/SetBool cuenta con campos distintos tanto para el request como para response. El cliente, o nodo A, cuando requiere de un servicio por parte del servidor, o nodo B, le manda un boolean de con contenido "True". El nodo B, realizará internamente el servicio requerido por el nodo A, su cliente, a lo que este le responderá mediante un boolean y un string, que, en esencia, le indica al cliente que su petición se ha realizado correctamente.

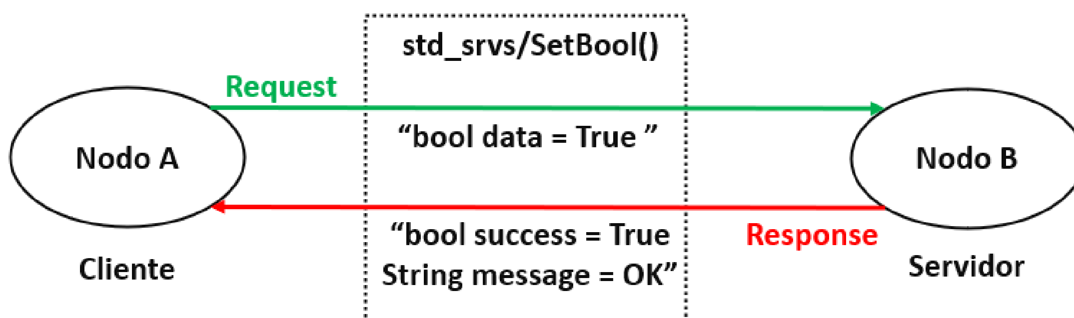


Figura 4.8 Comunicación Nodos Mediante Servicio Tipo std_srvs/SetBool

Con las acciones, en cambio, la comunicación no se realiza directamente entre nodos, sino que intervienen como intermediarios cinco tópicos que recogen los cinco campos de la estructura del mensaje empleada para la acción. La acción, define para cada uno de sus cinco campos el tipo de mensaje que va a emplear, como se muestra en la *Figura 4.9.*

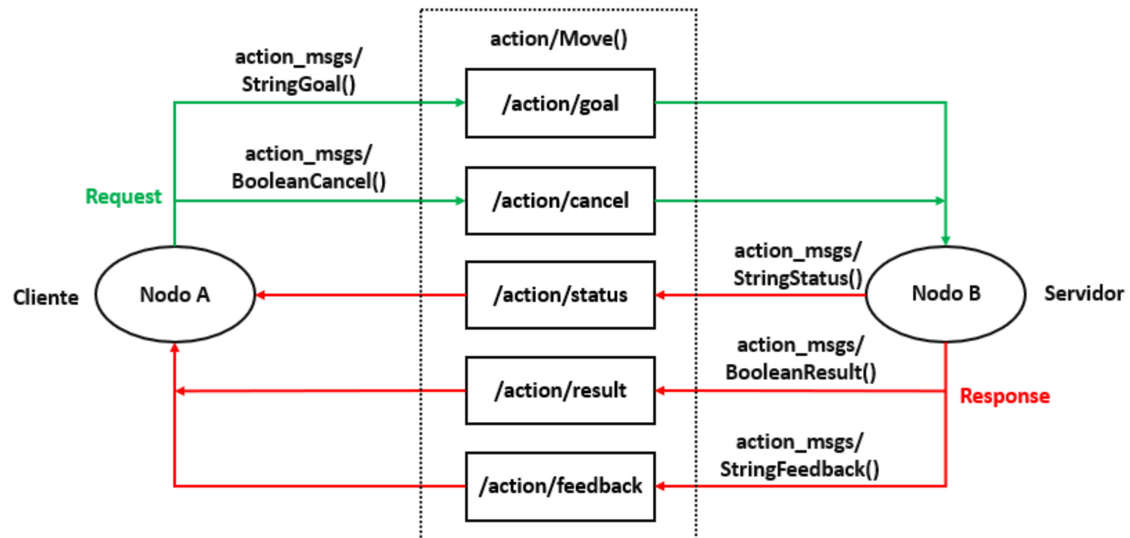


Figura 4.9 Comunicación Nodos Mediante Acción tipo action/Move()

Del mismo modo que existen una serie de mensajes predefinidos por ROS, también cabe la posibilidad de crear mensajes que sigan una estructura de datos personalizada [36]. Sin embargo, ROS únicamente permite personalizar los campos de dicha estructura de datos y su orden. El contenido de los campos y su tipo deberán de estar definidos dentro del entorno ROS y ser reconocibles por éste. Asimismo, al igual que es posible customizar las estructuras de datos de los mensajes que se vayan a emplear, también se permite modificar y personalizar las estructuras empleadas en la comunicación de nodos mediante servicios y acciones. Tal y como se especificará en el siguiente apartado, los mensajes, servicios y acciones cuya estructura de datos esté customizada, residen dentro del paquete o paquetes que vayan a emplear dicho tipo de mensaje.

4.1.2.1.4 Entornos de Trabajo

Dentro del entorno de ROS, en el cual puede haber múltiples equipos independientes dentro de una misma red Wi-Fi, pueden existir diferentes entornos de trabajo. Estos entornos de trabajo son capaces de implementar y abstraer todas las funcionalidades mínimas y necesarias de ROS para integrar varios equipos. A su vez, están formados por múltiples paquetes, en los que residen los archivos de configuración, tipos de mensajes y todo el código ejecutable que los nodos emplearán para realizar sus actividades y tareas. En pocas palabras, permiten generar pequeños entornos ROS, dentro de toda la red y entorno de ROS [37].

De esta forma, se pueden modularizar los distintos desarrollos ROS para las características de cada uno de los sistemas que se quieran integrar dentro de la red de ROS, abstrayéndose de aquellas funcionalidades que no le sean necesarias o bien generando nuevas herramientas que el resto de los equipos del sistema no necesiten. A pesar de ello, los distintos entornos son capaces de comunicarse entre ellos, e incluso generarse varios entornos de trabajo dentro de un mismo equipo, como se ilustra en la *Figura 4.10.*

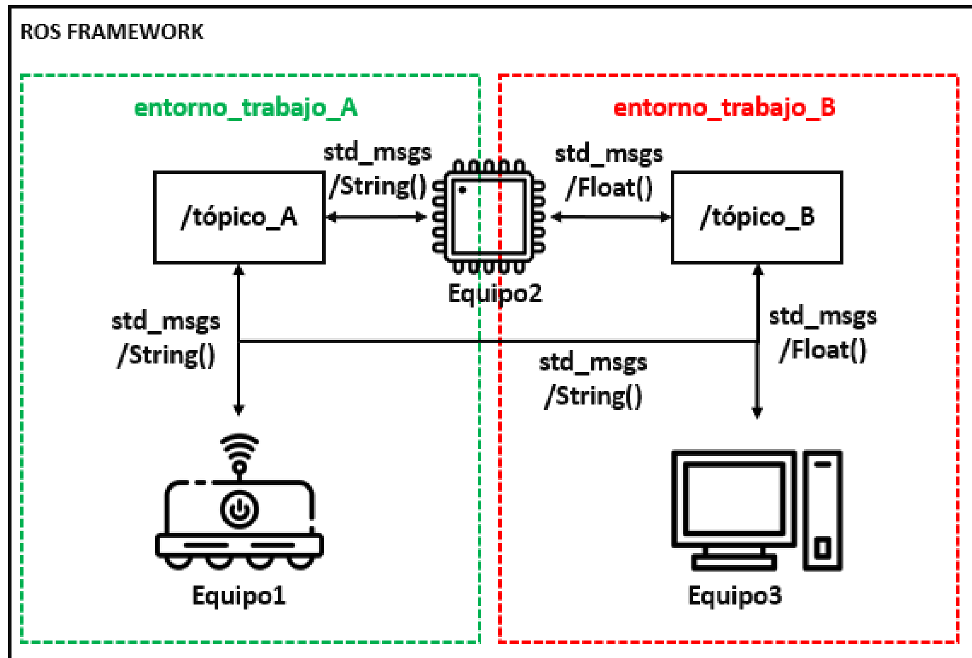


Figura 4.10 Distribución de varios Entornos de Trabajo dentro de una misma Red ROS

Como se puede ver en el anterior esquema, los entornos de trabajo pueden interconectarse y comunicarse entre sí mediante los tópicos, sin importar el tipo de equipo en el que se esté ejecutando ROS. Asimismo, pueden emplearse servicios y acciones para la comunicación entre los distintos entornos de trabajo, siempre y cuando, al igual que con los tópicos, el tipo de mensaje o estructura de datos empleada sea la misma para ambos entornos. Cabe destacar, que puede haber equipos en los que se integren varios entornos de trabajo, como es el caso del "Equipo2" de la *Figura 4.10*.

La distribución de los entornos de trabajo ROS se puede resumir en el esquema mostrado en la *Figura 4.11*. En dicha figura, puede apreciarse que los nodos ROS "residen" dentro de los paquetes de los entornos de trabajo. Los paquetes, a su vez, se definen en la carpeta "src" o source del entorno de trabajo, que, junto con las carpetas de "build", "install" y "devel", forman el conjunto del entorno. Las funcionalidades de cada una de las carpetas se resumen en:

- **src**: La carpeta "src", también conocida como "source space", se trata del lugar donde residen los paquetes propios del entorno de trabajo. También puede encontrarse todo el código relacionado para la construcción de paquetes.

- **build**: En esta carpeta residen todas las herramientas necesarias para construir los paquetes software que residen en la carpeta "src". Lo hará mediante la herramienta CMake (Cross Platform Make), una herramienta multiplataforma capaz de gestionar la construcción, pruebas y empaquetamiento de software.

- **devel**: También conocido como "development space", se trata de la carpeta en la cual el software construido se asienta antes de ser instalado en el entorno. En él, residen todas las dependencias del entorno de trabajo, de manera que el equipo o terminal que quiera trabajar dentro de dicho entorno, pueda hacerlo. De todos los archivos dentro de esta carpeta, destaca "setup.bash", un archivo encargado de declarar todas las dependencias, variables y directorios propios empleados por el entorno de trabajo.

- **install**: Se trata de la carpeta dónde se sitúan los paquetes software una vez son construidos, asentados e instalados dentro del entorno de ROS. En resumen, se trata del lugar de instalación de los paquetes software construidos que residen dentro de la carpeta "src", una vez han pasado a través de "build" y "devel". No necesariamente tiene que residir dentro del entorno de trabajo.

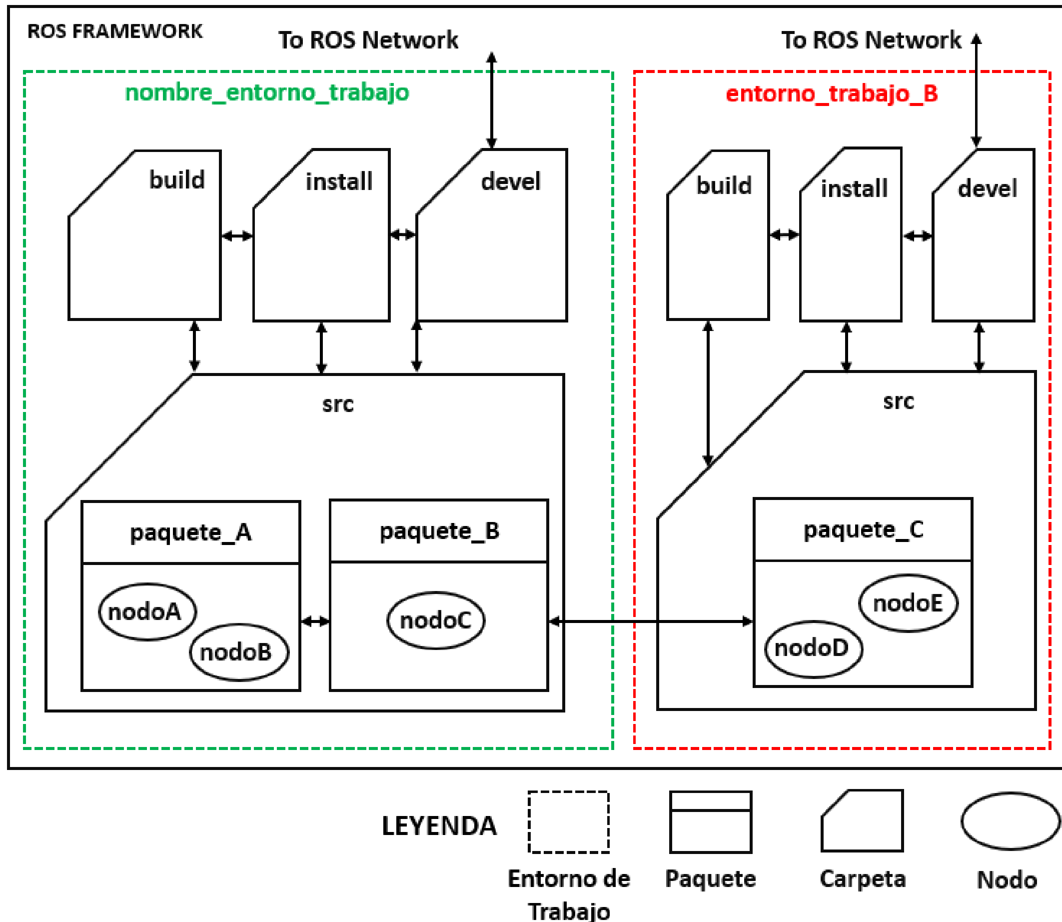


Figura 4.11 Distribución de un Entorno de Trabajo ROS

Tal y como se muestra en la *Figura 4.12* los paquetes cuentan con múltiples elementos. Todos ellos se repiten en todos los paquetes que se generen, siendo todos ellos necesarios para la correcta ejecución de un paquete; salvo los archivos **msg**, **srv** y **action**, los cuales son opcionales y únicamente se incluyen si se quieren integrar mensajes, servicios o acciones customizadas, respectivamente. Las funciones de cada uno de los elementos básicos de un paquete son las siguientes:

- **CMakeList.txt.** En este archivo se definen todas las pautas y condiciones necesarias para construir un paquete software mediante la herramienta. En él, se definen tanto las dependencias con otros paquetes, los tipos de mensajes, servicios y acciones que se van a emplear y las propiedades generales del paquete [38].

- **package.xml.** También conocido como "package manifest", se trata de un archivo ejecutable escrito en XML (Extensible Markup Language) cuya tarea principal es la de

definir el nombre del paquete, la versión, el nombre del autor o autores, encargados del mantenimiento de este y las dependencias con otros paquetes del entorno [39].

- **launch.**: En esta carpeta se recogen los archivos de tipo ".launch". Se tratan de archivos ejecutables los cuales sientan las pautas de ejecución de una funcionalidad dentro del paquete. Es decir, según qué archivo ".launch" se ejecute, se lanzarán una serie de nodos u otros, según lo que se indique dentro del archivo ejecutable lanzado. A su vez, estos archivos ejecutables también sirven como archivos de configuración de parámetros globales, de forma que varios paquetes puedan emplear las mismas variables. Estos archivos, se pueden lanzar mediante el comando "roslaunch" desde una terminal, indicando el nombre del paquete en el que se encuentra y el nombre del archivo. A su vez, los ".launch" son capaces de lanzar otros archivos ".launch", sin importar si estos se encuentran dentro del mismo paquete o no [40].
- **src.** En esta carpeta, reside el código fuente correspondiente al paquete. Este código, será el que emplearán los nodos que lancen los archivos de tipo ".launch" para llevar a cabo sus tareas y procedimientos. En esencia, se trata de la carpeta donde residen todas las funcionalidades que dicho paquete puede aportar en formato de código. Dentro de esta carpeta, pueden coexistir scripts escritos en varios lenguajes de programación, como Python, C++ o Java.
- **msg.** Es la carpeta en la cual reside la estructura de los mensajes customizados.
- **srv.** Se trata de la carpeta en la cual reside la estructura de los servicios customizados.
- **action.** Carpeta en la cual reside la estructura de las acciones customizadas.

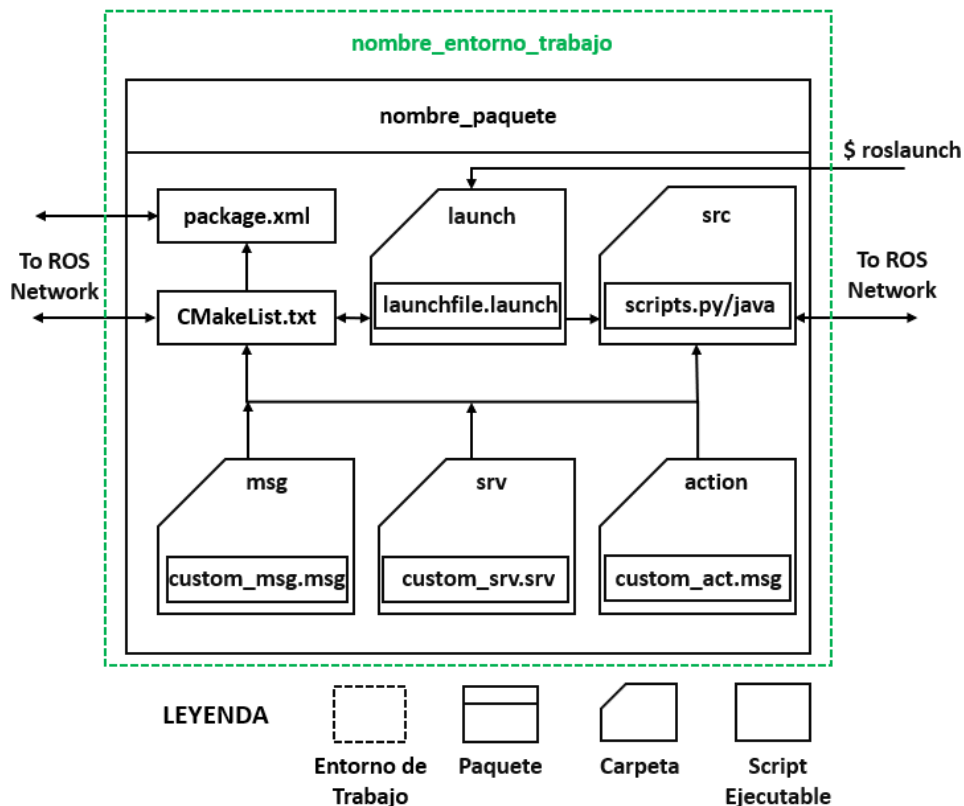


Figura 4.12 Distribución de un Paquete ROS

Cabe destacar que obviamente es posible crear otro tipo de carpetas dentro de un paquete ROS. Es más, a través de los archivos ".launch", se pueden referenciar estas carpetas de forma que añadan funcionalidades extra al paquete. Como se verá más adelante, este tipo de carpetas adicionales suelen emplearse para guardar distintos tipos de configuraciones, incluir librerías que no sean propias del entorno de ROS, o simplemente carpetas que recojan documentos multimedia.

4.1.2.1.5 Transformadas

En ROS existen una serie de nodos cuya única labor es la de indicar la posición de todos los elementos de un sistema con respecto a un punto de referencia. En el caso de un robot antropomorfo de seis grados de libertad, las transformadas serían las responsables de indicar y calcular la posición relativa de los seis sistemas de referencia móviles asociados a cada grado de libertad con respecto al sistema de referencia central o sistema de referencia mundo. Asimismo, las transformadas también permiten el calcular la posición relativa de elementos que no formen parte necesariamente del conjunto físico del elemento asociado al sistema de referencia mundo, como podrían ser: puntos definidos en el espacio, sensores externos al propio robot como cámaras o incluso otras unidades robóticas que pertenezcan al sistema [41], [42].

Para facilitar el entendimiento de las funciones que ocupan las transformadas, se aporta la *Figura 4.13*. En dicha ilustración, puede observarse una unidad de transporte, concretamente un Turtlebot2, equipado con dos periféricos: un sensor LiDAR de la gama Hokuyo y una cámara Kinect. Se muestran tres sistemas de referencia: *base_link*, asociado a la base móvil del AGV; *camera_link*, asociado a la cámara Kinect; y *hokuyo_link_frame*, asociado al sensor LiDAR. De los tres sistemas de coordenadas, el sistema de referencia central o mundo sería base_link, mientras que los otros dos sistemas de referencia definen la posición relativa de sus respectivos periféricos con respecto al sistema base_link. Ambos sistemas, son solidarios a base_link, es decir, que se desplazan junto a él, por lo que, en caso de moverse el AGV, tanto camera_link como hokuyo_link_frame se desplazarán exactamente la misma distancia que base_link. En pocas palabras, las transformadas de ROS permiten obtener las matrices de transformación entre los distintos sistemas de referencia recogidos dentro del entorno, de manera que sea posible definir la posición física de cada uno de los sistemas en el espacio.

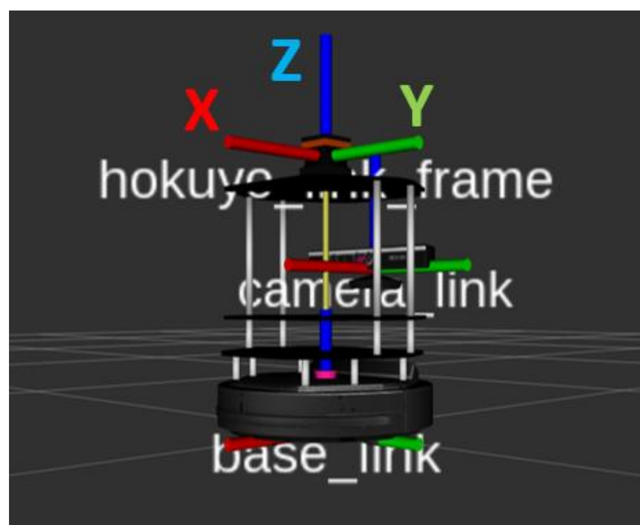


Figura 4.13 Ejemplo de Árbol de Transformadas en AGV

Los nodos, tópicos y mensajes asociados al cálculo y definición de las transformadas suelen estar indicados mediante la abreviación "tf". Estos, es habitual definirlos en los archivos ".launch", aunque también pueden definirse en los archivos ".urdf", los cuales se comentarán más adelante en el apartado asociado a Gazebo. De hecho, para cada sistema de coordenadas que se quiera incluir dentro del entorno proporcionado por ROS, ha de definirse un nodo del tipo "static_transform_publisher", cuya única labor es la de publicar de manera periódica una serie de datos que permitan calcular la posición del sistema definido. Concretamente, estos datos a publicar corresponden a los campos asociados al tipo de mensaje que emplea el tópico de tipo "static_transform_publisher", mostrados en la *Tabla 4.2*.

Campo	Contenido
x	Posición del sistema de referencia en metros en el eje x con respecto a su sistema de referencia padre.
y	Posición del sistema de referencia en metros en el eje y con respecto a su sistema de referencia padre.
z	Posición del sistema de referencia en metros en el eje z con respecto a su sistema de referencia padre.
qx	Rotación del sistema de referencia en el cuaternión x.
qy	Rotación del sistema de referencia en el cuaternión y.
qz	Rotación del sistema de referencia en el cuaternión z.
qw	Rotación del sistema de referencia en el cuaternión w.
frame_id	Nombre del sistema de referencia padre al que se asocia el sistema de referencia que se esté definiendo.
child_frame_id	Nombre que va a tomar el sistema de referencia que se está definiendo dentro del entorno de ROS.
period_in_ms	Frecuencia a la cual el nodo va a publicar la matriz de transformación que une el sistema padre o "frame_id" y el sistema al que representa o "child_frame_id"

Tabla 4.2 Campos del Tipo de Mensaje del Nodo static_transform_publisher/Tf()

De esta forma, si se quisiera definir un nodo del tipo *static_transform_publisher* de la cámara Kinect de la *Figura 4.13* con respecto a la base, es decir, camera_link con respecto a base_link, habría que definirlo tal y como se indica en la *Tabla 4.3*.

Campo	Contenido
x	-0.10 m
y	0.0 m
z	0.29 m
qx	0
qy	0
qz	0
qw	1
frame_id	camera_link
child_frame_id	base_link
period_in_ms	100

Tabla 4.3 Transformada de camera_link respecto a base_link

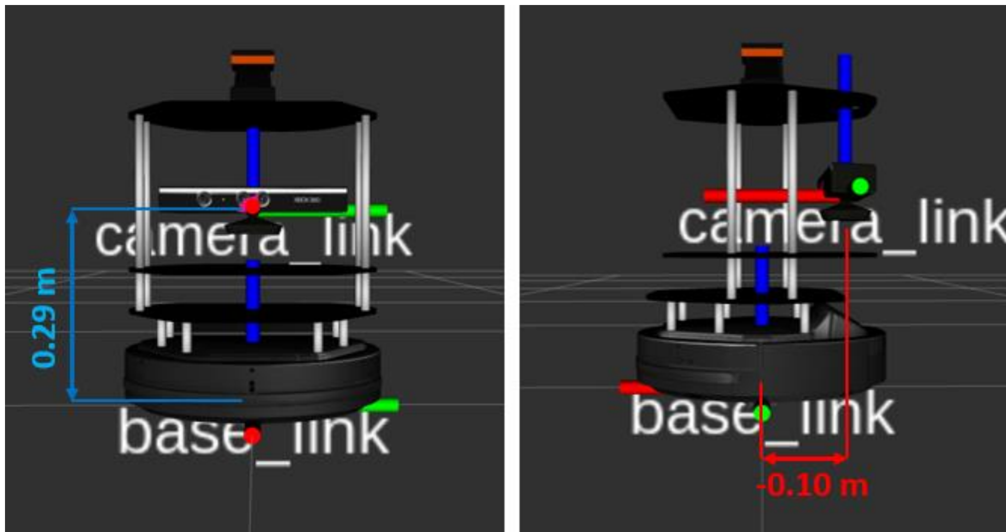


Figura 4.14 Posición relativa de camera_link respecto a base_link

El conjunto de sistemas de referencia y transformadas que componen un sistema de ROS se recoge dentro del árbol de transformadas. En dicho árbol, se define tanto la jerarquía entre todos los sistemas de referencia como toda la información necesaria para llevar a cabo el cálculo de sus matrices de transformación. En este árbol, pese a contar con decenas de sistemas de referencia, es necesario comprender la diferencia entre dos sistemas que suelen darse habitualmente en los proyectos de ROS:

- **base_link**: Tal y como se ha visto anteriormente, se asocia al punto central de la base móvil de la unidad de transporte, no necesariamente a su centro de gravedad. Se trata del sistema de referencia padre del cual dependen todos los elementos que conforman el cuerpo físico del robot.
- **base_footprint**: Es el sistema de referencia cero del robot, es decir, aquel que define cuál es su posición con respecto al suelo y sistema del cual depende base_link. Su principal diferencia, es que base_link se sitúa por encima de las ruedas del robot, mientras que base_footprint está directamente ligado al suelo.

4.1.2.1.6 Odometría

La odometría se trata de una funcionalidad con la que cuentan la amplia mayoría de los robots o elementos que implementen tareas de navegación o desplazamiento, sean autónomas o no. En pocas palabras, la odometría calcula cuál es la posición relativa del robot con respecto a un punto inicial o sistema de referencia fijo. Habitualmente, este sistema de referencia fijo es aquel punto en el cual el robot se ha inicializado, es decir, su punto de partida. Se trata de una funcionalidad indispensable para poder llevar a cabo algunos algoritmos de navegación, así como la generación de SLAM.

Esta funcionalidad hace uso de distintos sensores como giroscopios, resolvers o encoders para calcular cuánto se ha desplazado el robot y en qué eje o ejes cartesianos. De esta manera, de forma iterativa, el sistema es capaz de determinar el punto exacto en el que se encuentra con

respecto al sistema de referencia mundo o sistema de referencia inicial. Si el robot se desplaza un metro en el eje x, la odometría indicará que el punto actual en el que se encuentra el robot está a un metro en el eje x. Por otro lado, si el robot vuelve a la posición inicial, la odometría indicará que el desplazamiento dado en el eje x es cero, puesto que el punto actual en el que se encuentra el robot y el punto de referencia inicial coinciden sobre el plano, pese a haber un desplazamiento previo.

Concretamente, la odometría dentro de las aplicaciones desarrolladas en ROS, determina la distancia entre el sistema de referencia *base_footprint*, y el sistema odom, tal y como se muestra en los ejemplos de la *Figura 4.15* y la *Figura 4.16*. En ambas imágenes, se muestra a un AGV en un pasillo, donde en la primera imagen el transporte permanece en su punto inicial y en la segunda tras desplazarse unos pocos metros hacia atrás. En ROS, se denomina al sistema de referencia odom como el sistema de referencia el cual se toma como posición inicial, es decir, el sistema fijo del cual la odometría ha de calcular su posición relativa. Es decir, la odometría determinará la posición relativa de base_footprint, que se mueve solidario al AGV, con respecto al sistema odom.

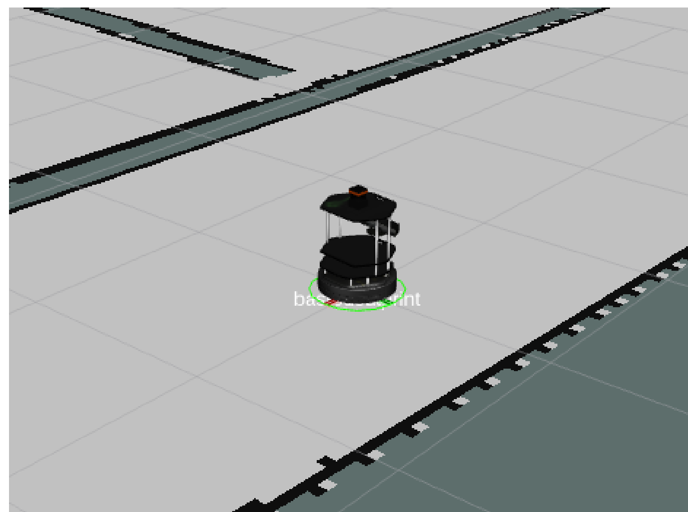


Figura 4.15 Relación base_footprint y odom inicial

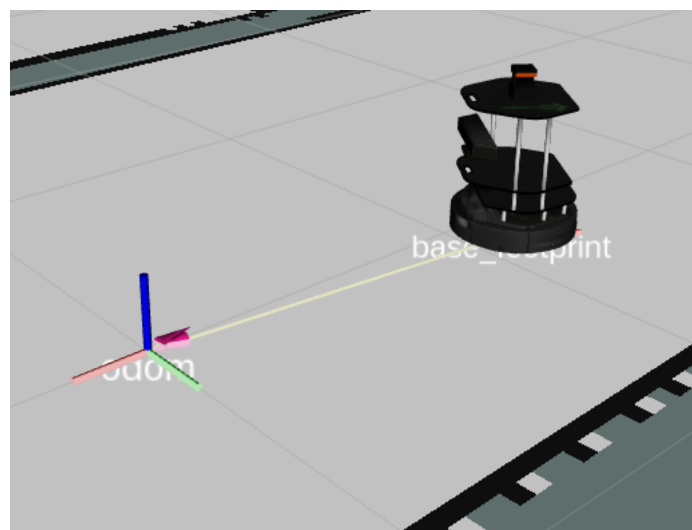


Figura 4.16 Relación base_footprint y odom tras un desplazamiento

En ROS, el nodo o nodos encargados de proporcionar la odometría del sistema lo harán a través del tópico "/odom" o derivados, empleando mensajes con estructuras de datos del tipo nav_msgs/Odometry. Esta estructura, indicará la posición del robot en los tres ejes cartesianos en metros, mientras que la orientación propia del robot con respecto a su eje vertical cartesiano z lo hará en cuaterniones.

4.1.2.2 ROS Navigation Stack

El ROS Navigation Stack, se trata de un paquete de ROS el cual permite la navegación de un robot de manera autónoma. Este paquete está diseñado principalmente para robots equipados con un sensor LiDAR planar y con una distribución física relativamente simétrica. La distribución general de todos los nodos que lo conforman se muestra en la *Figura 4.17* [43] ,[44] [45].

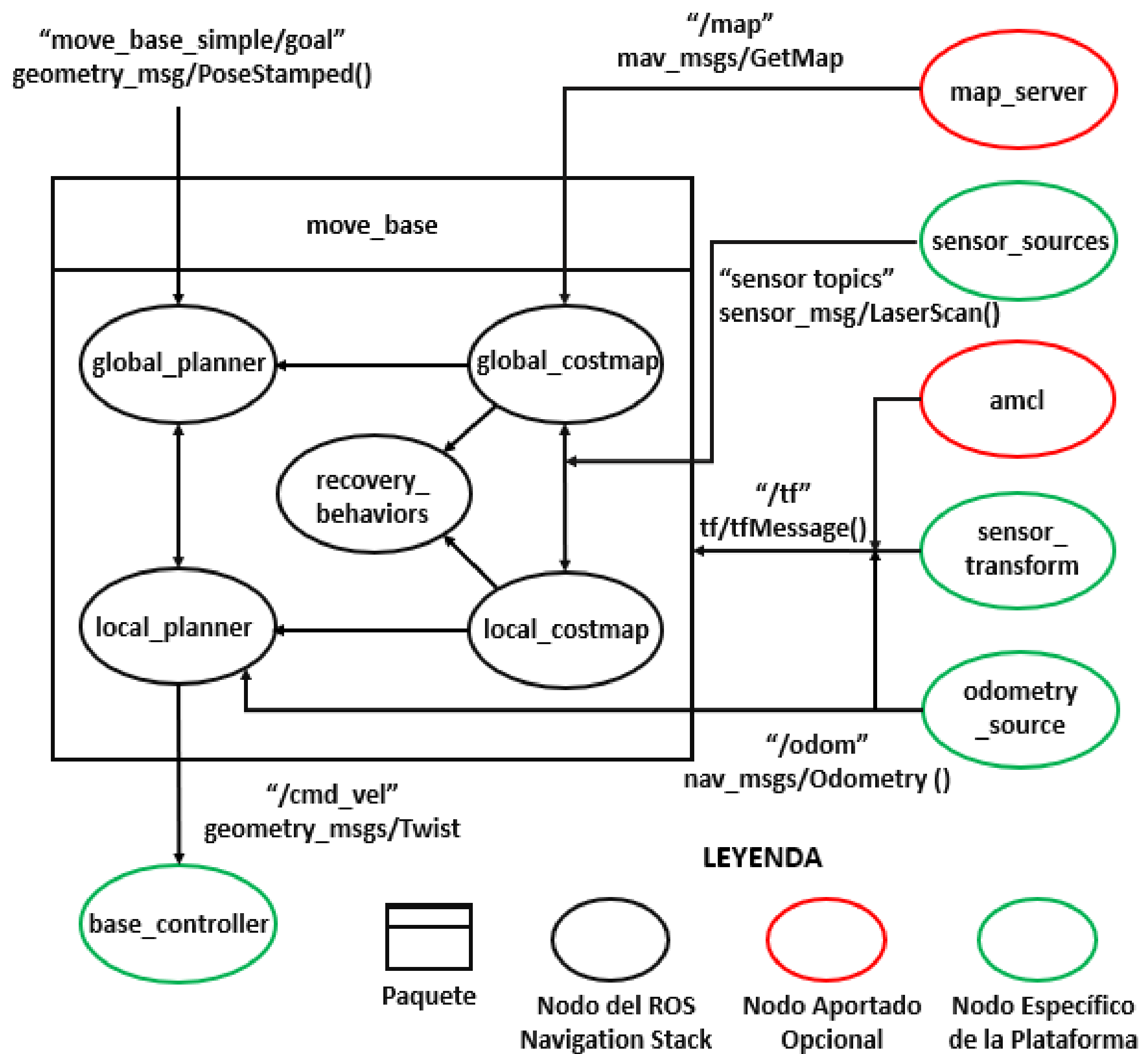


Figura 4.17 Distribución general de los nodos del ROS Navigation Stack

Tal y como se puede observar en la *Figura 4.17*, el paquete diferencia tres clases de nodos: Nodos del ROS Navigation Stack, los cuales están integrados dentro del paquete de *move_base*; nodos aportados opcionalmente, se trata de nodos necesarios para la navegación autónoma pero que pueden ser sustituidos por equivalentes siempre y cuando cumplan los requisitos de comunicación; y los nodos específicos de la plataforma, los cuales dependerán del modelo de la unidad de transporte o arquitectura en la que se integre el sistema.

El paquete move_base es el elemento central del ROS Navigation Stack y en el que se ejecutan todas las tareas relacionadas con la navegación. En resumen, este paquete recibe una coordenada objetivo a la que trasladarse a través de una acción de ROS y utilizando un mensaje del tipo geometry_msgs/PoseStamped. Posteriormente, el sistema hará uso del resto de argumentos de entrada como las lecturas de los sensores o la odometría del robot para calcular y ejecutar la trayectoria que une el punto en el que se encuentra el AGV y la coordenada objetivo. Para llegar al objetivo, el sistema estará continuamente realimentándose de todas sus entradas para publicar de manera periódica comandos del tipo geometry_msgs/Twist. Es decir, comandos los cuales el nodo encargado del movimiento de la unidad de transporte leerá para saber cuánto y cómo de rápido ha de moverse, de manera que pueda ejecutar la trayectoria calculada por el paquete move_base y desplazarse hasta la coordenada objetivo.

En esencia, el objetivo principal de move_base es el de actuar como controlador y cerciorarse de que el AGV ejecuta la referencia de entrada, es decir, la trayectoria calculada que une el punto inicial con la coordenada objetivo, minimizando el error entre la referencia y la salida, y esquivando los posibles obstáculos que puedan encontrarse. En lo relativo a los nodos que componen el esquema mostrado de la *Figura 4.17* sus funcionalidades son las siguientes:

- **map_server**: Nodo encargado de proporcionar el mapa del entorno en el cual la unidad de transporte va a navegar.
- **sensor_sources**: Se trata del conjunto de nodos que gestionan los sensores de la unidad de transporte, concretamente sensores LiDAR o cámaras RGB capaces de proporcionar la profundidad de la imagen.
- **amcl**: Nodo encargado de ejecutar el algoritmo de localización del AGV en el mapa.
- **sensor_transform**: Nodo o conjunto de nodos cuya responsabilidad es la de actualizar de manera periódica la posición de todas las transformadas que formen parte de la unidad de transporte.
- **odometry_source**: Nodo encargado de proporcionar la odometría de la unidad de transporte.
- **base_controller**: Nodo responsable de gestionar el cuerpo del AGV, concretamente, la cantidad de par que se aporta a los motores y su desplazamiento, además de otras funcionalidades propias de la unidad de transporte.
- **global_planner**: Nodo que calcula el plan global o trayectoria a seguir desde el punto inicial de la unidad de transporte hasta la coordenada objetivo.
- **local_planner**: Nodo que se encarga de calcular la trayectoria a seguir para poder llevar a cabo el plan del global_planner según lo que percibe del entorno.
- **global_costmap**: Se trata del nodo el cual calcula el mapa de coste global.
- **local_costmap**: Nodo encargado de calcular el mapa de coste local.

- **recovery_behavior**: En caso de que alguno de los planificadores no encuentre una trayectoria viable para alcanzar el objetivo, se ejecutará este nodo, siempre y cuando esté habilitado. Su función, es la de girar el AGV sobre sí mismo hasta recalibrar el algoritmo de localización del nodo amcl, de manera que pueda encontrar otro camino a seguir.

En resumen, el ROS Navigation Stack provee de las tres funcionalidades básicas para realizar navegación autónoma de manera óptima: mapeado del entorno, localización de la unidad de navegación en el entorno y planificación de trayectorias. Es por ello, que se procede a continuación a exponer cada una de dichas funcionalidades.

4.1.2.2.1 Mapeado

Para llevar a cabo una navegación autónoma, es imprescindible contar con un mapa del entorno sobre el que se va a operar. Pese a que existen algoritmos de navegación autónoma que permiten el realizar trayectorias sin la presencia de un mapa, la gran mayoría de estos algoritmos requieren de dicho mapa. Tal es así, que el ROS Navigation Stack cuenta con un algoritmo integrado dentro de sus funcionalidades que permite el mapeado del entorno, concretamente, el algoritmo Gmapping [46], [47].

Gmapping es una técnica basada en una técnica SLAM, concretamente la técnica SLAM de Filtro de Partículas Rao-Blackwellize. En esencia, este algoritmo representa la unidad de transporte o elemento de navegación como una serie de puntos o partículas en las cuales el robot podría estar situado. Este conjunto de puntos genera una especie de nube, la cual, en su conjunto, representan al cuerpo del AGV en el mapa del entorno.

Del mismo modo Gmapping representa el entorno que quiere reflejarse en el mapa mediante puntos que conforman la conocida como "Occupacy Grid". Esta red de ocupación representa todos los puntos del entorno en un rango de valores desde 0 hasta 100, dónde 0 significa que el punto está completamente libre y sin obstáculos y 100 que está ocupado. Esta codificación se publica en el tópico /map, generado por el ROS Navigation Stack, tópico el cual cuando se visualiza su Occupacy Grid tiene el aspecto mostrado en la *Figura 4.18* donde los puntos negros indican los obstáculos o delimitaciones del mapa y los puntos grises espacio libre por el cual el transporte puede desplazarse.

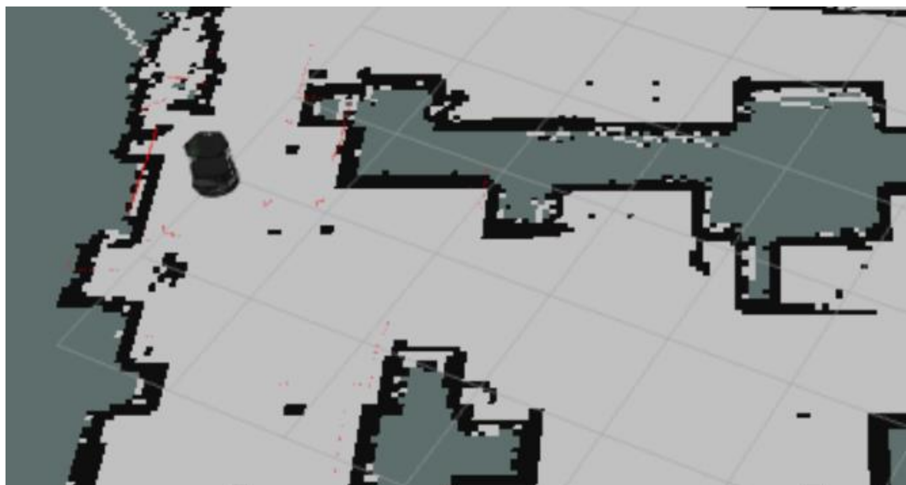


Figura 4.18 Mapa Convencional Generado por Gmapping

Pese a existir diversos algoritmos y técnicas que emplean SLAM, todas ellas cuentan con un funcionamiento similar en el que toman parte los mismos elementos. Al fin y al cabo, SLAM no es más que un método que permite la creación de un mapa y la localización de un vehículo autónomo en el mismo. Desde su primera aparición, los elementos que toman parte en dicha técnica no han variado en exceso, generando el lazo de control y diagrama de bloques que puede apreciarse en la *Figura 4.19* y el cual se aplica para el caso de Gmapping.

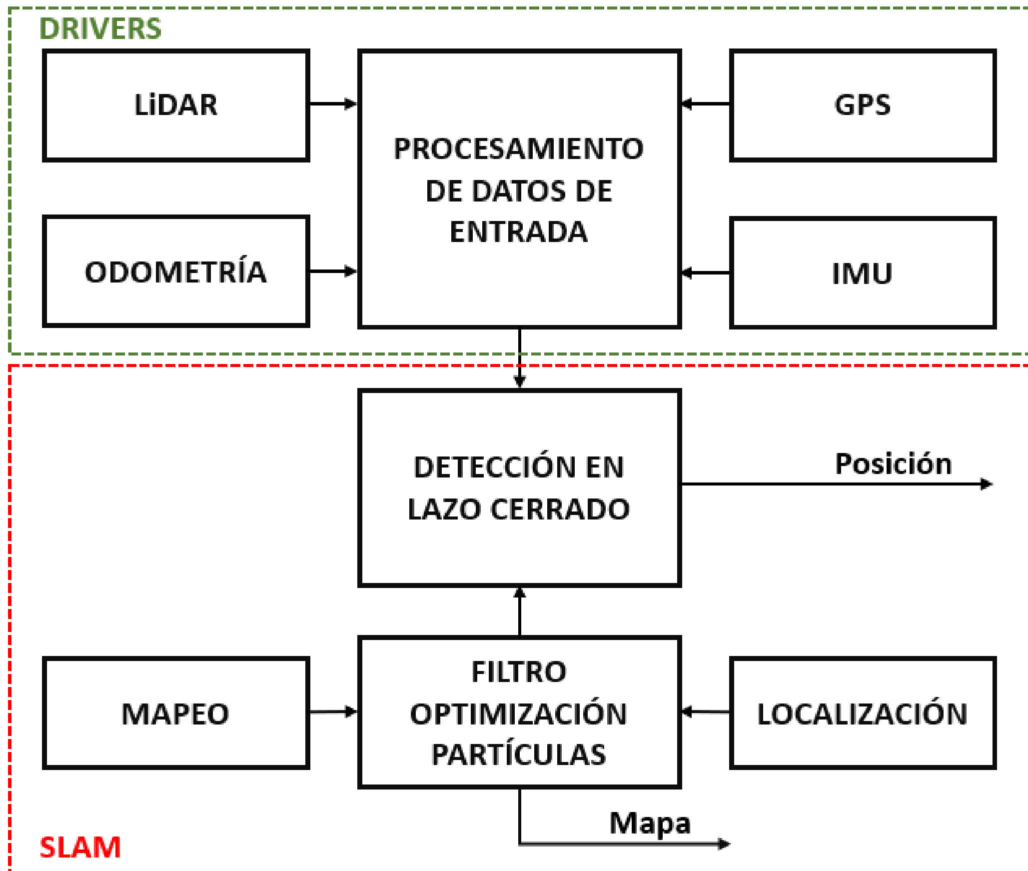


Figura 4.19 Diagrama de bloques de la técnica SLAM de Gmapping

A su vez, el mapa generado por Gmapping debe de guardarse en un formato entendible para sistema ROS. Es por ello que todos los datos históricos que el tópico /map ha publicado en su "Occupacy Grid" correspondiente, pueden guardarse en dos únicos archivos. El primero, cuenta con una extensión .yaml, el cual contiene toda la metadata necesaria de la red de ocupación y el nombre de la imagen. El segundo, cuenta con la extensión .pgm, que, en resumen, permite visualizar el mapa que contiene el .yaml codificado en su interior. Ambos archivos son generados por el nodo map_server, encargado de gestionar y proveer el mapa estático del entorno donde se requiera navegar. Cada vez que se quiera trabajar en un entorno u otro, simplemente ha de lanzarse el nodo map_server junto con su .yaml correspondiente, de manera que el tópico /map replique los datos adquiridos por el algoritmo Gmapping durante la etapa de mapeado. Es decir, el nodo efectúa las funciones correspondientes a un bagfile o bolsa de ROS.

Sin embargo, como se ha comentado previamente, existen diversos mapas y todos ellos, son empleados para llevar a cabo la navegación. Concretamente, se dividen en dos tipos de mapas:

- **Mapa Convencional**: Los mapas convencionales son aquellos que recogen las partículas que forman la red de ocupación y le dan un sentido físico, un sentido de mapa. Es decir, recogen si las partículas obtenidas del entorno son 0 o 100, libres u ocupadas, punto gris o punto negro, no hay punto medio.
- **Mapa de Costes**: Los mapas de costes recogen el cómo de peligroso o de costoso es para un vehículo autónomo el desplazarse de un punto a otro. Es decir, al igual que los mapas convencionales, cuentan con las partículas recogidas en la red de ocupación u "Occupacy Grid", sin embargo, introducen una serie de partículas que cuentan con valores entre el 0 y el 100, de manera que cuanto más cerca esté una partícula de un obstáculo o pared, más peligroso será el pasar por encima de ella por riesgo a colisión, es decir, mayor tono oscuro tomará en el mapa. Un ejemplo de su visualización se muestra en la *Figura 4.20*.

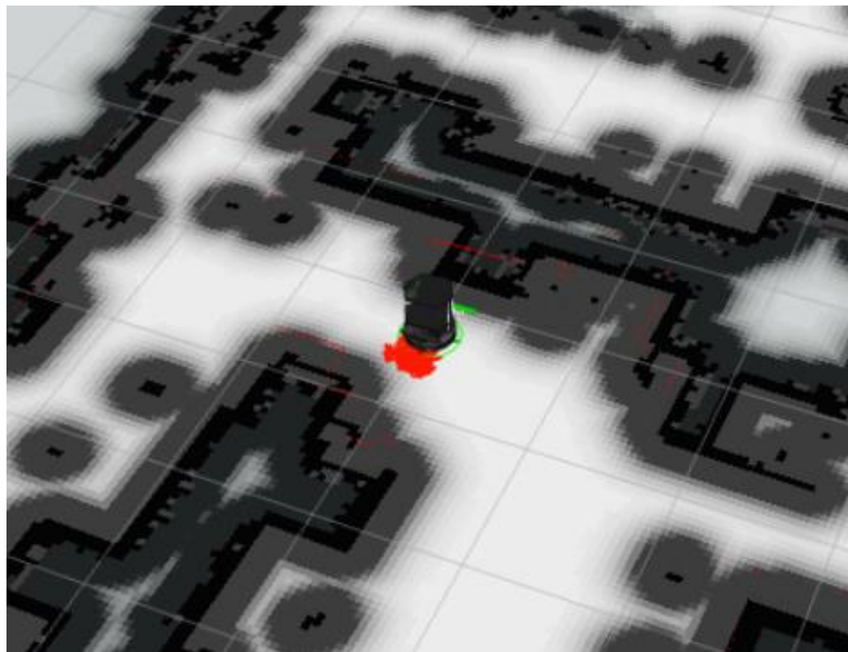


Figura 4.20 Mapa de Costes Generado por Gmapping

Del mismo modo, estos dos tipos de mapas cuentan con otras dos subdivisiones cada uno de ellos, como lo son las siguientes:

- **Mapa Convencional Estático.** Se trata del mapa base o mapa global, el que se genera en los dos archivos .yaml y .pgm por el nodo map_server y sobre el cual el vehículo autónomo se traslada y se localiza. Se trata del mapa de referencia global y en el cual se basa toda la navegación.
- **Mapa Convencional Dinámico.** Se trata del mismo mapa que el mapa estático, salvo con la excepción de que incluirá los obstáculos o elementos que puedan detectarse por los sensores del vehículo, como lo es el LiDAR para este caso de aplicación. Sin embargo, la inclusión de estos objetos en el mapa es local y dinámica, abarcando únicamente una

región pequeña alrededor del lugar en el que se encuentre el vehículo, y eliminando los obstáculos una vez el vehículo haya abandonado dicha zona.

- **Mapa de Costes Global.** Se trata del mapa que recoge los costes o peligrosidad de desplazamiento del vehículo basándose en el mapa convencional estático.
- **Mapa de Costes Local.** Se trata del mapa que recoge los costes o peligrosidad de desplazamiento del vehículo basándose en el mapa convencional dinámico.

El ROS Navigation Stack permite configurar diversos parámetros de los mapas de coste, como la resolución que emplea, el número de partículas que se van a emplear, o el sistema de referencia global al que se van a asociar del árbol de transformadas. Sin embargo, cuentan con la posibilidad de configurar un mayor número de parámetros asociados a la localización y la planificación de trayectorias, las cuales se mencionarán en los dos siguientes apartados.

Concretamente, estos archivos se les denomina habitualmente dentro de la comunidad de ROS como: costmap_common.yaml, costmap_global.yaml, costmap_global_static.yaml y costmap_local.yaml. Se recogen dentro de la carpeta de configuración del paquete que se esté empleando para llevar a cabo la navegación, y es necesario el indicar el direccionamiento de estos desde el archivo .launch que lance el sistema.

Sin embargo, todos estos parámetros pueden dejarse por defecto según lo que establezca el ROS Navigation Stack, salvo uno: El posicionamiento inicial del vehículo autónomo en el mapa, o el sistema de coordenadas que empleará la odometría para determinar la localización del vehículo en el mapa. Este parámetro es de vital importancia para llevar a cabo tanto la localización como la planificación de trayectorias. Los archivos necesarios para lanzar Gmapping y sus relaciones entre sí, se muestran en la *Figura 4.21*.

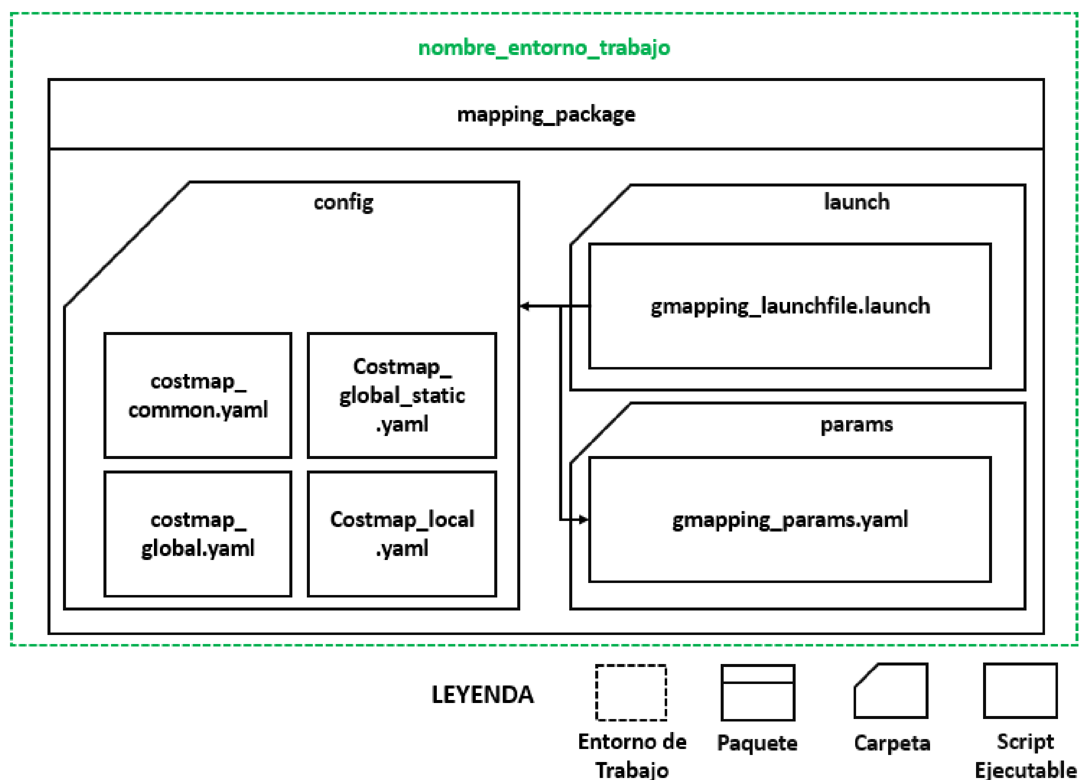


Figura 4.21 Archivos generales relativos a Gmapping

4.1.2.2.2 Localización

Toda aplicación en la que se lleva a cabo navegación autónoma requiere de un algoritmo que permita localizar a la unidad o vehículo que vaya a realizar la navegación. Dicha localización no es más que el posicionamiento cartesiano del vehículo con respecto a un sistema de coordenadas de referencia o mundo. Habitualmente, este sistema de coordenadas suele definirse en el punto de inicio del vehículo, es decir, en el punto el cual vaya a lanzarse e iniciarse su funcionamiento. De esta manera, la odometría, la cual empleará como referencia ese mismo punto, no diferirá del sistema de localización en su estimación, además de que este último sistema emplea la odometría para la ejecución de su algoritmo.

El ROS Navigation Stack emplea el algoritmo de localización Monte Carlo, concretamente, una variación de éste: Adaptative Monte Carlo Localization, o AMCL. Este algoritmo cuenta con un funcionamiento muy similar al filtro de partículas que emplea Gmapping, salvo con la excepción de que la nube de puntos la cual representa al vehículo en cuestión también predice la dirección más probable que va a tomar en el instante siguiente. Es decir, que implementa algoritmos propios de control predictivo empleados en navegación y conducción autónoma. Esto, supone una considerable reducción de computación al planificador de trayectorias empleado por el paquete move_base [20], [48].

Cuando se lanza por primera vez el algoritmo, lanza una especie de nube de puntos a lo largo del mapa para determinar dónde se encuentra el vehículo. Para ello, comparará la información que recibe por parte de la odometría del vehículo y sus sensores con la posición que ocupan dichas partículas iniciales por todo el mapa. Poco a poco, a medida que le va llegando más información, irá eliminando las partículas que no coincidan según localización con lo percibido del entorno, hasta congregar todas dentro de una región la cual en su conjunto representará al vehículo. En cierto modo, su funcionamiento es muy similar al algoritmo de optimización PSO (Particle Swarm Optimization), con la salvedad de que el AMCL únicamente trata de encontrar la posición en la que se encuentra un vehículo según el mapa aportado por el nodo map_server.

Es por ello, que antes de llevar a cabo cualquier aplicación relativa a la navegación autónoma que requiera de ejecución de trayectorias o movimiento controlado por el planificador, hay que calibrar el equipo y cerciorarse de que el algoritmo de localización de Monte Carlo se ha efectuado de manera correcta. La diferencia entre un vehículo no calibrado y con el AMCL recientemente inicializado con la de un vehículo con el algoritmo de localización ya preparado, se visualiza como flechas rojas en la *Figura 4.22*.

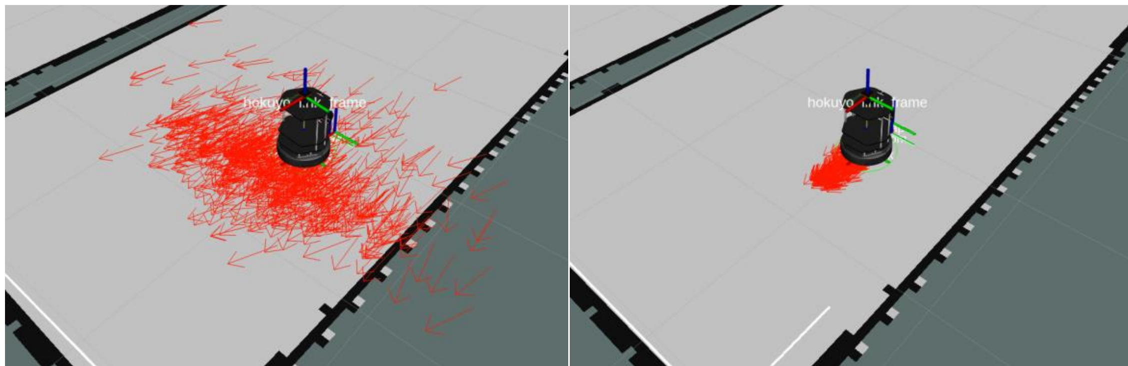


Figura 4.22 AMCL descalibrado (izquierda) y AMCL Calibrado (derecha)

Al igual que con Gmapping, la localización propia del AMCL puede configurarse en el ROS Navigation Stack a través de un único archivo: El correspondiente para lanzar el algoritmo de localización, amcl.launch.

4.1.2.2.3 Planificación

El planificador se trata del elemento encargado de trazar la trayectoria que el vehículo autónomo ha de seguir para desplazarse desde un punto de partida hasta el destino objetivo. Esta planificación permite al controlador encargado del desplazamiento del vehículo autónomo el tener una referencia de entrada, de manera que su única tarea sea la de seguir la trayectoria trazada por el planificador. El ROS Navigation Stack emplea dos planificadores diferentes [44], [49] :

- **Planificador Global**: Se trata del encargado de trazar una trayectoria desde el punto inicial donde se encuentra el vehículo hasta el punto final o destino. Lo hará basándose en lo recogido en el mapa convencional estático y en el mapa de costes global.
- **Planificador Local**: El planificador local es el encargado de trazar pequeñas trayectorias las cuales sigan el recorrido trazado por el planificador global. Lo hará basándose tanto en el mapa de costes local como en el mapa convencional dinámico. Es decir, se trata del planificador que permite modificar ligeramente la trayectoria trazada por el planificador global en caso de darse obstáculos u objetos en dicha trayectoria que imposibiliten realizar la trayectoria global.

Al igual que con la localización y el mapeado, existen diversos algoritmos de control que implementen el cálculo de trayectorias. La gran mayoría de ellos, son derivados del algoritmo de Dijkstra y el algoritmo de búsqueda A-Star, también conocido como A*. El ROS Navigation Stack implementa hasta siete algoritmos de búsqueda distintos, tres para el planificador global y cuatro para el planificador local, concretamente:

- **Navfn**: Asociado al planificador global, emplea el Algoritmo de Dijkstra para calcular el trayecto óptimo y corto entre un punto inicial y otro final.
- **carrot_planner**: Asociado al planificador global, este planificador permite ejecutar trayectorias en entornos en los que el vehículo tenga que moverse muy cerca de obstáculos. Requiere de altas tasas de refresco y gran capacidad de cómputo.
- **global_planner**: Asociado al planificador global, permite elegir e implementar entre el Algoritmo de Búsqueda A* o el Algoritmo de Aproximaciones Cuadráticas.
- **base_local_planner**: Asociado al planificador local, emplea dos algoritmos distintos, Trajectory Rollout y DWA (Dynamic Window Approach). Su funcionamiento se basa en la simulación de diferentes trayectorias del vehículo a distintas velocidades, teniendo en cuenta la dinámica de éste. Va verificando una a una aquellas que son óptimas y elimina aquellas que no sean posibles de realizar. La trayectoria y velocidad con mayor puntuación es aquella que finalmente se realiza.
- **DWA_local_planner**: Asociado al planificador local, se trata del DWA del base_local_planner abstraído a otro paquete independiente.
- **Eband_local_planner**: Asociado al planificador local, emplea el método de Elastic Band.

- **Teb_local_planner**: Asociado al planificador local, emplea el método de Time Elastic Band.

En el presente trabajo, se trabajará con los planificadores Navfn, para el planificador global, y el DWA_local_planner, para el planificador local. El planificador, puede ser modificado en los cuatro archivos contenidos de la carpeta config mostrados previamente en el apartado de mapeado, los mostrados en la *Figura 4.21*. Por otro lado, en la *Figura 4.23* se observa mediante una línea verde la trayectoria trazada sobre el mapa por el planificador local, para un desplazamiento desde el punto inicial donde se ha inicializado la unidad de transporte hasta un punto cercano a la pared. Nótese que la nube de puntos del AMCL se mueve junto con el vehículo sobre el mapa.

Figura 4.23 Ejemplo de Visualización de Trayectoria de Desplazamiento

En lo relativo a los parámetros y variables más significativas a modificar de los archivos de la carpeta de configuración en referencia a la planificación de trayectorias, se muestran los siguientes:

- **transform_toleranc**e: Se trata del parámetro que establece el máximo retardo que permite el planificador para dar como válidos los mensajes de los tópicos a los que se suscribe, suele rondar en los 0.5 Hz. En caso de no cumplirse, rechaza dichos datos.
- **robot_radius**: Establece el radio físico del vehículo, para determinar su tamaño y determinar cómo de cerca puede moverse junto a un objeto. Para vehículos no cilíndricos, su homólogo sería el parámetro “footprint”.
- **xy_goal_tolerance**: Determina cual es la diferencia mínima que ha de tener la odometría y el punto final de la trayectoria para que el planificador y dé por válida y finalizada su tarea de ejecución de trayectoria.
- **inflation_ratius**: Distancia mínima que puede haber entre el vehículo y obstáculo.

4.1.2.3 ROSJava

ROSJava es un paquete propio de ROS que permite llevar a cabo el desarrollo de nodos y soluciones ROS en Java. La presencia de este paquete en el presente proyecto se debe a la necesidad de integrar todo el entorno de ROS con la plataforma MAS encargada de gestionar las unidades de transporte. Dado que el MAS de este trabajo está desarrollado en Java, concretamente en la plataforma JADE, la cual se describirá en detalle más adelante en este mismo capítulo, el paquete ROSJava con un entorno Java [23], [24], [50].

Este paquete, en resumen, permite llevar a cabo un Gateway entre un entorno ROS y otro entorno Java. En esencia, mediante ROSJava se pueden desarrollar nodos escritos en el lenguaje de programación Java, es decir, clases java que sean capaces de ejecutar funcionalidades propias de ROS, como la publicación o suscripción a tópicos. Estas clases Java, se encuentran integradas dentro del entorno de ROS, es decir, son capaces de registrar un nodo en el entorno de ROS. A su vez, al tratarse de clases, pueden instanciarse a otro tipo de clases java que tengan las funcionalidades necesarias como para inscribirse dentro de la plataforma MAS, es decir, inscribir un agente en la plataforma.

En el ejemplo de la *Figura 4.24* se muestra toda la estructura del sistema Gateway que el paquete ROSJava permite realizar entre un entorno ROS y otro en Java como es el MAS. El nodo en cuestión, donde reside toda la funcionalidad del agente, reside en el “Nodo ROS”, dentro del entorno de ROS. Sin embargo, a través de la “Funcionalidad Agente ROS”, que no es más que una clase con funcionalidad de agente instanciada como objeto por el “Nodo ROS”, este último, cuenta con un agente que representa al “Agente ROS” en la plataforma MAS .

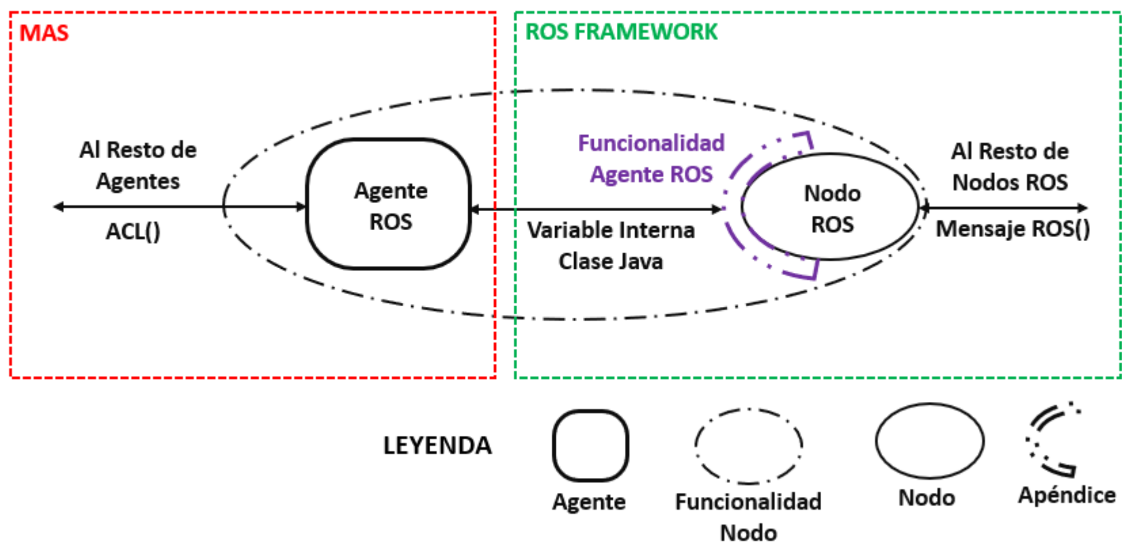


Figura 4.24 Ejemplo de Integración de Nodo en ROS y MAS

En lo relativo a la estructura y árbol de proyecto, las soluciones desarrolladas en ROSJava difieren ligeramente de los entornos y paquetes propios llevados a cabo en C++, C# o Python, como lo son los mostrados en el apartado de conceptos básicos de ROS. Los archivos CMakelist.txt y package.xml también se encuentran dentro del entorno de ROSJava, sin embargo,

cuentan con el añadido de build.gradle; así como las distintas clases javas que se generan al compilar sus respectivos scripts .java.

El build.gradle se trata de un archivo basado en Gradle, un sistema que permite ejecutar y construir paquetes software de manera automática, generando dependencias y estableciendo las dependencias a las cuales las clases Java del entorno tienen que direccionarse. Estas dependencias, pueden definirse a directorios internos del equipo donde resida el paquete, como a repositorios en internet.

En lo relativo a su estructura, aparece el concepto de "proyectos", los cuales permiten un nivel de encapsulación mayor a los entornos convencionales de ROS. Es decir, en el ROS convencional únicamente se cuenta con los entornos de trabajo y directamente en su interior, los paquetes. Sin embargo, ROSJava permite entornos de trabajo, que cuenten con múltiples paquetes ROS, y estos a su vez, varios proyectos en su interior. Del mismo modo, ROSJava cuenta con una serie de directorios y ejecutables propios del entorno de Java, empleando entre otras cosas la carpeta build para guardar las clases Java. El esquema y distribución de una solución desarrollada en ROSJava, tendrá la siguiente estructura de la *Figura 4.25*.

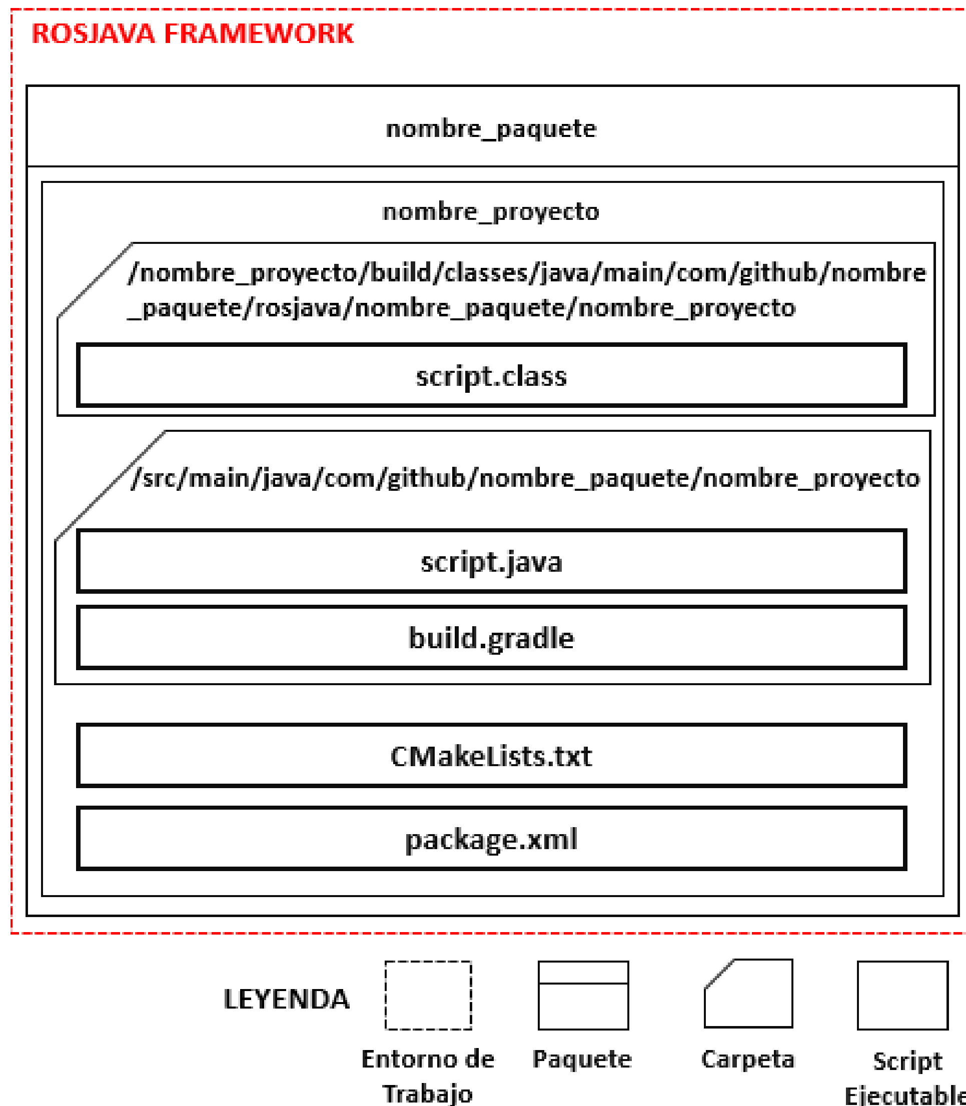


Figura 4.25 Estructura de un Entorno ROSJava

4.1.2.4 Gazebo

Gazebo es el simulador de ROS por excelencia, el cual permite generar entornos y gemelos virtuales que permiten replicar la estructura física y la dinámica de prácticamente cualquier robot. Asimismo, permite integrar toda clase de archivos tipo CAD (Computer Assisted Design) y derivados que habilitan la posibilidad de replicar el entorno o célula en la que va a trabajar el robot que se quiera integrar en Gazebo. Es más, prácticamente la totalidad de los paquetes que emplean las unidades robóticas integradas dentro de ROS en la realidad, también los emplean sus gemelos virtuales en Gazebo. Esta característica permite no solo validar el funcionamiento de un robot real sin la presencia física del mismo, sino que posibilita el depurar el código que vaya a emplearse en dicho robot, sin la necesidad de poner en peligro la integridad física de éste. La interfaz que ofrece Gazebo puede observarse en la *Figura 4.26* [51].

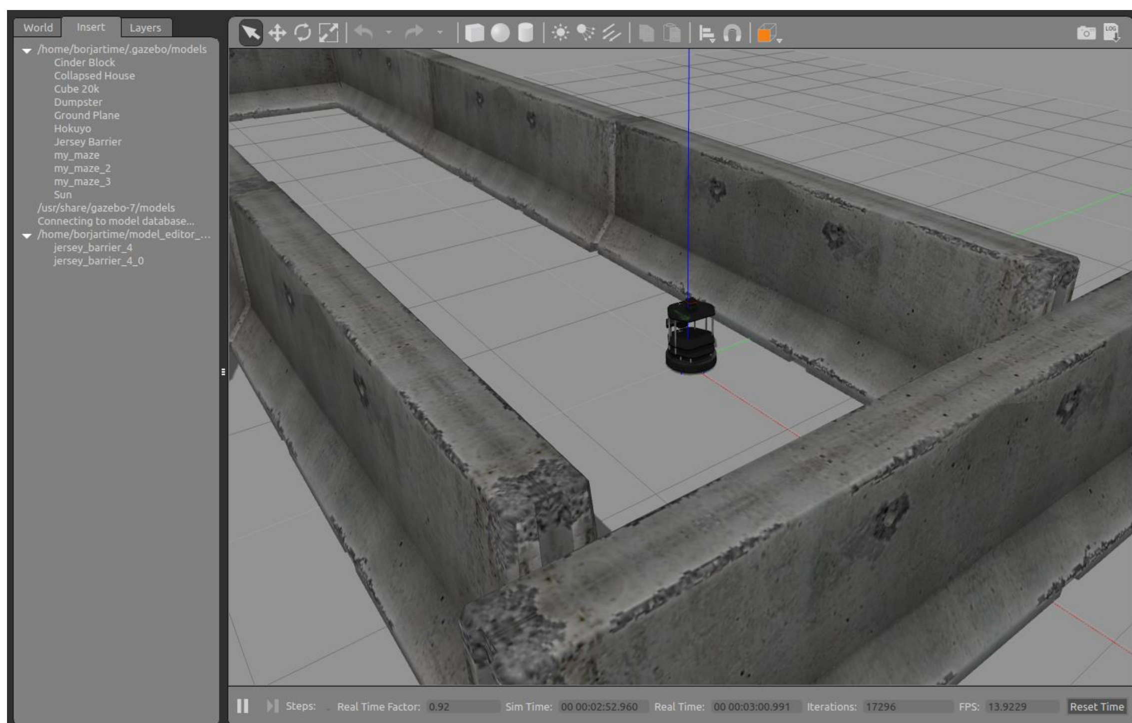


Figura 4.26 Interfaz de Usuario de Gazebo

Tal es la utilidad de esta herramienta que, para el desarrollo de las primeras soluciones de este proyecto, sobre todo aquellas relacionadas con la navegación autónoma, se empleó Gazebo. Para ello, se generó un entorno que simulase la distribución de un laberinto simple, del cual la unidad de transporte tenía que salir de manera autónoma. Evidentemente, para ello, se emplearon las funcionalidades propias del ROS Navigation Stack, de manera que se tuvo que obtener un mapa del entorno para poder navegar sobre él. Pese a ser Gazebo un entorno digital, como se muestra en la *Figura 4.27*, los resultados obtenidos replican el comportamiento que la unidad de transporte tendría en la realidad.

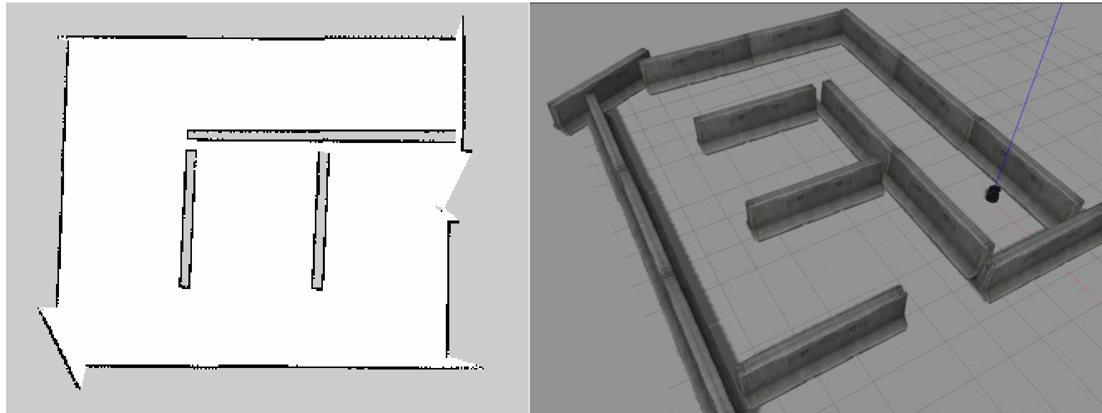


Figura 4.27 Mapa del Entorno de Gazebo (Izquierda) Entorno de Gazebo (Derecha)

Pese a que Gazebo pueda interpretar archivos del tipo CAD e introducirlos a su entorno (extensión .dae para Gazebo), la mayoría de los elementos que lo forman no cuentan con un archivo del tipo CAD per se. Esta herramienta emplea archivos descriptivos escritos en XML para definir cada uno de los objetos que integra, los cuales cuentan con una extensión ".urdf". Para interpretar estos archivos, Gazebo hace uso de una herramienta propia de ROS: Xacro [52].

Xacro es un macro del lenguaje de programación XML empleado en el entorno de ROS. En esencia, se trata de un conjunto de comandos que se invocan con palabras clave que emulan al comportamiento y estructura de XML. Este macro permite: definir la propia estructura de un robot, la dinámica de sus elementos, las relaciones y uniones entre cada una de las partes que forman el conjunto del robot, la definición de constantes globales del sistema e incluso realizar pequeñas operaciones matemáticas. Esta herramienta, viene incorporada dentro del propio middleware de ROS, recogida dentro de los paquetes ya instalados por defecto [53].

En la *Figura 4.28* se muestra un ejemplo de una sección del código en Xacro que define el sensor LiDAR hokuyo mostrado en la *Figura 4.13,* en la sección de transformadas de ROS. En esta sección, puede observarse como se define el nombre de su sistema de referencia, hokuyo_link, el sistema de referencia al que se asocia, base_link", y varios parámetros de funcionamiento, a saber: el tópico en el que se publican los datos, /scan_lidar; y la posición exacta de hokuyo_link en el árbol de transformadas, 0.45 metros sobre el eje vertical z; entre otros parámetros.

```
<!--<hokuyo_utm30lx_model name="hokuyo_laser" parent="base_link">-->
<xacro:hokuyo_utm30lx name="hokuyo_link" parent="base_link"
  ros_topic="scan_lidar"
  update_rate="10"
  ray_count="500"
  min_angle="-130"
  max_angle="130.0"
  >
  <origin xyz="0 0 +0.450" rpy="0 0 0" />
</xacro:hokuyo_utm30lx>
```

Figura 4.28 Ejemplo de parte de la definición de un Sensor LiDAR Hokuyo en Gazebo mediante Xacro

Esta característica de Gazebo y los archivos descriptivos definidos mediante Xacro, permite no solo diseñar elementos propios personalizados, sino emplear librerías y paquetes de otros usuarios e integrarlos dentro del archivo descriptivo ".urdf" que se requiera. Es más, Gazebo en su página web oficial cuenta con un repositorio del estilo de GitHub en el que se recogen cientos de modelos y gemelos virtuales de elementos robóticos para Gazebo descritos en Xacro. De esta manera, se pueden integrar elementos dentro de Gazebo únicamente copiando el conjunto de palabras clave que lo definen, tanto su estructura física como su dinámica.

En este caso de aplicación, al emplearse como unidad de transporte un Kobuki Turtlebot2, los archivos descriptivos propios de este robot se recogen en el paquete turtlebot_description. Sin embargo, dado que en su mayoría se emplean elementos y objetos que no se recogen dentro del propio Turtlebot2, como podría ser el sensor LiDAR, se emplean otros paquetes que cuentan a su vez con los archivos descriptivos de estos elementos externos al Turtlebot2. Todos estos paquetes y archivos descriptivos ".urdf" se invocan y se coordinan desde un archivo ".launch", el cual coordina, instancia y declara todas aquellas variables y directorios que los ".urdf" necesitan para que al ejecutarse el macro Xacro no se den errores de ejecución. Para mostrar el cómo se relacionan todos los archivos descriptivos, se muestra en la *Figura 4.29* un esquema de todos los paquetes y ".urdf" que toman parte en la generación del entorno de la *Figura 4.26*.

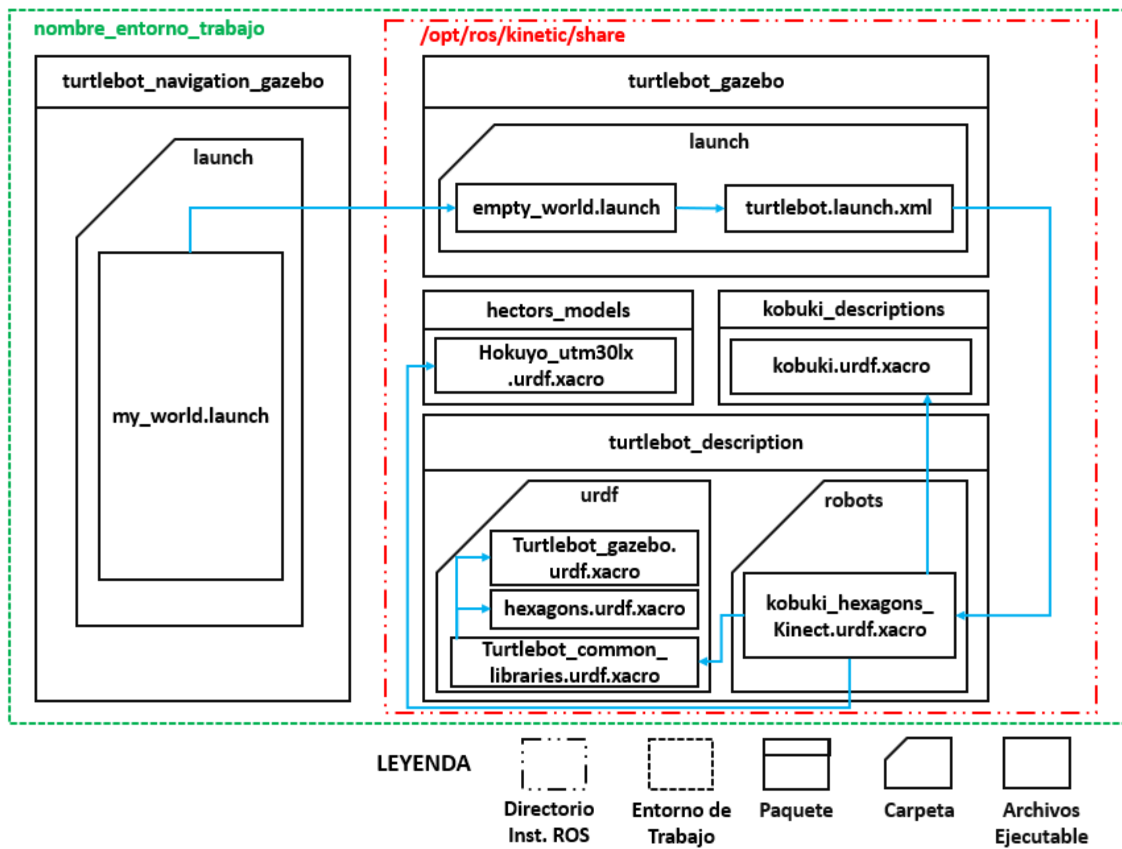


Figura 4.29 Distribución de los archivos y paquetes ROS empleados en Gazebo

Salvo por el archivo "my_world.launch" del ejemplo de la figura anterior, el resto de los paquetes y archivos son propios del entorno de ROS, por lo que se repiten en la mayoría de las aplicaciones. Entre ellos los más importantes se resumen en:

- **turtlebot.launch.xml.** Describe el Turtlebot2 en XML.
- **kobuki_hexagons_kinect.urdf.xacro.** Establece las dependencias del modelo del Turtlebot2 en gazebo y los directorios del resto de archivos descriptivos.
- **turtlebot_gazebo.urdf.xacro:** Archivo descriptivo que construye los modelos y objetos dentro de Gazebo y define algunas de sus dinámicas.

4.1.2.5 RViz

RViz se trata de un paquete ROS el cual proporciona una herramienta de visualización de los distintos elementos que conforman todo el entorno de ROS. En cierto modo, podría considerarse como una herramienta GUI (Graphical User Interface) del entorno de ROS, la cual permite generar un gemelo virtual de todos los elementos que lo conforman, estén estos integrados dentro de ROS o se estén ejecutando en un equipo o robot real. Por ejemplo, la *Figura 4.30* muestra la interfaz de usuario de RViz para el entorno de ROS mostrado en el apartado anterior de Gazebo, más concretamente la de la *Figura 4.26*. Esta herramienta también es capaz de interpretar los datos publicados en los tópicos y transformarlos a un formato que permita su visualización. Cabe destacar, que RVIz también hace uso de la herramienta Xacro para representar y definir los gemelos virtuales del entorno [54], [55].

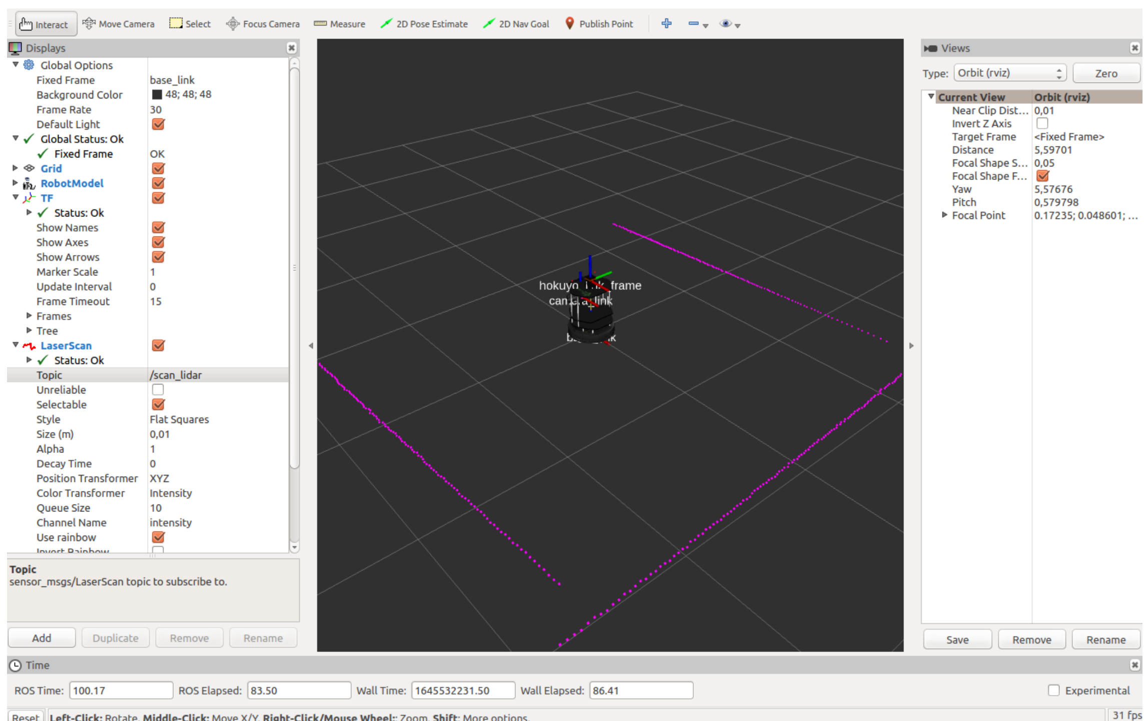


Figura 4.30 Interfaz de usuario de RViz

En la anterior imagen, se puede comprobar cómo RViz aporta un gemelo digital de la unidad de transporte introducida a Gazebo, la posición que ocupan diversos sistemas de referencia del árbol de coordenadas e incluso la lectura que aporta el sensor LiDAR del entorno en el que se encuentra, interpretando lo que el sensor LiDAR publica al tópico "/scan_lidar" y dándole un sentido físico y visual al representar esa información como puntos rosas dentro del entorno en el que se encuentra el AGV.

Asimismo, RViz permite visualizar muchos otros tipos de elementos propios de ROS que en esencia tengan relación con los tópicos del sistema, como podrían ser: imágenes, mapas, marcadores del entorno, temperatura e incluso personalizar la lectura de tópicos con mensajes customizados de ROS. Es más, RViz permite guardar las configuraciones de su interfaz, de manera que cuando se lance el nodo correspondiente a la ejecución de RViz, se abra directamente una configuración de previsualización definida, como lo podría ser el entorno en el que se estaba trabajando la última vez que se empleó RViz.

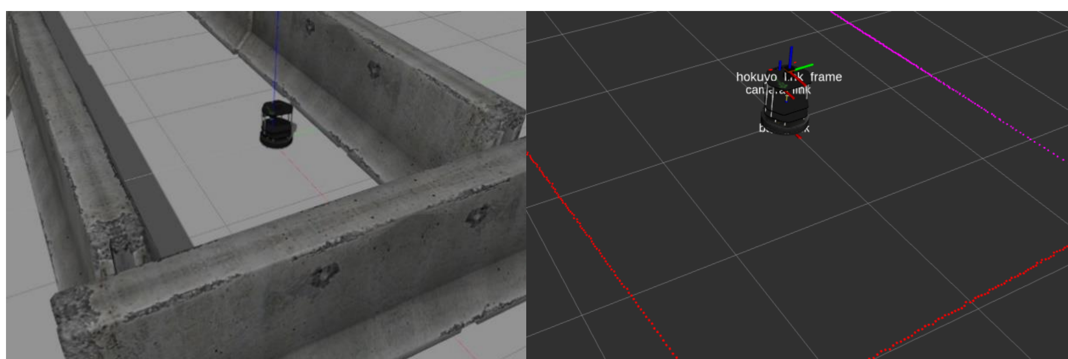


Figura 4.31 Comparativa Entorno en Gazebo (Izquierda) y Gemelo Visualizado en RViz (Derecha)

Evidentemente, como se ha comentado, RViz también puede visualizar el entorno de ROS en la realidad, no únicamente entornos virtuales como Gazebo. Para ello, se muestra la *Figura 4.32* donde con un AGV se realiza una prueba de navegación autónoma, empleando con RViz la visualización del mapa del entorno en el cual está navegando y visualizando lo que percibe del sensor LiDAR que lleva incorporado.

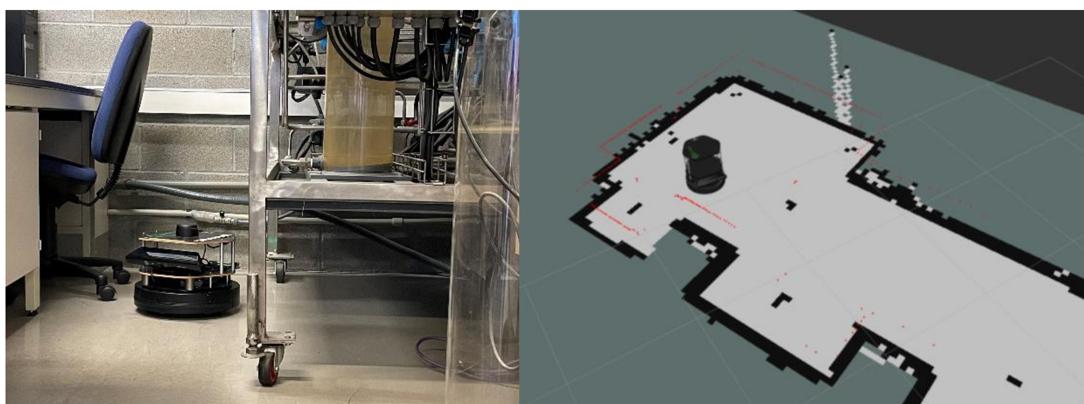


Figura 4.32 Comparativa Entorno Real (izquierda) y Gemelo Visualizado en RViz (Derecha)

4.1.2.6 ROS Distro

La versión o distribución de ROS con la que se va a trabajar en el presente proyecto, también denominada ROS Distro, es ROS Kinetic Kame. La principal diferencia entre todas las Distros de ROS es la versión del código de los paquetes que incorpora y su compatibilidad con los sistemas operativos de Linux. Dado que este trabajo se desarrolla dentro de una plataforma llevada a cabo en Ubuntu LTS 16.04, la Distro de ROS correspondiente para dicho sistema operativo es ROS Kinetic Kame. Por otro lado, esta versión de ROS se trata de una de las más populares entre los desarrolladores amateurs de ROS, siendo junto a ROS Indigo, la Distro con mayor número de soluciones y paquetes llevados a cabo [27], [56].

Figura 4.33 Logo Oficial de ROS Kinetic Kame

Dejando de lado las versiones más recientes de ROS2 (Humble, Galactic y Foxy), las últimas Distros de ROS y sus principales diferencias se muestran en la siguiente tabla.

ROS Distro	**Lanzamiento**	***Obsolescencia***	***Sistema Operativo***
ROS Noetic Ninjemys	Mayo de 2020	Mayo de 2025	Ubuntu 20.04
ROS Melodic Morenia	Mayo de 2018	Mayo de 2023	Ubuntu 18.04
ROS Lunar Loggerhead	Mayo de 2017	Mayo de 2019	Ubuntu 17.04
ROS Kinetic Kame	Mayo de 2016	Abril de 2021	Ubuntu 16.04
ROS Jade Turtle	Mayo de 2015	Mayo de 2017	Ubuntu 15.04
ROS Indigo Igloo	Mayo de 2014	Abril de 2019	Ubuntu 14.04

Tabla 4.4 Comparativa entre las Distros de ROS

4.1.3 Plataforma JADE

La plataforma JADE o Java Agent Development Network por sus siglas en inglés, se trata de una plataforma que permite tanto el recoger como el desarrollar un MAS o Sistema Multi-Agente. Esta herramienta, se trata de la plataforma propia aportada por el lenguaje de programación Java y la cual cuenta con una GUI (Graphical User Interface) que facilita considerablemente la interacción con los agentes que se recogen dentro del MAS [57].

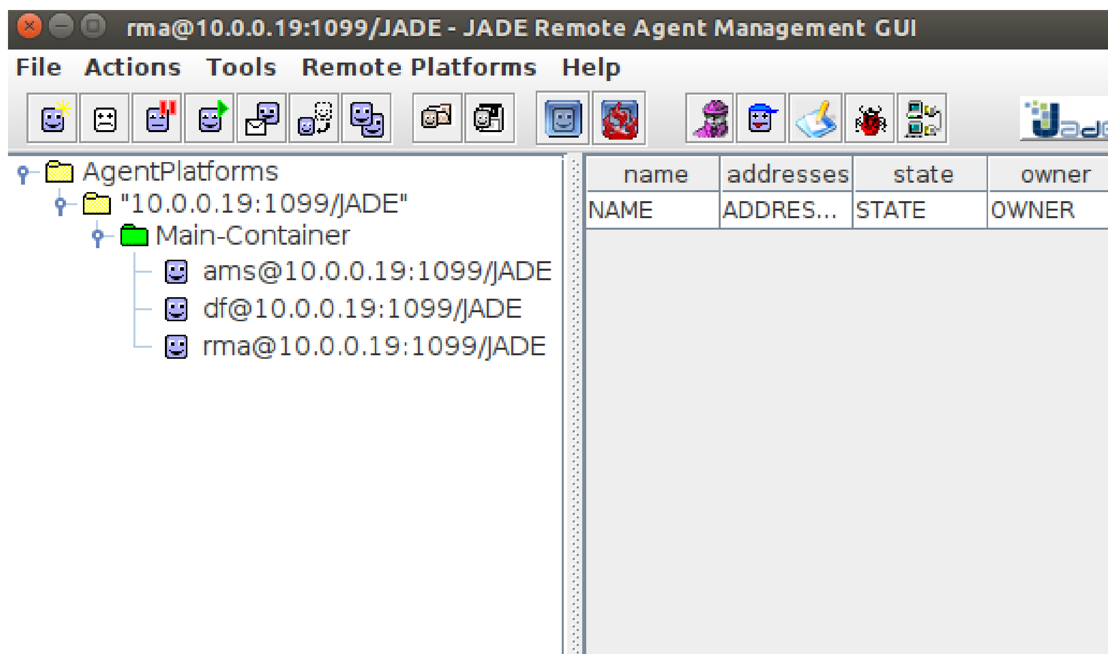


Figura 4.34 GUI de la Plataforma JADE

Del mismo modo, al lanzarse la GUI se arrancan tres agentes software los cuales se encargan de las tareas de gestión y registro más básicas, su presencia es obligatoria para el correcto funcionamiento de la plataforma. Concretamente, se tratan de tres agentes, los cuales son los siguientes:

- **Agent Management System (AMS)**: Controla el acceso al dominio y el uso general de la plataforma, así como de verificar que el registro de cada agente ha sido el correcto. En cierto modo, podría considerarse que su funcionalidad se asemeja a la de un ROS Master.
- **Directory Facilitator (DF):** Se encarga de almacenar los servicios que proporciona cualquier agente, de forma que sea rápido encontrarlo para el resto de los agentes.
- **Agent Communication Channel (ACC):** Canal de comunicación entre agentes que controla el intercambio de mensajes dentro de la plataforma.

Los agentes se recogen dentro de contenedores, de manera que puedan agruparse dentro de un mismo contenedor un conjunto de agentes asociados entre sí. Cada agente, puede tener diferentes estados, más concretamente, puede encontrarse en los estados de: Iniciado, donde el agente está registrado en la plataforma pero no listo para comunicarse; activo, donde el agente ya está listo para pasar a un funcionamiento normal; suspendido, o un equivalente a un estado de error; en espera, el cual está esperando a la finalización de una comunicación o a la activación de un evento; en tránsito, cuando se encuentra evaluando su estado interno; o desconocido, cuando por algún casual se ha perdido la comunicación con el agente.

Esta GUI recoge diversas funcionalidades como el “Sniffer”, el cual permite ver los mensajes que está mandando y recibiendo un agente dentro de la plataforma, como puede observarse en la *Figura 4.35*.

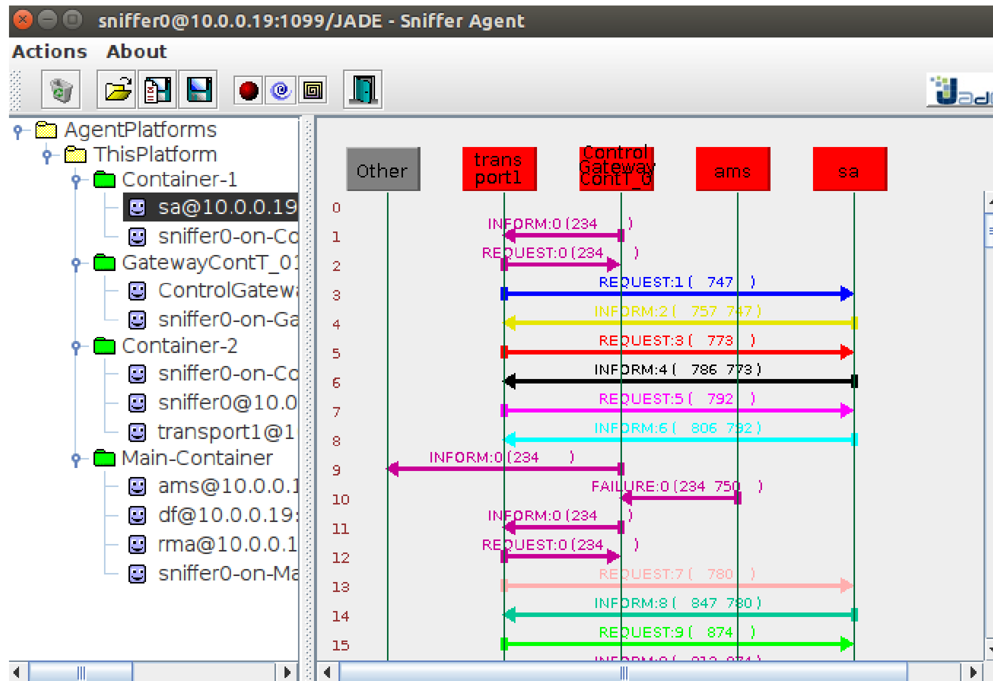


Figura 4.35 Sniffer de la GUI de la Plataforma JADE

Sin embargo, al tratarse de una plataforma para el desarrollo de agentes software, cuenta con diversas herramientas, no solo con una interfaz gráfica, tales como repositorios, clases java, librerías y métodos para la comunicación, gestión e intercambio de información entre los distintos agentes que tomen parte en el MAS. Principalmente, esta serie de paquetes software incluyen todas las funcionalidades necesarias para la comunicación a través de mensajes ACL, o “ACLMessages”, que son los mensajes propios de la normativa FIPA2000 sobre la que se construyen los agentes en JADE. Estos mensajes, cuentan con una serie de campos y estructura la cual el receptor puede filtrar para leer únicamente los mensajes que concuerden con la estructura de datos que requiera. Los mensajes ACL se dividen en los campos:

- **Sender**: AID o nombre del agente que manda el mensaje.
- **Receiver:** AID o nombre del agente que debe de recibir el mensaje.
- **Performative**: Performativa del mensaje, etiqueta la cual permite clasificar el tipo de mensaje. Los tipos de performativas existentes están recogidos en una lista cerrada por el estándar FIPA.
- **Ontology:** Ontología del mensaje, la cual permite clasificar un mensaje dentro de un campo de datos formado por la ontología.
- **ConversationID**: Etiqueta ID de la conversación, la cual permite clasificar el mensaje dentro de una ID formada por números enteros.
- **Content**: Contenido del mensaje, el cual debe de ser de tipo String para cumplir con el estándar.

4.1.4 Sistema Operativo

Tal y como se ha comentado anteriormente, varias de las aplicaciones desarrolladas en los antecedentes del presente proyecto se han llevado a cabo en el SO Ubuntu 16.04, principal razón por la cual se ha optado por trabajar con la Distro de ROS Kinetic Kame. Este sistema operativo es completamente gratuito y está basado en GNU/Linux. Cuenta con soporte técnico hasta 2024, además de una gran comunidad de usuarios. Por otro lado, al tratarse de un sistema operativo sin coste alguno, permite a sus usuarios el modificar el software independientemente de sus necesidades particulares.

La mayor limitación de este SO es que emplea la versión de Python 2.7.12, la cual dejó de tener soporte técnico en 2020, lo cual ha llegado a generar problemas de compatibilidad con algunos paquetes de ROS con código desarrollado en Python. Sin embargo, estos problemas únicamente se han dado con paquetes provenientes de otras Distros de ROS superiores a la Kinetic Kame, por lo que los paquetes de esta Distro se llevarán a cabo principalmente todos en la versión 2.7.12 de Python en caso de emplearse dicho lenguaje.

Figura 4.36 Logo de Ubuntu 16.04

4.1.5 IDE

En lo relativo a los entornos de desarrollo o IDE (Integrated Development Environment) empleados, se ha trabajado con dos: PyCharm, para todo el desarrollo relativo a las capas inferiores del sistema llevadas a cabo en Python; e IntelliJ Idea, empleado para la programación relacionada con Java. La licencia profesional de ambos IDE puede conseguirse de manera gratuita en la web de Jetbrains, siempre y cuando sea un docente o estudiante universitario aquel que la solicite.

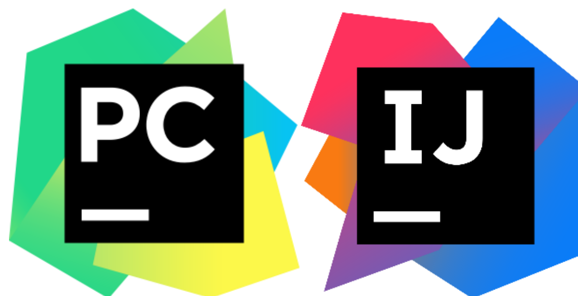


Figura 4.37 Icono de Acceso PyCharm (Izquierda) e InteliJ Idea (Derecha)

4.2 Herramientas Hardware

En este apartado, se expondrán las principales características y especificaciones de los elementos hardware empleados en el presente trabajo. En el capítulo seis, concretamente en la *Figura 6.3*, podrá verse la distribución e interconexión de todos estos elementos.

4.2.1 Kobuki iCleBot Turtlebot2

El Kobuki iCleBot Turteltbot2 es una base robótica móvil especialmente diseñada para un uso académico, desarrollada por la compañía koreana Yujin Robot. Se trata del segundo modelo de la gama "TurtleBot", una serie de robots ideados por dos estadounidenses cuyo objetivo era el de desarrollar un robot de bajo costo y accesible para acercar la robótica a un mayor número de personas y el crear una herramienta accesible al mundo académico. Tal es así, que el Turtlebot2 se ha convertido en uno de los robots más populares dentro del mundo académico y amateur, sobre todo dentro de la comunidad de ROS. Principalmente, esto no se debe únicamente a su bajo costo y amplia comunidad de desarrolladores, los cuales dan un soporte técnico muy amplio, sino que cuenta con un elevado grado de personalización, tanto de su estructura física como de programación software [58].

Figura 4.38 Kobuki iCleBot Turtlebot2

En lo relativo a sus especificaciones funcionales, no se trata de un robot especialmente potente ni rápido. Sin embargo, sí que cuenta con las capacidades suficientes como para tener un alto grado de autonomía y agilidad a la hora de desplazarse, al igual que capacidad de carga, tal y como se muestra en la *Tabla 4.5*.

Característica	Valor
Velocidad de Traslación Máxima	0.7 m/s (sin carga) 0.15 m/s (carga máxima)
Velocidad de Rotación Máxima	180 º/s
Capacidad de Carga Máxima	5 kg (superficie rígida) 4 kg (superficie rugosa)
Dimensiones de la Base	351.5 mm (diámetro) 124.8 mm (altura)
Peso de la Base	2350 g
Tiempo de Operación Estimado	2:30 h (batería estándar) 7 h (batería grande)
Tiempo de Carga Estimado	1.5 h (batería estándar) 2.6 h (batería grande)
Precio	560.00 - 699.00 €

Tabla 4.5 Especificaciones Funcionales Turtlebot2

En lo relativo a las especificaciones hardware del Turtlebot2, cuenta con múltiples elementos que facilitan tanto la integración de periféricos y otros elementos en el conjunto del robot como la interacción con este. Principalmente, cuenta con los elementos de la *Figura 4.39* en su parte posterior, fácilmente accesible tanto por los periféricos con los que se requiera equipar el AGV como por las personas operarias de planta.

Figura 4.39 Interfaz Hardware del Kobuki iCleBot Turtlebot2

En referencia a la figura anterior, cabe destacar que el Turtlebot2 cuenta con cuatro puertos de alimentación, que van desde los 5 VDC hasta los 12 VDC, teniendo cada uno de ellos una capacidad de 1 A hasta 5 A de descarga, lo que facilita la integración de distintos tipos de periféricos. Por otro lado, los puertos de propósito general o GPIO, son capaces de implementar tanto comunicación serie TTD como USB, además de poder configurar puertos digitales tanto de entrada como se salida. Los tres LEDs disponibles son tricolor, y los tres botones accesibles pueden configurarse según programación. La controladora del Turtlebot2 es accesible vía USB tipo B, además de ser posible la modificación de su firmware mediante ese mismo acceso.

Por otro lado, cuenta con una serie de características y funcionalidades interesantes que se emplearán para el presente trabajo, tales como:

- **Parachoques**: La base robótica del Turtlebot2 cuenta con tres sensores de parachoques, situados los tres en la parte frontal de este. Están situados de manera que uno es capaz de detectar colisiones a la izquierda, otro a la derecha, y el último en el centro.
- **Odometría**: En lo relativo a la odometría, el Turtlebot2 cuenta con una resolución más que suficiente para implementar un odometría muy exacta. Concretamente, es capaz de detectar hasta 2578 pulsos por cada giro completo de las ruedas, es decir, que por cada milímetro se detectarán hasta 11.7 pulsos.

- **Giroscopio**: El giroscopio incorporado en la base cuenta con una robustez considerablemente alta en lo relativo a la necesidad de calibración. Sin embargo, es necesario el calibrar dicho sensor cada cierto tiempo siempre y cuando se requiera de una aplicación de precisión. Su resolución estando perfectamente calibrado, es de 110 °/s.
- **Sensores de Acantilado**: Al igual que los parachoques, cuenta con tres unidades: Uno a la izquierda, otro la derecha y uno en el centro. La función de estos sensores es la de detener en seco al robot en caso de que se detecte un desnivel de más de 5 cm entre dos superficies.
- **Audio**: La base móvil cuenta con un buzzer o zumbador capaz de emitir diez sonidos diferentes, todos ellos configurables por programa.

En lo relativo a sus características estructurales, el Turtlebot2 está diseñado para incorporar varias bases o platos en forma de torre, de manera que puedan incorporarse diversos periféricos a distintas alturas, como se muestra en la *Figura 4.40*.

Figura 4.40 Diseño Estructural Turtlebot2

4.2.2 RPLiDAR A2

El RPLiDAR A2 se trata de un sensor de bajo costo basado en tecnología LiDAR. A diferencia de otros dispositivos de la misma tecnología, el RPLiDAR únicamente es capaz de emitir haces de luz láser en un único plano, es decir, en dos dimensiones. Su diseño, le permite rotar en 360 º sobre sí mismo, lo que le permite no tener puntos ciegos dentro de su rango de operación.

Figura 4.41 Sensor RPLiDAR A2

La principal diferencia entre un sensor LiDAR 2D y uno 3D se muestra en la *Figura 4.42*, y es en la forma en la que emiten sus haces de luz láser al entorno. Un robot equipado con un sensor LiDAR planar 2D, será incapaz de detectar elementos por debajo de la altura a la que emite los haces láser, como lo está el gato, mientras que sí que será capaz de detectar obstáculos que estén a su misma altura, como lo es el humano. Los LiDAR 3D, en esencia, funcionan igual que los 2D, con la salvedad de que son capaces de efectuar varias capas o niveles de lectura desde el punto donde están situados. Habitualmente, 5 capas ya se considera un sensor LiDAR 3D. Debido a su ineficiencia para la detección de ciertos obstáculos, los sensores LiDAR 2D únicamente son empleados en tareas de navegación autónoma para la localización del robot y la percepción del entorno u obstáculos considerablemente grandes, como lo son las paredes [21], [59].

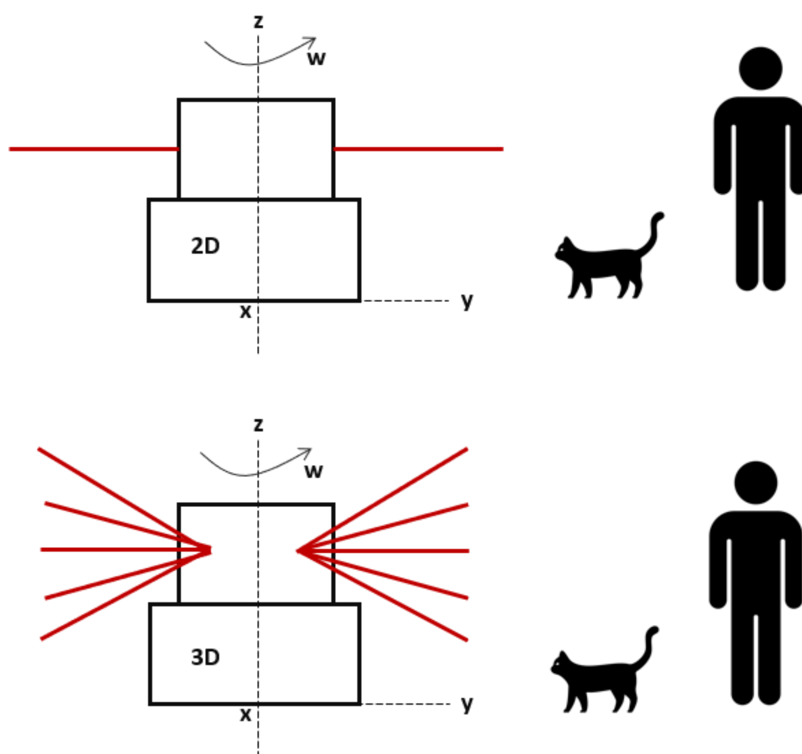


Figura 4.42 Detalle del Rango de Lectura de un Sensor LiDAR 2D vs 3D

En lo referente a sus características y especificaciones técnicas del RPLiDAR A2, se recogen en la siguiente *Tabla 4.6.*

Característica	Valor
Rango de Escaneo	360 º
Tipo de Escaneo	Planar 2D
Número de Muestras	4000 muestras/segundo
Distancia de Escaneo	6 m
Velocidad de Rotación	600 rpm
Resolución	0.9 º
Alimentación	5 VDC 1.5 A
Precio	280 – 330 €

Tabla 4.6 Especificaciones RPLiDAR A2

Este sensor LiDAR, desarrollado por el fabricante chino SLAMTEC, se trata de uno de los sensores más empleados para aplicaciones de navegación autónoma dentro de la comunidad de ROS. Tal es así, que cuenta con drivers tanto software como hardware para conectarse al bus USB de la controladora del Turtlebot2, de manera que su integración dentro del entorno ROS es sencilla. Este sensor, funciona y da soporte al cliente a través del paquete rplidar_ros, el cual establece la configuración del sensor, los parámetros necesarios para la comunicación con el mismo y el driver que interpreta los haces láser leídos y los convierte a un formato en el cual ROS sea capaz de trabajar.

4.2.3 Cámara Kinect XBOX 360

La cámara Kinect XBOX 360 se trata de un elemento muy conocido dentro de la comunidad ROS. Esta cámara desarrollada por Microsoft fue diseñada especialmente para la consola de videojuegos XBOX como un periférico que fuese capaz de detectar la posición y movimiento de los usuarios, dando una experiencia de realidad aumentada. Sin embargo, su bajo costo y su software de libre acceso han permitido que la Kinect 360 se haya convertido en uno de los elementos más empleados en ROS en lo relativo a la percepción de imagen y visión artificial [60].

Figura 4.43 Cámara Kinect XBOX 360

Esta cámara, no solo proporciona la posibilidad de tomar imágenes del entorno, sino que está equipada con una cámara infrarroja que puede suplir a un sensor LiDAR planar capaz de emitir haces de luz láser en un rango poco menor a 180 grados. También, está equipada con un micrófono capaz de reconocer la voz humana, y un motor que le permite inclinar la lente de manera autónoma para abarcar mayor rango de visión.

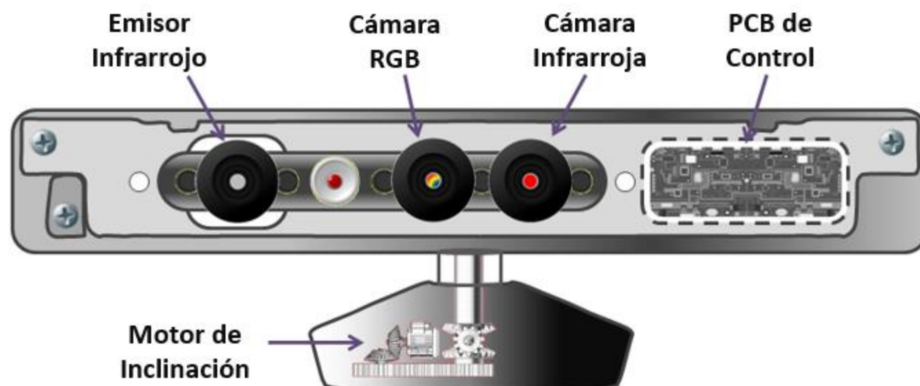


Figura 4.44 Distribución de los Elementos de la Cámara Kinect XBOX 360

En lo relativo a sus especificaciones y capacidades hardware, se recoge en la *Tabla 4.7* lo que la cámara Kinect ofrece. Cabe destacar que su protocolo de comunicación es el USB 2.0, por lo que únicamente puede conectarse en un puerto USB que trabaje bajo ese protocolo.

Característica	Valor
Campo de Visión Horizontal	57 º
Campo de Visión Vertical	43 º
Campo de Visión Infrarroja	175 º
Rango de Inclinación	± 27 º
Detección de Profundidad de Imagen	1.2 – 3.5 m
Flujo de Datos	30 frames / s // 640 x 480 32-bit
Audio	16-bit 16 kHz
Comunicaciones	USB 2.0
Precio	10.00 – 35.00 €

Tabla 4.7 Especificaciones Cámara Kinect XBOX 360

Existen diversos paquetes ROS para integrar la cámara Kinect dentro del entorno de ROS, más concretamente: freenect_launch, openni_launch y openni2_launch; entre otros. Todos ellos, proporcionan nodos capaces de tomar imágenes del entorno, detección de profundidad de las imágenes, operaciones de preprocesado básicas en estas, detección y lectura de códigos AR y QR y lectura y emisión de haces de luz infrarrojos. En el presente trabajo, por cuestiones de compatibilidad con la distro de ROS Kinectic, se empleará el paquete freenect_launch.

4.2.4 *Odroid XU4*

Para el control y gestión local de las unidades de transporte, se ha optado por emplear, ordenadores de placa reducida o SBC (Single Board Computer) a modo de CPU. Mas concretamente, se ha empleado un SBC de la gama Odroid, en concreto, el modelo XU4 mostrado en la *Figura 4.45.* Estos dispositivos, actúan como elemento central de cada una de las unidades de transporte, proporcionando a cada uno de los periféricos del robot y a la propia base móvil conectividad e integración dentro de la red de ROS.

Figura 4.45 Odroid XU4

A diferencia de otros SBC como las Raspberry o las Jetson Nano de NVIDIA, las Odroid proporcionan un equilibrio más que asequible entre sus prestaciones y su precio. En pocas palabras, se trata de un dispositivo que no está sobredimensionado para las funciones que va a cumplir como controladora local de cada uno de los AGV, pero cuenta con suficiente capacidad como para implementar lazos de control de altos requerimientos como es el ROS Navigation Stack. La Raspberry por su parte, dispositivo que suele venir incluida con la compra del Kobuki Turtlebot2, no es capaz de implementar el lazo de control requerido por el ROS Navigation Stack. En lo relativo a las especificaciones técnicas de la Odroid XU4, se muestra la *Tabla 4.8*. [22].

Característica	Valor
Procesador	Samsung Exynos5422 Cortex - A15 2Ghz
RAM	2Gbyte LPDDR3 RAM PoP
ROM	eMMC5.0 HS400 Flash Storage Interface
Sistema Operativo	Linux Kernel 5.4 LTS
Puertos USB 2.0	1 ud.
Puertos USB 3.0	2 ud.
Puertos Ethernet	1 ud.
Puertos HDMI	1 ud.
Alimentación	5 VDC - 4 A
Dimensiones	83 x 58 x 20 mm
Precio	60.00 – 85.00 €

Tabla 4.8 Especificaciones de la Odroid XU4

4.2.5 Router TL-WR802N

Para poder proporcionar a los AGV un acceso a la red, es necesario que dispongan de un elemento que les permita acceder a una red Wi-Fi, de manera que sean capaces de integrarse dentro del entorno de ROS y comunicarse con los distintos equipos y nodos que residen dentro de esta. Para ello, se emplean los router de la gama TL-WR802N, como el de la *Figura 4.46*. Estos dispositivos del fabricante estadounidense TP-Link, se conectan a la Odroid XU4 vía Ethernet, proporcionando acceso a la red a la unidad de transporte. De hecho, los TL-WR802N proporcionan la posibilidad de trabajar tanto como clientes, routers, puntos de acceso o incluso repetidores de una red Wi-Fi asociada; pudiendo incluso configurar tanto su propria dirección IP como la de aquellos elementos que dependan de él como la Odroid XU4.

Figura 4.46 Router TL-WR802N

Este dispositivo, como se ha podido ver, se caracteriza por su versatilidad, dotando al usuario todo lo necesario para poder trabajar como un router convencional. Sus especificaciones técnicas, se muestran en la *Tabla 4.9.*

Característica	**Valor**
Tasa de Transmisión Wi-Fi	N300: 2.4 GHz: 300 Mbps (802.11n)
Modos de Trabajo	Router, Punto de Acceso, Repetidor, Cliente, WISP
Puerto Ethernet	1 ud – 10/100 Mbps WAN/LAN
Alimentación	5 VDC - 1 A
Dimensiones	57 x 57 x 18 mm
Precio	22-00 – 37.00 €

Tabla 4.9 Especificaciones Router TL-WR802N

4.2.6 Estación AirPort Extreme A1354

Así como cada una de las unidades de transporte cuenta con un router TL-WR802 que proporciona de acceso a la red Wi-Fi donde reside el maestro ROS, es necesario un elemento que sea capaz de generar la red. Para ello, se emplea un AirPort Extreme modelo A1354, una estación Wi-Fi de la compañía Apple que es capaz de trabajar como enrutador, conmutador de red o bien como un punto de acceso inalámbrico. Este elemento, no reside dentro de ninguna de las unidades de transporte, puesto que ejerce de punto de acceso para todos los AGV del sistema.

Figura 4.47 Estación AirPort Extreme A1354

Al igual que los dispositivos TL-WR802N, la estación A1354 cuenta con una interfaz de configuración que permite customizar los servicios que esta ofrece. Dicha interfaz, se encuentra recogida dentro del software AirPort Utility, disponible para todas las plataformas en la propia página web del fabricante.

Característica	**Valor**
Tasa de Transmisión Wi-Fi	2.4 - 5 GHz: 300 Mbps (802.11n)
Modos de Trabajo	Router, Punto de Acceso, Repetidor.
Puertos USB 2.0	1 ud
Puertos Ethernet LAN	3 ud.
Puertos Ethernet WAN	1 ud.
Dimensiones	33 x 165.1 x 165.1 mm
Precio	25.00 – 85 .00 €

Tabla 4.10 Especificaciones Estación AirPort Extreme A1354

4.2.7 Brazo Robótico WidowX

Para dotar a las unidades de transporte la posibilidad de manipular objetos, se emplea un brazo robótico antropomórfico de seis grados de libertad de la gama WidowX, como el de la *Figura 4.48.* Este brazo, se trata de un componente muy empleado dentro de la comunidad de desarrolladores de ROS debido a su alta versatilidad y facilidad de uso, puesto que se trata de un elemento especialmente diseñado para aplicaciones académicas [61].

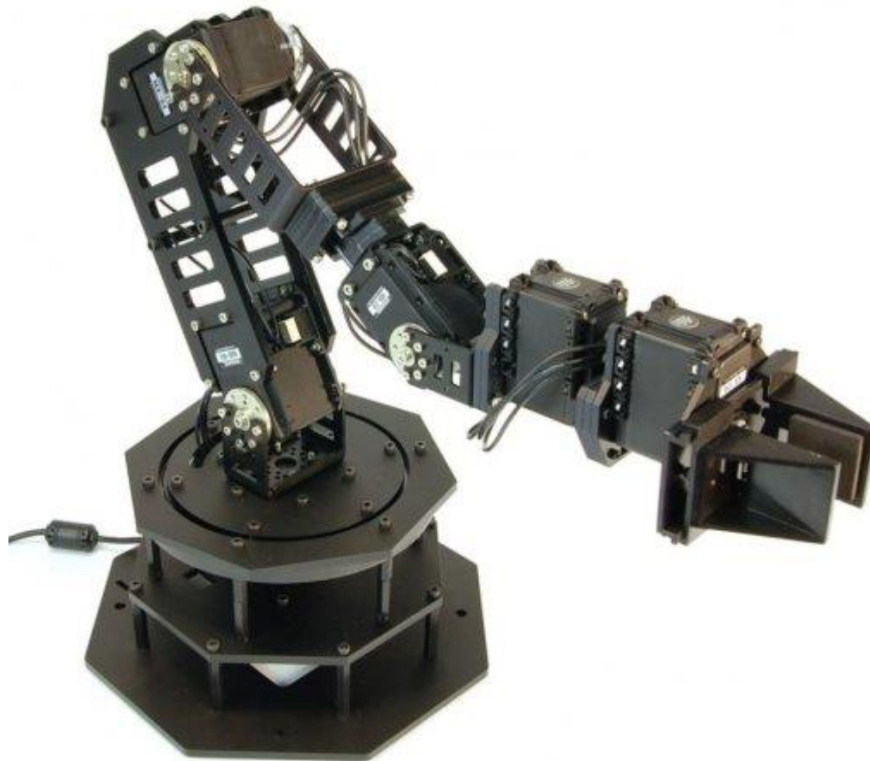


Figura 4.48 Brazo Robótico WidowX

Tal es así, que para llevar a cabo la comunicación e integración con el resto de los elementos que forman parte del entorno de ROS, este robot cuenta con un Arduino Mega en su base el cual sirve como Gateway entre ROS y los diferentes actuadores que gobiernan el brazo. Este Arduino, ejecuta las funciones propias del paquete de ROS arbotix, el cual, en esencia, generará un nodo denominado "/arbotix" que recibirá las posiciones o puntos a los que debe de dirigirse el TCP (Tool Centre Point). Por otro lado, en vez de trabajar con cinemática directa, también cabe la posibilidad de especificar las configuraciones que cada uno de los servos del brazo deben de tomar, dependiendo de la solución que quiera llevarse a cabo, aunque no es lo habitual para este caso de aplicación. Se muestra en la *Figura 4.49* el esquema de comunicaciones con dicho brazo.

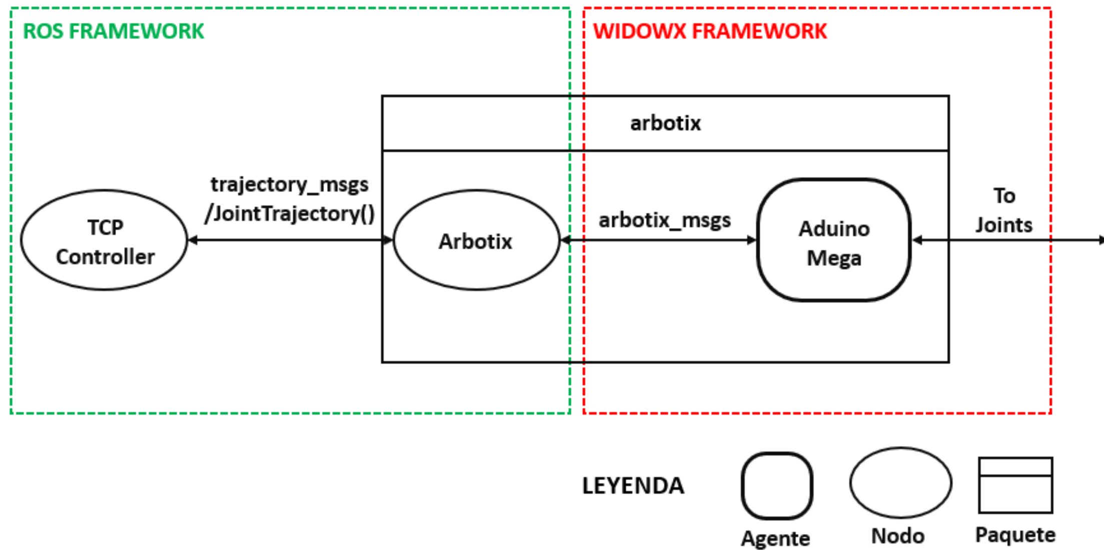


Figura 4.49 Comunicación ROS con Arbotix y Brazo Robótico

Tal y como puede observarse en la anterior figura, los mensajes ROS con los que trabaja el brazo robótico son del tipo trajectoy_msgs/JointTrajectory, los cuales toman los siguientes campos:

- **Header**: Identifica la trayectoria que se va a hacer, además del tiempo que tardará desde que empieza a realizarla hasta que la termina.
- **Joint Names**: Identifica a cada uno de los seis servos.
- **Points**: Incluye la acción que cada servo tiene que hacer para llegar desde el punto actual, hasta el siguiente. Esta información, puede aportarse mediante posición final, velocidad, aceleración o esfuerzo.

En lo relativo a su espacio de trabajo, se muestran los siguientes esquemas de la *Figura 4.50* y la *Figura 4.51*

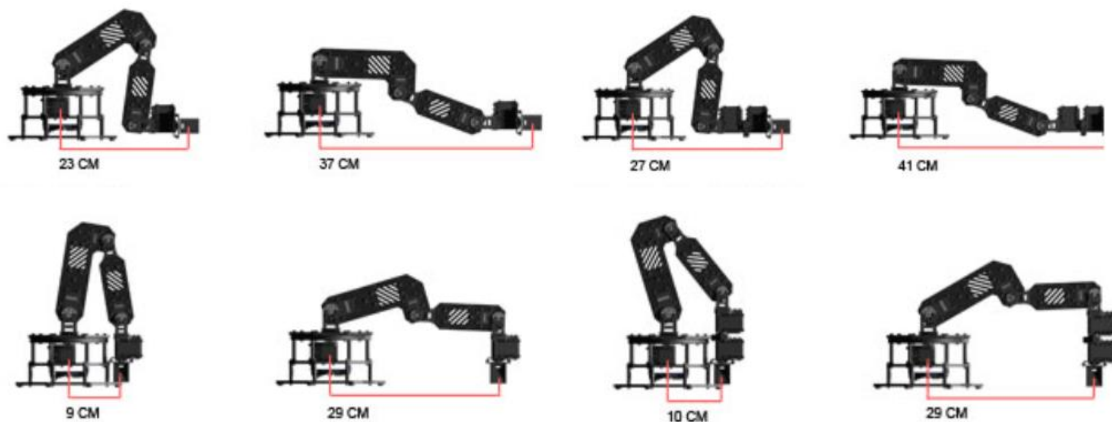


Figura 4.50 Espacio de Trabajo Singularidades

Figura 4.51 Espacio de Trabajo Planta

En lo referente a las especificaciones técnicas del brazo, se aporta la *Tabla 4.11.* Cabe destacar, que el paquete arbotix que gobierna el controlador del brazo robótico cuenta con la posibilidad de ser programado tanto online mediante guiado o aprendizaje de trayectorias, como offline, empleando las herramientas de RViz y Gazebo. De este modo, el nodo encargado de llevar a cabo la manipulación o TCP Controller, únicamente tendrá que recoger dichos puntos físicos y posteriormente enviárselos al nodo Arbotix en el orden requerido para llevar a cabo la manipulación y movimiento del brazo.

Característica	**Valor**
Alimentación	12 VDC – 10 A
Actuador Dynamxel MX-64	2 ud.
Actuador Dynamxel MX-28	2 ud.
Actuador Dynamxel AX-12A	2 ud.
Alcance Vertical	510 mm
Alcance Horizontal	370 mm
Fuerza	30 mm / 400 g ; 20 mm / 600 g ; 10 mm / 800 g
Fuerza Pinza	500 g
Fuerza de Elevación de la Muñeca	500 g
Peso	1330 g
Precio	1600 – 1800 €

Tabla 4.11 Especificaciones Brazo Robótico WidowX

4.2.8 Concentrador USB HUB UH720

Dado que ni la Odroid XU4 ni la base robótica Turtlebot2 proporcionan puertos USB suficientes para integrar todos los periféricos de cada unidad de transporte dentro de un mismo bus digital, es necesario integrar en el sistema un elemento capaz de ampliar los puertos USB disponibles. Este elemento, se trata del concentrador o "hub" USB de la gama UH720.

Figura 4.52 Hub UH720

En lo relativo a las especificaciones técnicas de este concentrador, cabe destacar que todos los puertos USB que aporta trabajan bajo el protocolo USB 3.0, tal y como se muestra en la *Tabla 4.12*.

Característica	Valor
Sistemas Soportados	Windows, Mac OS X y Linux
Puertos USB 3.0	7 ud. (2 de ellos especializados para carga)
Puertos USB 3.0 Micro B	1 ud. (Interfaz)
Alimentación	12 VDC - 4 A
Dimensiones	165 x 65.5 x 17.5 mm
Precio	60.00 – 85.00 €

Tabla 4.12 Especificaciones del Hub UH720

4.2.9 Convertidor Buck LM-2596

Pese a que la base robótica Turtlebot2 cuenta con varios puertos de alimentación, en función de las tareas que esté efectuando, puede darse el caso de fallos de alimentación o bajadas del nivel de potencia que suministra a dichos puertos. Esto, se debe a que la batería interna del Turtlebot2 es aquella que suministra la energía tanto a la propia base, como al resto de periféricos que formen la unidad de transporte. Sin embargo, en momentos puntuales, como cuando ha de suministrarse el par de arranque a los motores, pueden darse picos en otros suministros que

influyan en la alimentación de los periféricos. Por otro lado, pese a que el fabricante asegura que en los puertos de alimentación de la base robótica siempre se va a dar la tensión especificada, la realidad es que su potencial eléctrico de salida se va a ver afectado a medida que disminuye el nivel de voltaje de la batería.

Para que no se produzcan estos picos ni bajadas del potencial eléctrico en los puertos de alimentación, con objeto de que el suministro eléctrico sea constante a los periféricos de cada unidad de transporte, se ha empleado un módulo convertidor basado en el integrado LM2596 de Texas Instruments. Este elemento, da suministro eléctrico al concentrador USB UH720, de manera que este último, sea capaz de alimentar todos los periféricos conectados a sus puertos USB. Gracias a la presencia de este módulo convertidor, se asegura que el suministro eléctrico sea constante, atenuando en mayor medida los picos de alimentación producidos por la batería del Turtlebot2, asegurando que ninguno de los periféricos del AGV se reinicia por fallos de alimentación puntuales.

Figura 4.53 Módulo Convertidor Buck LM2596

En lo referente a sus especificaciones técnicas, tal y como puede comprobarse en la *Tabla 4.13*, se trata de un módulo muy sencillo y barato, el cual permite a través de un potenciómetro ajustar el CD (Cicle Duty) del Buck, de manera que pueda ajustarse la tensión de salida según las necesidades de la aplicación.

Característica	Valor
Alimentación	4.5 VDC – 40 VDC
Dimensiones	55 x 20 x 8 mm
Precio	0.74 – 1.80 €

Tabla 4.13 Especificaciones Módulo Convertidor Buck LM2596

4.2.10 MacBook Air A1369

Para llevar a cabo el desarrollo de este trabajo, se ha empleado un MacBook Air modelo A1369, al cual se le ha instalado el sistema operativo Ubuntu LTS 16.04, el requerido para trabajar con la distro de ROS Kinetic. Es necesario el instalar la distro de ROS en físico, puesto que debido a todos los recursos de tarjeta gráfica que consume el simulador Gazebo y la interfaz de RViz, es muy difícil hacer funcionar todas las herramientas necesarias de manera simultánea para el desarrollo del presente proyecto en una máquina virtual sobre otro sistema operativo. Tal es así, que en dicho portátil reside todo lo necesario para ejecutar un entorno ROS, así como todas las herramientas de programación, desarrollo y simulación que se han presentado anteriormente; además de las herramientas suficientes para lanzar un sistema MAS con JADE.

Figura 4.54 MacBook Air A1369

En lo relativo a las especificaciones técnicas del equipo que sean de utilidad o bien estén relacionadas con las necesidades del presente trabajo, se muestra la *Tabla 4.14*.

Característica	**Valor**
Procesador	Core i5 (I5-2557M)
Memoria ROM	256 GB
Memoria RAM	4 GB DDR3 SDRAM
Puertos USB 2.0	2 ud.
Batería	50 Wh – LiPo – 7 h
Precio	300.00 – 420.00 €

Tabla 4.14 Especificaciones MacBook Air

CAPITULO 5

DATOS DE PARTIDA

5 Datos de Partida

En este capítulo, se detallan todos los datos y elementos con los que se contaba al inicio del desarrollo del presente trabajo, así como de una breve descripción del entorno de trabajo en el que se pretende operar y sus principales características a considerar.

5.1 Entorno de Operación

El entorno de trabajo en el que las unidades autónomas de transporte deberán de operar se trata de uno de los laboratorios correspondiente al Departamento de Ingeniería de Sistemas y Automática, situado en el edificio B de la Escuela de Ingeniería de Bilbao. Concretamente, se trata del laboratorio correspondiente al GCIS, del cual se muestra en la *Figura 5.1* un plano simplificado de la distribución que toma el laboratorio en una vista de planta y sin escalar. Del mismo modo, se muestra a partir de la *Figura 5.2* unas fotos del entorno en la realidad. Tal y como puede comprobarse, el entorno está subdividido en 8 secciones diferentes y cuenta con 3 células.

- **Célula KUKA**: Se trata de una célula que cuenta con un robot antropomórfico de KUKA.
- **Célula UR**: Se trata de una célula que cuenta con un robot colaborativo de UR.
- **Células Virtuales**: Se trata del lugar que ocupan un conjunto de gemelos virtuales que simulan el comportamiento de varios tipos de robots dentro de la planta, de manera que puedan tenerse más células y procesos dentro del abanico de opciones productivas.
- **Sección A**: Se trata de la zona de operación central de las unidades de transporte autónomas, lugar en el que se sitúan las estaciones de carga de los AGV.
- **Sección B**: Se trata de la zona de operación dónde se encuentran las células virtuales.
- **Sección C**: Se trata de la zona de paso entre la célula UR y la célula KUKA.
- **Sección D**: Se trata de una zona del entorno sin una funcionalidad definida.
- **Sección E**: Se trata de una zona de paso entre la sección X y la sección central A.
- **Sección U**: Se trata de la zona de operación dónde se encuentra la célula correspondiente al robot colaborativo de Universal Robots.
- **Sección K**: Se trata de la zona de operación dónde se encuentra la célula KUKA, además de la entrada y salida del entorno de operación.
- **Sección X**: Se trata de una zona de operación que interconecta la Sección E y la Sección U, la cual está caracterizada por contar con un ventanal traslúcido que afecta a la navegación basada en LiDAR.

El principal objetivo de definir el entorno de operación en una serie de secciones y regiones, es la de estandarizar todas las zonas de manera que cuando vaya a definirse una coordenada o punto de traslado en concreto, esta tenga el prefijo correspondiente a la letra correspondiente a su sección, de manera que un punto situado en la Sección C, se indique mediante “C1”, siendo C el indicador de que se trata de un punto en la Sección C y el 1 el punto correspondiente a todos los definidos dentro de dicha sección.

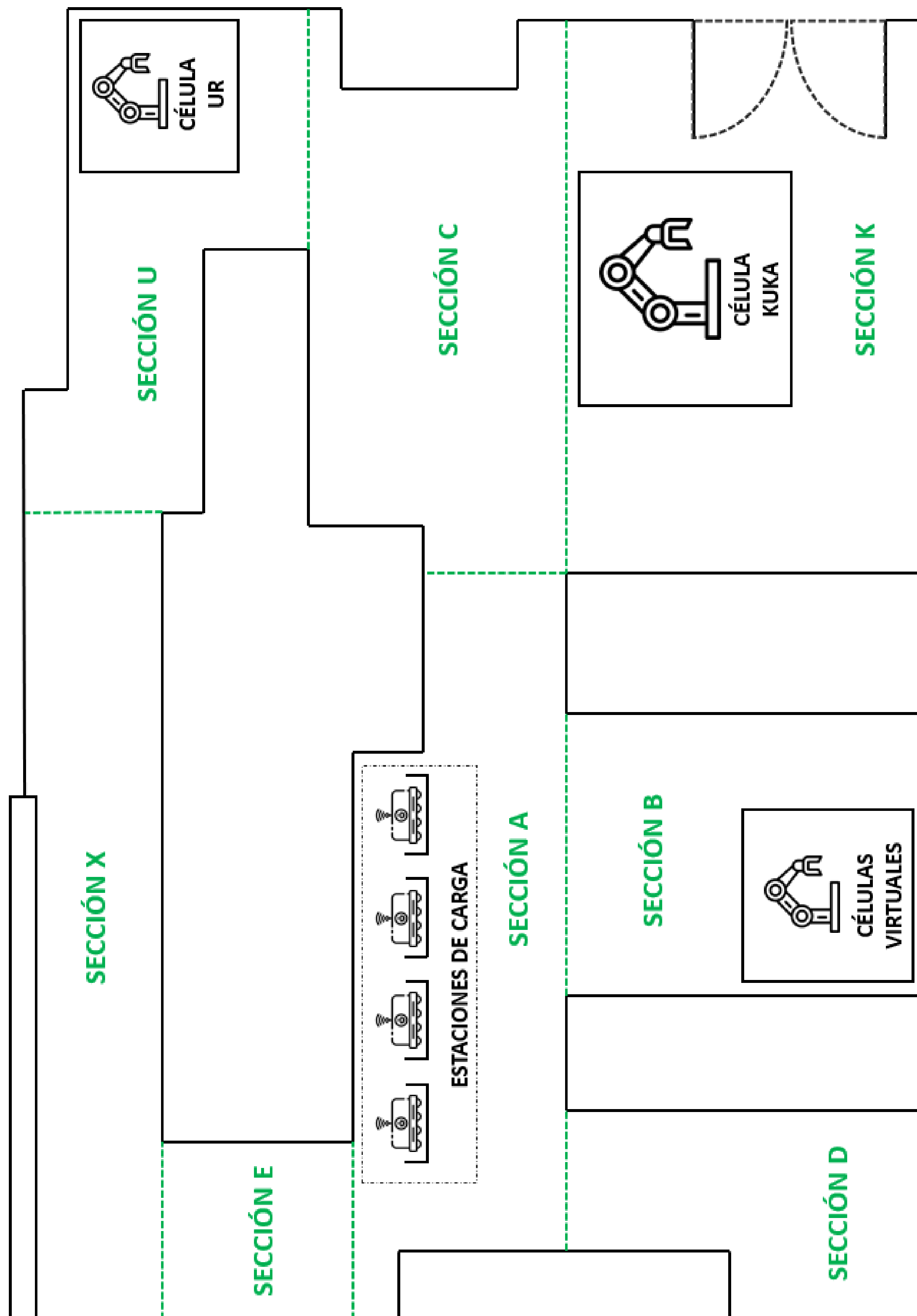


Figura 5.1 Plano Simplificado del Entorno de Operación de las Unidades de Transporte

Figura 5.2 Secciones C y K

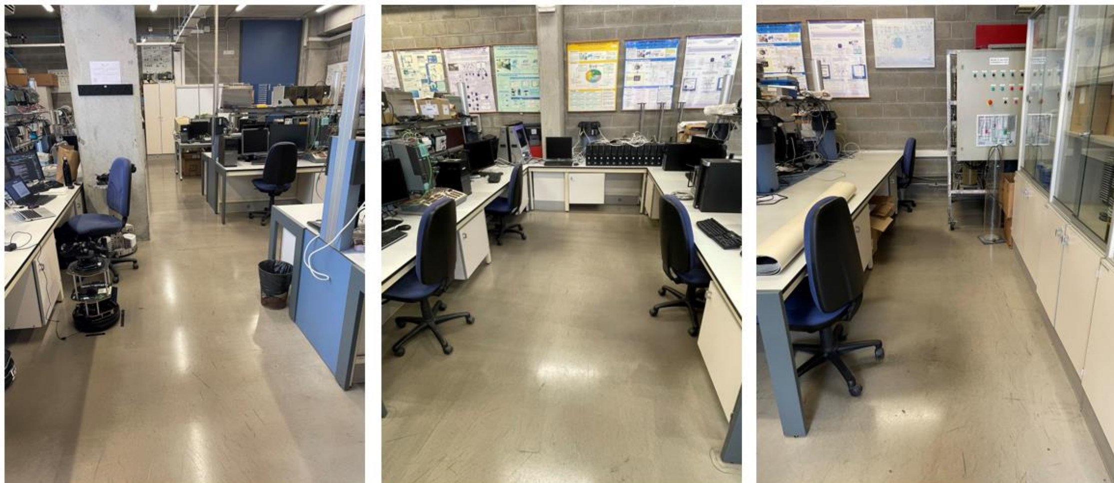
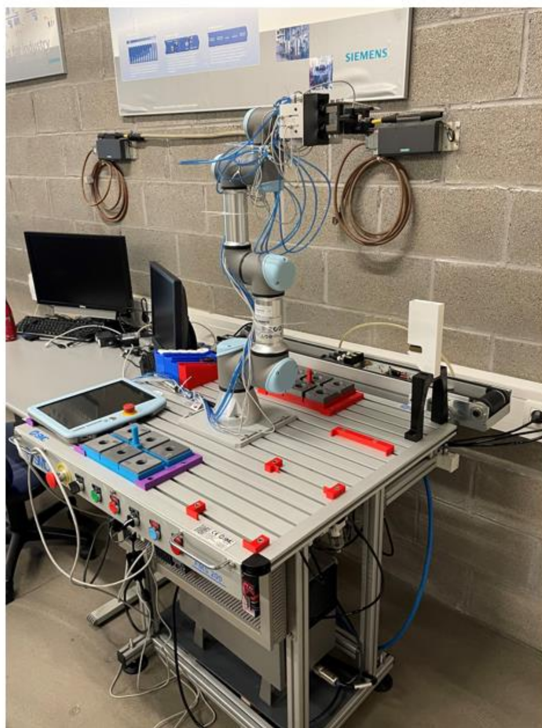


Figura 5.3 Secciones A (Izquierda), B (Centro) y D (Derecha)

Figura 5.4 Secciones X y U (Izquierda) y E (Derecha)



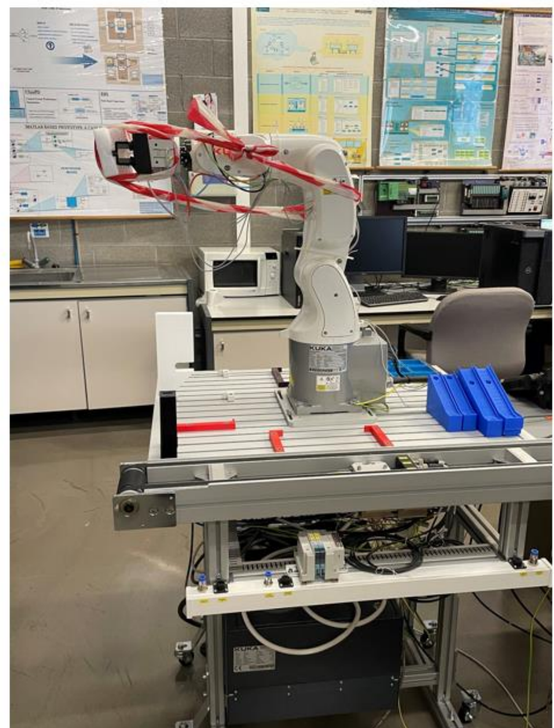


Figura 5.5 Células Robóticas UR (Izquierda) y KUKA (Derecha)

5.2 Material Hardware

Tal y como se ha comentado anteriormente, uno de los requisitos del presente trabajo era el empleo de las unidades robóticas de la gama Kobuki iCleBot Turtlebot2 como AGV, concretamente, el requisito R1.1. Esto, viene establecido previamente por la presencia de dichos modelos en el inventario del propio departamento, tal y como se emplearon en varios antecedentes del proyecto. Del mismo modo, desde un inicio, se contaban con varios elementos y hardware, de los cuales se muestra un inventario en la *Tabla 5.1*. Evidentemente, la presencia de dichos elementos alentó al empleo de estos para el desarrollo del presente trabajo, sobre todo en lo relativo a las cuestiones económicas.

Elemento	Unidades	Estado
Kobuki iCleBot Turtlebot2	4 ud	Tres bases operativas, una con problemas de alimentación y batería.
RPLiDARA2	4 ud	Los cuatro LiDAR en condiciones óptimas.
Cámara Kinect	1 ud	Cámara en estado óptimo.
Odroid XU4	4 ud	Tres de operativas, una con problemas de inicialización.
Router TL-WR802N	5 ud	Los cinco en estado óptimo, dos de ellos sin configurar correctamente.
Estación AirPort A1354	1 ud	Estación sin configurar, hubo que realizarse la configuración desde cero.
Brazo WidowX	2 ud	Ambos brazos en estado óptimo.
Hub UH720	4 ud	Todos los concentradores en estado óptimo.
Convertidor LM-2596	4 ud	Todos los convertidores en estado óptimo.

Tabla 5.1 Material Hardware Inicial

Del mismo modo, el equipo personal donde se llevó a cabo la codificación y el desarrollo más técnico no fue aportado desde un inicio, por lo que se optó por formatear un MacBook ya obsoleto e instalar el sistema operativo requerido por el requisito R1.2.

5.3 Material Software

En cuanto al material software con el que se cotaba en el inicio, se recogen las siguientes herramientas:

- **Licencia Jetbrains:** Para el uso de IDE profesionales para codificar tanto en Python como en Java.
- **Repositorio en OneDrive:** Para el almacenamiento de todo el material e información relativa al proyecto.
- **Tutoriales JADE**: Hasta cuatro tutoriales y ejemplos realizados por el propio GCIS para la formación relativa a la plataforma y arquitectura MINECO.
- **Tutoriales ROSJava**: Hasta ocho ejemplos de clases Java realizadas por el GCIS para la formación relativa al uso del paquete ROSJava.

5.4 Comparativa Unidades Robóticas

Cabe destacar, que en la etapa inicial y de viabilidad del presente trabajo se planteó la posibilidad de emplear por parte del GCIS otro tipo de unidades robóticas a los Turtlebot2, más concretamente su siguiente modelo, el Turtlebot3. Esta posibilidad, vino impulsado por la alternativa de realizar el presente trabajo en ROS2, en vez de en ROS convencional, el cual se espera que en unos años quede obsoleto y sin soporte técnico. Sin embargo, tras un profundo análisis de dichas unidades robóticas Turtlebot3, se optó por realizar el presente proyecto con los Turtlebot2 por dos principales razones:

- **Solución Escalable**
  - El proyecto, se realizaría en el entorno de ROS de todas formas, por lo que podría migrarse llegado el momento desde ROS convencional a ROS2 sin un esfuerzo considerable.
- **Incertidumbre Logística.**
  - Dada la actual crisis de componentes y semiconductores, existía una gran incertidumbre en cuanto a la llegada de los Turtlebot3, por lo que no podía estancarse el proyecto esperando su llegada.

Sin embargo, pese a que uno de los objetivos secundarios del proyecto, concretamente el S2, era el dejar abierta la posibilidad de que el sistema fuese escalable en un futuro, se optó por adquirir las unidades robóticas Turtlebot3 para futuros desarrollos y migración al entorno de ROS2. De tal modo, se muestra una breve comparativa entre ambas unidades robóticas en la

Tabla 5.2 para definir las principales diferencias entre ambos.

Característica	Turtlebot2	Turtlebot3
Lanzamiento	2015	2020
Precio	560.00 - 699.00 €	499.00 – 1199 €
Batería	2:30 – 7 h	2:30 h
Peso	2350 g	985 – 1890 g
CAD Abierto	No	Si
Plataformas Soportadas	ROS	ROS / ROS2
Gazebo	Si	Si
Soporte Técnico	No	Si
Cámara Incorporada	No	Si

Tabla 5.2 Comparativa Turtlebot2 con Turtlebot3

En resumen, el Turtlebot3 ha sido diseñado para abarcar un mayor público, mayoritariamente amateur o académico, al hacerlo más accesible y sencillo a su predecesor. A diferencia de este, cuenta con dos modelos diferentes: El Buger, de menor tamaño y peso; y el Waffle Pi, el cual permite incorporar un brazo robótico tipo WidowX. La posibilidad de acceder al CAD de manera gratuita permite el diseñar piezas que se adapten a la estructura propia del robot de manera sencilla y con conocimientos simples de diseño por ordenador, de forma que podrían realizarse modelos de impresión 3D de manera muy simple. Con todo, la posibilidad de que muchos de los drivers necesarios para su funcionamiento estén integrados dentro de todas las distros de ROS2, facilita enormemente el desarrollo de proyectos en dicho entorno sin la necesidad de tener que diseñar puentes o parches de un entorno a otro.

CAPITULO 6

DESARROLLO DE LA SOLUCIÓN

.

6 Desarrollo de la Solución

En este capítulo, se detallan todos los aspectos de la solución llevada a cabo, así como la justificación de las decisiones de diseño tomadas.

6.1 Estructura General de la Solución

La solución propuesta en el presente trabajo se desglosa en tres sistemas claramente diferenciados entre sí: el Sistema de Navegación, correspondiente a las capas más bajas de la arquitectura; el MAS o Sistema Multi-Agente, dónde residen los agentes encargados de la gestión de cada transporte; y el Sistema Gateway, el cual habilita la comunicación entre los dos sistemas previamente mencionados, o lo que es lo mismo, realiza el puente entre el entorno de ROS y el entorno de Java. En la *Figura 6.1* se muestra un esquema de comunicación simplificado de dichos sistemas.

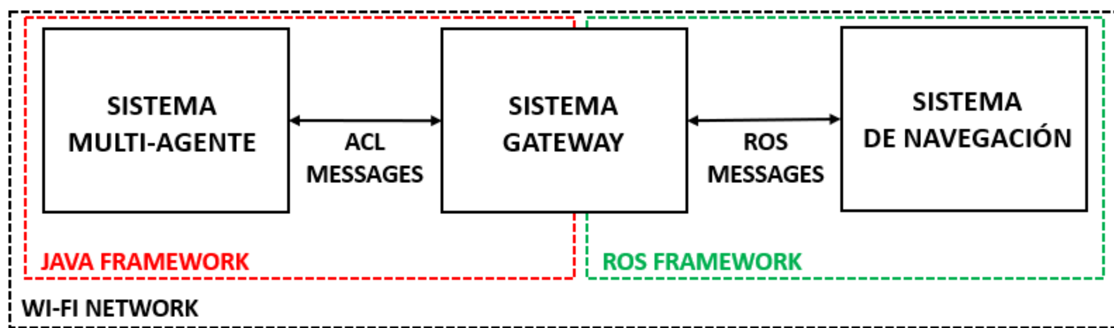


Figura 6.1 Esquema de Comunicación de los Sistemas de la Solución

Cada uno de estos sistemas cuenta con una función y objetivos claramente predefinidos, los cuales responden a las necesidades y especificaciones del presente trabajo. Estas funciones se resumen como sigue a continuación:

- **Sistema de Navegación**. Se trata del conjunto de nodos, periféricos y elementos implicados en la navegación autónoma de las unidades de transporte empleadas para el traslado y reabastecimiento de material a las células.
- **Sistema Gateway**. Este sistema integra todos los paquetes, librerías y nodos necesarios para llevar a cabo la comunicación entre un sistema desarrollado en un entorno Java, como es el MAS, y otro desarrollado en ROS, de manera que los transportes puedan recibir encomiendas desde los agentes que residen en el MAS.
- **Sistema Multi-Agente**. Se trata del conjunto de agentes que gestionan todo el conjunto de la planta. En este trabajo, se entiende y limita el MAS al conjunto de agentes que se relacionan directamente con las unidades autónomas de transporte, además de aquellos agentes necesarios para la coordinación de éstos.

Tanto el Sistema de Navegación como el Sistema Multi-Agente están completamente desacoplados el uno del otro, de manera que la gestión de las tareas que realiza cada uno de ellos está completamente modularizado. La interacción entre ambos se realiza a través de dos tópicos ROS: uno enfocado a la escritura y petición de comandos por parte de un operario de planta o agente al Sistema de Navegación, y otro centrado en la lectura de datos relevantes del Sistema de Navegación por parte de los agentes. El Sistema Gateway incorpora todas las herramientas necesarias para, por una parte, interpretar los mensajes ROS de ambos tópicos, y, por otra parte, convertirlos a un formato entendible para las clases Java sobre las que se ejecutan los agentes. Estos dos tópicos para lectura y escritura se recogen dentro del sistema como /flexmansys/coordenada/TUN y /flexmansys/state/TUN, tal y como muestra la *Figura 6.2*.

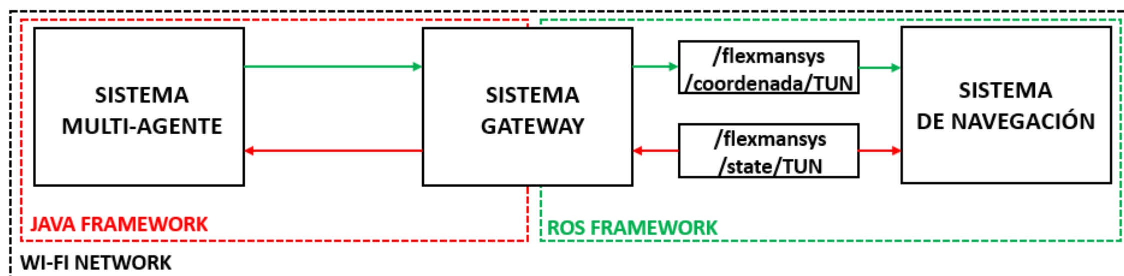


Figura 6.2 Esquema de Comunicación de los Sistemas a través de Tópicos

Sin embargo, para llevar a cabo el desglose de sistemas de la solución propuesta, ha sido necesario llevar a cabo una serie de configuraciones y distribuciones complementarias. Es por ello, que, a lo largo del presente capítulo, se detallará todo lo necesario para comprender el conjunto de la solución propuesta, además del funcionamiento más específico de cada uno de los tres sistemas que engloban al presente trabajo. De este modo, se llevará a cabo el siguiente orden de desarrollo para los siguientes apartados:

1. **Distribución de las Unidades de Transporte**: En este apartado se mostrará la distribución de todos los elementos y periféricos para cada unidad de transporte.
2. **Distribución y Configuración de Redes**: En este apartado se presentará la configuración de redes llevada a cabo para la comunicación e integración de todos los sistemas.
3. **Sistema de Navegación**: Donde se desarrollará todo lo relativo a la navegación, así como elementos propios de dicho sistema como la percepción de imagen.
4. **Sistema Gateway**: Donde se desarrollará todo lo relativo a la interconexión ROS-Java
5. **Sistema de Agentes de Transporte**: Donde se recoge todo lo relativo a la integración de la solución propuesta en el MAS implementado en JADE.
6. **Distribución y Operación en Planta**: Apartado en el cual se mostrará la distribución final de las unidades de transporte en el entorno de navegación y se desglosarán los distintos comportamientos que el sistema tendrá para según qué situaciones o peticiones de transporte.

6.2 Distribución de las Unidades de Transporte

En lo relativo a la distribución de las unidades de transporte, teniendo en cuenta los datos de partida, se ha optado por el montaje y puesta a punto de cuatro unidades de transporte. Dichas unidades y sus características vienen recogidas en la siguiente *Tabla 6.1*.

Unidad Transporte	Etiqueta TUN	Sensor RPLiDAR	Brazo WidowX	Cámara Kinect	Tipo Configuración
T_01	Leonardo	Si	Si	Si	Completa
T_02	Raphael	Si	Si	No	Media
T_03	Donatello	Si	No	No	Mínima
T_04	Michelangelo	Si	No	No	Mínima

Tabla 6.1 Equipamiento Unidades de Transporte

De dicha tabla, es importante destacar una serie de puntos los cuales engloban a toda la solución desarrollada en cuanto a la distribución de unidades de transporte:

- Existen tres tipos de distribuciones según los elementos que cuentan equipados, es decir: AGV Completo (Navegación, Manipulación y Visión incorporados), AGV Medio (Navegación y Manipulación), y AGV Mínimo (Únicamente Navegación).
- Únicamente dos unidades de transporte cuentan con la posibilidad de manipular objetos, como lo son aquellos equipados con un brazo robótico WidowX: T_01 y T_02.
- Únicamente una unidad de transporte cuenta con la posibilidad de detectar obstáculos y códigos AR o QR, como lo es aquella equipada con la cámara Kinect: T_01.
- Todas las unidades de transporte comparten ciertos elementos y periféricos básicos, como lo son: El router TL-WR802N, la Odroid XU4, el convertidor LM2596, la base robótica Kobuki iCleBot Turtlebot2 y el hub UH720.
- Finalmente, se destaca la presencia de la etiqueta TUN (Transport Unit Name), la cual da nombre a las diferentes unidades de transporte. Dicha variable, se encuentra recogida dentro del archivo de configuración de redes hosts.cfg y en el fichero bash a modo de constante, de manera que la dirección MAC (Media Access Control) de cada una de las unidades de transporte quede recogida dentro de una etiqueta lingüística. Esto, da la posibilidad de que tanto el sistema de navegación como el MAS se entiendan a la hora de asignar según qué tareas a que unidad de transporte, de manera que no se asigne una tarea de manipulación a un transporte que no cuenta con un brazo robótico WidowX. Estas etiquetas lingüísticas se han recogido de los nombres aportados a cada transporte en los antecedentes del presente trabajo.

Pese a que existan tres tipos de configuraciones, la distribución de los periféricos y sus interconexiones es muy similar para cada unidad de transporte. Tal es así, que se muestran en la *Figura 6.3* y la *Figura 6.4* el esquema para un AGV de configuración completa.

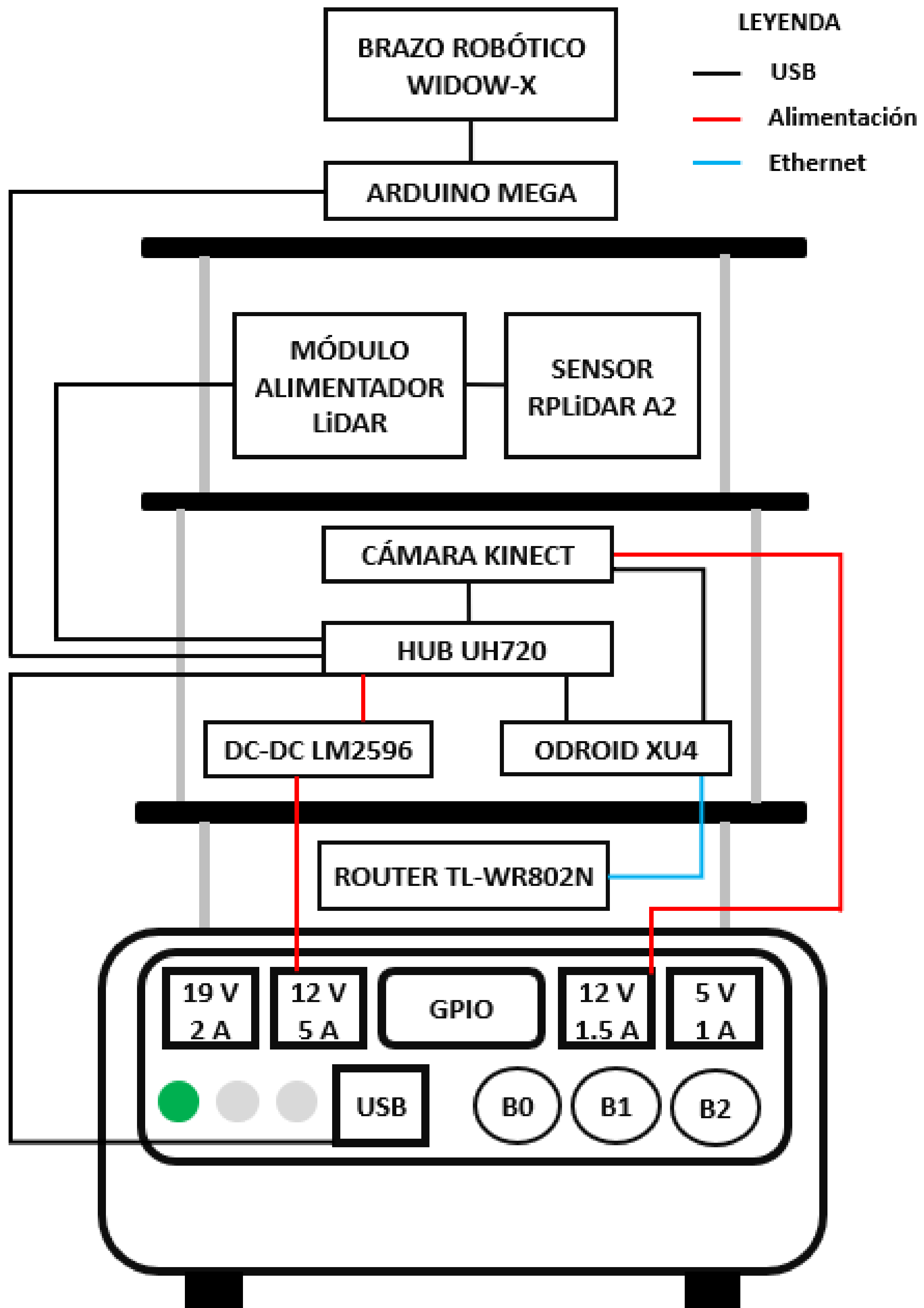


Figura 6.3 Esquema de Conexiones de Elementos de un AGV Completo Vista Trasera

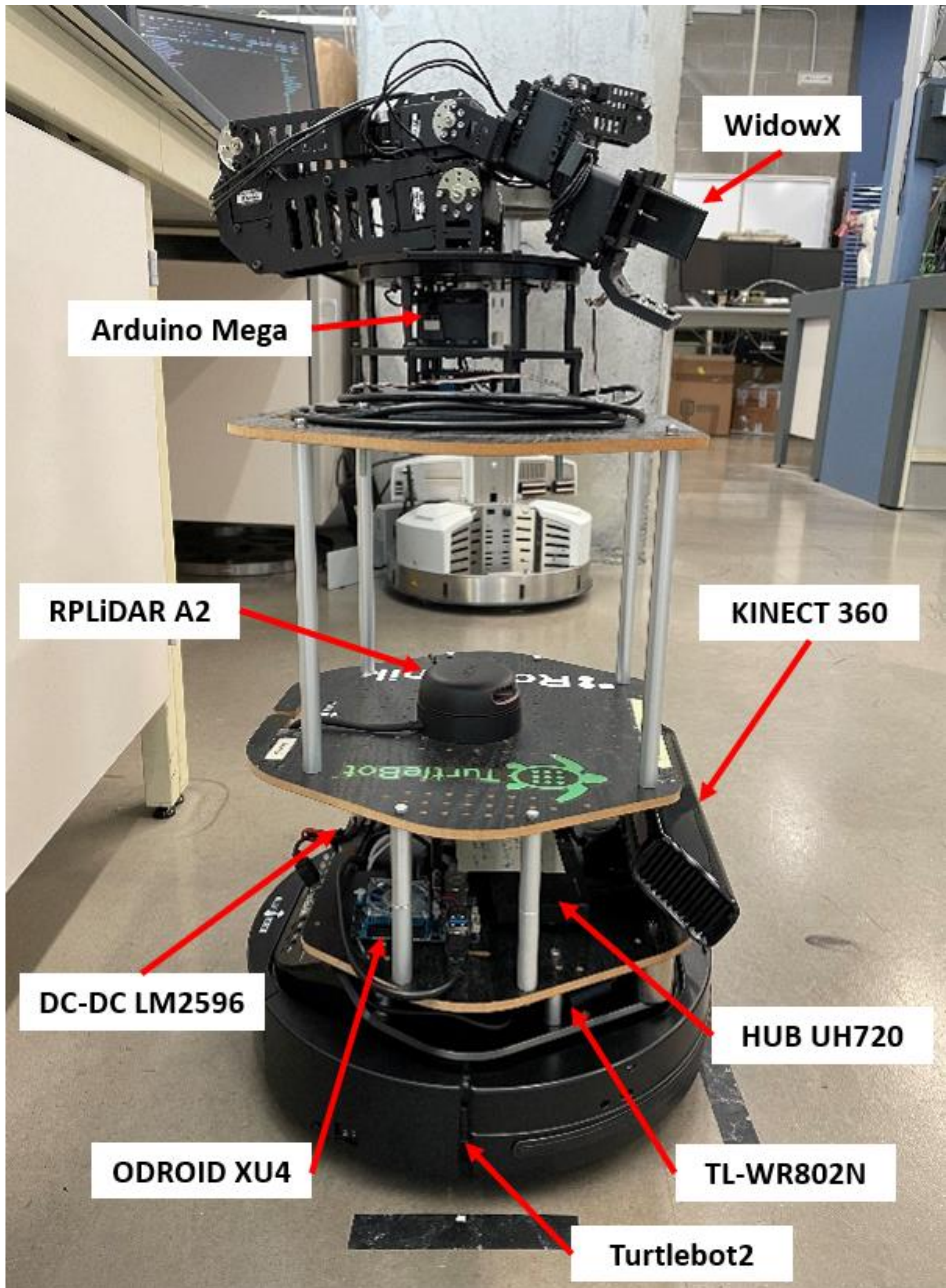


Figura 6.4 Distribución de Periféricos en AGV Completo

6.3 Distribución y Configuración de Redes

En lo que concierne al desarrollo del entorno de ejecución del entorno de ROS, se ha optado por una red Wi-Fi convencional a través de la cual se comuniquen todos los componentes y elementos necesarios para la ejecución del sistema al completo. Esta red es generada por la estación Airpot Extreme A1354 y se trata de una red de área local o LAN (Local Area Network) sin acceso a internet.

Tal y como se ha comentado anteriormente, para cada AGV se emplea una SBC como elemento central encargado de toda la computación y gestión relativa de cada transporte. Esta SBC es una Odroid XU4, la cual de por sí no cuenta con un módulo Wi-Fi que le permita conectarse a una red de dichas características. Es por ello que se emplea un router de la gama TL-WR802N, el cual se conecta a través del puerto ethernet de la Odroid. De este modo, la unidad de transporte correspondiente puede conectarse e integrarse a la red Wi-Fi en cuestión.

En esencia, son estos tres elementos mencionados anteriormente, la estación Airport, el router TL-WR802N y la SBC Odroid XU4 los que permiten llevar a cabo la comunicación con las capas más altas del sistema, como lo son los agentes que residen en el MAS, además del resto de las unidades de transporte que conforman el Sistema de Navegación. Se muestra en la *Figura 6.5* una simplificación de la distribución que toman estos elementos en la comunicación entre el MAS y el AGV de TUN Leonardo.

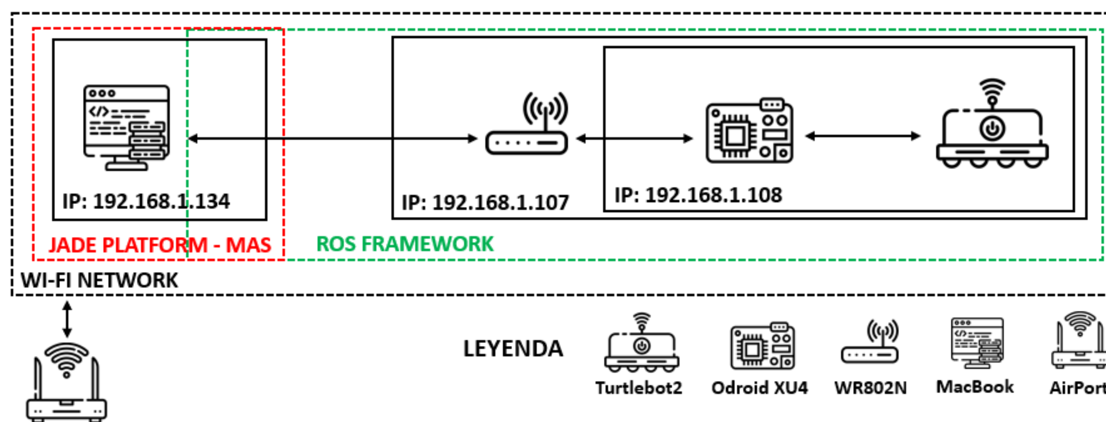


Figura 6.5 Esquema de Comunicación MAS-AGV

Como ha podido observarse, en la comunicación toman parte tres direcciones IP distintas, estando la correspondiente a la Odroid XU4 detrás del router TL-WR802N. Esto se deb, a que estos dispositivos alojan un servidor DHCP (Dynamic Host Configuration Protocol) que permite asignar direcciones IP específicas según la dirección MAC de un dispositivo cualquiera, de manera que siempre que se inicie el router este asigne la misma dirección IP a cada dispositivo. En este caso, al tener únicamente un elemento conectado por cada transporte, es decir, una Odroid XU4, cada router asignará únicamente una dirección IP, la de su Odroid XU4 asociada. Esta funcionalidad puede configurarse accediendo a la interfaz que provee el fabricante para el dispositivo o bien conectándose mediante ethernet con un equipo.

La presencia de cuatro unidades de transporte en el sistema requiere la necesidad de cuatro direcciones IP diferentes de manera que los nodos ROS que lancen cada uno de ellos puedan coincidir en puerto, pero nunca en un mismo socket. Tal es así, que para comunicarse con

cada transporte dentro de la red LAN, únicamente es necesario enrutar la dirección correspondiente a la Odroid XU4. El conjunto de estas direcciones IP se recoge en la *Tabla 6.2*.

Unidad Transporte	Etiqueta TUN	IP Odroid XU4	TL-WR802N IP
T_01	Leonardo	192.168.1.108	192.168.1.107
T_02	Raphael	192.168.1.106	192.168.1.105
T_03	Donatello	192.168.1.104	192.168.1.103
T_04	Michelangelo	192.168.1.101	192.168.1.100

Tabla 6.2 Direcciones IP Transportes

Para identificar cada uno de los elementos que se integran en una red LAN, ROS emplea dos variables: ROS_MASTER, la cual recoge la dirección IP del equipo dónde reside y se ejecuta el nodo maestro de ROS; y ROS_HOSTNAME, la cual establece la dirección IP a la cual todos los nodos ROS de dicho equipo deben direccionarse para que todos los nodos y tópicos del sistema puedan encontrarse entre sí al consultar el registro del nodo maestro.

Estas dos variables, se definen en el fichero bash de cada una de las Odroid XU4, además del equipo en el que se ejecute el nodo maestro y el MAS. El conjunto de IP que conforman el sistema y la etiqueta bajo la que se guardan en el bash correspondiente para cada equipo se define en la *Tabla 6.3*. Nótese, que el ROS_HOSTNAME también define el puerto asociado al nodo maestro, puesto que el puerto 11311 está reservado en ROS para la comunicación con dicho nodo.

Elemento	ROS_HOSTNAME	ROS_MASTER
T_01	192.168.1.108	http://192.168.1.134:11311
T_02	192.168.1.106	http://192.168.1.134:11311
T_03	192.168.1.104	http://192.168.1.134:11311
T_04	192.168.1.101	http://192.168.1.134:11311
Equipo Maestro	192.168.1.134	http://192.168.1.134:11311
Wi-Fi Network	Arkham_Asylum	

Tabla 6.3 Direcciones IP Sistema de Navegación

Finalmente, cabe destacar que la integración de las unidades de transporte dentro de una misma red LAN permite no solo que se integren al entorno ROS, sino que es completamente necesario para la puesta en marcha de todos los elementos necesarios para el Sistema de Navegación, la cual se realiza desde el equipo maestro mediante el protocolo SSH (Secure Shell). Además, cabe mencionar que , dado que la red generada por la estación Airport no cuenta con acceso a internet, los relojes internos de todos los equipos no pueden sincronizarse con la hora mundial, elemento clave para la comunicación dentro de ROS. Para ello, se emplea Chrony, una aplicación que implementa el protocolo NTP (Network Time Protocol), permitiendo que todos los elementos del sistema se sincronicen a una única referencia horaria, en este caso, a la del equipo maestro.

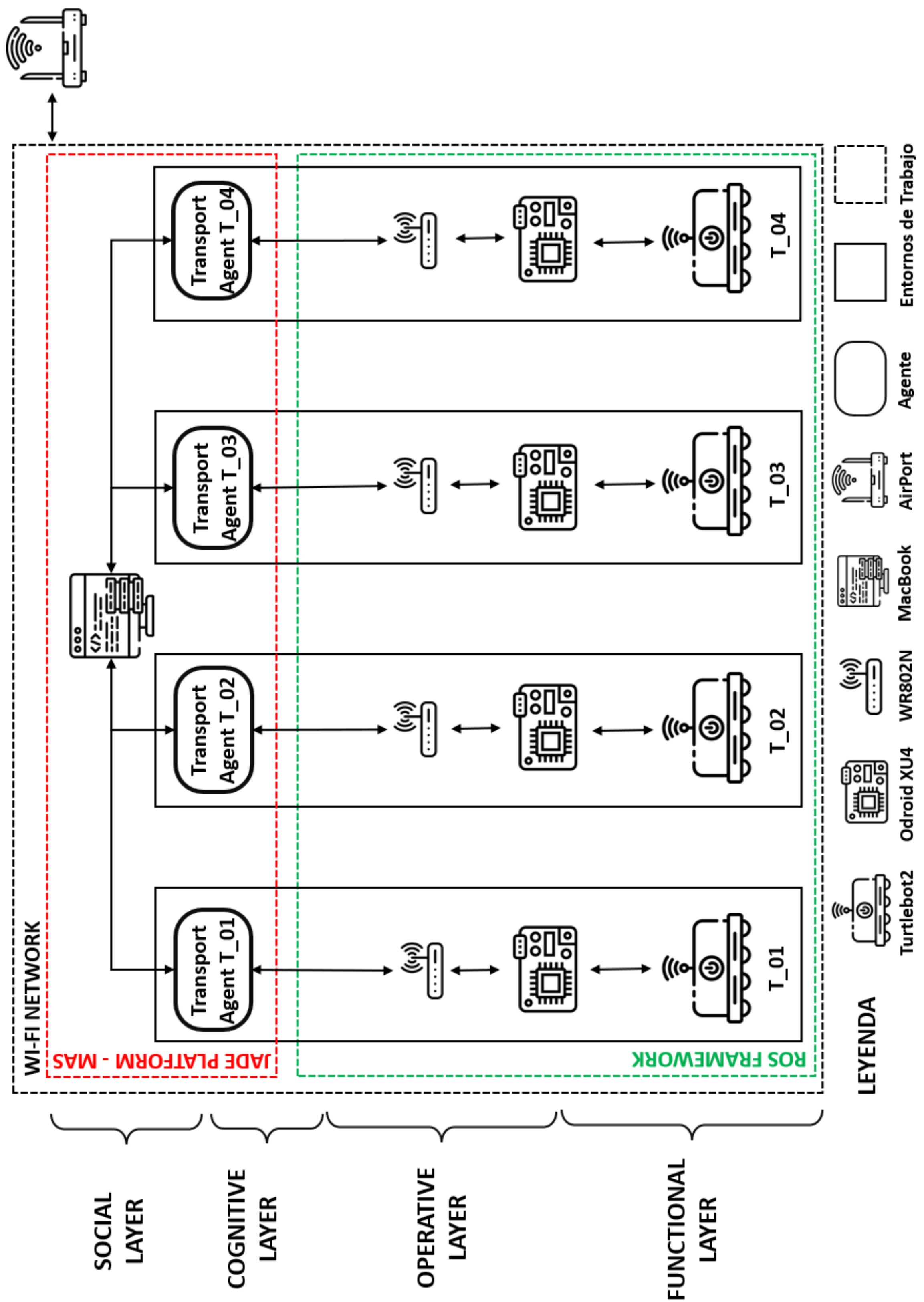


Figura 6.6 Distribución y Configuración General de los Elementos en la Red

6.4 Sistema de Navegación

Tal y como se ha mencionado al inicio de este capítulo, el Sistema de Navegación es una caja negra con la cual el Sistema Multi-Agente únicamente puede interactuar mediante dos tópicos: uno para la lectura del estado de cada transporte y otro para la solicitud de peticiones de operación a las unidades de transporte.

De este sistema, es importante destacar la estructura tanto general como la estructura software que compone al conjunto de elementos que lo forman. Por otro lado, es importante mencionar que cada una de las unidades de transporte tienen un comportamiento equivalente al de una máquina de estados, de manera que la situación en la que se encuentre cada transporte sea fácilmente identificable para los agentes del MAS. Finalmente, hay que destacar la presencia del Sistema de Percepción de Imagen, encapsulado dentro de las funcionalidades del sistema de navegación. Todos estos elementos se irán desglosando a lo largo del presente apartado.

6.4.1 Estructura General del Sistema de Navegación

En lo relativo a la estructura general del Sistema de Navegación, hay que comprender que toman parte prácticamente la totalidad de los periféricos de cada unidad de transporte. Del mismo modo, al tratarse de un sistema prácticamente encapsulado y modularizado en su totalidad, este podría desacoplarse completamente del Sistema Gateway y del Sistema Multi-Agente y ser operado de manera manual a través de una persona operaria u otro sistema de gestión no basado en agentes. Esta característica, permite al Sistema de Navegación plena flexibilidad, de manera que pueda ser integrado en diversas arquitecturas, siempre y cuando éstas cuenten con algún elemento que les permita trabajar y comunicarse mediante ROS.

Dado que el sistema en su conjunto cuenta con una distribución específica tanto software como hardware, se muestran dos esquemas diferentes: El primero en la *Figura 6.7*, donde se muestra un esquema más generalista de todo el conjunto; y el segundo en la *Figura 6.8*, donde se hace hincapié en todos los elementos software que permiten y son necesarios para ejecutar el Sistema de Navegación.

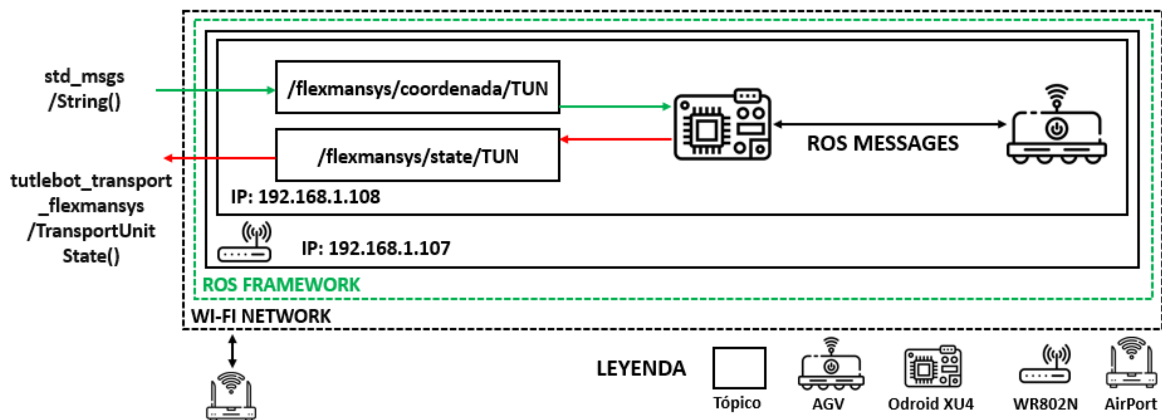


Figura 6.7 Estructura General Sistema Navegación

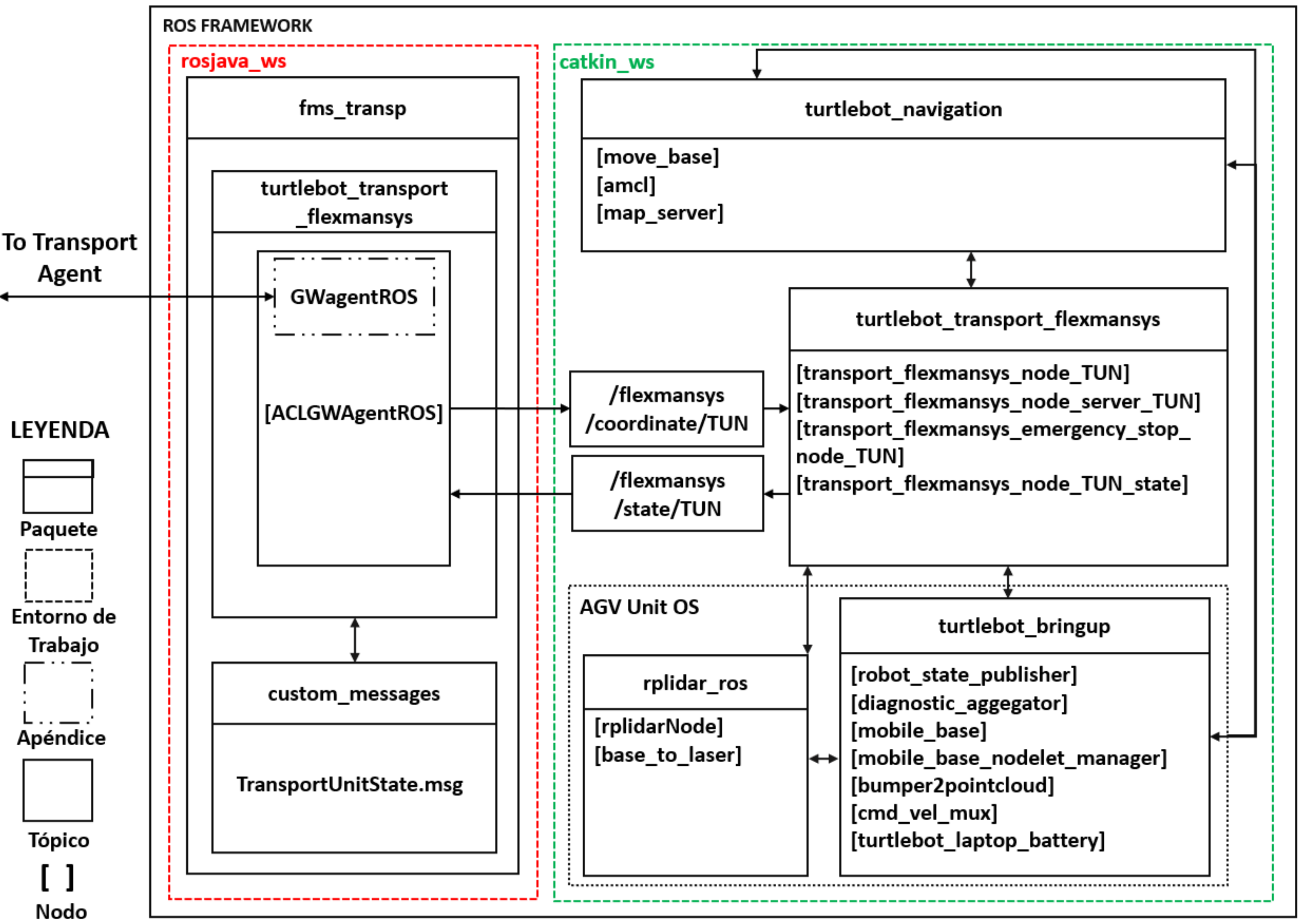


Figura 6.8 Estructura General Software del Sistema de Navegación

6.4.2 Estructura Software del Sistema de Navegación

El sistema de navegación reside en la capa funcional y operativa de la arquitectura, y se ejecuta en varios dispositivos integrados dentro del Framework creado por ROS. La estructura general del sistema, así como las interacciones entre todos sus elementos, se muestra en la *Figura 6.8,* mientras que en la *Figura 6.9* puede observarse la distribución física de cada uno de los componentes del sistema de navegación, es decir, dónde reside y se ejecuta cada uno. Este Framework se divide a su vez en otros dos entornos: "catkin_ws", donde reside todo lo relativo a la navegación, comunicación y gestión de los elementos más bajos de la arquitectura: y "rosjava_ws", cuya labor es la de permitir la comunicación entre las unidades de transporte y los agentes a modo de Gateway.

6.4.2.1 Entorno de Trabajo catkin_ws

En lo relativo a catkin_ws, toda su funcionalidad se divide en cuatro paquetes ROS: turtlebot_transport_flexmansys, turtlebot_navigation, turtlebot_bringup y rplidar_ros; el paquete arbotix para el brazo WidowX se contendría dentro de este entorno. Pese a que toda la funcionalidad se recoge dentro del paquete turtlebot_transport_flexmansys, el paquete generado y diseñado exclusivamente para la integración del sistema de navegación en cualquier unidad de transporte, los archivos ".launch" de dicho paquete referencian librerías de los otros cuatro paquetes. Esto supone que es necesario que el equipo en el que se vaya a ejecutar el sistema de navegación tenga estos paquetes ya instalados, los cuales se pueden obtener de manera gratuita en la propia web de ROS. A continuación, se resumen las funcionalidades de cada uno de los paquetes y nodos más destacables dentro de los mismos:

6.4.2.1.1 Paquete turtlebot_transport_flexmansys

Paquete encargado de la gestión y coordinación de todo el sistema de navegación. En él, reside toda la lógica y control de las unidades de transporte, así como las funciones de seguridad e interfaz, además de la configuración de los parámetros de navegación que posteriormente empleará el paquete turtlebot_navigation. Al ser el paquete central, se encarga de invocar o lanzar a través de sus archivos ".launch" al resto de paquetes que componen el sistema de navegación. A su vez, también contempla las funcionalidades necesarias para invocar nodos del tipo gmapping y generar un mapa del entorno, así como la posibilidad de integrar cámaras RGB mediante el paquete "ar_track_alvar". En lo relativo a sus nodos más significativos:

- **transport_flexmansys_node_TUN:** Se trata del nodo main o nodo principal de todo el sistema de navegación. En él se ejecuta y coordina la máquina de estados en la que se basa el sistema, además de gestionar la comunicación con la plataforma de agentes.

- **transport_flexmansys_node_server_TUN:** Se trata de un nodo servidor con respecto al nodo main. Su función, se centra en enviarle al nodo move_base, correspondiente al paquete turtlebot_navigation, la coordenada la cual la unidad de transporte ha de desplazarse.

- **transport_flexmansys_emergency_stop_node_TUN:** Este nodo permite la parada segura y de emergencia de todo el sistema. Lo hace suscribiéndose de manera periódica al tópico "/flexmansys/coordinate/TUN", de manera que, si se recibe una "E", el nodo

mate y detenga la ejecución de todos los nodos implicados en la navegación, sin importar el estado en el que se encuentre el AGV.

- **transport_flexmansys_node_TUN_state:** Su función es la de informar el estado en el que se encuentra la unidad autónoma de transporte. Lo hará publicando en el tópico "/flexmansys/state/TUN" de manera periódica cada dos segundos.

6.4.2.1.2 Paquete turtlebot_navigation

Se trata del paquete relativo al ROS Navigation Stack. Se encarga de la ejecución de la navegación autónoma de las unidades de transporte, localización de la unidad en el mapa y la planificación de trayectorias. También integra las funcionalidades necesarias para llevar a cabo el mapeado del entorno. Cuenta con tres principales nodos:

- **move_base:** Se trata del nodo central del turtlebot_navigation, siendo su función la de enviar los comandos de velocidad correspondientes a la unidad de transporte para que ésta cumpla con la trayectoria fijada. En esencia, se trata de un paquete ROS, que integra en su interior los nodos global_planner, global_costmap, recovery_behaviors, local_planner y local_costmap; tal y como se ha expuesto anteriormente en el apartado de ROS Navigation Stack.
- **amcl:** Es el nodo responsable de la ejecución del AMCL, o algoritmo de Monte Carlo el cual permite localizar a la unidad de transporte en el mapa empleado para la navegación. En pocas palabras, situará al AGV en el mapa según su posición relativa con el entorno.
- **map_server:** Su función es la de proveer del mapa de navegación.

6.4.2.1.3 Paquete turtlebot_bringup

En este paquete se recogen todos los nodos que gestionan las funcionalidades mínimas de la unidad de transporte. Es decir, se trata del paquete que gestiona el sistema de navegación al más bajo nivel, proporcionando los drivers necesarios para que el AGV sea capaz de desplazarse, proporcionar una odometría y comunicarse dentro de un entorno ROS. De todos ellos, destacan:

- **mobile_base**: Nodo gestor de todos los elementos que conforman la base del robot, así como la odometría, el sensor IMU, los parachoques o los botones.
- **cmd_vel_mux**: Es el nodo responsable de los comandos de velocidad introducidos a la unidad de transporte.
- **robot_state_publisher**: Este nodo es el encargado de publicar todas las transformadas del robot, actualizándose de manera periódica haya desplazamiento o no de los sistemas de coordenadas que conforman todos los elementos del robot.

6.4.2.1.4 Paquete rplidar_ros

Se trata del paquete que gestiona la comunicación e interpretación de datos del sensor LiDAR equipado en la unidad de transporte. El único nodo destacable, es:

- **rplidarNode**: Su funcionalidad se resume en definir la configuración del LiDAR empleado, así como la configuración y comunicación del sensor empleado.

6.4.2.2 Entorno de Trabajo rosjava_ws

En lo referente a rosjava_ws, se trata del entorno de trabajo que integra todas las funcionalidades necesarias para realizar la comunicación entre los agentes y los nodos ROS. En resumen, se trata del entorno que actúa como Gateway entre ambas plataformas: la de JADE, desarrollada en Java y en la que residen los agentes; y la de ROS, donde se encuentran todos los nodos de la capa funcional de la arquitectura. Trabaja en la capa de gestión y únicamente está formado por dos paquetes: fms_transp, donde residen todas las clases java y nodos ROS del entorno; y custom_messages, cuya única labor es la de proporcionar los tipos de mensajes customizados que se emplean dentro de catkin_ws.

6.4.2.2.1 Paquete fms_transp

En este paquete se recogen todos los nodos y clases java que se emplearán para la comunicación entre los agentes y los nodos ROS del entorno catkin_ws. En esencia, se trata de un entorno formado en su totalidad por clases java, en el cual no hay nodos ROS per se. Sin embargo, al tratarse de un paquete el cual está desarrollado mediante ROSJava, existirán clases java que cuenten con la capacidad de lanzar nodos ROS e integrarse en la plataforma de JADE. A su vez, las clases java residen dentro del proyecto denominado turtlebot2, donde las clases más significativas se resumen en:

- **ACLGWAgentROS**: Se trata de la clase que integra las funcionalidades de ROS necesarias para lanzar un nodo. Dicho nodo, del mismo nombre que la clase, es el encargado de comunicarse con el entorno catkin_ws a través de los dos tópicos ya mencionados anteriormente: /flexmansys/coordinate/TUN y /flexmansys/state/TUN. A su vez, cuenta con la capacidad de integrarse dentro de la plataforma JADE, configurando la comunicación con los agentes y el nombre que tomará dentro de la plataforma. Su comportamiento y funcionalidades de agente se definen dentro de la clase GWagentROS, a la cual instancia.
- **GWagentROS**: En esta clase java, se define el comportamiento que el agente GWAgent va a tomar en lo relativo a sus actividades dentro de la plataforma MAS. Se trata de una especie de apéndice que proporciona al nodo ROS la capacidad de mandar y recibir mensajes del resto de agentes que integrarán la plataforma.
- **StructCommand**: Se trata de la clase dónde se recogen los campos necesarios para la comunicación entre los agentes de transporte y el agente GWAgent.

Cabe destacar, que tal y como se ha mencionado anteriormente, la clase **ACLGWAgentROS** lanza un nodo ROS que le permite integrarse dentro del entorno ROS como si fuese un nodo ROS más. Dicho nodo, es capaz de suscribirse y publicar en tópicos de la misma manera la cual el resto de los nodos del sistema pueden hacerlo, con la excepción de que, en vez de estar desarrollado en Python o C++, está desarrollado en Java. Es por ello, que cuenta con la posibilidad de emplear las librerías, métodos y recursos necesarios para integrarse dentro de la plataforma JADE. El comportamiento de este nodo ROS dentro de la plataforma de agentes, se lleva define en la clase GWagentROS. Se aporta la *Figura 6.9* para ilustrar la distribución de estos elementos dentro del sistema.

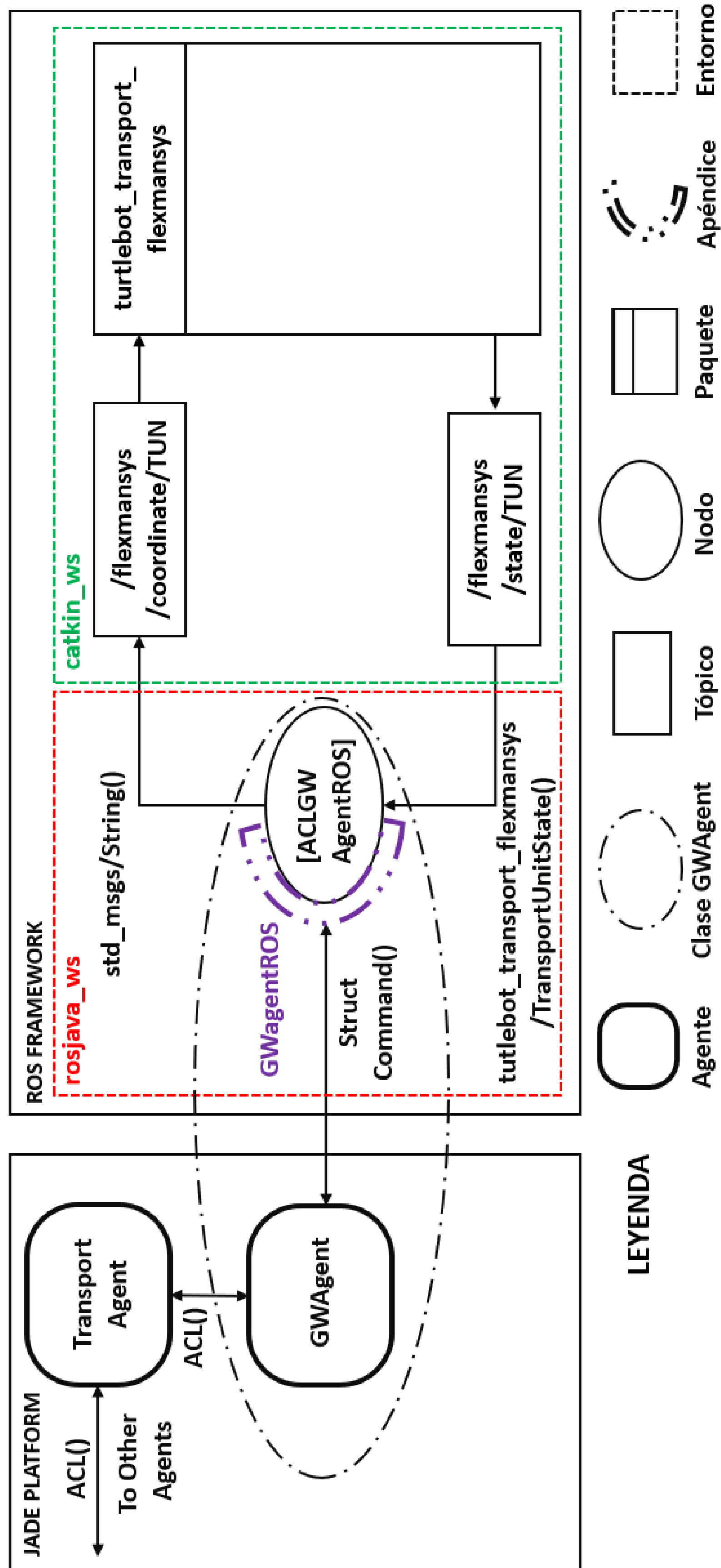


Figura 6.9 Estructura de la Comunicación ROS-JADE

6.4.3 Scripts del Sistema de Navegación

Antes de dar paso al desarrollo del funcionamiento del Sistema de Navegación, se darán unas breves indicaciones de todos los scripts de Python que contiene el sistema y la función que realiza cada uno de ellos. Cabe destacar, que cado uno de estos scripts tiene una clase asociada, por lo que la codificación del Sistema de Navegación está orientada a objetos.

- **transport_flexmansys_battery.py**: Script de lectura de la batería del AGV.
- **transport_flexmansys_bumper.py**: Script de lectura del parachoques del AGV.
- **transport_flexmansys_buttons.py**: Script de lectura del estado de los botones del AGV.
- **transport_flexmansys_coordinates_bagfile.py**: Script que contiene el conjunto de coordenadas recogidas y sus coordenadas tanto cartesianas como cuaterniones asociados.
- **transport_flexmansys_distance_calculator.py**: Script que calcula la distancia desde un punto inicial a otro punto final.
- **transport_flexmansys_emergency_stop_node.py**: Script en el que reside el nodo de emergencia.
- **transport_flexmansys_imu.py**: Script de lectura del giroscopio del AGV.
- **transport_flexmansys_kinect.py**: Script de lectura y gestión de la cámara Kinect.
- **transport_flexmansys_led_manager.py**: Script de gestion de los LED del AGV.
- **transport_flexmansys_localization.py**: Script que gestiona la calibración del AGV.
- **transport_flexmansys_main.py**: Script que contiene el nodo main del sistema.
- **transport_flexmansys_manipulation_activity.py**: Script que gestiona la manipulación.
- **transport_flexmansys_movement_manager.py**: Script que gestiona el movimiento del AGV y recoge todos los movimientos no asociados al ROS Navigation Stack.
- **transport_flexmansys_odom.py**: Script de lectura de la odometría.
- **transport_flexmansys_send_goal.py**: Script que envía a move_base la coordenada a la que debe de desplazarse la unidad de transporte.
- **transport_flexmansys_sound_manager.py**: Script que gestiona el buzzer del AGV.
- **transport_flexmansys_state_active.py**: Script que gestiona el estado Active.
- **transport_flexmansys_state_idle.py**: Script que gestiona el estado Idle.
- **transport_flexmansys_state_localization.py**: Script que gestiona el estado Localization
- **transport_flexmansys_operative.py**: Script que gestiona el estado Operative.
- **transport_flexmansys_state_publisher.py**: Script que gestiona el funcionamiento del nodo encargado de publicar el estado del AGV en el /flexmansys/state/TUN.

6.4.4 *Máquina de Estados del Sistema de Navegación*

El funcionamiento y comportamiento de cada una de las unidades de transporte integradas en el sistema de navegación, se basa en una máquina de estados. Dicha máquina se representa en el diagrama de estados de la *Figura 6.10*.

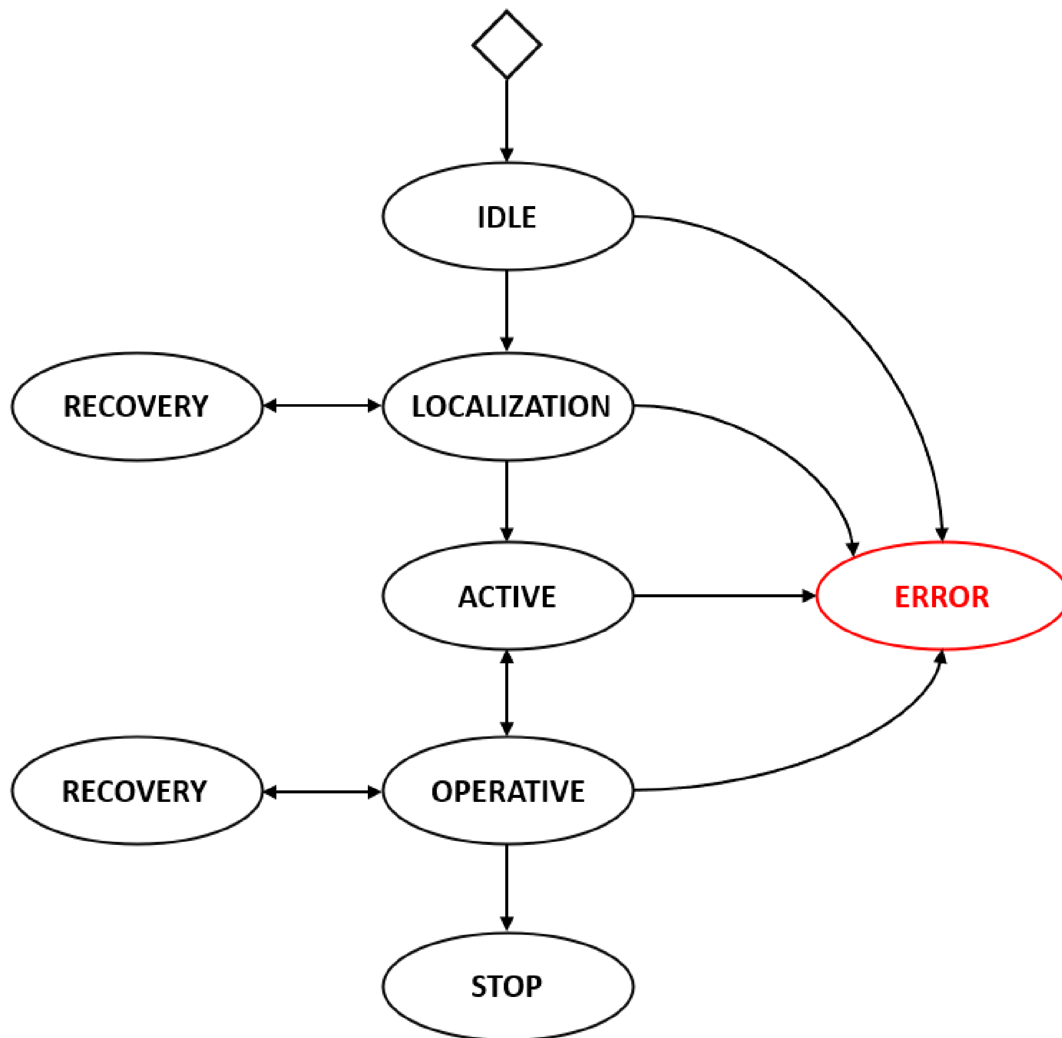


Figura 6.10 Diagrama de la Máquina de Estados del Sistema de Navegación

La máquina de estados cuenta con un total de siete posibles estados, siendo "Idle" el estado inicial al cual todas las unidades de transporte entrarán nada más lanzarse los nodos ROS del paquete "/turtlebot_transport_flexmansys". Dentro de dicha máquina, únicamente se dará un comportamiento cíclico entre los estados "Active" y "Operative", el equivalente al funcionamiento normal de cada robot de transporte. En caso de que cualquier AGV entre en los estados de "Stop" o "Error", el sistema deberá de volver a ejecutarse desde cero si se quiere volver a poner a la unidad de transporte en marcha.

6.4.4.1 Estado Idle

Tal y como se ha comentado anteriormente, el estado "Idle" se trata del estado inicial al que toda unidad de transporte entrará nada más inicializarse el sistema de navegación. Este estado, tiene como objetivo comprobar que las funcionalidades básicas del AGV en cuestión responden correctamente, además de verificar que se establece la comunicación con las capas superiores de la arquitectura, es decir, con los agentes. Si la unidad de transporte en la que se lance en sistema de navegación entra en el estado Idle, se considerará a la unidad como apta para formar parte del conjunto de AGVs que forman la flota de transporte. Sin embargo, el sistema previamente deberá pasar por los estados "Localization" y "Active" para poder llevar a cabo tareas que impliquen navegación autónoma.

El funcionamiento del estado Idle se muestra en el diagrama de flujo de la *Figura 6.11*. En esencia, una vez en Idle, la unidad de transporte estará continuamente suscribiéndose al tópico en el que recibe las peticiones desde los agentes. En caso de que reciba una "X", la unidad de transporte avanzará al estado "Localización", mientras que si recibe un "STOP" avanzará al estado "Stop". En caso de recibir cualquier otra coordenada o petición que no sean las dos anteriores, el sistema automáticamente ignorará el mensaje recibido.

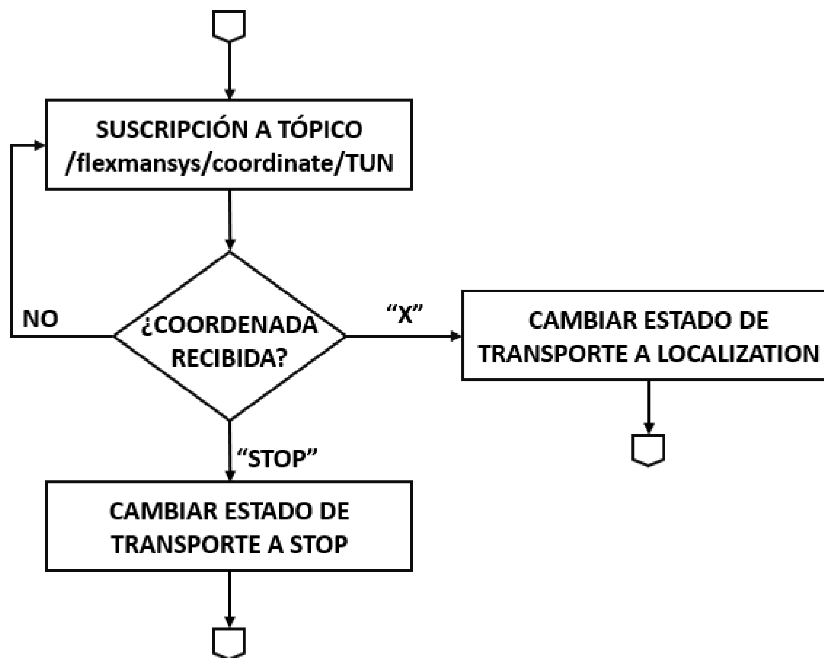


Figura 6.11 Diagrama de Flujo del Estado Idle

6.4.4.2 Estado Localization

Para que un AGV pueda ser considerado disponible para asignarle tareas que requieran de navegación autónoma, previamente, ha de calibrarse. Para ello, existe el estado "Localization", cuyo objetivo es el de situar a la unidad de transporte en cuestión dentro del mapa empleado para la navegación. Lo hará a través del método AMCL, algoritmo el cual se integra dentro de los nodos cargados por el paquete move_base del ROS Navigation Stack. En la *Figura 6.12* se muestran las pautas que siguen los transportes en el estado Localization.

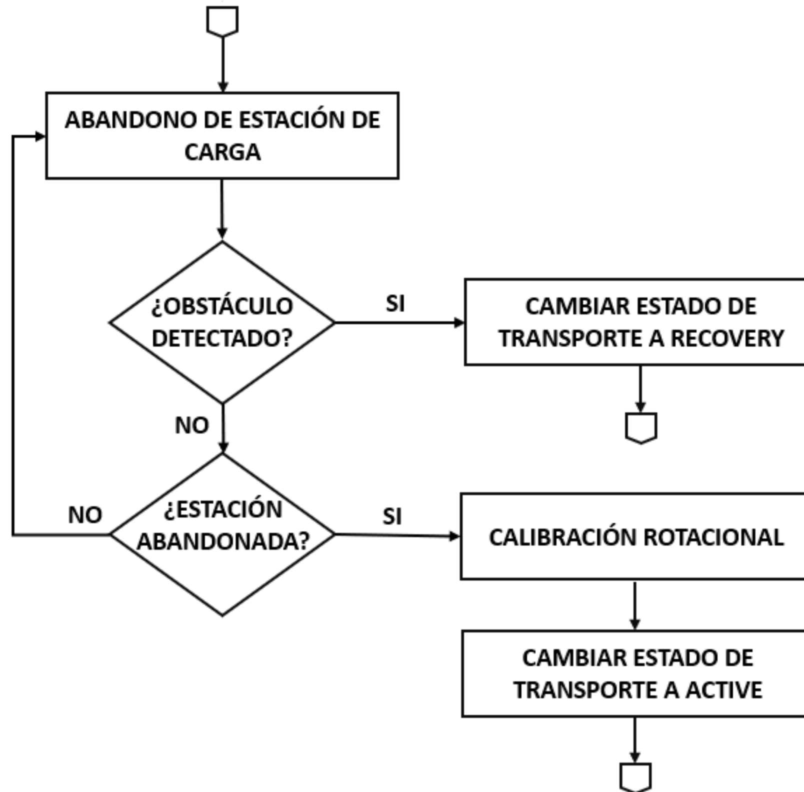


Figura 6.12 Diagrama de Flujo del Estado Localization

Previamente, el transporte abandonará su estación de carga o posición inicial, desplazándose 0.3 metros en línea recta. Es necesario que todos los transportes se trasladen dicha distancia, por un lado, porque la estación de trabajo de algunos transportes se encuentra encajonada entre varios elementos y sería imposible llevar a cabo una correcta calibración en el mapa, y por el otro, para que el nodo ROS encargado de ejecutar el algoritmo de AMCL detecte movimiento y se active antes de llevar a cabo la calibración de la unidad de transporte. Una vez el AGV abandona la estación, este realizará dos giros de 360 grados sobre el eje vertical z, es decir, que realizará un giro completo sobre sí mismo tanto en sentido horario como en antihorario. Posteriormente, se considerará a la unidad correctamente calibrada y se dará paso al estado "Active" de la máquina de estados

Cabe destacar, que el sistema de navegación puede entrar al estado "Recovery" desde el estado de Localization. A este estado, el cual se detallará más adelante su funcionamiento completo, únicamente entrará el AGV si se encuentra un obstáculo que le impida abandonar su estación de trabajo. Pese a ser una situación poco probable, en principio, puede darse el caso de que el posicionamiento inicial del robot no sea la adecuada o bien sea necesario despejar algún objeto que se encuentre en la trayectoria que vaya a realizar la unidad de transporte.

6.4.4.3 Estado Active

El estado "Active", indica que la unidad de transporte correspondiente está lista para recibir encomiendas y desplazarse hasta la coordenada que su respectivo agente de transporte indique. La principal diferencia entre este "Idle" y "Active", es que el AGV ya se encuentra situado y calibrado dentro del mapa de navegación, por lo que se considera óptimo para la navegación dentro de este. Se detalla su funcionamiento en la *Figura 6.13*.

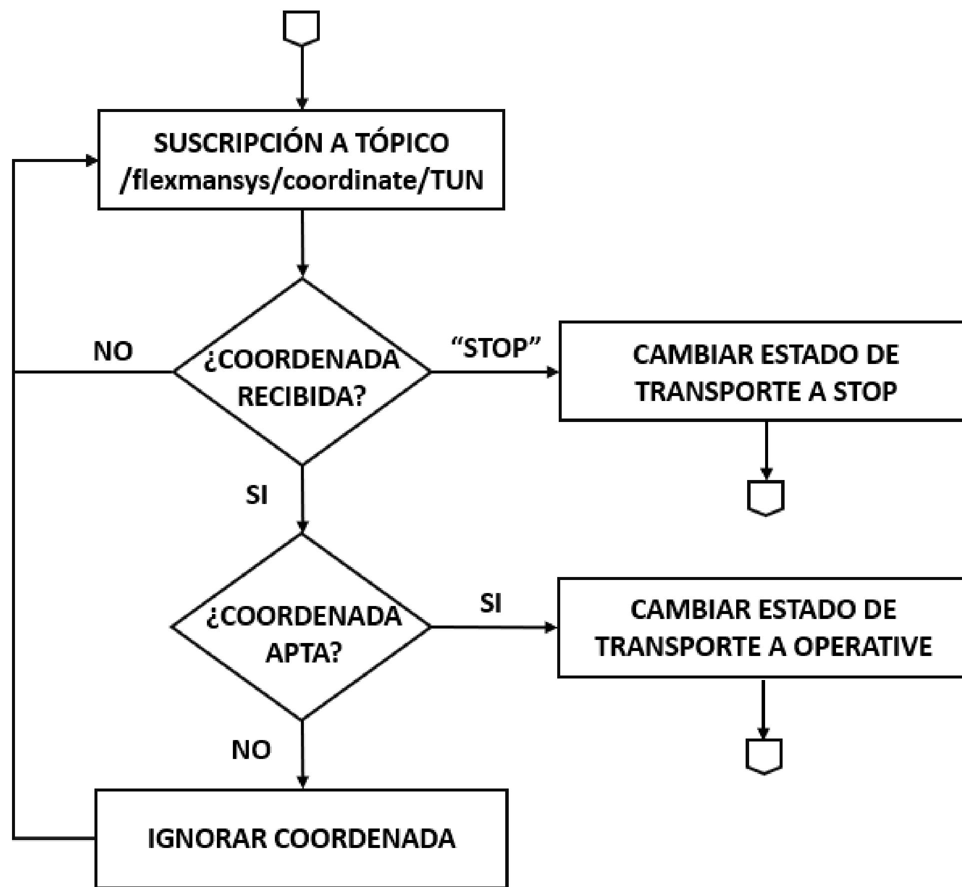


Figura 6.13 Diagrama de Flujo del Estado Active

El estado Active cuenta con un funcionamiento muy similar al estado "Idle", puesto que, en esencia, lo único que les diferencia es su capacidad de navegar de manera autónoma en torno a un mapa. En resumen, en el estado active la unidad estará cíclicamente suscribiéndose al tópico "/flexmansys/coordinate/TUN", de manera que pueda recibir las encomiendas desde el agente de transporte asignado. En caso de recibirse una coordenada apta, es decir, una coordenada recogida dentro del "bagfile" o librería de coordenadas de la unidad de transporte, se dará paso al estado "Operative". Evidentemente, si la coordenada no es válida, simplemente se ignorará el comando recibido. En caso de leerse un "STOP" desde el tópico, el AGV entrará en el estado de Stop.

6.4.4.4 Estado Operative

El estado "Operative" indica que la unidad de transporte se encuentra realizando una tarea u operación asignada. Dichas tareas, no están únicamente relacionadas con la navegación o movimiento de la base del AGV, puesto que también se considera como operación a toda tarea que implique traslado, recogida o manipulación de objetos. Se trata del estado más complejo de todos y exigente de todos, puesto que implica la cooperación directa de todos los nodos y paquetes implicados en el sistema de navegación dentro de un marco temporal y de sincronización considerablemente exigente. De manera resumida y sin entrar en detalles, las pautas que sigue el estado Operative se muestran en la *Figura 6.14.*

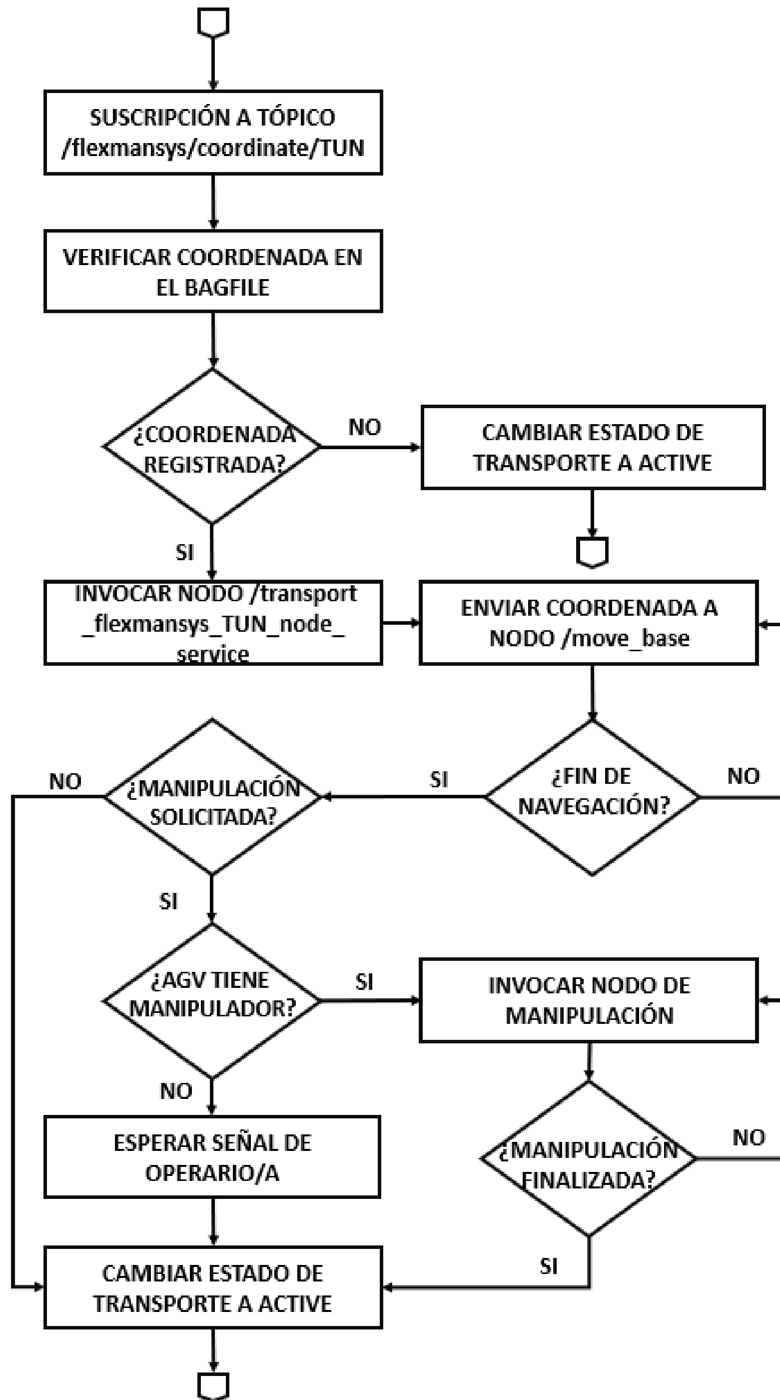


Figura 6.14 Diagrama de Flujo del Estado Operative

Nada más entrar al estado Operative, la unidad de transporte volverá a comprobar que la coordenada o petición recibida desde la plataforma de agentes es válida. En caso de ser válida, el nodo main recogerá las coordenadas de la coordenada requerida del "bagfile" y se las enviará al nodo "/move_base", mientras que, si no es válida, volverá al estado Active. La comunicación entre el nodo principal de navegación y el nodo "/move_base", se hace a través del nodo servidor "transport_flexmansys_node_server_TUN", el cual crea una jerarquía de cliente-servidor con "/move_base", siendo este último el servidor de la comunicación. El nodo "/move_base", será el coordinador de la navegación del AGV desde el punto de partida en el que se encontraba la unidad

de transporte cuando recibe la petición de traslado, hasta alcanzar la posición de la coordenada requerida. Una vez "/move_base" considere que el AGV se encuentra dentro de los márgenes aceptables configurados tanto en el planificador global como en el local, indicará que la navegación ha terminado a su cliente, el nodo "transport_flexmansys_node_server_TUN", que a su vez devolverá el "control" de la unidad de transporte a su cliente, el nodo de navegación principal.

Tras completar la navegación de manera correcta, se comprobará si se ha solicitado la manipulación de material, operación la cual únicamente dos transportes pueden completar al estar equipados con un brazo robótico. En caso de contar con un brazo, se invocarán las funcionalidades del nodo de manipulación, mientras que, en caso de no contar con uno, simplemente se esperará a que la persona operaria correspondiente manipule el material y esta le indique que ha terminado de manipular su material. Para ello, se habilita en todos los AGV un pulsador, el cual se ha modelado en los Turtlebot2 como el botón B0. Finalmente, cambiará al estado Active de nuevo para recibir una nueva encomienda.

6.4.4.5 Estado Stop

El estado "Stop" se trata del estado al cual toda unidad de transporte debe de entrar antes de ser apagada. Se trata del equivalente a la parada o finalización del funcionamiento normal del AGV. En esencia, este estado únicamente se encarga de detener a la unidad de transporte y de terminar la ejecución de todos los nodos implicados en el sistema de navegación, salvo los relacionados con el paquete "turtlebot_bringup". Es por ello, que, pese a que el AGV es capaz de entrar al estado Stop en cualquier punto del mapa, se recomienda que previamente el robot se encuentre en su estación de carga correctamente posicionado. Su funcionamiento es simple y no requiere de mayor explicación a la mostrada en la *Figura 6.15*

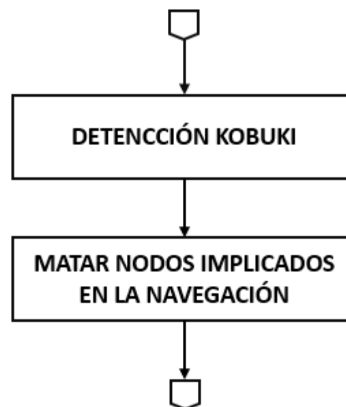


Figura 6.15 Diagrama de Flujo del Estado Stop

6.4.4.6 Estado Error

El estado "Error", se trata del equivalente de la parada de emergencia de la unidad de transporte. A diferencia del estado de "Stop" o parada normal, el AGV en cuestión puede entrar a "Error" sin importar lo que esté haciendo en el momento el cual ha sido invocado el estado de "Error" en el sistema.

Esto, se debe a la existencia del nodo "transport_flexmansys_emergency_stop_node", totalmente independiente al nodo main del sistema de navegación. Su labor, es la de suscribirse cíclicamente al tópico dónde se publican las coordenadas de transporte, es decir, "/flexmansys/coordenada/TUN", y detener la ejecución de todos los nodos implicados en el sistema de navegación de la unidad de transporte en caso de que se invoque de manera forzada la parada de la unidad de transporte. De este modo, el transporte se detiene en el punto exacto en el cual se ha invocado la parada de emergencia, notificando tanto a través del nodo de estado del sistema de navegación como a través del buzzer y LED incorporados en el AGV que se encuentra en el estado de emergencia. Será necesario la acción de una persona operaria para hacer que el AGV salga de dicho estado. Las etapas que sigue el sistema de navegación en este estado son muy similares al estado Stop, tal y como se observa en la *Figura 6.16*.

Figura 6.16 Diagrama de Flujo del Estado Error

6.4.4.7 Estado Recovery

El estado "Recovery" se trata de un estado especial el cual no forma parte de la secuencia principal de funcionamiento de la unidad de transporte. El AGV en cuestión, únicamente podrá entrar a "Recovery" si este mientras se desplaza encuentra un obstáculo el cual interfiere con su trayectoria y el cual le es imposible esquivar. Es por ello, que únicamente es posible entrar al estado "Recovery" desde los estados "Localization" y "Operative".

La existencia de este estado y la posibilidad de toparse con un obstáculo el cual el sistema de navegación no haya podido detectar, se debe a las propias limitaciones hardware de las

unidades de transporte empeladas. El sistema de navegación hace uso del LiDAR planar RPLiDAR A2 para percibir su entorno, lo cual le permite trazar trayectorias dinámicas que le permitan esquivar obstáculos no incluidos en el mapa de navegación. Sin embargo, al tratarse de un LiDAR planar, es decir, de dos dimensiones, este únicamente es capaz de detectar obstáculos a la misma altura del haz de luz láser que emite. Es por ello, que obstáculos situados por debajo de dicho LiDAR, son imperceptibles para el nodo encargado del trazado de trayectorias.

Para contrarrestar dicha limitación, los AGV harán uso de su parachoques delantero para detectar dichos obstáculos. En caso de que el parachoques se presione, es decir, que se ha encontrado con un obstáculo el cual no ha podido detectar o bien no puede desplazar de su trayectoria al ser demasiado pesado, la unidad de transporte entrará en el estado “Recovery”. En esencia, seguirá un comportamiento muy sencillo, mostrado en la *Figura 6.17*

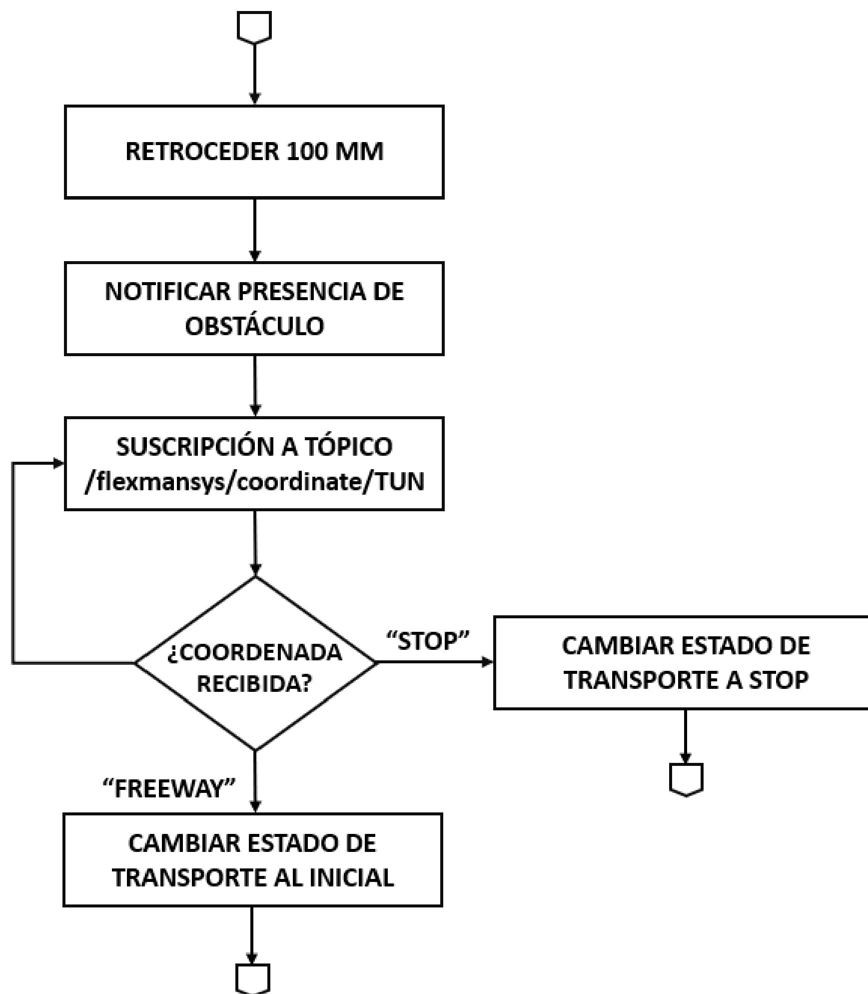


Figura 6.17 Diagrama de Flujo del Estado Recovery

En caso de detectar un obstáculo mientras se encuentra en movimiento, la unidad de transporte retrocederá 0.1 metros. Posteriormente, notificará la presencia de un obstáculo tanto a través del nodo indicador de estado, mediante las dos variables asignadas para ello “detected_obstacle_bumper” y/o “detected_obstacle_camera”. Tras ello, entrará en un bloqueo de los motores, impidiendo cualquier consigna de movimiento exterior. Posteriormente, se

suscribirá de manera cíclica al tópico "/flexmansys/coordinate/TUN" hasta recibir una de las dos coordenadas o comandos que admite: STOP, para indicarle que cese su funcionamiento bien porque el obstáculo detectado no se puede despejar o por mayores inconvenientes; o FREEWAY, mediante el cual se hará saber a la unidad de transporte que puede seguir con su trayectoria previa al haber sido despejado el obstáculo que había detectado,

6.4.4.8 Indicadores de Estado

Los AGVs integrados dentro del sistema de navegación cuentan con una serie de indicadores tanto hardware como software para saber en todo momento el estado en el que se encuentran.

En lo relativo al hardware, el sistema cuenta con dos indicadores ya integrados dentro del hardware de los Turtlebot2: Tres indicadores LED, situados en la parte posterior de la base del Turtlebot2; y un buzzer. Las personas operarias en planta que, o bien no tengan ninguna herramienta capaz de conectarse a la red ROS, o bien no tengan los conocimientos necesarios para interpretar la información del sistema de navegación, podrán conocer el estado en el que se encuentra cada AGV únicamente comprobando la iluminación de los LED posteriores. Estos diodos, se iluminarán según la lógica de la *Figura 6.18*

IDLE LOCALIZATION ACTIVE OPERATIVE STOP ERROR RECOVERY
LED 0
LED 1
LED 2

Figura 6.18 Indicadores de Estado LED

Por su parte, el buzzer únicamente emitirá sonido en tres ocasiones: En el estado Idle, dónde emitirá un sonido durante 1 segundo e indicando que la unidad ya está inicializada; en el estado Recovery, emitiendo un sonido distinto a Idle pero durante 1 segundo igualmente; y en el estado Error, dónde emitirá sonido durante 5 segundos de manera repetida hasta llegar a 10 zumbidos consecutivos.

En lo relativo al software, el sistema cuenta con dos interfaces: Los propios mensajes de la consola, los cuales se publican desde el nodo main a la capa de información de ROS y pueden ser visualizados desde la terminal del propio equipo dónde se esté ejecutando el sistema; y un nodo ROS íntegramente dedicado a la lectura de los parámetros principales de la unidad de transporte. Dicho nodo, se trata del nodo el "flexmansys_transport_node_TUN_state", cuya única labor es la de publicar dichos parámetros en el tópico "flexmansys/state/TUN". Los campos que se publican en dicho tópico, así como sus campos, se desglosan en la *Tabla 6.4*

Tipo Datos Tópico	Campo 1	Campo 2	Tipo Dato
turtlebot_transport_flexmansys /TransportUnitState	kobuki_general	transport_unit_name	String
		transport_unit_state	String
		battery	Float32
	kobuki_obstacle	detected_obstacle_bumper	Bool
		detected_obstacle_camera	Bool
		transport_in_dock	Bool
	kobuki_position	recovery_point	String
		odom_x	Float64
		odom_y	Float64
		rotation	Float64
	odroid_date	year	TimeDate
		month	TimeDate
		day	TimeDate
		hour	TimeDate
		minute	TimeDate
		seconds	TimeDate

Tabla 6.4Campos de turtlebot_transport_flexmansys/TransportUnitState

El tipo de dato TransportUnitState se trata de una clase no definida dentro del entorno ROS, es decir, una variable customizada. A su vez, dicho tipo de datos emplea como campos otros cuatro tipos de datos no definidos dentro del entorno ROS: kobuki_general, kobuki_obstacle, kobuki_position y odroid_date. Se muestra en la *Tabla 6.5* el contenido de cada uno de los campos del tipo de dato TransportUnitState.

Variable	Contenido
transport_unit_name	Nombre o TUN de la unidad de transporte asociada.
transport_unit_state	Estado que se encuentra la máquina de estados.
battery	Nivel de batería del AGV expresado en VDC.
detected_obstacle_bumper	Flag que indica si el parachoques del kobuki se ha presionado.
detected_obstacle_camera	Flag que indica si la cámara equipada ha detectado un obstáculo.
transport_in_dock	Indica si el AGV ha verificado que se encuentra en su estación de carga.
recovery_point	Variable que indica si el AGV ocupa uno de los puntos de recovery.
odom_x	Devuelve la posición x de la odometría del AGV, expresado en metros.
odom_y	Devuelve la posición y de la odometría del AGV, expresado en metros.
rotation	Valor de rotación en la que encuentra el AGV, expresado en grados.
year	Año del reloj interno de la odroid.
month	Mes del reloj interno de la odroid.
day	Día del reloj interno de la odroid.
hour	Hora del reloj interno de la odroid.
minute	Minuto del reloj interno de la odroid.
seconds	Segundo del reloj interno de la odroid.

Tabla 6.5 Contenido de las Variables de TransportUnitState

6.4.5 Sistema de Detección de Obstáculos

En un sistema dinámico donde múltiples unidades de transporte operan y conviven con el personal de planta, es necesario implementar sistemas que permitan la detección de obstáculos u objetos que puedan influir en la trayectoria del AGV. Tal es así, que cada una de las cuatro unidades de transporte cuentan con dos elementos que les permiten detectar obstáculos: El sensor LiDAR A2 y el parachoques que incorporan en la parte frontal de la base robótica Turtlebot2.

Por un lado, el tópico que publica de manera periódica el sensor LiDAR es monitoreado constantemente por el nodo del ROS Navigation Stack encargado de calcular el plan local, es decir, la trayectoria que debe de seguir en el instante siguiente la unidad de transporte para poder seguir el plan global. En caso de detectarse una serie de puntos que no concuerden con lo recogido en el mapa estático, es decir, que lo recogido en el mapa dinámico y en el estático difiera, indicará que existe la presencia de un objeto nuevo que no existía durante el mapeado del mapa estático. Inmediatamente después, el planificador local trazará un nuevo plan que permitirá esquivar dicho objeto, pese a que tenga que desviarse del plan global inicial que se basa en el mapa estático.

Por otro lado, se cuenta con el parachoques frontal, el cual se activa en caso de colisionarse con un objeto cualquiera. Ahora bien, existe un principal inconveniente, y es que ambos sistemas de percepción, tal y como puede verse en la *Figura 6.19*, sirven para detectar obstáculos a la misma altura del sensor LiDAR de 2D, de manera que objetos por debajo de dicho rango de detección no podrán ser detectados hasta que el transporte colisione frontalmente con el objeto en cuestión. Es por ello, que este sistema de detección únicamente funcionaría en un entorno ideal en el que los obstáculos existentes fuesen de la misma altura del LiDAR.

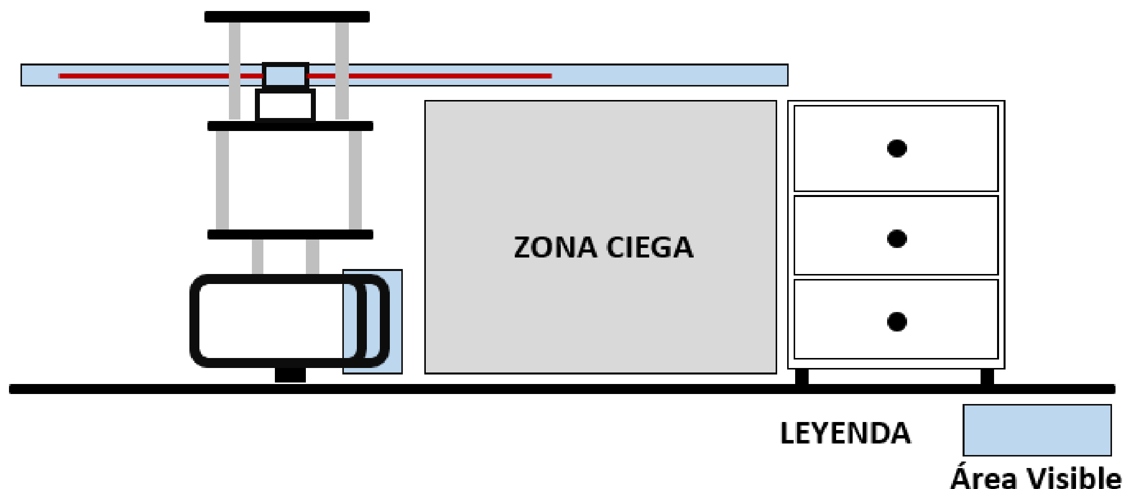


Figura 6.19 Áreas de Detección de Obstáculos

6.4.6 Sistema de Percepción de Imagen

En lo relativo al Sistema de Percepción de Imagen, se trata de una funcionalidad añadida del Sistema de Navegación. Como se ha mencionado en el anterior apartado, la detección de obstáculos puede suponer un problema para según qué elementos se interpongan en la trayectoria del transporte. Es por ello, que era necesario el implantar un sistema adicional al parachoques del Turtlebot2 y al sensor LiDAR A2. El objetivo era claro, dar un elemento que permitiese detectar obstáculos en la zona ciega de la *Figura 6.19*.

Es por ello, que se planteó la posibilidad de llevar a cabo un sistema de percepción de obstáculos mediante visión artificial y mediante una cámara Microsoft Kinect XBOX 360. Se optó por esta alternativa por dos razones: Principalmente, porque se trataba de una cámara muy barata con relación a sus homólogas y muy empleada dentro de la comunidad de ROS; y, por otro lado, puesto que el suelo en el que iba a operar el transporte era completamente monocolor, lo que facilitaría enormemente la labor del procesamiento de imagen para detectar obstáculos. Sin embargo, dado que únicamente se contaba con una única cámara para los cuatro transportes y debido a que las condiciones de operación que el cliente requería para la solución del presente trabajo no contemplaban obstáculos, se optó por dejar la percepción de obstáculos mediante visión artificial como un área de trabajo futura.

A pesar de ello, dado que principalmente se iba a operar con el transporte T_01, equipado con una cámara, se optó por llevar a cabo un sistema de percepción que permitiese localizar puntos específicos en el mapa, concretamente, mediante códigos AR. De esta manera, mediante visión artificial, se podría verificar que el transporte se encuentra en el punto especificado por la odometría, como podría ser la estación de carga, o puntos estratégicos en el mapa. Estos códigos, similares a lo QR, son considerablemente más simples, lo que no suponen una excesiva carga computacional para el equipo. Para lanzar dicho sistema, se emplean dos ficheros launch recogidos dentro del paquete turtlebot_transport_flexmansys.

- **turtlebot_transport_flexmansys_freenect.launch**: Se lanza en la Odroid XU4, es el paquete encargado de procesar la imagen de la cámara Kinect y de publicar dichos datos en diversos tópicos [62].
- **turtlebot_transport_flexmansys_ar_trac_alvar_indiv**: Se lanza en el equipo en el cual se encuentre el nodo maestro ROS, paquete encargado de detectar y leer los códigos AR que pueda haber en las imágenes de la cámara [63].

De este modo, no únicamente se da una vía alternativa a futuro para detectar obstáculos por debajo de la línea de visión del sensor LiDAR sin la necesidad de que estos colisionen con el parachoques, como se muestra en la *Figura 6.20* , sino que se da una herramienta más a las unidades de transporte para detectar códigos y etiquetas que se introduzcan en el entorno, de manera que exista redundancia con el sistema de localización sacado a partir de la odometría. Este sistema de percepción es el empleado para modificar el campo transport_in_dock del tipo de mensaje ROS TransportUnitState, verificando que el sistema detecta el código AR correspondiente de su estación a la hora de posicionarse sobre ella.

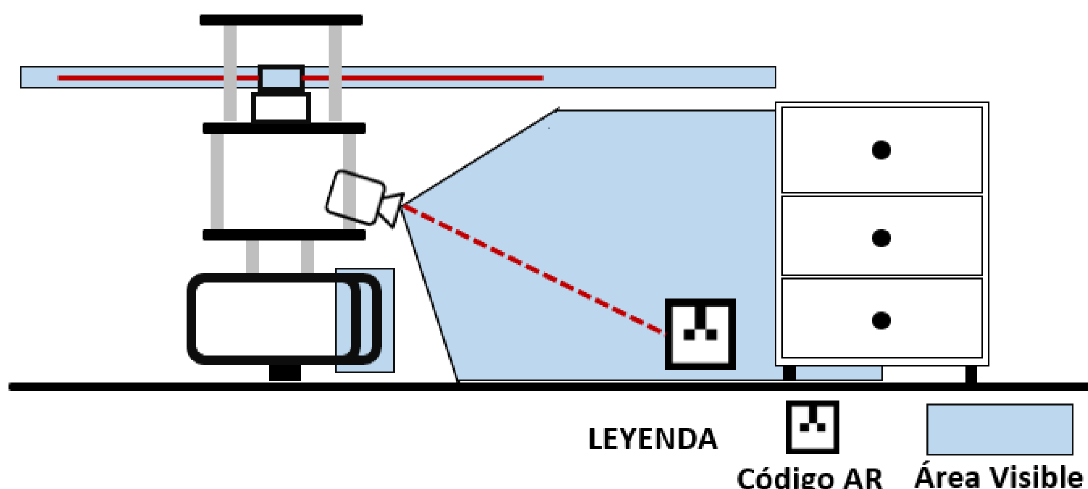


Figura 6.20 Áreas de Detección de Obstáculos con Cámara

6.5 Sistema Gateway

Tal y como se ha comentado en la parte inicial del presente capítulo, el Sistema Gateway es aquel que integra todos los paquetes, librerías y nodos necesarios para llevar a cabo la comunicación entre un sistema desarrollado en un entorno Java, como es el MAS JADE, y otro desarrollado en ROS. En esencia, todos sus elementos se recogen dentro del entorno de trabajo de rosjava_ws, anteriormente desarrollado en el apartado *6.4.2.2*. Este sistema, trabaja tanto en el entorno de ROS como en el MAS, por lo que la estructura general del sistema sería la mostrada en la *Figura 6.21.*

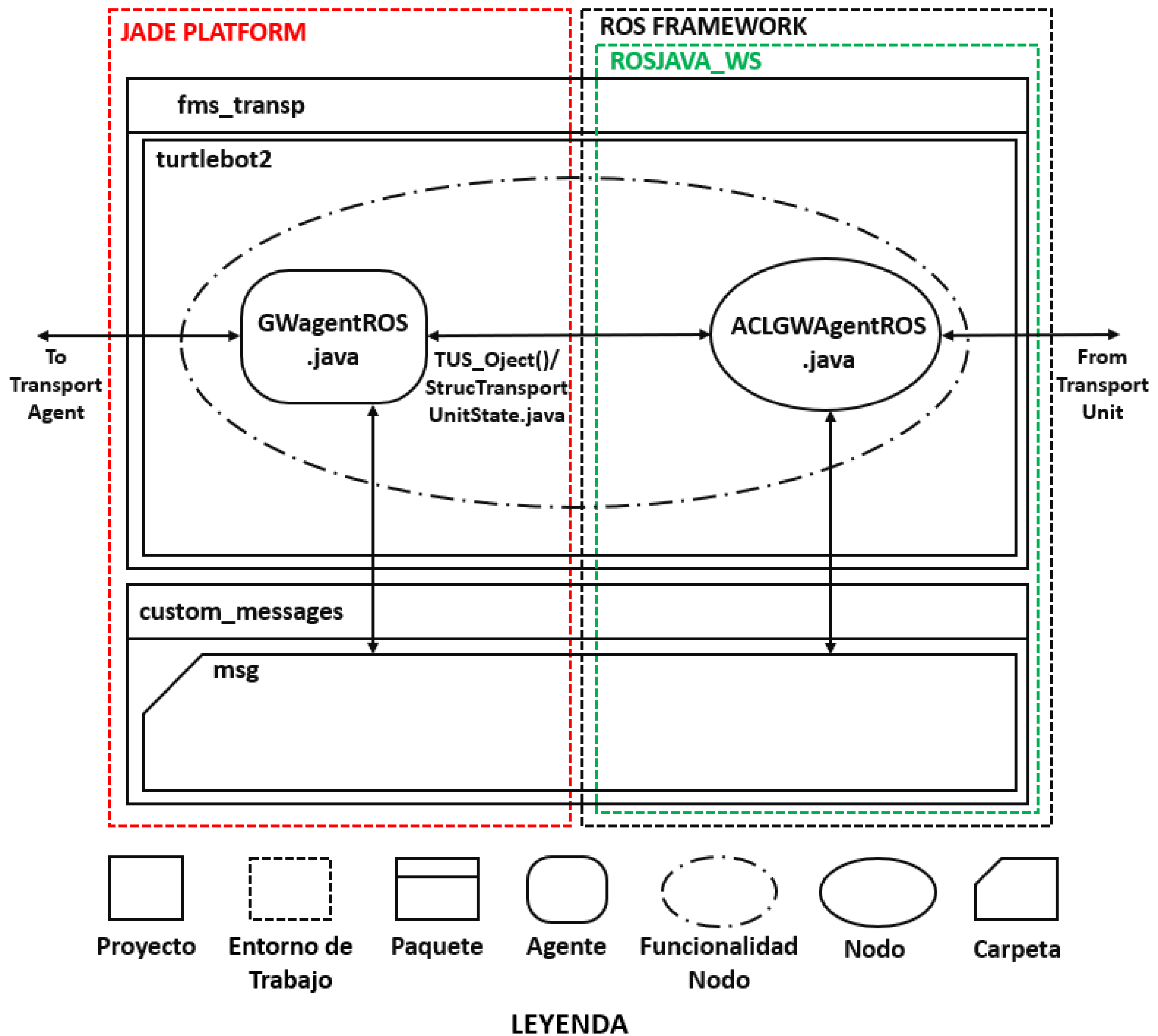


Figura 6.21 Estructura General Software Sistema Gateway

Como puede apreciarse, el Sistema Gateway está formado por dos clases java, las cuales están recogidas dentro de un nodo ROS. De ambas clases, ACLGWAgentROS.java es la clase principal o main, la cual cuenta con la funcionalidad propia de integrarse en el entorno de ROS. Por otro lado, GWagentROS.java es aquella que se integra dentro del MAS, a la cual se instancia la clase principal. La comunicación entre ambas clases se lleva a cabo mediante una variable interna denominada TUS_Object, estructura de datos contenida dentro de una tercera clase java

que replica la estructura de datos contenida en el mensaje ROS TransportUnitState.msg para la comunicación entre el nodo responsable de indicar el estado de la unidad de transporte y los nodos suscritos al tópico donde se publica la información: /flexmansys/state/TUN; del cual la clase java ACLGWAgentROS recoge su información.

Por otro lado, se puede ver como el sistema está formado por dos paquetes: fms_transp, relativo al procesamiento de datos entre ROS y Java; y custom_messages, donde residen los mensajes customizados de ROS empleados en la comunicación con el nivel funcional de la arquitectura y por tanto con las unidades de transporte. Del mismo modo, también se cuenta con el proyecto turtlebot2, que encapsula la funcionalidad necesaria para la comunicación con dicha gama de AGV, y la carpeta msg, en la cual se guardan todos los campos de datos mostrados en la *Tabla 6.4*.

Al igual que con el Sistema de Navegación, el Sistema Gateway cuenta con cierto comportamiento de máquina de estados, la cual irá avanzando o bien cesando su actividad según lo que reciba desde la unidad de transporte y el agente transporte. Es decir, a diferencia del Sistema de Navegación, cuenta con un carácter no tan secuencial ni lineal, sino que más bien distribuido en varias clases ejecutándose de manera simultánea. Este comportamiento, puede apreciarse en el esquema de la *Figura 6.22*, esquema en el cual se muestran todas las acciones de los elementos implicados en el Sistema Gateway. Nótese que la mayor parte de las funciones realizadas por el agente transporte, o “TransportAgent” en el esquema, se abstraen, puesto que será explicado en mayor detalle en el siguiente apartado.

En lo referente al comportamiento del Sistema Gateway, se muestra en la *Figura 6.22* un ciclo de completo, es decir, desde su inicialización, pasando por la etapa de calibración de la unidad de transporte hasta la ejecución de un listado de tareas. Para facilitar el entendimiento, en el esquema se han indicado siete números, los cuales marcan los eventos importantes dados en la ejecución, de manera que:

1. En este primer paso se inicializa la clase java ACLGWAgentROS, la clase principal del sistema. Esta inicialización, requerirá como argumentos de entrada el TUN de la unidad de transporte la cual va a servir como Gateway y el número que tomara el agente en el MAS, como, por ejemplo, T_01. Se llevarán a cabo las tareas de configuración más básicas.

2. Posteriormente, al inicializar la clase principal, esta se instanciará a la clase correspondiente a GWagentROS, aquella que cuenta con toda la funcionalidad de agente para operar en la operar JADE. Si todo se ejecuta de manera correcta, la clase principal seguirá con su ejecución. Durante su ejecución, leerá el tópico correspondiente de estado de su unidad de transporte, /flexmansys/state/TUN, procesará los datos, y se los enviará a la clase GWagentROS de manera periódica desde entonces cada vez que se actualicen los datos en el tópico. Dado que no está el agente transporte inicializado, esta clase no procesará dichos datos hacia arriba.

3. Una vez la clase principal se ejecuta de manera normal, se da paso a la inicialización del agente transporte. Al igual que con la clase ACLGWAgentROS, habrá que indicarle en sus argumentos de entrada el TUN de la unidad de transporte y el número que tomará esta dentro del MAS; además del archivo correspondiente donde se escribirán los planes de transporte, funcionalidad que se explicará en el siguiente apartado. Tras ello, el agente transporte se inicializará en el MAS bajo el homónimo “auxma-local”, una especie de agente temporal, que tras comprobar si el agente Gateway está vivo, cambiará su nombre de auxma-local a transportX, siendo X el número correspondiente al número de agente transporte de la plataforma.

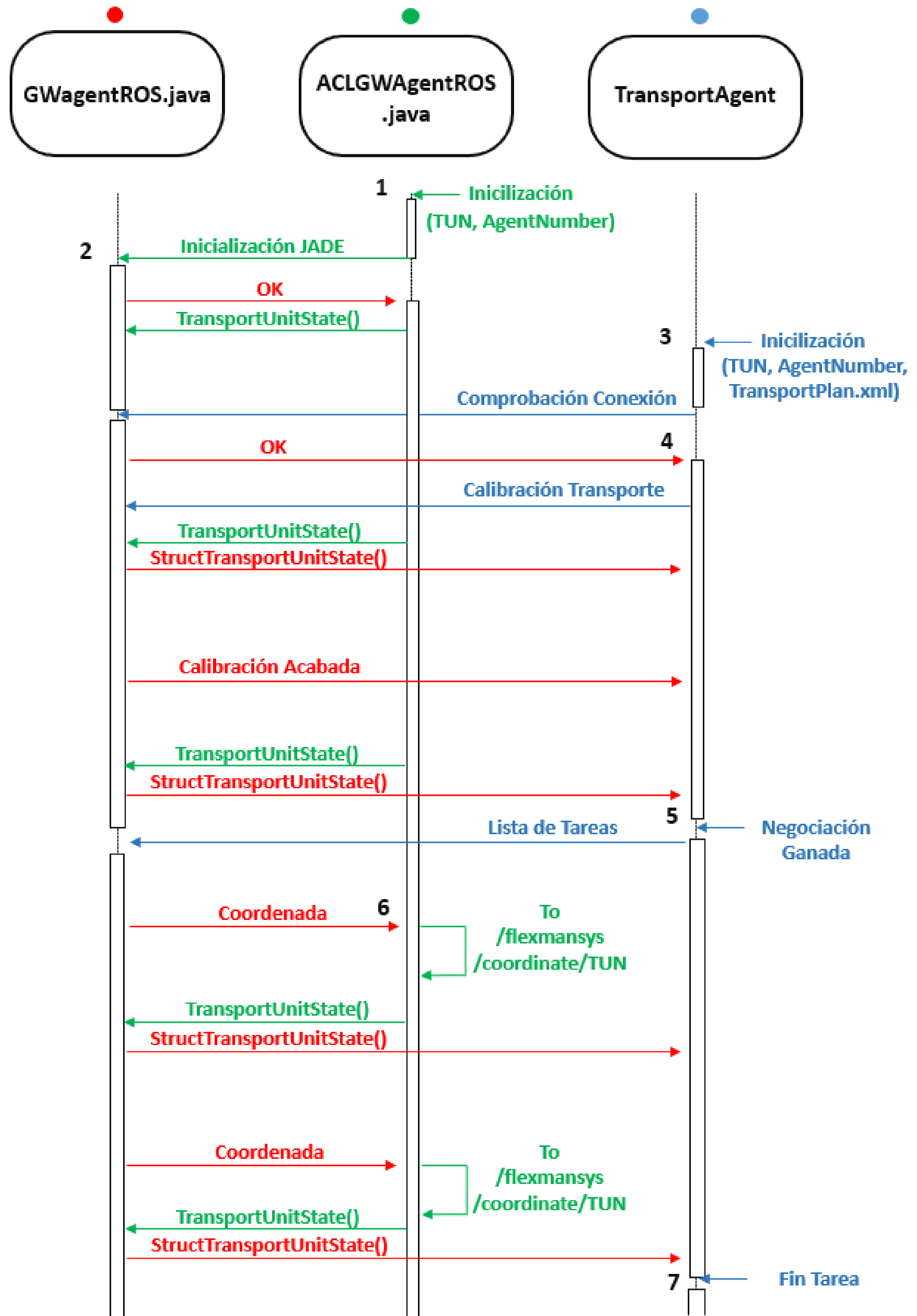


Figura 6.22 Esquema de Funcionamiento en Ejecución del Sistema Gateway

4. Posteriormente, si tras comprobar la conexión con el agente Gateway, o lo que es lo mismo, con el transporte en sí, este se encuentra en el estado de Idle, estado correspondiente a la máquina de estados del Sistema de Navegación, el agente transporte le indicará que debe calibrarse. Dado que el agente transporte ya está inicializado, este irá recibiendo de manera periódica los datos publicados en el tópico de estado del transporte.

5. Tras la calibración, el sistema continuará enviando los datos recibidos en el tópico de estado de manera periódica hacia el MAS, hasta que su agente transporte gane una negociación en función de los datos que ha obtenido por parte del Gateway.

6. Cuando gane la negociación, se le asignará una lista de tareas o plan, el cual se le enviará al Sistema Gateway, y este, a su vez a través del ACLGWAgentROS, publicará en el tópico de coordenadas /flexmansys/coordenada/TUN las coordenadas de las tareas que ha recibido. Pese a que una lista de tareas puede estar compuesta por hasta 30 coordenadas distintas, tal y como indica el requisito R2, estas se irán pasando al transporte de una a una.

7. El agente transporte sabrá en función de lo que recibe de los datos de estado del transporte el cuándo ha acabado su lista de tareas, por lo que le indicará que debe volver a su puesto de carga y notificará al agente solicitante del servicio que su transporte representado ha terminado sus tareas.

Tal y como se ha comentado, el Sistema Gateway puede recibir dos tipos de encomiendas: Tareas o coordenadas sueltas, que no son más que un único string; o un listado de tareas, que se tratan de un array de string. A pesar de ello, el Sistema Gateway debe de publicar las tareas de una en una en el tópico de coordenadas. Es por ello, que, la clase GWagentROS se encarga desglosar ese array de coordenadas y suministrarlas de una en una al transporte.

El Sistema Gateway sabrá si el mensaje recibido desde el agente transporte es una única coordenada o un listado de coordenadas a través de la ontología del mensaje ACL que reciba. Si se recibe una ontología de "ComandoCoordenada", será una única tarea, mientras que, si recibe "PlanCoordenadas", recibirá un conjunto de estas. Por otro lado, ambos tipos de mensajes cuentan con otra diferencia fundamental, y es el momento en el cual ellos pueden escribir en el tópico de coordenadas /flexmansys/coordenada/TUN. En lo relativo a "ComandoCoordenada", podrá escribir siempre que se reciba un mensaje desde el agente transporte con dicha ontología, independientemente del estado en el que se encuentre el Sistema de Navegación del agente transporte; como por ejemplo cuando requiere de una parada de emergencia o de parada. En cambio, las coordenadas recogidas dentro del mensaje con ontología "PlanCoordenadas" únicamente pueden escribirse en el tópico si se ha dado un flanco de activación del estado Operative al Active desde la primera tarea requerida, es decir, si el transporte ha vuelto al estado Active por su propio pie, lo que indicaría que la tarea se ha efectuado sin errores.

La comunicación que se da entre el agente transporte y el GWagentROS para indicar las coordenadas recogidas en el listado de tareas o plan de coordenadas, se inspira principalmente en la estructura seguida en las comunicaciones serie: Con una cabecera o bit de start, unos datos posteriores, y un bit de cierre. También, cuenta con características propias de la comunicación de TCP/IP, en la cual se indica desde el receptor que se han recibido los datos cada vez que estos son recibidos. Se muestra en la *Figura 6.23* un ejemplo de dicha transmisión de datos, donde el bit de start indica el número de tareas que se van a escribir, y donde el bit de finalización se indicaría mediante un mensaje con contenido "END". Del mismo modo, las coordenadas a desplazarse serían A1, C3 y K2, en dicho orden de llegada al Gateway.

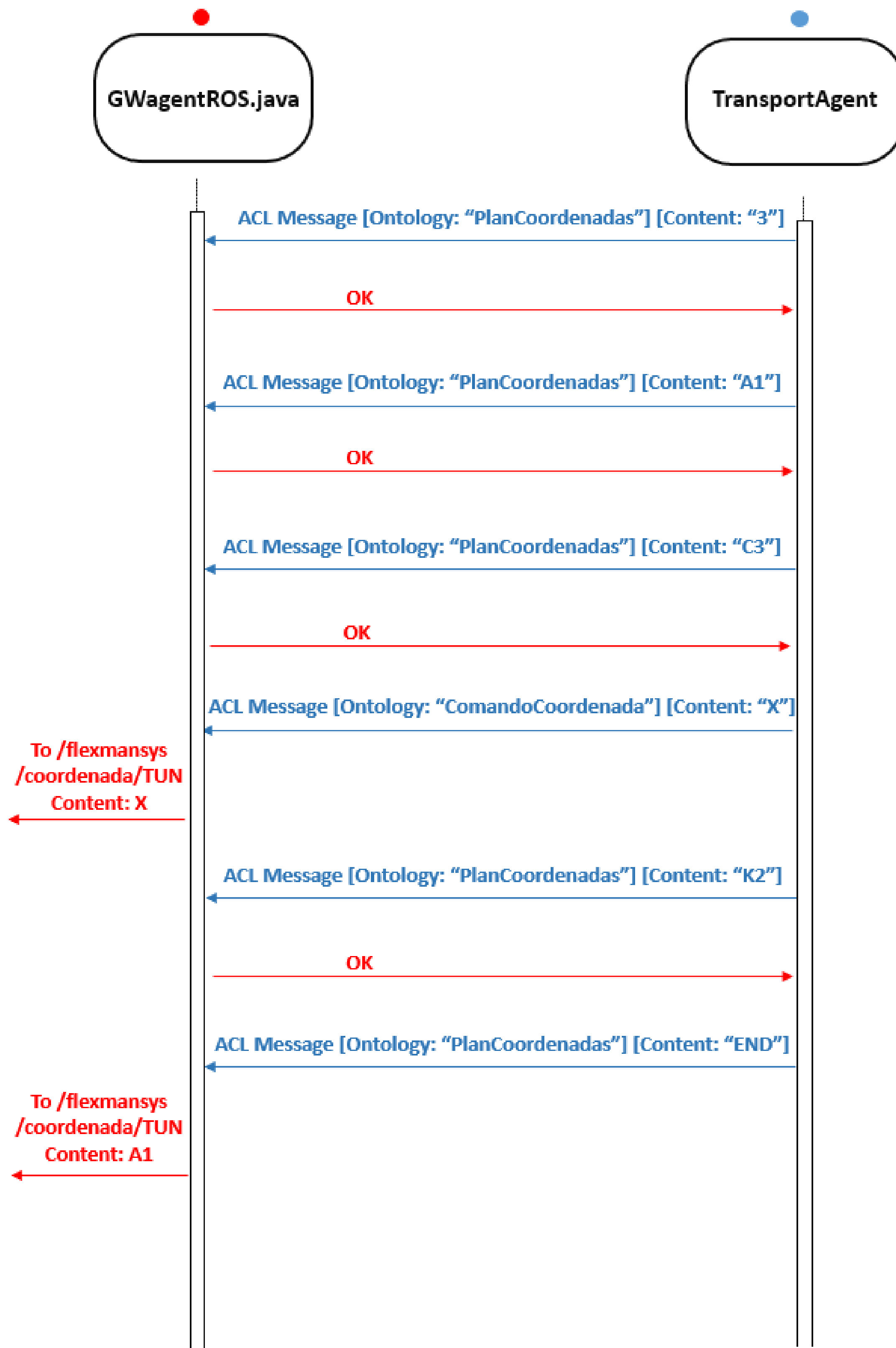


Figura 6.23 Ejemplo Escritura de Tareas desde Agente Transporte

6.6 Sistema de Agentes de Transporte

Tal y como se ha comentado al inicio del presente capítulo, el Sistema de Agentes de Transporte se trata del conjunto de agentes que gestionan y representan todo lo relativo a los AGV dentro del MAS de la planta donde se integran dichos transportes. Dentro de este sistema-multi agente, existirán hasta cuatro agentes de tipo transporte, dado que tal y como indica la especificación E1 pueden darse hasta cuatro AGV dentro del Sistema de Navegación. Es por ello, que la conexión entre el MAS y los transportes quedaría tal y como se muestra en la *Figura 6.24*

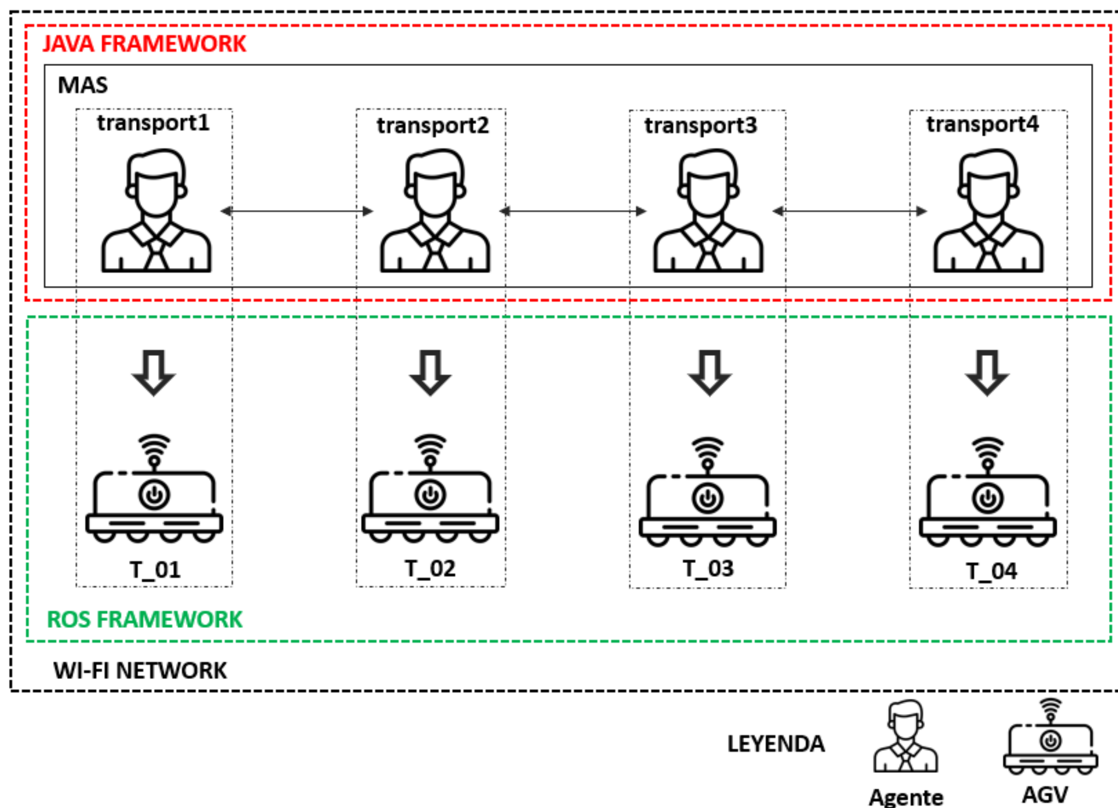


Figura 6.24 Esquema Simplificado de la Estructura Formada por los Agentes Transporte y los AGV

Sin embargo, el MAS en el que se integran los agentes transporte se trata de un conglomerado de archivos, ficheros y clases java que permiten no solo la gestión de transportes, sino que también cuenta con agentes enfocados a procesos, a células o incluso a la gestión, como se desarrolló en el apartado de antecedentes del capítulo tres en el punto 3.2.3. Evidentemente, todas estas clases no son necesarias para poder implementar a los agentes transporte, siendo única y exclusivamente necesarias dos: TransportAgent.java y Transport_Functionality.java. Son estas dos clases java las que en su conjunto forman al agente transporte. A pesar de ello, estas clases y los métodos que estas emplean son herencias o extensiones de otras clases recogidas dentro del MAS, las cuales se muestran en la *Figura 6.25*.

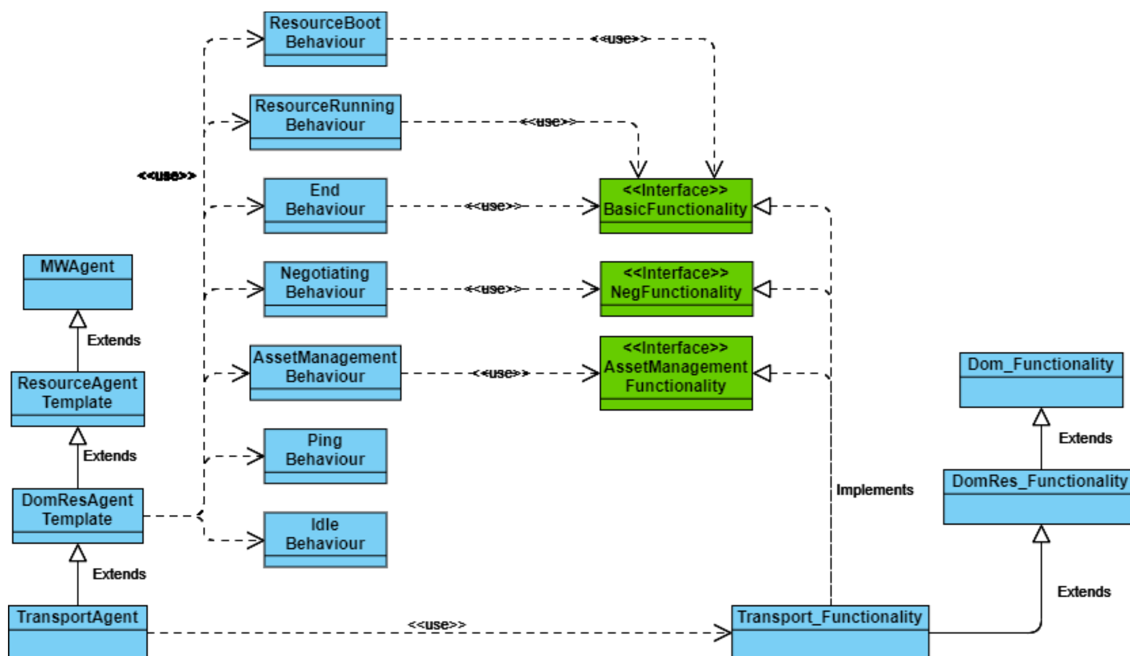


Figura 6.25 Esquema de Herencias de Clases Relativas al Agente Transporte

En lo relativo a las funcionalidades que ocupan las dos clases java que forman el agente transporte, se recogen:

- **TransportAgent**: Se refiere a la clase main del agente transporte, se encarga de inicializar al agente y de instanciar su funcionalidad mediante la clase Transport_Functionality. Define las variables comunes que van a emplearse y designa un archivo para recoger su plan de transporte.
- **Transport_Functionality**: Recoge toda la funcionalidad del agente transporte, desde su inicialización en la plataforma JADE hasta las tareas más simples de lectura y escritura de datos desde y hasta la unidad de transporte. Por otro lado, también implementa el algoritmo empleado para determinar el valor de cada transporte en la negociación y la escritura de planes en su archivo de plan de transporte.

6.6.1 Funcionamiento General

En lo que concierne al funcionamiento general de los agentes transporte, podría resumirse a través de los siguientes métodos recogidos dentro de Transport_Functionality:

- **init**(): Este método se invoca desde la clase TransportAgent, en esencia, será aquel que inicializa el agente dentro de la plataforma JADE, verificando previamente que el Sistema Gateway está activo y recogiendo los argumentos de entrada que se han introducido a la hora de ejecutar el agente transporte para dar nombre al agente en el MAS. Se comunicará con el SMA o System Model Agent para recibir un nombre tras verificar que el Sistema Gateway responde, dándole el nombre al agente de transportX, siendo X el numberAgent.

- **execute()**: Este método es llamado desde checkNegotiation, el cual le introduce como argumentos de entrada las operaciones que ha recibido desde calculateNegotiationValue. En resumen, se encarga de coger todas esas operaciones, las cuales están contenidas dentro de un String, y de separarlas para posteriormente escribirlas en el archivo donde se recogen los planes de transporte, es decir, TransportPlan.xml.

- **calculateNegotiationValue()**: En este método se implementan los algoritmos de cálculo del valor del agente de transporte en la negociación. Este valor puede determinarse o bien por posición o distancia desde el punto inicial a la primera coordenada requerida, o bien por nivel de batería.

- **checkNegotiation()**: En este método se dan las comprobaciones necesarias para determinar si el agente transporte es el ganador o perdedor de una negociación. En caso de ser el ganador, invocará al método execute para introducir en el plan de transporte recibido desde el solicitante de la negociación.

- **sendDataToDevice()**: Se trata del método que se comunica con el Sistema Gateway, aquel que en esencia le envía la lista de tareas o plan de transporte para publicarlas posteriormente en el tópico de coordenadas /flexmansys/coordenada/TUN.

- **rcvDataFromDevice()**: Este método es aquel que de manera periódica lee los datos de estado de la unidad de transporte recibidos desde el Sistema Gateway y actualiza los campos correspondientes de la clase TransportAgent. Por otro lado, también es el encargado de mandar el comando de calibración al transporte en caso de ser necesario y de eliminar el plan de transporte completado del fichero TransportPlan.xml.

Al igual que con el desarrollo del Sistema Gateway, se muestra el esquema de la *Figura 6.26* para describir el funcionamiento del agente transporte, abstrayendo, en este caso, lo relativo al Sistema Gateway. Cabe destacar, que en este esquema se considera que el agente Gateway ya están arrancados y en pleno funcionamiento, lo que corresponde hasta el momento previo al paso 3 de la anterior *Figura 6.22,* cuyo equivalente en este nuevo esquema sería el paso 1. Se describen a continuación cada uno de los pasos de funcionamiento del agente transporte:

1. Primeramente, con el Sistema de Navegación y el Sistema Gateway ya en marcha, se inicializa el Agente transporte, introduciendo de entrada tres argumentos: El TUN del transporte, el número que tomará el agente transporte en la plataforma, y el fichero de plan de transporte al que va a asociarse. Posteriormente, la clase TransportAgent se instanciará a Transport_Functionality para llevar a cabo el proceso de inicialización.

2. Una vez se entre al método de init(), el agente transporte comprobará que su Gateway correspondiente está activo y que recibe información desde el transporte. Durante dicha comprobación, el agente transporte tomará el nombre de “auxma-local” en la plataforma JADE, un nombre auxiliar y temporal. Tras verificar que el Sistema Gateway está vivo y que ha recibido el mensaje desde el agente transporte para iniciar la comunicación, se dará paso al cambio de nombre del agente transporte.

3. Llegados a este punto, se le solicitará al SMA o System Model Agent el cambio de nombre de “auxma-local” a transportX, siendo X el número correspondiente al número de agente de transporte, el cual debe de coincidir necesariamente con el introducido en los argumentos de entrada. Si el SMA está lanzado y ejecutándose de manera correcta, no habrá mayores problemas y el agente transporte cambiará de nombre.

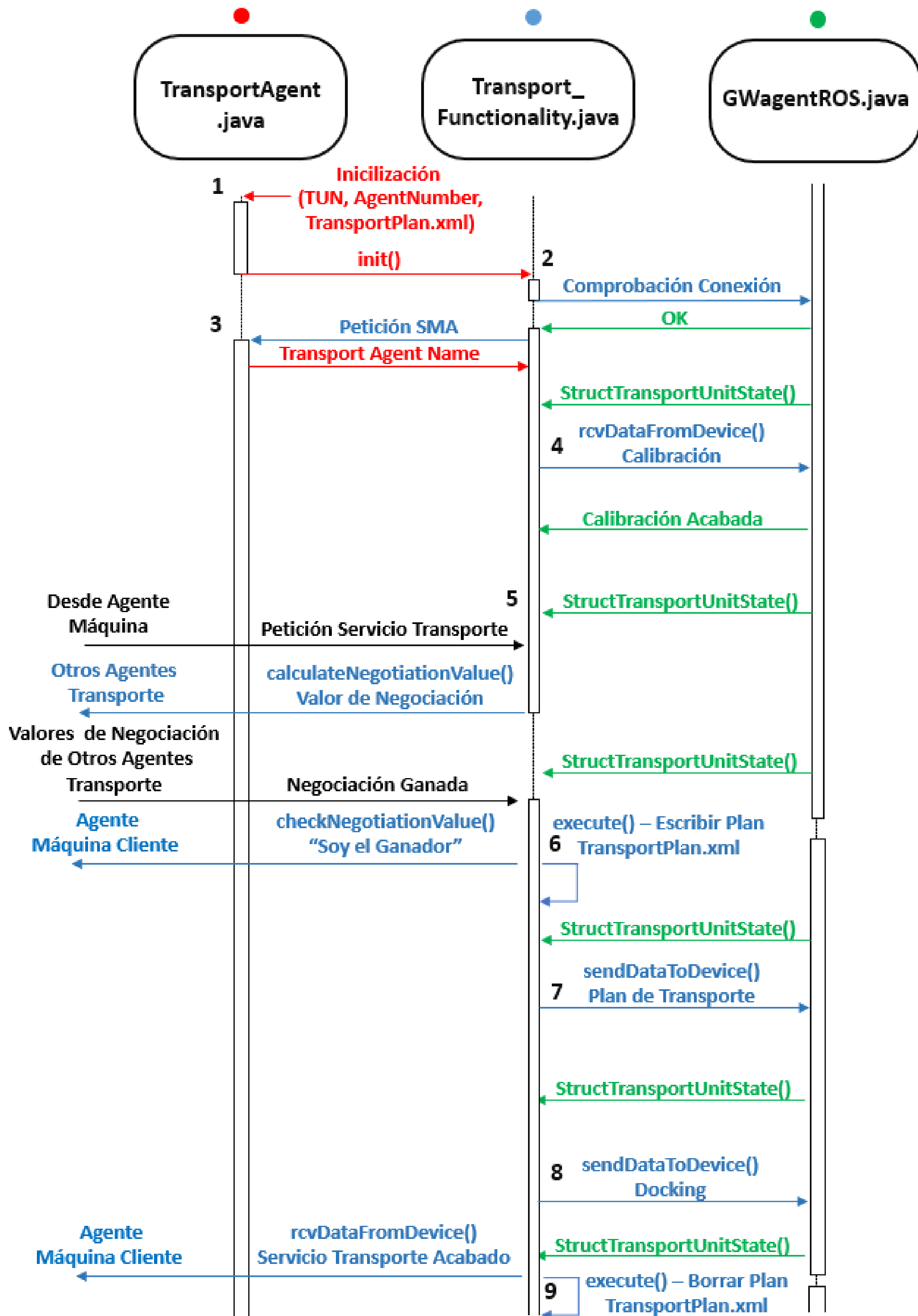


Figura 6.26 Esquema de Funcionamiento del Agente Transporte

4. Si tras la inicialización del agente transporte este ha llegado a este punto, la unidad de transporte necesariamente debe de estar en el estado de "Idle", esperando su calibración. Es por ello, que, desde el propio método de lectura de datos desde el transporte, es decir, rcvDataFromDevice() se le manda el comando de calibración al AGV.

5. El agente transporte sabrá que ha finalizado su calibración una vez reciba desde el tópico de estado /flexmansys/state/TUN que la unidad se encuentra en el estado "Active", momento el cual comenzará a escuchar las peticiones de negociación de otros agentes. En este caso, una agente máquina, que podría ser una célula o proceso cualquiera de la planta, le solicita realizar un servicio de transporte. Esta solicitud se trata de un mensaje tipo ACL el cual sus campos se desglosan según la *Tabla 6.6.* y donde se recoge el plan que deberá seguir el cliente, es decir, el transporte representado por el agente transporte. Posteriormente, entra el método calculateNegotiationValue(), el cual calculará el valor de la negociación según el negotiationCriteria introducido. Este valor, se enviará a la clase NegotiationBehaviour, la cual comparará el valor calculado de todos los agentes transporte solicitados para el servicio.

6. En caso de ganar la negociación, entrará el método checkNegotiationValue(), en la cual se solventarán empates entre agentes o situaciones en las que no existan varios agentes a negociar, actuando como confirmador de que el agente transporte en cuestión ha sido el ganador. Tras ello, se invocará desde checkNegotiationValue() al método execute(), que introducirá las coordenadas del plan de transporte solicitadas por la agente máquina que solicita el servicio transporte en el fichero TransportPlanX.xml, siendo X el número de agente de transporte en la plataforma. Finalmente, el método checkNegotiationValue() notificará al agente solicitante del servicio quién es el que va a realizar el transporte, es decir, qué agente transporte ha ganado la negociación.

7. Una vez ganada la negociación, la agente máquina cliente notificado y el plan escrito en el TransportPlanX.xml, se da paso al envío de la lista de tareas o plan al Sistema Gateway, que este a su vez siguiendo el ejemplo de la *Figura 6.23*, enviará los datos al Sistema de Navegación. Lo hará a través del método sendDataToDevice(), el cual se apoyará en el fichero KeyPositions.xml para traducir de coordenadas escritas, como "KukaStation", a coordenadas entendible por el transporte, como "K2".

8. El agente transporte sabrá cuando ha acabado de ejecutar el servicio mediante los datos de estado que le llegan desde el tópico /flexmansys/state/TUN. Será entonces, cuando solicitará al transporte que vuelva a su estación de carga, únicamente en caso de no haber ningún otro plan en el fichero de plan de transporte TransportPlanC.xml. Si hay otro plan dentro de dicho transporte, procederá a ejecutarlo volviendo al paso 7.

9. Finalmente, antes solicitar al transporte que vuelva a su estación de carga, el método rcvDataFromDevice() borrará el plan de transporte ejecutado del TransportPlanX.xml. De este modo, al no haber ningún plan dentro de dicho fichero, el agente transporte sabrá que debe de indicar a la unidad de transporte que vuelva a su estación de carga. Por otro lado, también indicará al agente solicitante del servicio, es decir, la agente máquina del paso 5, que el servicio ha sido finalizado correctamente.

6.6.2 Lógica de Negociación

El sistema de negociación reside dentro de la clase NegotiationBehaviour, dentro de la cual se implanta toda la lógica que siguen los agentes a la hora de negociar entre ellos para una determinada tarea. Por su parte, los dos métodos relativos a la negociación mencionados anteriormente, calculateNegotiationValue y checkNegotiationValue, son aquellos que implementan, por así decirlo, las normas que los agentes transporte van a seguir para determinar el cómo de aptos o no aptos son para una determinada tarea. Estas normas, se recogen dentro de dos algoritmos distintos: Uno el cual centra su negociación en el nivel de batería restante de las unidades de transporte, y otro que la centra en la distancia que hay desde el punto en el que se encuentra el transporte y la coordenada objetivo.

La negociación la invoca cualquier agente que requiera del servicio de una unidad de transporte, es decir, del traslado o reabastecimiento de material. Habitualmente, los agentes que invocan la negociación son aquellos que representan a máquinas o procesos en específico, los cuales generan una arquitectura tipo cliente y servidor con el agente transporte correspondiente. Para invocar la negociación, se envía un mensaje ACL a los agentes transporte implicados, cuyos campos se muestran en la *Tabla 6.6*.

Campo	Contenido	
Performative	CFP (Call For Proposal)	
Receiver	Nombre del agente receptor del mensaje dentro de la plataforma	
Ontology	Ontología del mensaje para la negociación, debe de indicarse “negotiation”.	
ConversationID	ID de la conversación, concretamente debe indicarse “1234”	
Content	**targets**	Transporte al que se refiere la llamada a servicio transporte.
	negotiationCriteria	Parámetro que establece bajo qué parámetro se va a negociar, si bien por batería o por posición.
	negAction	Tipo de negociación que va a seguirse de las recogidas en el NegotiationBehaviour, concretamente se indica “supplyConsumables” para la negociación con agentes.
	externalData	Nombre de las coordenadas requeridas para el plan de transporte y máquina solicitante del servicio de transporte.

Tabla 6.6 Estructura Mensaje Solicitante del Servicio

Cabe destacar que dentro del campo externalData del mensaje de negociación se recogen las coordenadas que van a incluirse en el plan de transporte del agente transporte que gane la negociación. Estas coordenadas o tareas se indican mediante su nombre convencional, es decir, el nombre de la estación o punto al que desplazarse. Posteriormente, se hace uso del fichero KeyPositions.xml mencionado anteriormente para establecer a qué coordenadas en concreto se refiere. En el caso de querer desplazarse hasta el punto de recogida de la célula del robot Kuka, se indicaría dentro de externalData como kukaStation, mientras que una vez se comprueba en el fichero KeyPositions.xml, la coordenada que se indica al transporte en el tópico de coordenadas /flexmansys/coordenada/TUN será “K2”.

6.6.2.1 Algoritmo de Batería

El pack de baterías de los transportes Turtlebot2 está formado por celdas del tipo Litio-ion con una configuración 4s1p, es decir, 4 celdas en serie y una en paralelo. Esto, le da una tensión nominal de 14.8 VDC con una capacidad de 2200 mAh, lo que equivale a en torno a 2 o 3 horas de autonomía, en función del estado de salud de las celdas y la cantidad de tareas asignadas a ejecutar y periféricos conectados. De tal modo, cada celda del pack cuenta con una tensión nominal de 3.7 VDC con una capacidad de 2200 mAh.

La principal característica de las celdas tipo Litio-ion a la hora de calcular la cantidad de energía que contienen, es decir, calcular su SOC (State of Charge), es la no linealidad de la curva de descarga de estas celdas. Esta curva, cuenta con dos regiones exponenciales negativas con una pendiente muy pronunciada y una región plana, donde la tensión varía muy poco. Esta región plana, supone el 80 % de todo el recorrido de energía de la celda, lo que hace que en ocasiones sea muy difícil estimar la cantidad de energía real con la que cuenta el pack de celdas. Esto, se debe a que al tener tan poca variación de voltaje de un punto a otro, una lectura de 14.5 VDC de la batería podría equivaler a un 55 % de energía restante, o bien al 60 % de energía restante. Por otro lado, no es lo mismo medir el voltaje de la batería cuando esta está en reposo que cuando se está descargando por un motivo cualquiera.

La curva de descarga correspondiente al Turtlebot2 se aporta en la *Figura 6.27*, gráfica que provee la página oficial de documentación de Yujin Robots. Nótese que hay dos gráficas: Una para la configuración 4s2p, la azul que corresponde al pack de baterías grande de 7 h de autonomía; y la 4s1P, la roja con la que están equipadas las unidades robóticas del presente trabajo.

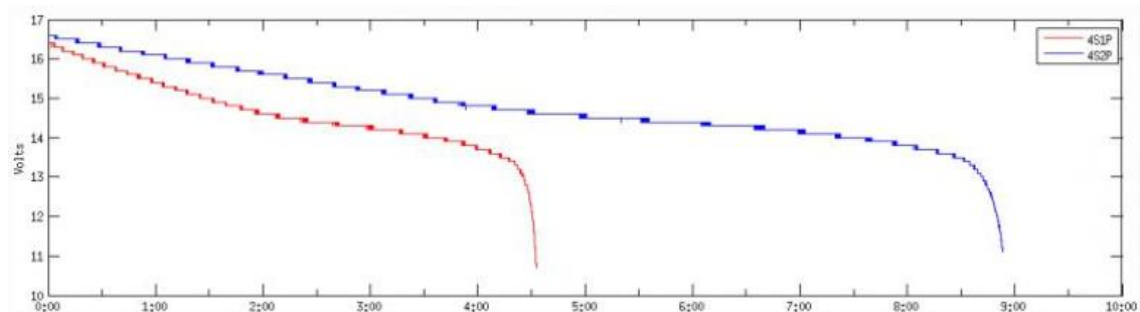


Figura 6.27 Curva de Descarga de la Batería del Turtlebot2

Dado que al no disponer de un modelo exacto de la batería de los transportes y al ser necesario el desarrollar un estimador del SOC individualizado para cada transporte según la cantidad de energía que consume y el estado de salud de su pack de baterías, se optó por hacer un modelo más simple pero mucho más inexacto en el que simplemente se linealizase la curva de descarga previamente mostrada en rojo. Para ello, se definieron tres regiones distintas, en las cuales gobernaría un estimador según el valor de batería publicado en el tópico de estado /flexmansys/state/TUN. Estas regiones, están definidas por los límites recogidos en la siguiente *Tabla 6.7*.

Región	Nivel de Energía	Nivel de Batería	Función Linealizada
Higland	100 % - 80 %	16.8 VDC – 14.8 VDC	Energía = 10 * Batería - 68
Midland	80 % - 20 %	14.8 VDC – 13.6 VDC	Energía = 50 * Batería – 660
Lowland	20 % - 0 %	13.6 VDC – 13.2 VDC	Energía = 50 * Batería – 660

Tabla 6.7 Regiones Linealizadas de la Curva de Descarga

El valor de la negociación se obtiene multiplicando por 10 al valor obtenido de energía de la función linealizada que corresponda para el nivel de batería leído. Nótese, que a pesar de que se establezca como la región baja o Lowland al último tramo de energía, el correspondiente al 20 % y al 0 %, a la hora de asignar las tareas a los agentes transporte, hay que considerar y tener en cuenta si el transporte va a tener suficiente energía como para realizar todo el trayecto que supone el plan que se le ha cargado. Por otro lado, el hacer trabajar al pack de celdas por debajo del 20 % puede ser muy perjudicial para el estado de salud de estas, por lo que un transporte que tenga un nivel de batería de 13.6 o inferior, directamente, su valor de negociación será 0 y quedará prácticamente anulado de la conversación para la negociación.

6.6.2.2 Algoritmo de Posición

En muchas de las aplicaciones y desarrollos que pudieron estudiarse durante la realización del estado del arte, se vio que muchas aplicaciones se ayudaban de simples conceptos trigonométricos para determinar la distancia desde el punto de origen al punto destino. Sin embargo, el entorno de operación de las unidades de transporte, es decir, el laboratorio en el que se iba a operar no era apto para emplear este método al encontrarse situaciones en las que el cálculo fuese inefectivo. Esto se debe a la presencia de muebles y células en medio del entorno de trabajo, lo que haría que en caso de solicitarse una coordenada que estuviese al otro lado de dicho obstáculo, el cálculo fuese inefectivo al no contar con la distancia del trayecto a realizar para esquivar dicho elemento.

Es por ello, que se optó por aprovechar la subdivisión en secciones realizada del entorno de trabajo, como se ha podido apreciar en el capítulo anterior de datos de partida. En esencia, todos los trayectos que va a realizar la unidad de transporte ya están recogidos previamente, puesto que las posibilidades de traslado de una célula a otra no son muchas. De este modo, al saber si la coordenada destino estaba en la misma sección a donde se sitúa el transporte, se sabe que puede aplicarse trigonometría y calcular el módulo del vector que une ambos puntos para determinar la distancia que los separa. En caso de encontrarse en secciones distintas, se sabe la distancia que hay desde un extremo de la sección a otro, por lo que se calcularía dicha distancia y se sumaría la distancia desde el extremo de la última sección al punto solicitado nuevamente mediante el cálculo del módulo del vector que separa ambos puntos.

6.7 Distribución y Operación en Planta

En este apartado se mostrará un breve ejemplo del cómo se efectúa una operación de un agente transporte en el entorno de operación. Por otro lado, se recogerá al final en una tabla todos los comandos registrados en la solución y su significado.

6.7.1 Operación en Planta

Para describir el funcionamiento del sistema en su conjunto, lo mejor es mostrar un ejemplo del cómo solucionaría un transporte un servicio de traslado dentro del entorno de operación. Para ello, se hará uso del plano mostrado en el capítulo anterior de Datos de Partida, concretamente la *Figura 5.1,* donde un transporte trabajará en las secciones A, B y D. Considerando el inicio y la puesta en marcha, todo servicio de transporte aportado por un AGV se puede dividir en 6 etapas, las cuales se describirán a continuación.

Primeramente, se comienza en la etapa inicial, donde se colocan todos los AGV en sus estaciones de carga respectivas, es decir, lo que para el Sistema de Navegación es su punto de partida o punto inicial, la coordenada con la que se relaciona el sistema de coordenadas de la odometría. Tras ello, se encienden los transportes con los que se quiera operar, para este caso, se considerará que trabajaremos con las unidades T_01 y T_02. Posteriormente, se van ejecutando desde una terminal asociada a dichos transportes los respectivos sistemas de la solución desarrollada en el siguiente orden: Primero el Sistema de Navegación, el segundo el Sistema Gateway, y finalmente el MAS con sus respectivos agentes de transporte, el transport1 y el transport2. Se muestra en la *Figura 6.28* la distribución en planta de esta etapa.

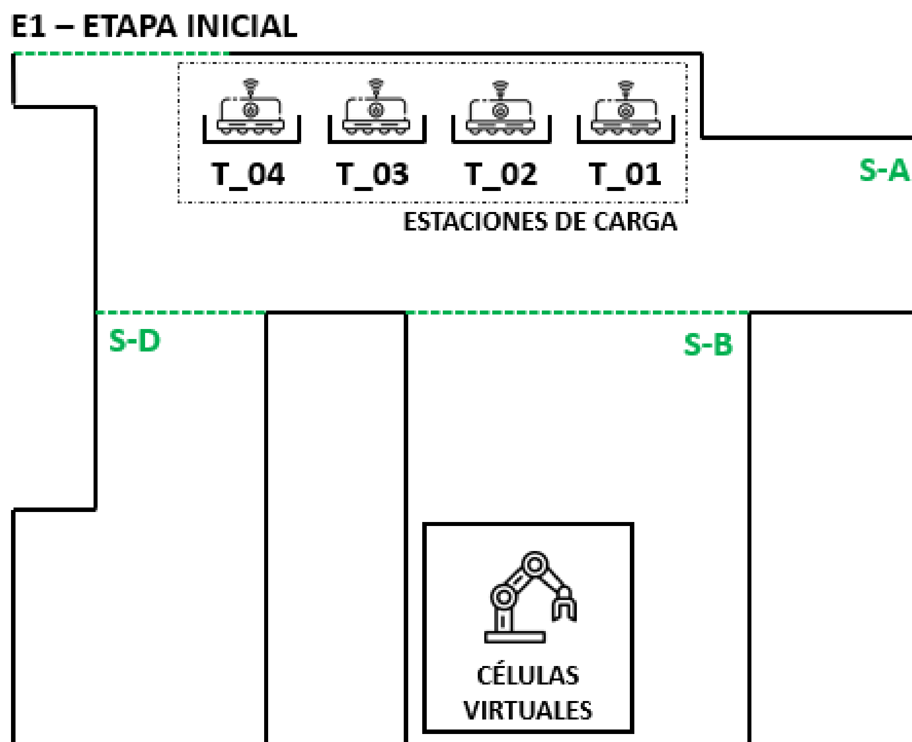


Figura 6.28 Etapa Inicial

La segunda etapa da comienzo cuando los agentes transporte conectan con sus respectivas unidades de transporte, las cuales estarán en el estado de Idle esperando a que se le envíen instrucciones a realizar. Tras ello, el agente transporte enviará el comando de calibración o “X” a la unidad de transporte, comando el cual el Sistema Gateway publicará en el tópico de coordenadas /flexmansys/coordenada/TUN. Una vez se reciba la orden, los AGV saldrán de sus respectivas estaciones de carga y darán paso a la calibración del algoritmo de localización, el AMCL, donde ejecutarán dos vueltas de 360º sobre sí mismas, una en sentido horario y otra en sentido antihorario. En caso de efectuarse correctamente la operación, la distribución en planta será similar a la

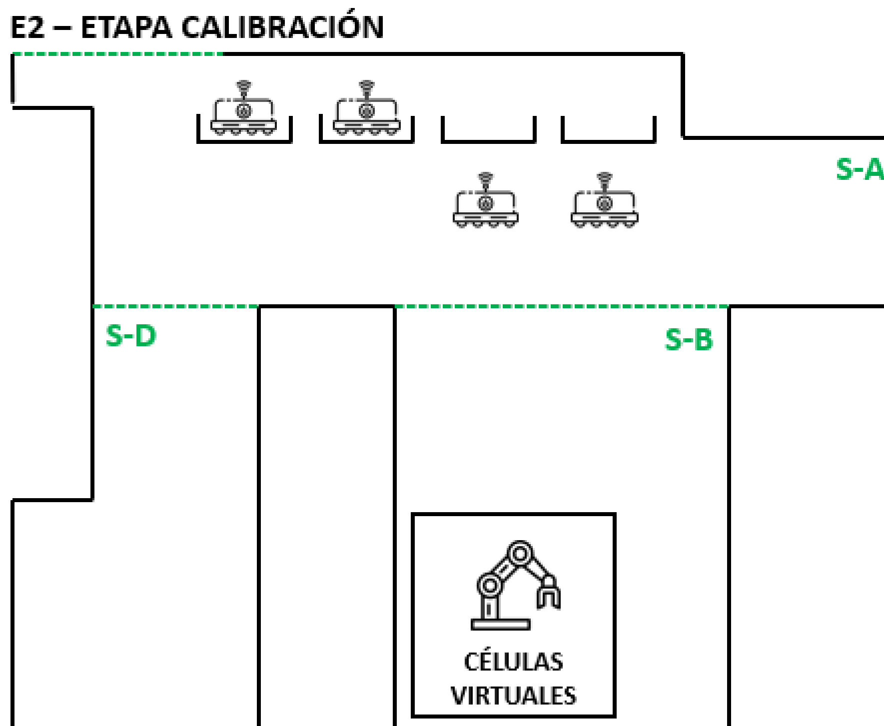


Figura 6.29 Etapa de Calibración

Pasada la segunda etapa, se da paso a la tercera, donde ambos transportes estarán calibrados y activos para recibir tareas, es decir, estarán en el estado Active de la máquina de estados de su Sistema de Navegación. Pasado un tiempo, se supondrá que se recibe un mensaje ACL de uno de los agentes máquina representante de una de las células virtuales de la planta, el cual solicita el reabastecimiento de su dispensador de material. Para ello, detecta que hay dos agentes transporte activos en el MAS, por lo que envía la encomienda a ambos agentes. Se supondrá que requiere de un plan de transporte que implica dos puntos diferentes: supplyStation y virtualKukaInput. El contenido del mensaje que enviará al agente transporte que representa a la unidad T_01 se muestra en la *Tabla 6.8*.

Campo	Contenido	
Performative	CFP	
Receiver	transport1	
Ontology	negotiation	
ConversationID	1234	
Content	**targets**	transport1
	negotiationCriteria	battery
	negAction	supplyConsumables
	externalData	supplyStation;kukaInput,machine1

Tabla 6.8 Contenido Mensaje Servicio Transporte a Agente Transporte de AGV T_01

Tras recibir ese mensaje, ambos transportes negociarán mediante el algoritmo de batería, por lo que el ganador será aquel que tenga mayor nivel de energía, es decir, mayor SOC. En este caso, se supondrá que se trata del transporte T_01 el ganador de la negociación. Este, recibirá a su tópico de coordenadas correspondiente la primera coordenada, la relativa a supplyStation, la cual se va a suponer que es la “A3”. Inmediatamente, el transporte pasará del estado Active al estado Operative, y rotará sobre sí mismo hasta encarar la posición de la coordenada A3. La situación tercera se muestra en la *Figura 6.30*. El transporte T_02, al no tener ningún plan asignado en su TransportPlan2.xml al no ganar la negociación, volverá a su estación de carga.

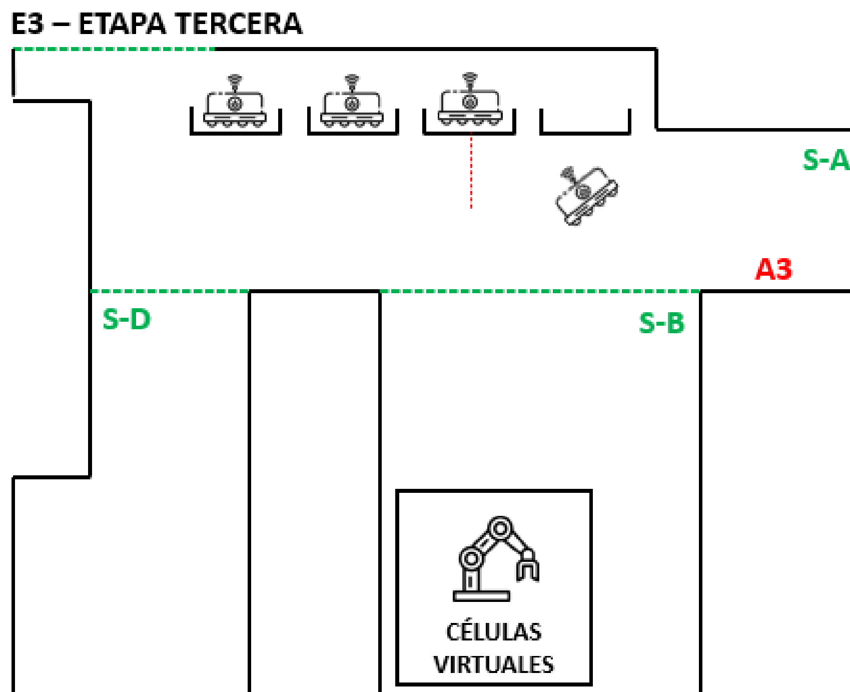


Figura 6.30 Etapa Tercera

Posteriormente, se cederá el control de los motores al paquete move_base, el cual trazará la trayectoria de navegación desde el punto inicial en el que se encuentra hasta el A3, dando lugar a la cuarta etapa mostrada en la *Figura 6.31*.

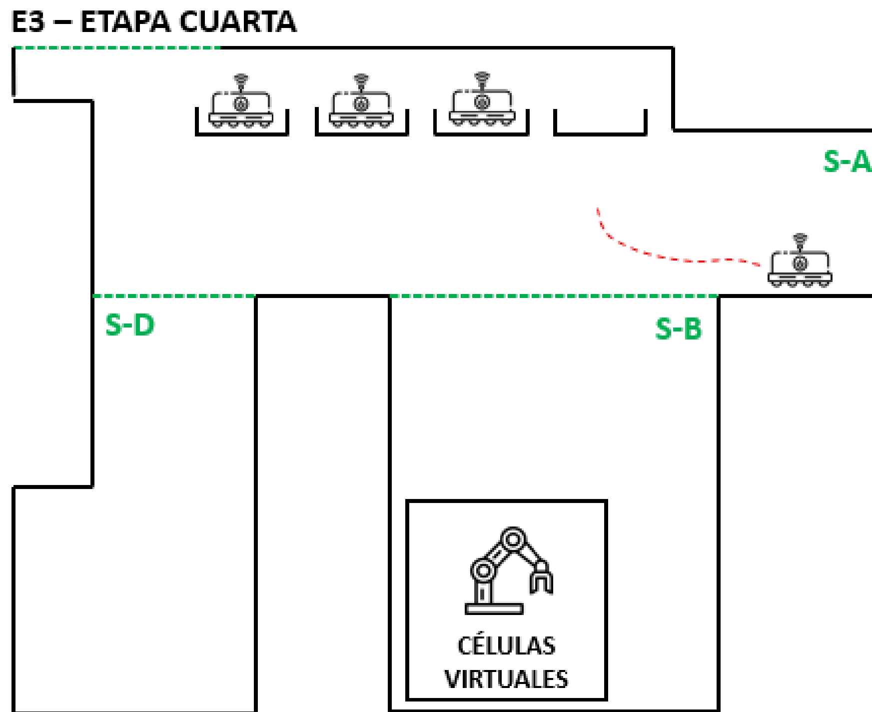


Figura 6.31 Etapa Cuarta

Una vez la actividad de manipulación sea finalizada en dicha estación, se repetirá el proceso para ir a la coordenada correspondiente de "virtualKukaInput", correspondiente a la siguiente *Figura 6.32*.

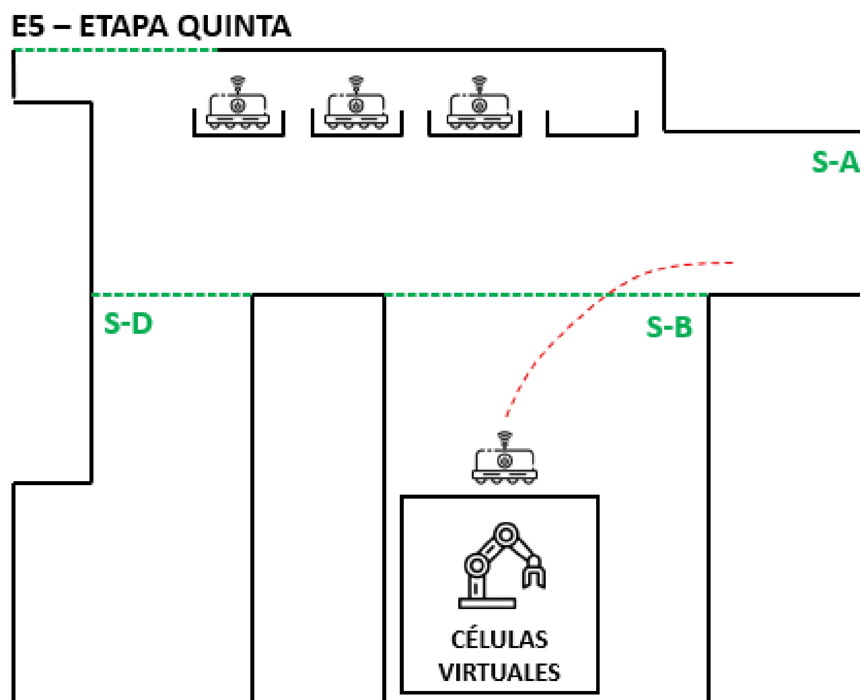


Figura 6.32 Etapa Quinta

Tras la finalización de las actividades de manipulación en la coordenada de entrada de la célula virtual, el agente transporte correspondiente a T_01 verificará si existe algún que otro plan en su TransportPlan1.xml respectivo e indicará al agente máquina solicitante del servicio, machine1 en este caso, que el servicio de transporte ha sido finalizado. En caso de no ser así, le indicará a la unidad de transporte que vuelva a la estación de carga mediante el comando "DOCK" en el tópico de coordenadas, dándose la situación de la *Figura 6.33*.

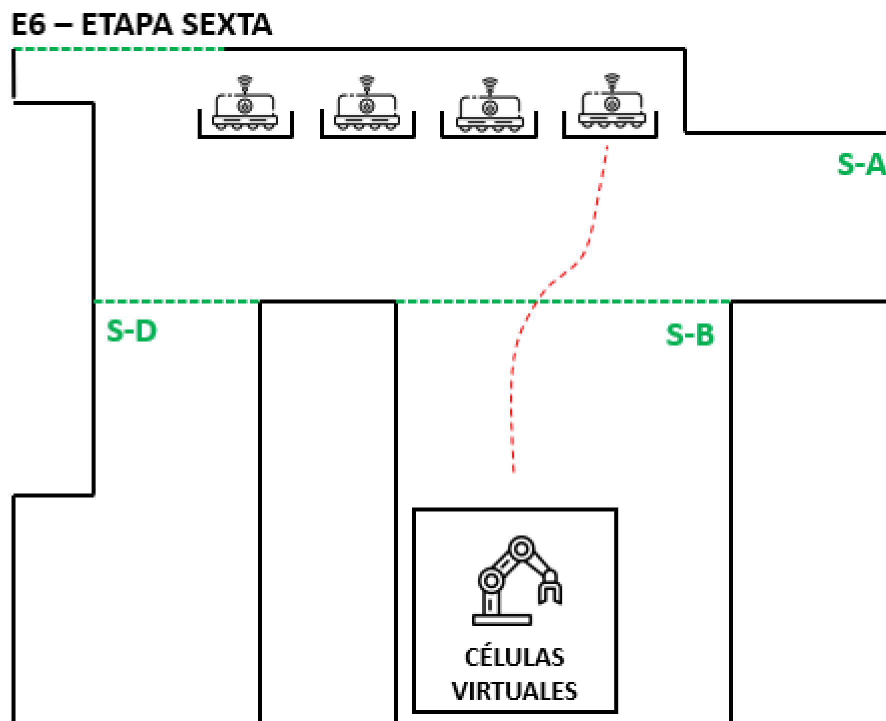


Figura 6.33 Etapa Sexta

El transporte, en caso de no ser indicado de lo contrario, permanecerá en su estación de carga en el estado Active, esperando a nuevas encomiendas por parte su agente transporte. Por otro lado, pese a que la unidad de transporte T_02 haya vuelto a su estación de transporte en la etapa 2 al no tener ningún plan asignado, ambas unidades, tanto T_01 como T_02 no tendrán que volver a pasar por la etapa de calibración, al ya haberla realizado anteriormente. En esencia, ambos transportes estarán listos para recibir nuevas peticiones de servicio.

6.7.2 Comandos Definidos en el Sistema

Se muestra en la *Tabla 6.9* todos los comandos los cuales tienen ya un valor predefinido en el sistema. Estos comandos se publican en el tópico /flexmansys/coordanada/TUN, bien sea por un operario, por el nodo de emergencia o por el agente transporte correspondiente a través del Sistema Gateway o clase ACLGWAgentROS.java. Se obvian aquellas coordenadas o comandos recogidos dentro de cada unidad de transporte y que corresponden a posiciones en el mapa.

Comando	Significado
X	Comando de calibración, en caso de publicarse este comando en el tópico de coordenadas la unidad de transporte sabrá que el agente transporte está vivo y correctamente lanzado en el MAS, por lo que pasará del estado Idle a Localization, dando paso a la calibración para posteriormente pasar a Active.
E	Comando de solicitud de emergencia, en caso de publicarse este comando en el tópico de coordenadas el nodo de emergencia matará todos los nodos ROS del Sistema de Navegación, incluyéndose a sí mismo, parando en seco al transporte y activando los indicadores de emergencia: LED y sonido.
STOP	Comando de solicitud de parada, en caso de publicase este comando en el tópico de coordenadas y en caso de que el transporte se encuentre en reposo, se evolucionará al estado de Stop del sistema.
FREEWAY	Comando de confirmación de obstáculo retirado, en caso de publicarse este comando en el tópico de coordenadas se hará saber al transporte, que se encuentra en el estado de Recovery, que el objeto que obstaculizaba su trayectoria ya ha sido retirado.
DOCK	Comando de estacionamiento, en caso de publicarse esta coordenada en el tópico de coordenadas se hará saber al transporte que debe volver a su estación de carga, es decir, a su punto inicial de partida.

Tabla 6.9 Comandos Definidos en el Sistema

CAPITULO 7

RESULTADOS Y ANÁLISIS

7 Resultados y Análisis

En este capítulo, se detallarán todas las pruebas realizadas y los resultados obtenidos de las mismas. Los resultados y pruebas realizadas se desglosarán según los objetivos principales del presente trabajo: navegación autónoma y negociación entre agentes. Finalmente, se aportará una prueba general del sistema, así como la validación de sus sistemas de seguridad.

Para cada prueba, se aportará una tabla de verificación de prueba, la cual recogerá el objeto de la prueba, los pasos a tomar, los resultados esperados y los resultados obtenidos. Tras cada tabla de verificación, para cada prueba, se procederá a ilustrar mediante pruebas gráficas los resultados obtenidos para cada paso.

7.1 Navegación

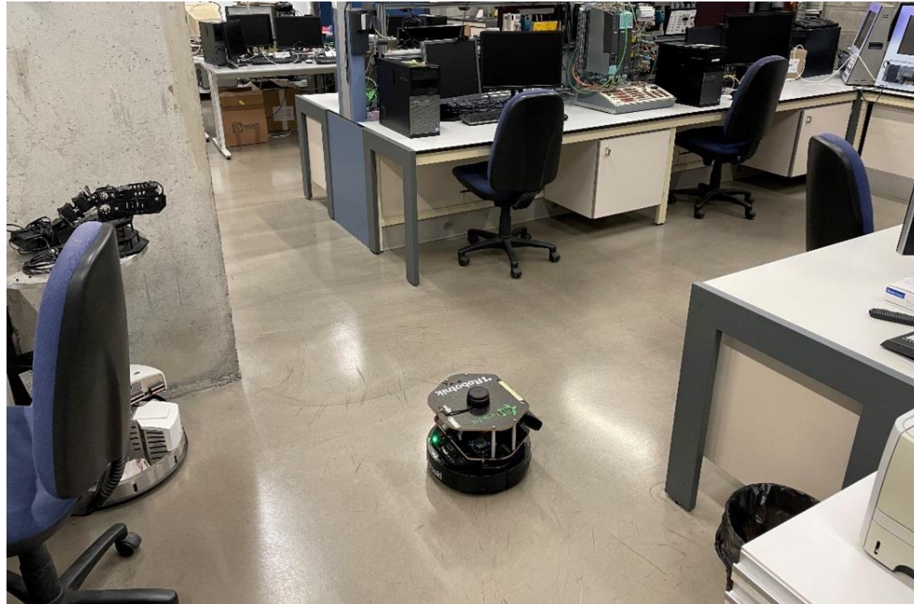
En lo referente a la navegación, se han identificado tres objetivos principales: la creación de mapas del entorno en el cual se va a operar; la navegación simple (sin la presencia de obstáculos); y la navegación completa (se introducen tanto obstáculos como funciones de percepción del entorno).

7.1.1 Prueba 1: Creación de Mapas del Entorno

Título de Prueba: Creación de Mapas del Entorno
Descripción: La prueba consiste en realizar un mapa del entorno en el que se desplaza la unidad de transporte a medida que ésta se traslada. El desplazamiento de esta unidad de transporte la comandará un operario mediante un panel de control que le permita mover el transporte según sus indicaciones y a la velocidad que precise.
Pasos: 1. Poner en marcha tanto el transporte como el Sistema de Navegación. 2. Lanzar el archivo launch turtlebot_transport_flexmansys gmapping.launch. 3. Mover el transporte por todo el entorno hasta obtener el mapa requerido. 4. Guardar el mapa en un formato entendible para el sistema de navegación.
Objetivos y Requisitos Correspondientes: P1, R1.4.
Resultados Esperados: El sistema debe de ser capaz de realizar un mapa de cualquier entorno en el que sea lanzada la petición de mapeo.
Resultados Obtenidos: Funciona correctamente, el sistema funciona y se pueden guardar mapas de diversos entornos. En este caso, se ha obtenido un mapa del laboratorio en el que opera la unidad de transporte.

Tabla 7.1 Tabla de Verificación de Prueba 1

Mediante las funciones propias del Sistema de Navegación desarrollado, como las aportadas por el ROS Navigation Stack por defecto, se puede realizar el mapa de un entorno cerrado empleando como panel de control para el movimiento una consola terminal a la cual se introducen consignas de movimiento mediante un teclado. Mediante la ayuda de la interfaz propia de ROS RViz, puede verse de manera dinámica el mapa que se va dibujando. En la *Figura 7.1* se muestra un ejemplo de la prueba en proceso.



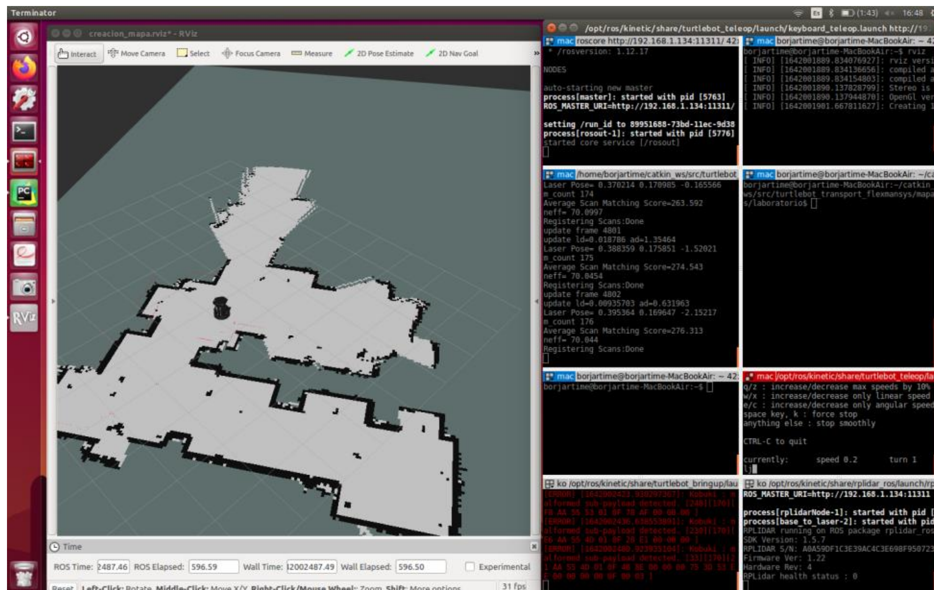


Figura 7.1 Mapeado durante la Prueba 1

El resultado de la prueba puede observarse en la *Figura 7.2* donde se muestra el mapa del laboratorio donde operarán los AGV. Este mapa será sobre el que opere todo el Sistema de Navegación para las posteriores pruebas.

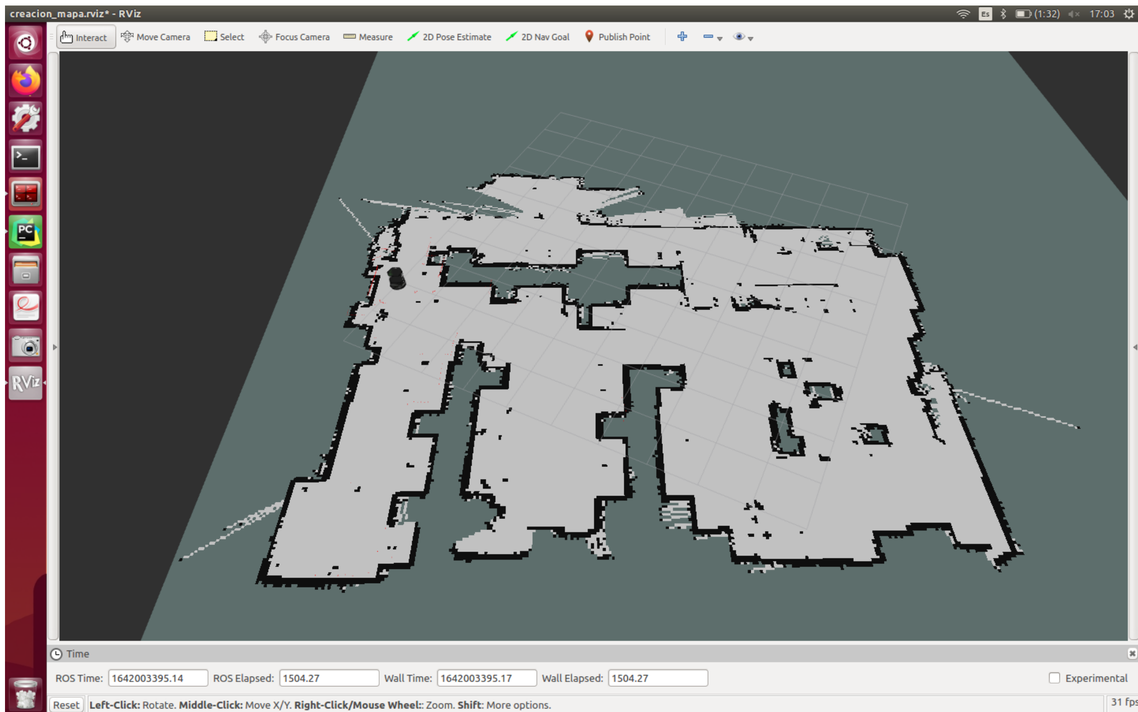


Figura 7.2 Mapa Obtenido de la Prueba 1

7.1.2 Prueba 2: Navegación Simple

Título de Prueba: Navegación Simple
Descripción: La prueba consiste en realizar un desplazamiento simple desde la estación de carga de la unidad de transporte hasta dos coordenadas en el mapa.
Pasos: 1. Poner en marcha tanto el transporte como el Sistema de Navegación. 2. Lanzar el archivo launch turtlebot_transport_flexmansys_launchfile. 3. Calibración, localización y puesta a punto del transporte. 4. Desplazamiento hasta el punto A petición por consola terminal. 5. Desplazamiento hasta el punto B a petición por consola terminal. 6. Vuelta a la estación de carga.
Objetivos y Requisitos Correspondientes: P1, S1, R1.4.
Resultados Esperados: El sistema debe de ser capaz de realizar un desplazamiento entre dichos puntos recogidos en el mapa.
Resultados Obtenidos: Funciona correctamente, el sistema de navegación es capaz de localizar al transporte en el mapa y de generar trayectorias acordes a los mapas de coste y estático generados de la Prueba 1.

Tabla 7.2 Tabla de Verificación de Prueba 2

Para ilustrar las verificaciones que se han llevado a cabo para la presente prueba, se ha empleado nuevamente RViz. Concretamente, se muestran los cuatro mapas empleados para la navegación a partir del mapa obtenido en la prueba de creación de mapa, el transporte, el sistema de coordenadas de la odometría, los puntos generados por el algoritmo de localización de Monte Carlo y las trayectorias generadas por el planificador global. Como puede observarse en la *Figura 7.3* y la *Figura 7.4*, la trayectoria desde el punto A al B se produce correctamente.

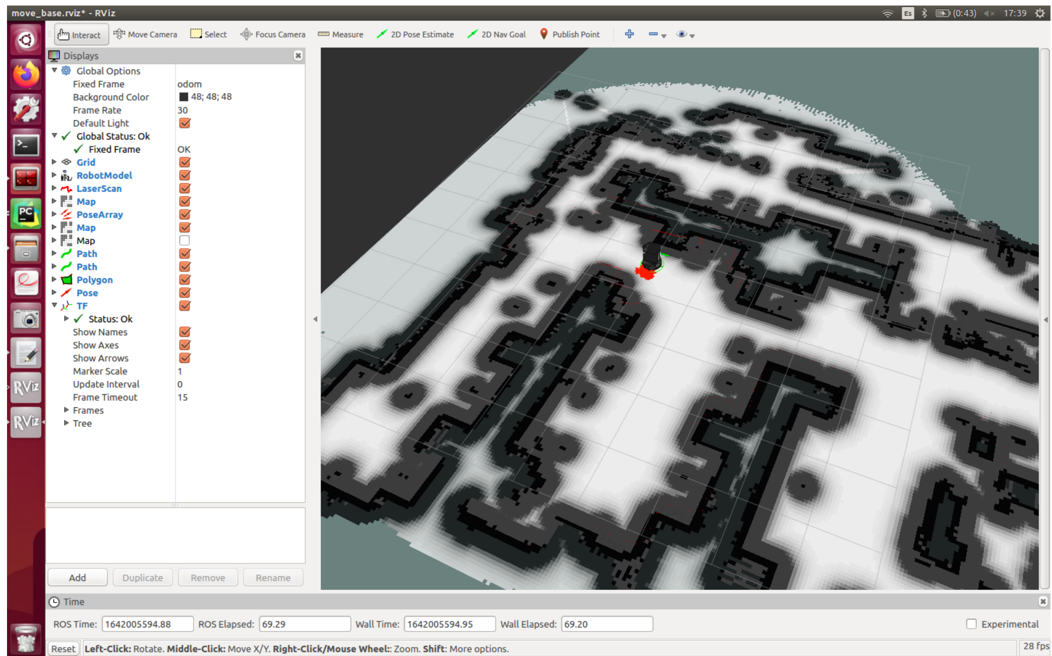


Figura 7.3 Transporte en Estación de Carga en la Prueba 2

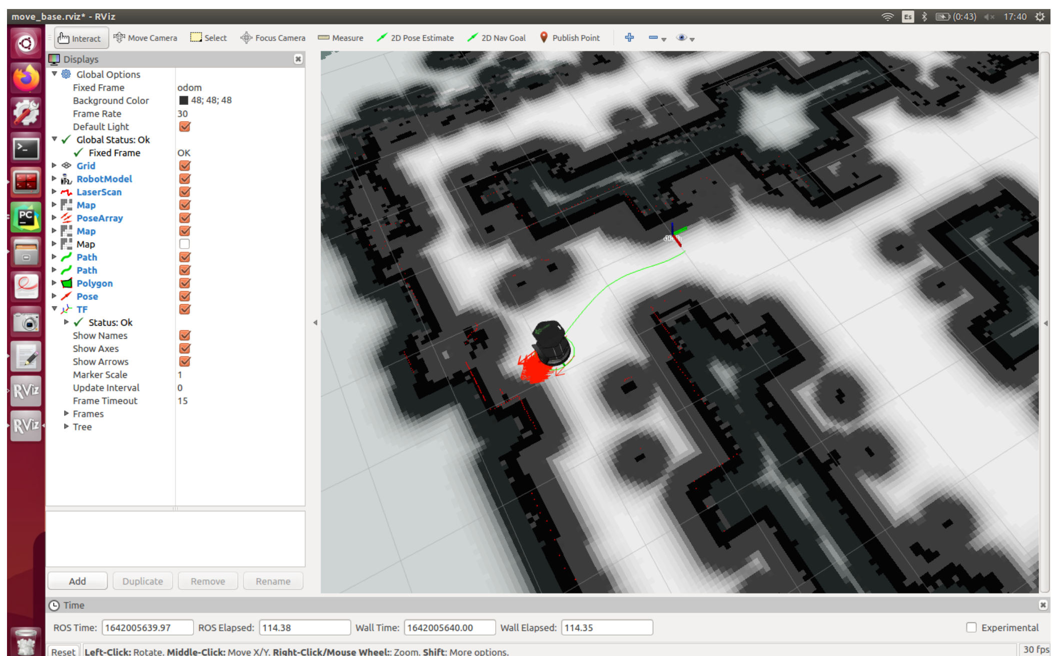


Figura 7.4 Desplazamiento del Transporte desde la Estación de Carga hasta el Punto A en la Prueba 2

Finalmente, en la *Figura 7.5* puede verse como la trayectoria desde el punto A (origen) hasta el B (estación de carga) se realizó correctamente.

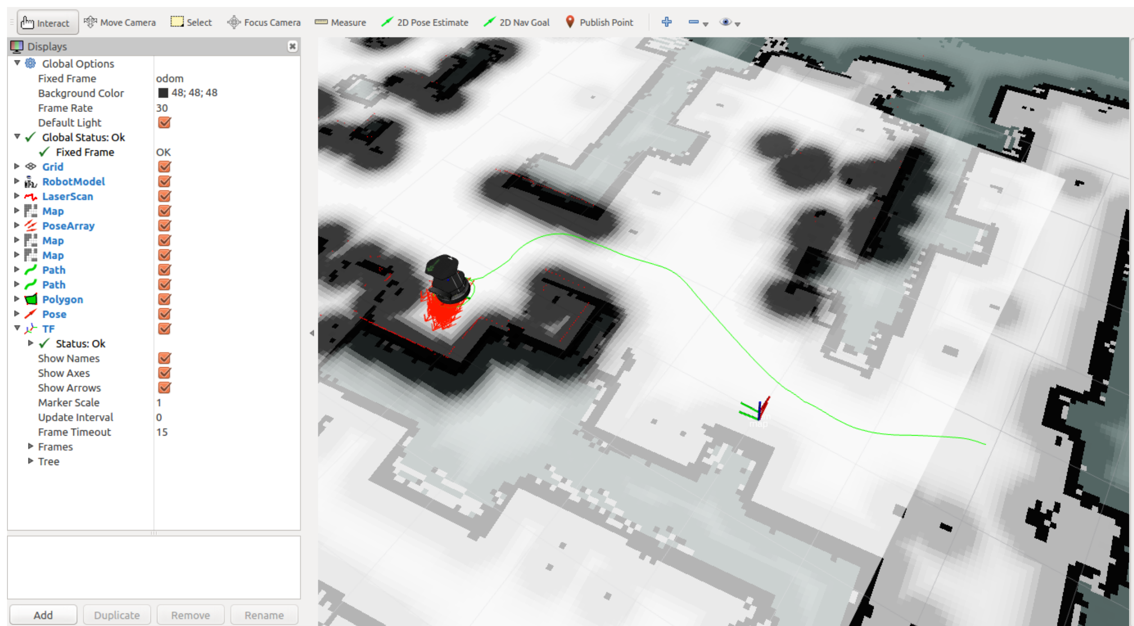


Figura 7.5 Desplazamiento del Transporte desde el Punto A hasta el punto B en la Prueba 2

Figura 7.6 Transporte en Punto A (Izquierda). Transporte en Punto B (Derecha)

En las dos siguientes figuras, puede observarse lo que recibe el Sistema de Navegación real en funcionamiento a través de la interfaz proporcionada por RViz, situada arriba a la izquierda. Por su parte, los tres terminales de la derecha muestran, de arriba abajo: la interfaz Sistema Navegación, el tópico de estado del transporte y tópico de coordenadas.

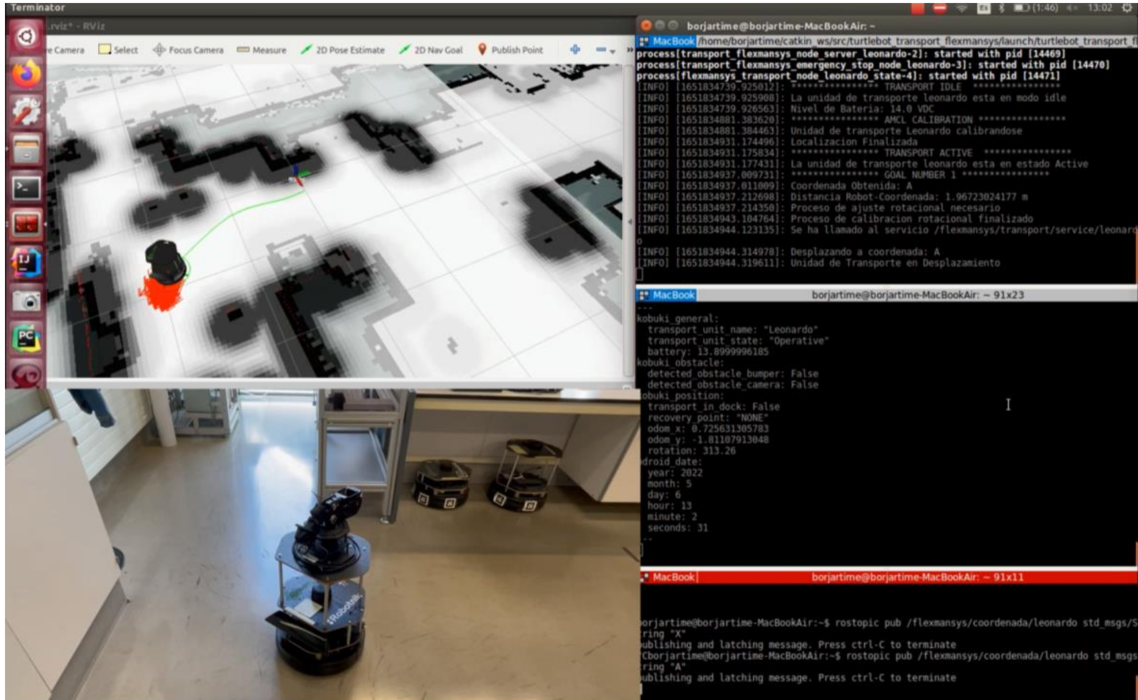


Figura 7.7 Comparativa Realidad con RViz: Trayectoria desde la Estación de Carga al Punto A

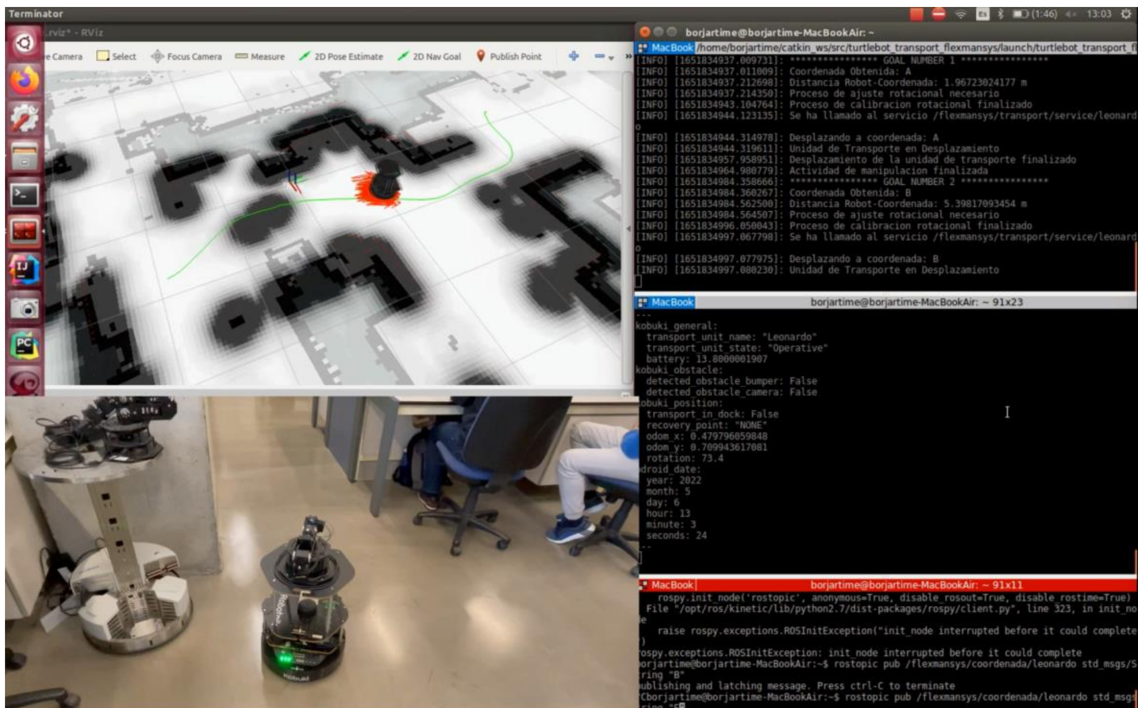


Figura 7.8Comparativa Realidad con RViz: Trayectoria desde el Punto A al Punto B

7.1.3 Prueba 3: Navegación con Obstáculos y Percepción del Entorno

Título de Prueba: Navegación con Obstáculos y Percepción del Entorno
Descripción: La prueba consiste en realizar un desplazamiento simple, pero con la presencia de un obstáculo dinámico que no esté recogido en el mapa estático del transporte. A su vez, se probará la capacidad del sistema de percepción de detectar códigos AR de identificación.
Pasos: 1. Poner en marcha tanto el transporte como el Sistema de Navegación. 2. Lanzar el archivo launch turtlebot_transport_flexmansys_launchfile 3. Lanzar el Sistema de percepción y detección de códigos AR. 4. Replicar la navegación simple de la Prueba 2 con un obstáculo añadido.
Objetivos y Requisitos Correspondientes: P1, S1 ,R1.
Resultados Esperados: El sistema debe de ser capaz de realizar un desplazamiento entre dichos puntos recogidos en el mapa y de esquivar el obstáculo, así como detectar códigos AR.
Resultados Obtenidos: Funciona correctamente, el sistema de navegación es capaz de localizar al transporte en el mapa y de generar trayectorias acordes a los mapas de coste y estático generados de la Prueba 1 y de modificar las trayectorias según los obstáculos no recogidos en el mapa. A su vez, es capaz de detectar los códigos AR.

Tabla 7.3 Tabla de Verificación de Prueba 3

La realización de esta prueba se ha dividido en dos: primero, se ha verificado que el sistema de percepción es capaz de detectar los códigos AR del entorno a través de la cámara de visión; segundo, se ha comprobado que el Sistema de Navegación es capaz de detectar obstáculos no recogidos en el mapa estático empleado para la navegación y modificar su trayectoria de manera dinámica. Para llevar a cabo ambas pruebas, se empleó nuevamente RViz.

En lo que concierne al sistema de percepción, como se ve en la *Figura 7.9,* se colocó un transporte frente a un código AR situado en el suelo. El sistema fue no solo capaz de detectar la presencia del código AR sobre el suelo, sino que lo integró en el árbol de transformadas mediante el sistema de coordenadas de nombre “ar_marker_0”, siendo el 0 el número representado en el código AR. Además, se pudo obtener la distancia que separaba al transporte del código, de manera que se pudo comprobar que los códigos AR pueden emplearse como elementos de ayuda a la identificación de objetos y navegación.

Por otro lado, para validar que el planificador de trayectorias del Sistema de Navegación es capaz de modificar y ejecutar trayectorias según lo percibido del entorno, se optó por llevar a cabo el mismo desplazamiento de la prueba anterior, es decir, entre un punto “A” y otro “B”. Sin embargo, al entorno de operación dónde residen estos puntos, es decir, la sección A del entorno de operación, se le introdujeron tres obstáculos nuevos a través de cajas de cartón. La configuración del entorno de operación para ambas trayectorias, tanto desde la estación de carga al punto A como del punto A al B, puede verse en la *Figura 7.10*.

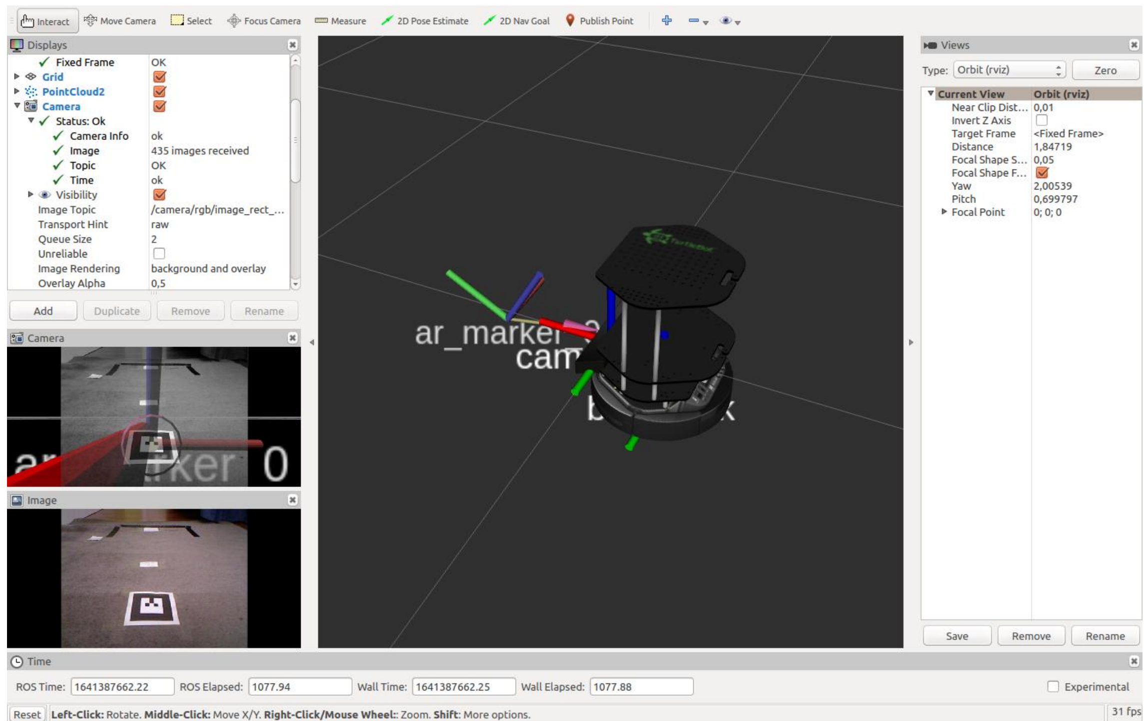


Figura 7.9 Percepción de Código AR en la Prueba 3

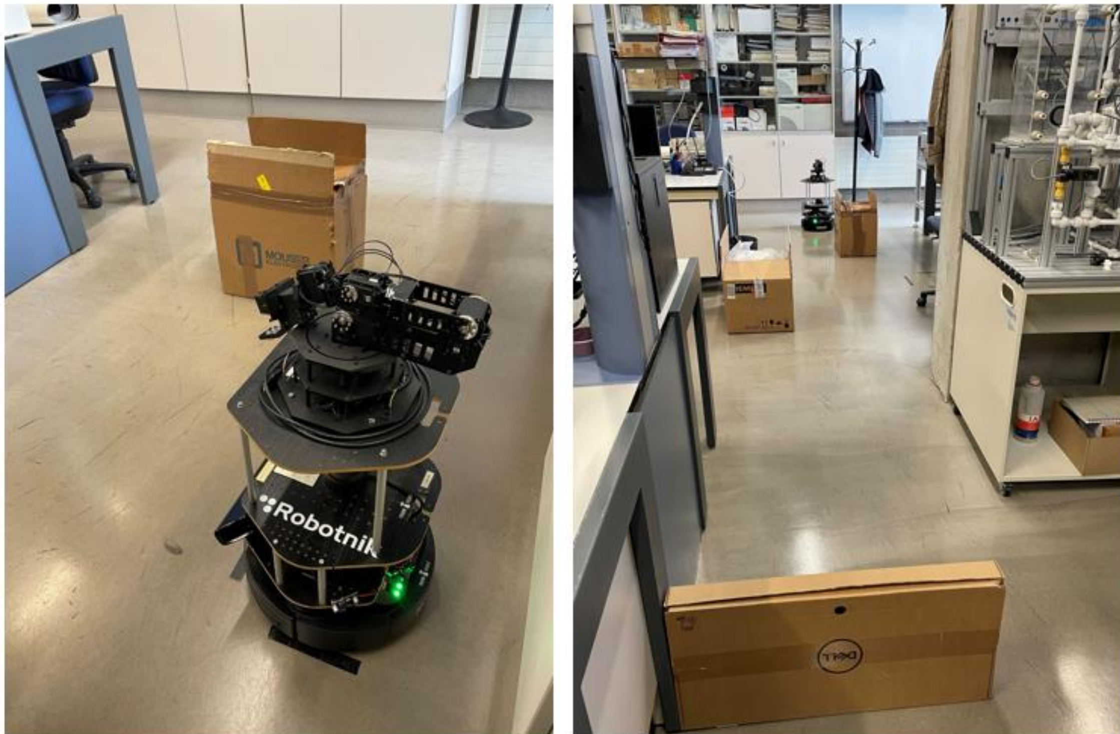


Figura 7.10 Obstáculos Trayectoria Dock al Punto A (Izquierda) y del Punto A al Punto B (Derecha)

Puede observarse las trayectorias ejecutadas en ambos desplazamientos en la *Figura 7.11* y la *Figura 7.12*. En ambas, puede comprobarse como la trayectoria planificada ha sido modificada según los obstáculos introducidos en el entorno.

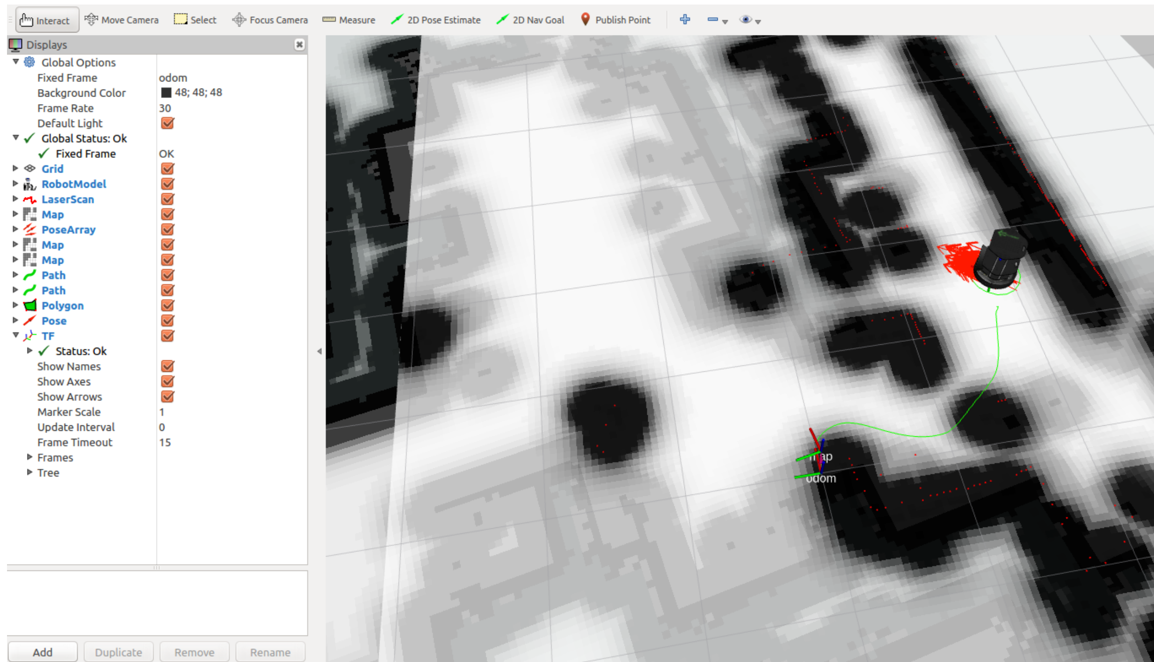


Figura 7.11 Trayectoria desde la Estación de Carga al Punto A con Obstáculos

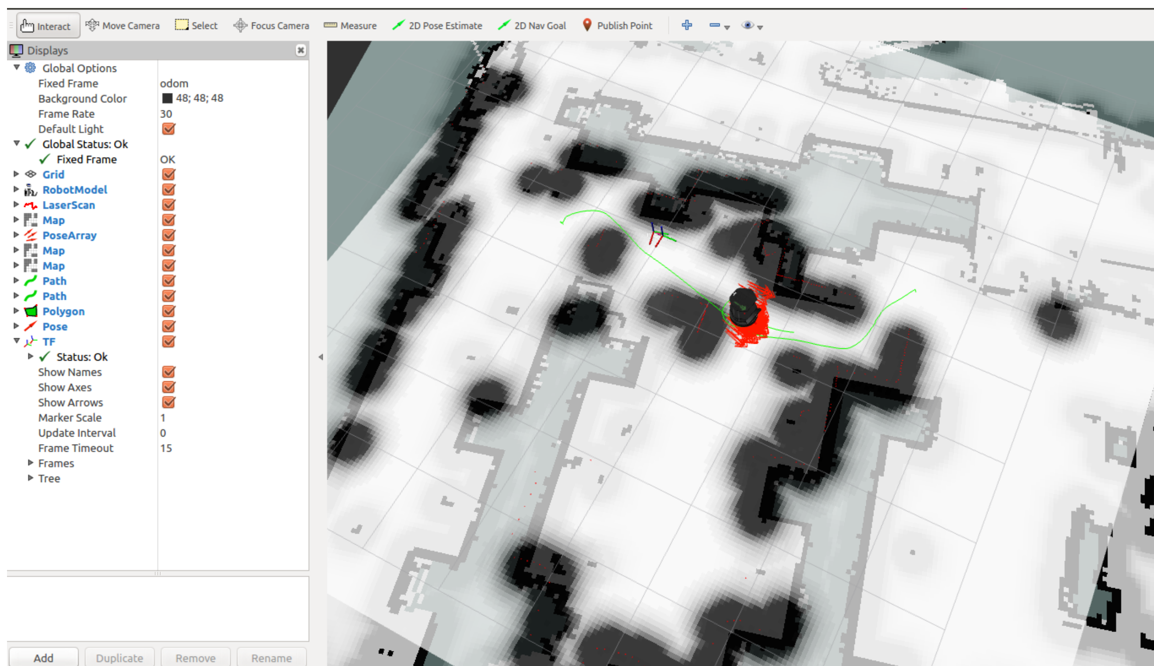


Figura 7.12 Trayectoria desde el Punto A al Punto B con Obstáculos

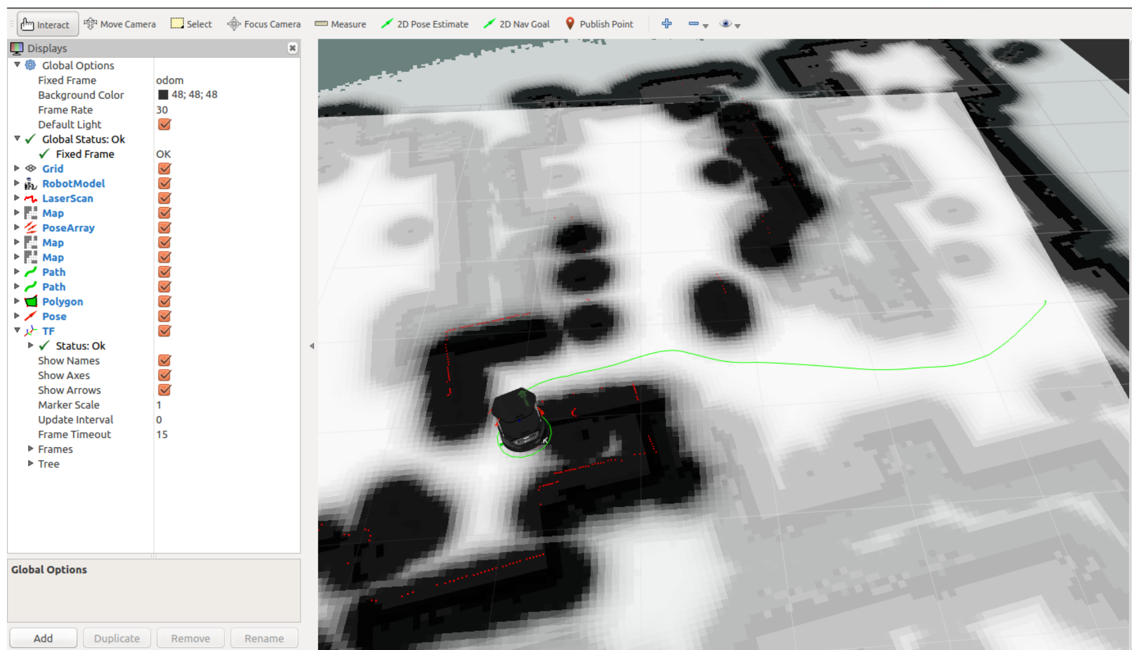


Figura 7.13 Fin de Trayectoria desde el Punto A al Punto B con Obstáculos

Finalmente, pudo comprobarse que la interfaz propia del Sistema de Navegación y los mensajes mostrados en la terminal por éste concuerdan con las tareas ejecutadas y que se realiza correctamente, como se muestra en la *Figura 7.14*

```
Mac                    borjartime@borjartime-MacBookAir: ~ 98x25
[INFO] [1651578935.624796]: **************** TRANSPORT ACTIVE  ****************
[INFO] [1651578935.627795]: La unidad de transporte leonardo esta en estado Active
[INFO] [1651578940.557313]: **************** GOAL NUMBER 1 ****************
[INFO] [1651578940.560145]: Coordenada Obtenida: A
[INFO] [1651578940.764007]: Distancia Robot-Coordenada: 1.95248089776 m
[INFO] [1651578940.766186]: Proceso de ajuste rotacional necesario
[INFO] [1651578946.369360]: Proceso de calibracion rotacional finalizado
[INFO] [1651578947.388390]: Se ha llamado al servicio /flexmansys/transport/service/leonardo
[INFO] [1651578947.725782]: Desplazando a coordenada: A
[INFO] [1651578947.728817]: Unidad de Transporte en Desplazamiento
[INFO] [1651578964.538580]: Desplazamiento de la unidad de transporte finalizado
[INFO] [1651579018.080336]: Actividad de manipulacion finalizada
[INFO] [1651579019.386316]: **************** GOAL NUMBER 2 ****************
[INFO] [1651579019.387482]: Coordenada Obtenida: B
[INFO] [1651579019.589037]: Distancia Robot-Coordenada: 5.4267757247 m
[INFO] [1651579019.589955]: Proceso de ajuste rotacional necesario
[INFO] [1651579030.752336]: Proceso de calibracion rotacional finalizado
[INFO] [1651579031.761325]: Se ha llamado al servicio /flexmansys/transport/service/leonardo
[INFO] [1651579031.768886]: Desplazando a coordenada: B
[INFO] [1651579031.770430]: Unidad de Transporte en Desplazamiento
[INFO] [1651579068.584598]: Desplazamiento de la unidad de transporte finalizado
[INFO] [1651579068.585362]: Actividad de manipulacion finalizada
[INFO] [1651579092.460696]: **************** GOAL NUMBER 3 ****************
[INFO] [1651579092.461338]: Coordenada Obtenida: A
[INFO] [1651579092.662581]: Distancia Robot-Coordenada: 5.37986179584 m
```

Figura 7.14 Interfaz Mostrada por el Sistema de Navegación

7.2 Negociación

En lo que concierne a la negociación, se entiende a negociación como el mecanismo que envuelve la integración del entorno de ROS y sus transportes en el sistema MAS implementado sobre la plataforma JADE. Para ello, se han definido dos pasos o pruebas: Integración del Sistema de Navegación en el MAS, verificando que los agentes y unidades de transporte pueden comunicarse entre sí; y la Negociación entre Agentes de Transporte, verificando que siempre sale ganador el agente transporte mejor calificado.

7.2.1 Prueba 4: Integración del Sistema de Navegación en el MAS

Título de Prueba: Integración del Sistema de Navegación en el MAS
Descripción: La prueba consiste en la integración de todos los componentes necesarios del Sistema de Navegación en el MAS, de manera que éste sea capaz de comunicarse con el Sistema de Navegación.
Pasos: 1. Poner en marcha el MAS. 2. Poner en marcha el Sistema de Navegación. 3. Poner en marcha el Sistema Gateway. 4. Comunicación desde el Sistema de Navegación hasta el MAS mediante la información publicada en el tópico de estado del transporte. 5. Comunicación desde el MAS hasta el Sistema de Navegación con un plan de transporte o lista de coordenadas que incluyan los puntos A2 y C4.
Objetivos y Requisitos Correspondientes: P2, R2.1.
Resultados Esperados: Se pueden lanzar varios agentes transporte según las unidades de transporte que se quieran integrar en el MAS. Además, éstos pueden leer la información de estado que los AGV publican en su tópico de estado (/flexmansys/state/TUN) y de escribir comandos de coordenadas en el tópico de coordenadas (/flexmansys/coordenada/TUN).
Resultados Obtenidos: Funciona correctamente: el MAS lanza correctamente varios agentes transporte según los argumentos de entrada introducidos y éstos son capaces de comunicarse con las capas más bajas de la arquitectura, es decir, con el Sistema de Navegación. Se verifica que el Sistema Gateway ejecuta correctamente su trabajo y que convierte de manera idónea la información desde el entorno de ROS al entorno de Java y viceversa sin perderse datos por el camino.

Tabla 7.4 Tabla de Verificación de Prueba 4

Para verificar que el MAS se comunica de manera correcta con el Sistema de Navegación y que es capaz de transferirle un plan de coordenadas, se ha abstraído este último empleando una terminal que publicase un mensaje ROS cíclico que replicase el mensaje de estado de una unidad de transporte. Dicho mensaje replicado, se muestra en la *Figura 7.15*.

```
borjartime@borjartime-MacBookAir: ~ 98x13
^Cborjartime@borjartime-MacBookAir:~$ rostopic pub /flexmansys/state/leonardo turtlebot_transport_
exmansys/TransportUnitState "kobuki_general:
  transport_unit_name: Leonardo
  transport_unit_state: Active
  battery: 15.0
kobuki_obstacle:
  detected_obstacle_bumper: false
  detected_obstacle_camera: false
kobuki_position: {transport_in_dock: false, recovery_point: '', odom_x: 12.0, odom_y: 10.0,
  rotation: 0.0}
odroid_date: {year: 0, month: 0, day: 0, hour: 14, minute: 20, seconds: 40}"
publishing and latching message. Press ctrl-C to terminate
```

Figura 7.15 Mensaje de Estado de Unidad de Transporte Replicado

Para avanzar en la secuencia de funcionamiento del agente transporte, se ha ido modificando el estado de la unidad de transporte, concretamente, desde Idle hasta Operative, pasando por Localization y Active. Como puede verse en la *Figura 7.16*, los datos introducidos en dicho mensaje ROS para suplir al Sistema de Navegación, llegan hasta el agente transporte.

```
Run:  GUI ×   SMA ×   TAgent1 ×
Leonardo Transport isn't active
cmd=set null battery=15.0
16:51:05.647 [transport1] INFO  MWAgent sendCommand 240 - mm(set null battery=15.0) > element not found
cmd=set null currentPos_X=10.0
16:51:05.653 [transport1] INFO  MWAgent sendCommand 240 - mm(set null currentPos_X=10.0) > element not found
cmd=set null currentPos_Y=10.0
16:51:05.661 [transport1] INFO  MWAgent sendCommand 240 - mm(set null currentPos_Y=10.0) > element not found
Leonardo Transport started a new task at 14:20:40 h
Leonardo Transport is active
cmd=set null battery=15.0
16:51:09.653 [transport1] INFO  MWAgent sendCommand 240 - mm(set null battery=15.0) > element not found
cmd=set null currentPos_X=10.0
16:51:09.662 [transport1] INFO  MWAgent sendCommand 240 - mm(set null currentPos_X=10.0) > element not found
cmd=set null currentPos_Y=10.0
16:51:09.671 [transport1] INFO  MWAgent sendCommand 240 - mm(set null currentPos_Y=10.0) > element not found
Leonardo Transport ended his task at 14:20:40 h
Number of Accomplished Tasks: 2
```

Figura 7.16 Interfaz Mostrada por el Agente Transporte

Por otro lado, para verificar el funcionamiento del agente transporte y su comunicación inversa, es decir, desde el MAS hasta el Sistema de navegación, se optó por leer el tópico de coordenadas en un terminal aparte, el cual se muestra en la *Figura 7.17*. En dicho terminal, pudo comprobarse cómo en función del estado en el que se encuentra la unidad de transporte, manda unas tareas u otras. Primeramente, requiriendo la calibración del transporte al encontrarse en Idle, y posteriormente, solicitando el desplazamiento hacia ciertos puntos del mapa y al finalizar mandarle a su estación de carga. Del mismo modo, es capaz de enviar una lista de tareas, respetando y esperando a que el transporte se esté en estado activo.

Esta comunicación full-duplex o bidireccional puede comprobarse mediante lo mostrado por el agente Gateway en su propia interfaz, el cual muestra tanto los datos recibidos por el tópico de estado de la unidad de transporte que posteriormente mandará al agente transporte, y los datos que recibe desde el agente transporte a la unidad de transporte. Información mostrada en la *Figura 7.18* y que coincide con las anteriores tres figuras.

```
borjartime@borjartime-MacBookAir: ~ 98x18
^Cborjartime@borjartime-MacBookAir:~$ rostopic echo /flexmansys/coordenada/leonardo
data: "DOCK"
---
data: "X"
---
data: "X"
---
data: "X"
---
data: "A2"
---
data: "C4"
---
data: "DOCK"
---
data: "DOCK"
---
```

Figura 7.17 Datos Publicados en el Tópico de Coordenadas

```
borjartime@borjartime-MacBookAir: ~/IdeaProjects/FlexManSys 98x52
INFO:
*************** GW AGENT ***************
Nombre AGV: Leonardo
Estado AGV: Active
Bateria AGV: 15.0
---------------------------------------------------
Obstaculo bumper: false
Obstaculo camera: false
---------------------------------------------------
AGV en Dock: false
Recovery Point:
Odom X: 12.0
Odom Y: 10.0
Rotation: 0.0
---------------------------------------------------
Year: 0
Month: 0
Day: 0
Hour: 14
Minute: 20
Seconds: 40

May 04, 2022 5:04:46 PM jade.wrapper.gateway.GatewayBehaviour action
INFO: ControlGatewayContT_01 started execution of command com.github.rosjava.fms_transp.turtlebot2
.StructCommand@f3e2f54
May 04, 2022 5:04:46 PM jade.wrapper.gateway.GatewayBehaviour releaseCommand
INFO: ControlGatewayContT_01 terminated execution of command com.github.rosjava.fms_transp.turtleb
ot2.StructCommand@f3e2f54
May 04, 2022 5:04:46 PM jade.wrapper.gateway.DynamicJadeGateway execute
INFO: Requesting execution of command com.github.rosjava.fms_transp.turtlebot2.StructCommand@72bed
408
May 04, 2022 5:04:46 PM jade.wrapper.gateway.GatewayBehaviour action
INFO: ControlGatewayContT_01 started execution of command com.github.rosjava.fms_transp.turtlebot2
.StructCommand@72bed408
May 04, 2022 5:04:46 PM jade.wrapper.gateway.GatewayBehaviour releaseCommand
INFO: ControlGatewayContT_01 terminated execution of command com.github.rosjava.fms_transp.turtleb
ot2.StructCommand@72bed408
Bloqueando el GWagentROS hasta recibir otro mensaje de TransportAgent
Se ha recibido un mensaje desde el TransportAgent con Ontologia: PlanCoordenadas
2
Se ha recibido un mensaje desde el TransportAgent con Ontologia: PlanCoordenadas
++++++++++++++++++++++++++++++++++++++++++++++++
Se ha recibido la siguiente coordenada: A2
++++++++++++++++++++++++++++++++++++++++++++++++
Se ha recibido un mensaje desde el TransportAgent con Ontologia: PlanCoordenadas
++++++++++++++++++++++++++++++++++++++++++++++++
Se ha recibido la siguiente coordenada: C4
++++++++++++++++++++++++++++++++++++++++++++++++
```

Figura 7.18 Datos Recibidos por el Agente Gateway tanto desde el MAS como desde el Sistema de Navegación

Finalmente, para mayor redundancia, se comprueba en la GUI de JADE la presencia de todos los agentes necesarios y los nombres que ha tomado cada uno en la plataforma, verificando que son correctos en la *Figura 7.19*. También, en la *Figura 7.20*, se puede comprobar como la performativa de los mensajes intercambiados entre el agente transporte y el Gateway corresponden entre sí.

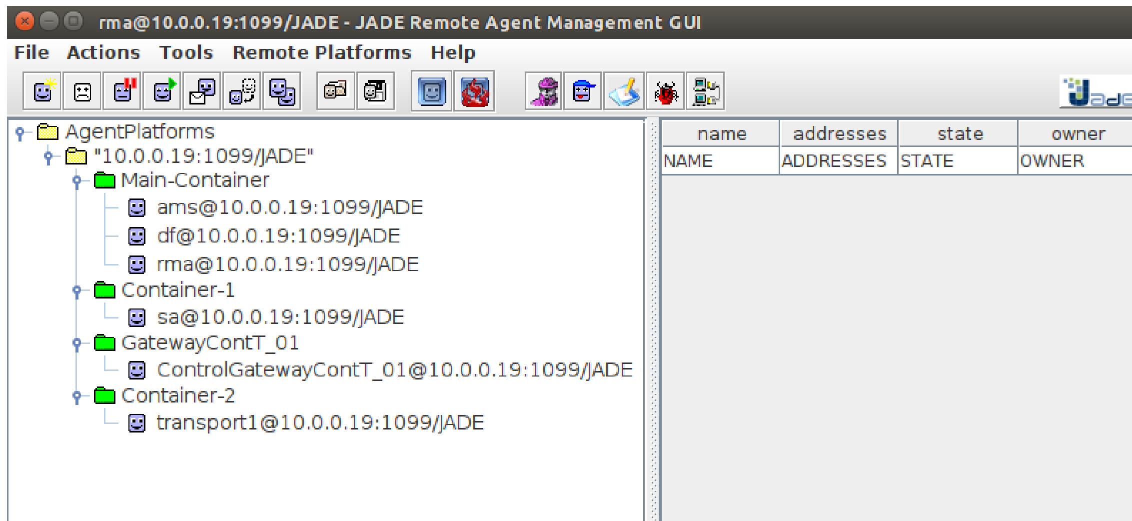


Figura 7.19 GUI de JADE con el Sistema de Agentes al Completo

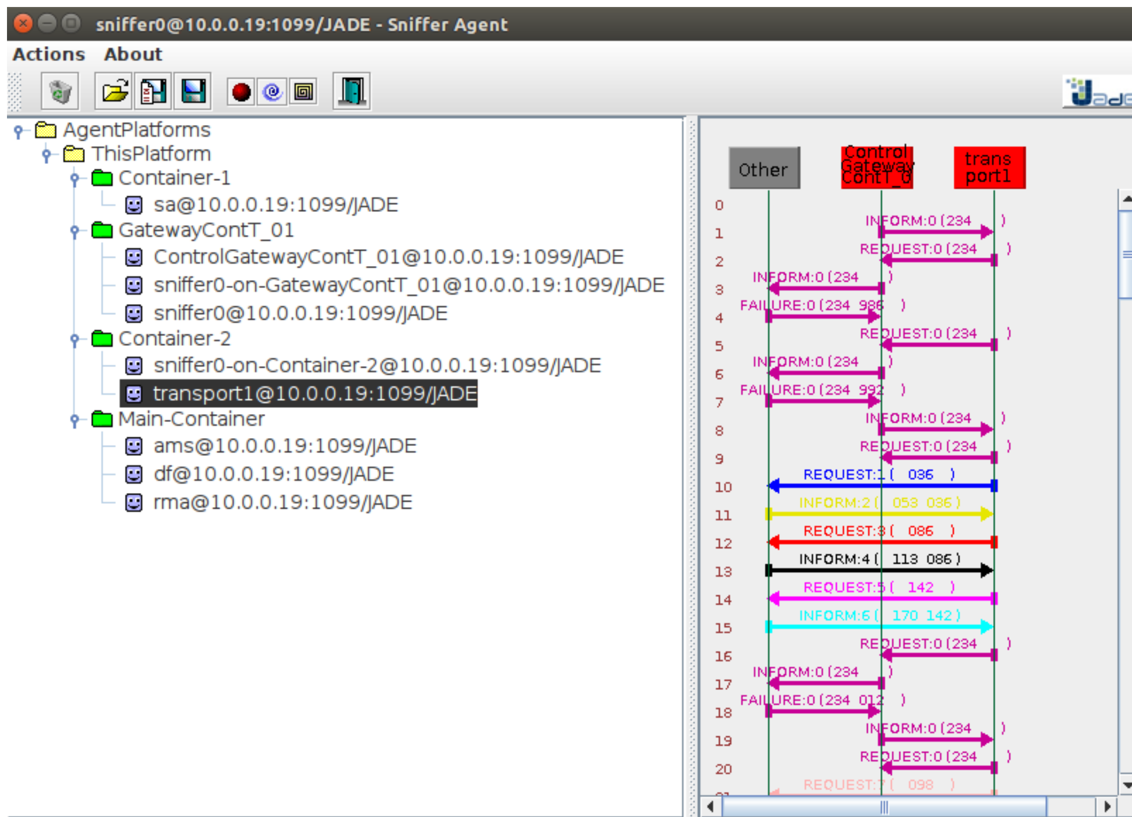


Figura 7.20 GUI de JADE con el Sniffer de Mensajes entre el Agente Gateway y el Agente Transporte

7.2.2 Prueba 5: Negociación entre Agentes de Transporte

Título de Prueba: Negociación entre Agentes de Transporte
Descripción: La prueba consiste en el lanzamiento de múltiples agentes de transporte dentro del MAS, cada uno con las características de una unidad de transporte independiente, de manera que cada uno de los agentes de transporte negocie con el resto para determinar quién de sus representados será el ganador y será el que lleve a cabo la lista de tareas designada.
Pasos: 1. Poner en marcha tanto el transporte como el Sistema de Navegación. 2. Dar de alta a varios agentes transporte y transportes en el MAS. 3. Realizar de la negociación entre agentes de transporte 4. Establecer de un ganador de la negociación. 5. Comunicar con el transporte y verificar que recibe la lista de tareas.
Objetivos y Requisitos Correspondientes: P2, P3, R2.
Resultados Esperados: Todos los agentes transporte en el MAS deben de ser capaces de comunicarse con su agente transporte y recoger toda su información de estado. Posteriormente, deberán de ser capaces de negociar para realizar tareas de transporte y de ponerse de acuerdo entre ellos para determinar a un único agente transporte ganador de la negociación.
Resultados Obtenidos: Funciona correctamente, el MAS lanza correctamente varios agentes transporte según los argumentos de entrada introducidos y es capaz de hacer negociar a múltiples agentes de transporte de manera que determinen cuál de ellos es el óptimo para realizar la tarea encomendada.

Tabla 7.5 Tabla de Verificación de Prueba 5

La realización de esta prueba está relacionada con la anterior, de manera que se solicita la misma lista de tareas o plan de transporte para la comunicación entre el agente transporte y el Sistema de Navegación, el cual, igualmente, se encuentra abstraído a través de una terminal que replica el mensaje de estado de la unidad de transporte T_01. Cuenta con la principal diferencia de que, en este caso, el plan de transporte no se extrae directamente del fichero de TransportPlan1.xml, sino que es construido a través de un mensaje que solicita la realización de un plan de transporte. Dicho mensaje, tiene contenidos los puntos por los que debe de pasar el transporte, concretamente en el campo “externalData”, correspondiendo a “dockingStation” y “kukaInput”, de la *Figura 7.21*. Estas dos coordenadas corresponden al plan de transporte que se construirá en el fichero TransportPlan1.xml y que posteriormente se mandará a la unidad de transporte, los cuales coinciden en el fichero de KeyPositions.xml con las dos coordenadas de la prueba anterior: A2 y C4.

Por otro lado, esta solicitud de servicio de transporte se hace desde el propio agente Gateway, el cual se lanzará a su agente de transporte respectivo para que negocie según el nivel de batería suyo y del resto de unidades de transporte. El resultado de la negociación, en este caso, es el correcto, generando el mensaje de la *Figura 7.22,* el cual puede verse en la interfaz del propio agente transporte y que verifica el correcto comportamiento de un agente ganador en la negociación entre dos agentes transporte ejecutándose en paralelo. Este segundo agente transporte

se lanza de la misma forma que el primero en la prueba anterior, con la excepción de que su nivel de batería será de 13.0 VDC en vez de 15 VDC, de forma que pierda la negociación.

```
// ************************* PRUEBA NEGOCIACION *************************
if (neg_requested == false) {
    ACLMessage msg_neg = new ACLMessage(ACLMessage.CFP);
    msg_neg.addReceiver(TransportAgent);
    msg_neg.setOntology("negotiation");
    msg_neg.setConversationId("1234");

    String targets = TransportAgentAID; //el ID que tenga el agente.
    String negotiationCriteria = "battery"; //o battery o el que utilices.
    String negAction = "supplyConsumables";
    String externalData = "dockingStation;kukaInput,machine1";
    //las posiciones separadas por ; y los campos del externalData separados entre si por ,

    msg_neg.setContent("negotiate " + targets + " criterion=" + negotiationCriteria +
            " action=" + negAction + " externaldata=" + externalData);
    send(msg_neg);
    neg_requested = true;
}
```

Figura 7.21 Mensaje de Solicitud de Negociación

```
Run:  GUI  SMA  TAgent1
16:50:25.527 [transport1] INFO  NegotiatingBehaviour action  188 - message 1234=negotiate transport1 criterion=battery action=supplyConsumables externaldata=doc
externaldata=dockingStation;kukaInput,machine1
16:50:25.530 [transport1] INFO  Transport_Functionality checkNegotiation 374 - negotiation(id:1234) partial winner transport1(value:1)
16:50:25.530 [transport1] INFO  Transport_Functionality checkNegotiation 383 - ejecutar supplyConsumables
16:50:25.531 [transport1] INFO  Transport_Functionality checkNegotiation 387 - id=
WON!
16:50:25.536 [transport1] DEBUG NegotiatingBehaviour initNegotiation 737 - Targets lenght: 1
16:50:25.536 [transport1] DEBUG NegotiatingBehaviour initNegotiation 751 - Sent Negotiation Propose msg
16:50:25.587 [transport1] INFO  ControlBehaviour action  101 - ****************ACLMessage.FAILURE (control):( (action ( agent-identifier :name transport1@10.0.0
16:50:25.587 [transport1] INFO  ControlBehaviour action  104 - msg.getPerformative()==ACLMessage.FAILURE (sender=machine1)
cmd=report (getcmp machine1) type=notFound cmpins=machine1
16:50:25.600 [transport1] INFO  MWAgent sendCommand 240 - mm(report (getcmp machine1) type=notFound cmpins=machine1) > cmd not found:report
16:50:25.603 [transport1] INFO  ControlBehaviour action  115 - (INFORM
 :sender  ( agent-identifier :name sa@10.0.0.19:1099/JADE  :addresses (sequence http://borjartime-MacBookAir:7778/acc ))
 :receiver  (set ( agent-identifier :name transport1@10.0.0.19:1099/JADE  :addresses (sequence http://borjartime-MacBookAir:7778/acc )) )
 :content  "cmd not found:report"
 :reply-with  transport1@10.0.0.19:1099/JADE1651675825598  :in-reply-to  "report (getcmp machine1) type=notFound cmpins=machine1_1651675825588"  :ontology  con
```

Figura 7.22 Mensajes de la Negociación del Agente Transporte

7.3 Prueba de Concepto General

La prueba de concepto general tiene como objeto la verificación de todo el sistema en conjunto, es decir, la navegación y gestión autónoma de un transporte mediante un agente transporte integrado en el MAS. Para ello, se ha planteado una única prueba: Navegación Simple a Petición de Agente Transporte, donde se darán de alta varios sistemas de transporte con sus respectivos agentes transportes, los cuales negociarán entre sí para que el AGV ganador de la negociación lleve a cabo el plan de transporte designado por el MAS.

7.3.1 Prueba 6: Navegación Simple a Petición de Agente Máquina

Título de Prueba: Navegación Simple a Petición de Agente Máquina
Descripción: La prueba consiste en solicitar desde un agente de máquina dentro del MAS, es decir, un agente que representa a una célula de la planta donde residen las unidades de transporte, un servicio de transporte determinado. Para ello, este agente máquina solicitará a los agentes transporte de la plataforma el servicio indicando tanto el plan de transporte que debe de realizar el AGV y el parámetro que han de tener los agentes transporte para la negociación. Por otro lado, se notificará en esta petición de transporte el agente que solicita el servicio, además deaquel que será el que recibirá el servicio,es decir, se definirán un receiver y un requester. El agente que recibirá el servicio deberá de iniciar su actividad una vez se le indique desde el agente transporte la finalización del transporte solicitado.
Pasos: 1. Poner en marcha tanto el transporte como el Sistema de Navegación. 2. Dar de alta al transporte en el MAS y asignarle un agente transporte. 3. Dar de alta a la célula en el MAS y asignarle un agente máquina. 4. Solicitar un servicio de transporte desde el SMA. 5. Realización de la negociación entre agentes de transporte. 6. Comunicación entre agente transporte y AGV 7. Desplazamiento AGV según plan de tareas del agente transporte 8. Notificación al MAS de finalización de tareas. 9. Ejecución de las tareas correspondientes al agente máquina.
Objetivos y Requisitos Correspondientes: P1, P2, P3, S1, S2, R1, R2, R3.
Resultados Esperados: Al igual que en pruebas anteriores, la unidad de transporte seleccionada ha de poder navegar de manera autónoma y alcanzar todos los puntos y coordenadas del plan de tareas que el agente transporte le indique. Del mismo modo, el MAS deberá de ser capaz de trabajar junto con el sistema de navegación y de comunicarse con él. La unidad de transporte deberá de ejecutar el plan de transporte definido en el fichero de plan de transporte correspondiente a su agente. Este agente, iniciará su actividad en la planta una vez el transporte ganador de la negociación termine el servicio de transporte.
Resultados Obtenidos: Funciona correctamente. El MAS se comunica correctamente con el Sistema de Navegación, de manera que la unidad de transporte seleccionada ejecuta de manera correcta y sin la intervención humana un listado de tareas proporcionado por su agente de transporte. Por otro lado, tanto los agentes de transporte como lo agentes máquina se comunican entre ellos de manera satisfactoria. Se considera la solución desarrollada óptima para la problemática planteada en el presente trabajo.

Tabla 7.6 Tabla de Verificación de Prueba 6

Para llevar a cabo esta prueba, era necesario realizar un acondicionamiento previo, puesto que la GUI de la plataforma JADE y la agente máquina se iban a ejecutar en equipos distintos al empleado hasta ahora, es decir, el equipo maestro que dentro de la red Wi-Fi generada por la estación Airport Extreme. Dichos equipos están constituidos por una serie de torres PC, como las mostradas en la *Figura 7.23,* en las cuales se ejecutarán los agentes del MAS. Lo relativo a ROS, se seguirá ejecutando en el equipo maestro anterior, el cual se integrará a la misma red Wi-Fi.

Figura 7.23 Equipos Empleados para la Prueba

La agente máquina para esta prueba será la representante de la célula del robot KUKA, el cual cuenta con un gemelo virtual como se muestra en la *Figura 7.24*.

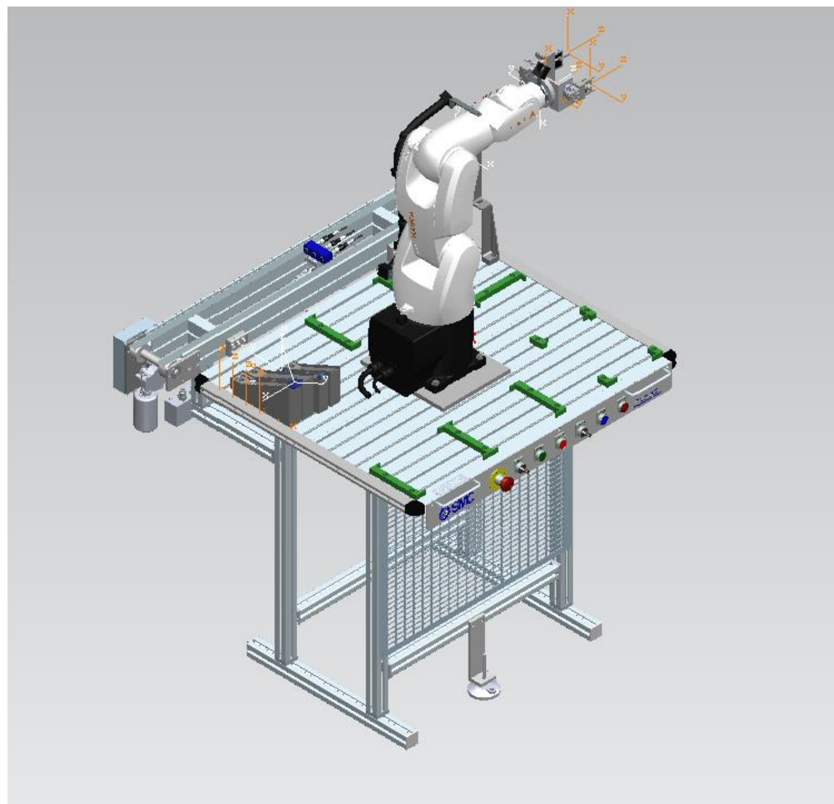


Figura 7.24 Célula Virtual Robot KUKA para Agente Máquina

Al emplearse un gemelo virtual para la prueba, la célula correspondiente se puede situar sobre cualquier punto del entorno de operación, es decir, de la planta. Por ello, para la prueba, se decidió el asignar el punto B de las anteriores pruebas de navegación para situar la célula virtual.

De esta manera, una vez el transporte realizase el servicio de transporte y ejecutase el plan de navegación situándose sobre el punto B, se comunicaría desde el agente transporte al agente máquina que recibe el servicio de transporte la finalización de la manipulación y traslado de material. Tras ello, la célula virtual del robot KUKA comienza a operar según su plan de operación interno. En la *Figura 7.25* se muestra a dicho agente máquina interactuar con la unidad de transporte y el gemelo virtual de la célula KUKA.

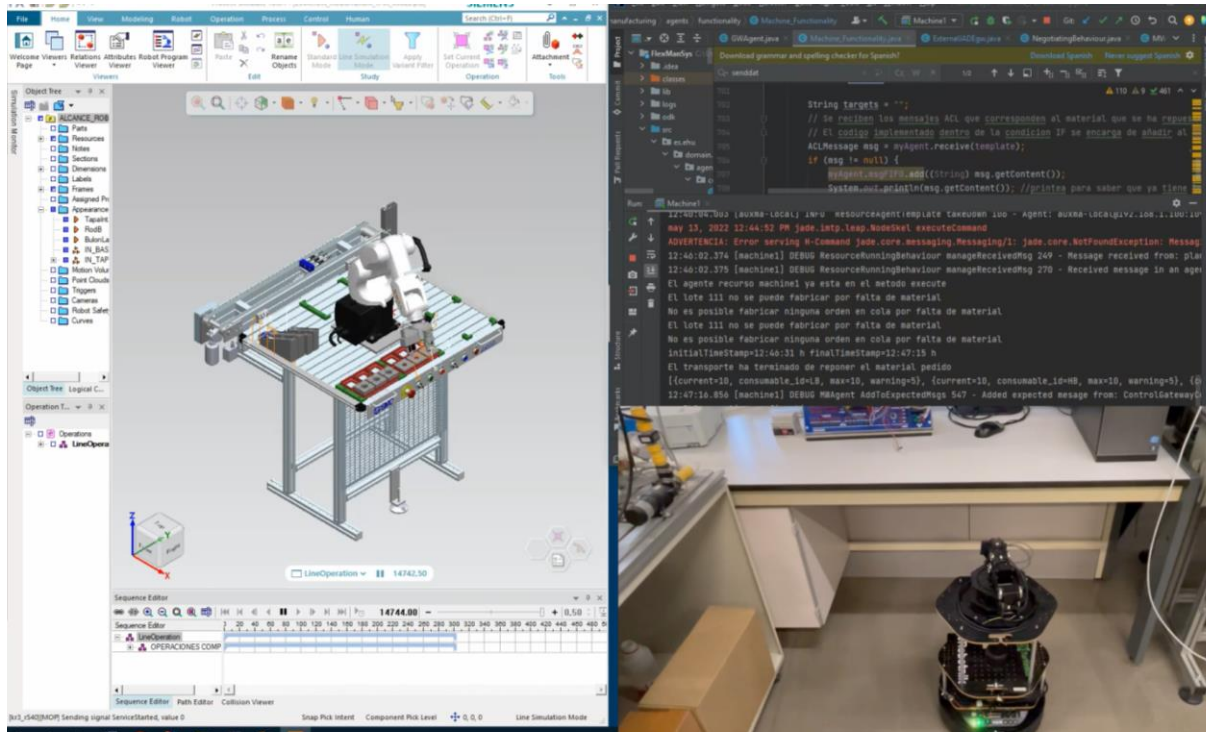


Figura 7.25 Interacción Transporte con Agente Máquina y Gemelo Virtual

Finalmente, para verificar que la estación Airport contaba con suficiente ancho de banda como para soportar varios agentes agente transporte, se empleó un medidor de ancho de banda durante la operación de transporte de un único AGV. El mayor pico mostrado fue de en torno a 261 kB/s, una cantidad considerablemente inferior a los 300 MB/s que aporta la estación y que cuenta con margen para integrar a los otros tres transportes.

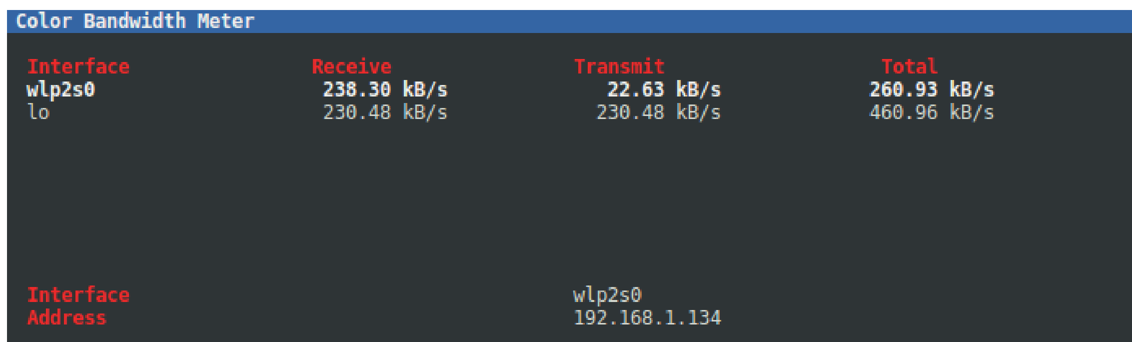


Figura 7.26 Ancho de Banda Consumido por el Sistema de Navegación

7.4 Sistemas de Seguridad

En lo referente a los diferentes sistemas de seguridad que integran toda la estructura del presente trabajo, muchos de ellos se recogen dentro del MAS, estando estos desarrollados y verificados fuera del marco del proyecto. Es por ello, que en lo que concierne al trabajo desarrollado, será necesario únicamente verificar el funcionamiento de la parada de seguridad del Sistema de Navegación, es decir, que el sistema es capaz de entrar en el estado de "Error" durante su operación normal.

7.4.1 Prueba 7: Parada de Emergencia

Título de Prueba: Parada de Emergencia
Descripción: La prueba consiste en solicitar la parada de emergencia del transporte durante su operación normal.
Pasos: 1. Poner en marcha tanto el transporte como el Sistema de Navegación. 2. Solicitar la realización de una trayectoria simple. 3. Solicitar la parada de emergencia 4. Verificar que el sistema es incapaz de desplazarse tras entrar en el estado "Error".
Objetivos y Requisitos Correspondientes: -
Resultados Esperados: El sistema debe de ser capaz de realizar una parada de emergencia sea cual sea la actividad que esté realizándose.
Resultados Obtenidos: Funciona correctamente, el sistema de navegación es capaz de llevar a cabo una parada de emergencia inmediata desde el momento en el que se solicita y de entrar al estado de "Error", deteniéndose por completo. A su vez, los indicadores de error se ejecutan de manera óptima haciendo saber que la unidad de transporte ha entrado en estado de "Error".

Tabla 7.7 Tabla de Verificación de Prueba 7

Para comprobar el funcionamiento de la parada de emergencia de las unidades de transporte, se ha simulado la ejecución de una trayectoria simple en la cual una persona operaria solicitaría la parada de emergencia publicando en el tópico de coordenadas de dicho AGV el comando de emergencia "E". En esta ocasión, se considera que la solicitud de parada de emergencia se ha realizado durante el transcurso de un desplazamiento, en el cual la unidad de transporte no ha respetado el coeficiente de seguridad entre un obstáculo y el centro de la unidad. El resultado de la prueba muestra que la unidad de transporte indica que ha entrado en el estado de error de manera correcta, como puede verse en la *Figura 7.27*, además de que será incapaz de volver a ponerse en marcha, al matar los nodos encargados de la gestión del Sistema de Navegación, como se ve en la *Figura 7.28*. También se comprobó el funcionamiento de la alarma producida por el buzzer de la base robótica al entrar al estado de "Error".

Figura 7.27 Unidad de Transporte en Estado de Error o Parada de Emergencia

```
[transport_flexmansys_node_server_leonardo-2] process has finished cleanly
log file: /home/borjartime/.ros/log/0d481a08-cad4-11ec-a93d-b88d12182a4a/transport_flexmansys_node_server_leonardo-2*.log
[INFO] [1651578590.233462]: **************** FORCED SHUTDOWN  ****************
[INFO] [1651578590.234282]: Unidad de transporte Leonardo se ha detenido forzosamente mientras
[INFO] [1651578590.234970]: la unidad operaba. Leonardo, ha entrado al estado de Operative
[INFO] [1651578590.235658]: Para reiniciar, la unidad de transporte debe volver a su estacion
[INFO] [1651578590.236324]: de origen y detener todos los demas nodos restantes, incluido roscore
[transport_flexmansys_node_leonardo-1] process has finished cleanly
log file: /home/borjartime/.ros/log/0d481a08-cad4-11ec-a93d-b88d12182a4a/transport_flexmansys_node_leonardo-1*.log
[transport_flexmansys_emergency_stop_node_leonardo-3] process has finished cleanly
log file: /home/borjartime/.ros/log/0d481a08-cad4-11ec-a93d-b88d12182a4a/transport_flexmansys_emergency_stop_node_leonardo-3*.log
^C[flexmansys_transport_node_leonardo_state-4] killing on exit
[INFO] [1651578679.460199]: **************** TSUP LEONARDO DETENIDO  ****************
shutting down processing monitor...
... shutting down processing monitor complete
done
```

Figura 7.28 Mensaje de Terminal tras la Solicitud de Parada de Emergencia

CAPITULO 8

PLANIFICACIÓN

8 Planificación y presupuesto

En este capítulo se presentan tanto la planificación de las etapas a llevar a cabo para la realización del presente trabajo como el presupuesto necesario. Para ello, previamente se presentará la línea temporal que se siguió para su ejecución, en la cual se mostrarán las diferentes etapas, fechas y tiempos a manejar, además de los materiales y objetivos para cada una de ellas. Posteriormente junto con el presupuesto, también se hará una breve explicación de las consideraciones que se han seguido y se incluirán las referencias de fábrica de todos los elementos que figuran en el BOM (Bill of Materials) del proyecto.

8.1 Línea Temporal

Dado que el presente trabajo se desarrolla dentro del marco temporal de un trabajo fin de máster, toda la planificación se ha ajustado al periodo comprendido entre el inicio del curso académico y su finalización, es decir, entre septiembre de 2021 y junio de 2022. Por otro lado, ha de considerarse que, al llevarse a cabo en el marco de una tesis doctoral, se han priorizado todos los objetivos y requisitos mínimos necesarios para la finalización del proyecto, de manera que muchos de los puntos a optimizar se plantean como posibles mejoras a futuro.

En términos generales, la planificación del proyecto se dividirá en diez etapas, las cuales están divididas en cuatro fases. Estas fases de dividen de forma cronológica de la siguiente manera:

1. **Fase de Viabilidad**: Se trata de la fase inicial del proyecto, cuyas etapas se centran en el estudio de todas las posibles alternativas y vías de desarrollo del trabajo, así como en recoger todos los requisitos y especificaciones iniciales.
    - **Duración:** 9 semanas.
2. **Fase de Diseño**: Las etapas pertenecientes a la fase de diseño tendrán como objetivo el diseño de toda la estructura del sistema, así como la definición de todos los recursos que van a emplearse y de qué manera se emplearán.
    - **Duración:** 13 semanas.
3. **Fase de Desarrollo**: En esta fase, las etapas cuentan con el principal objetivo de desarrollar la parte técnica del proyecto, es decir, en traducir los diseños, requisitos y especificaciones de las dos fases anteriores a un producto o solución funcional.
    - **Duración:** 16 semanas.
4. **Fase de Validación**: Se trata de la fase final del proyecto en la cual se buscará el verificar y validar el correcto funcionamiento del desarrollo llevado a cabo en la fase anterior. En pocas palabras, se llevarán a cabo las pruebas que comprueban si la solución llevada a cabo cumplimenta los requisitos y especificaciones de la fase de viabilidad.
    - **Duración:** 21 semanas.

De este modo, las etapas que se han establecido para el desarrollo del proyecto se definen como las siguientes:

1. **Etapa de Definición de Requisitos y Alcance del Proyecto**: En esta etapa la persona responsable del proyecto deberá definir los requisitos y especificaciones mínimas requeridas para la realización de la solución. Del mismo modo, deberá instalarse en el entorno de trabajo y comprobar todas las herramientas con las que cuenta.
    - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
    - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina.
    - **Objetivo**: Identificar listado de requisitos y especificaciones a cumplimentar junto con el alcance mínimo del trabajo.
    - **Duración**: 2 semanas.
    - **Fase**: Viabilidad.
2. **Etapa de Estudio de Antecedentes y Estado del Arte**: En esta etapa se recopilará toda la documentación e información acerca de los trabajos previos realizados en la materia. Por otro lado, se llevará a cabo un estudio del estado del arte sobre el marco en el que se desarrolla el proyecto para comprender las soluciones que se están desarrollando en el mundo académico y el industrial en este ámbito.
    - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
    - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina. Mendeley Reference Manager.
    - **Objetivo**: Realización del estado del arte y recopilación y estudio de los antecedentes y soluciones llevadas a cabo por otros desarrolladores tanto en el ámbito industrial como en el académico.
    - **Duración**: 3 semanas.
    - **Fase**: Viabilidad.
3. **Etapa de Análisis de Alternativas y Viabilidad**: En esta etapa, tras la realización del estado del arte y contextualización de las anteriores dos etapas, se definirán cuáles son las posibles alternativas para llevar a cabo el trabajo, indicando las correspondientes ventajas y desventajas entre ellas. Se escogerá una alternativa en función de una serie de criterios derivados de los requisitos.
    - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
    - **Recursos Materiales** Ordenador Personal. Microsoft Office 2021. Material Oficina. Mendeley Reference Manager.
    - **Objetivo**: Establecimiento de varias alternativas de soluciones, elección de la alternativa a tomar y justificación de esta frente al resto de alternativas.
    - **Duración**: 2 semanas.

   - **Fase**: Viabilidad

4. **Etapa de Formación y Entrenamiento**: En esta etapa se realizará la formación y entrenamiento de la persona responsable del proyecto en las herramientas que se vayan a emplear para su realización. En lo que concierne al presente trabajo, la formación necesaria deberá de centrarse en la herramienta JADE y el Framework Robótico ROS, así como breves conceptos de programación tanto en Java como en Python y C/C++. También, deberá de familiarizarse con los repositorios y herramientas con las que cuenta a su disposición para la consulta de información sobre ambas herramientas.
   - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
   - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina. ROS Kinectic Kame. JADE. Entorno de Programación y Depuración.
   - **Objetivo**: Obtener dominio suficiente de las herramientas a emplear en el presente trabajo como para llevar a cabo soluciones de manera autónoma.
   - **Duración**: 4 semanas.
   - **Fase**: Diseño.
5. **Etapa de Prediseños**: En esta etapa, conocidas ya las posibilidades que permiten realizar tanto la plataforma JADE como ROS, se realizará un esquema y estructura general de la solución final. Del mismo modo, se desarrollarán los puntos principales de dicha estructura para verificar que la interconexión y el concepto diseñado son funcionales.
   - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
   - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina. ROS Kinectic Kame. JADE. Entorno de Programación y Depuración.
   - **Objetivo**: Realización de la estructura general de la solución y desarrollo de pruebas mínimas de concepto que demuestren que la estructura diseñada es viable a realizar.
   - **Duración**: 2 semanas.
   - **Fase**: Diseño.
6. **Etapa de Diseños**: En esta etapa, se revisará el concepto de la Etapa de Prediseños, verificando que concuerda con los requisitos y especificaciones ~~del cliente~~. Se llevarán a cabo las correcciones pertinentes y posteriormente se dará paso al desarrollo final de la solución.
   - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
   - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina. ROS Kinectic Kame. JADE. Entorno de Programación y Depuración. Periféricos y Unidades Robóticas.

   - **Objetivo**: Corrección de los pequeños errores o desajustes de la estructura general de prediseñada para que esta cumplimente los requisitos y especificaciones mínimas de diseño.
   - **Duración**: 3 semanas.
   - **Fase**: Diseño.

7. **Etapa de Montaje y Codificación**: En esta etapa se realiza el desarrollo de la solución final, plasmándola tanto en código como en el montaje de los elementos y periféricos hardware empleados.
   - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
   - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina. ROS Kinectic Kame. JADE. Entorno de Programación y Depuración. Periféricos y Unidades Robóticas.
   - **Objetivo**: Realización de la solución final del trabajo, de manera que esta sea funcional y cumpla con el concepto general prediseñado.
   - **Duración**: 10 semanas.
   - **Fase**: Desarrollo.
8. **Etapa de Pruebas**: En esta etapa se llevan a cabo todas las pruebas necesarias para verificar y validar que la solución diseñada en la etapa previa de montaje y codificación cumplimenta con los requisitos y especificaciones de la primera etapa del proyecto.
   - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
   - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina. ROS Kinectic Kame. JADE. Entorno de Programación y Depuración. Periféricos y Unidades Robóticas.
   - **Objetivo**: Realización de las pruebas de verificación y validación necesarias como para asegurar de que el sistema es funcional, seguro y cumple con los requisitos y especificaciones mínimas establecidas.
   - **Duración**: 7 semanas.
   - **Fase**: Validación.
9. **Etapa de Acondicionamiento**: En esta etapa se realizan los cambios y correcciones necesarias en la solución final para eliminar los errores o fallos detectados en la Etapa de Pruebas previa.
   - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
   - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina. ROS Kinectic Kame. JADE. Entorno de Programación y Depuración. Periféricos y Unidades Robóticas.

- **Objetivo**: Corrección de los errores detectados en la Etapa de Pruebas sobre la solución final desarrollada en la Etapa de Montaje y Codificación.
- **Duración**: 6 semanas.
- **Fase**: Validación.

10. **Etapa de Documentación**: En esta etapa se plasma en una documentación formal toda la solución realizada, así como toda la información y anotaciones recogidas durante las nueve etapas anteriores que sean de interés para la comprensión de la solución final.
    - **Recursos Humanos**: Graduado en Ingeniería Electrónica Industrial y Automática o similares. Máster en Ingeniería de Control, Automatización, Robótica o similares. Ingeniero/a Junior. Responsable de Proyecto.
    - **Recursos Materiales**: Ordenador Personal. Microsoft Office 2021. Material Oficina. Mendeley Reference Manager.
    - **Objetivo**: Realización de la documentación y memoria final del proyecto.
    - **Duración**: 14 semanas.
    - **Fase**: Validación.

Para llevar a cabo el seguimiento de la planificación, se ha desarrollado un esquema tipo Gantt los cuales recogen las distintas tareas que han de realizarse para completar cada etapa. Dicho esquema, se muestra dividido en cuatro secciones, es decir, una correspondiente a cada fase, en la *Figura 8.1*, *Figura 8.2*, *Figura 8.3* y la *Figura 8.4*. Nótese que la gran mayoría de estas etapas se desarrollan en paralelo a sus contiguas, debido a la gran interdependencia entre muchas de las tareas, las cuales su cumplimiento da pie al inicio o parte del desarrollo de otras etapas de la planificación, como es en el caso de la Etapa de Pruebas o la Etapa de Documentación.

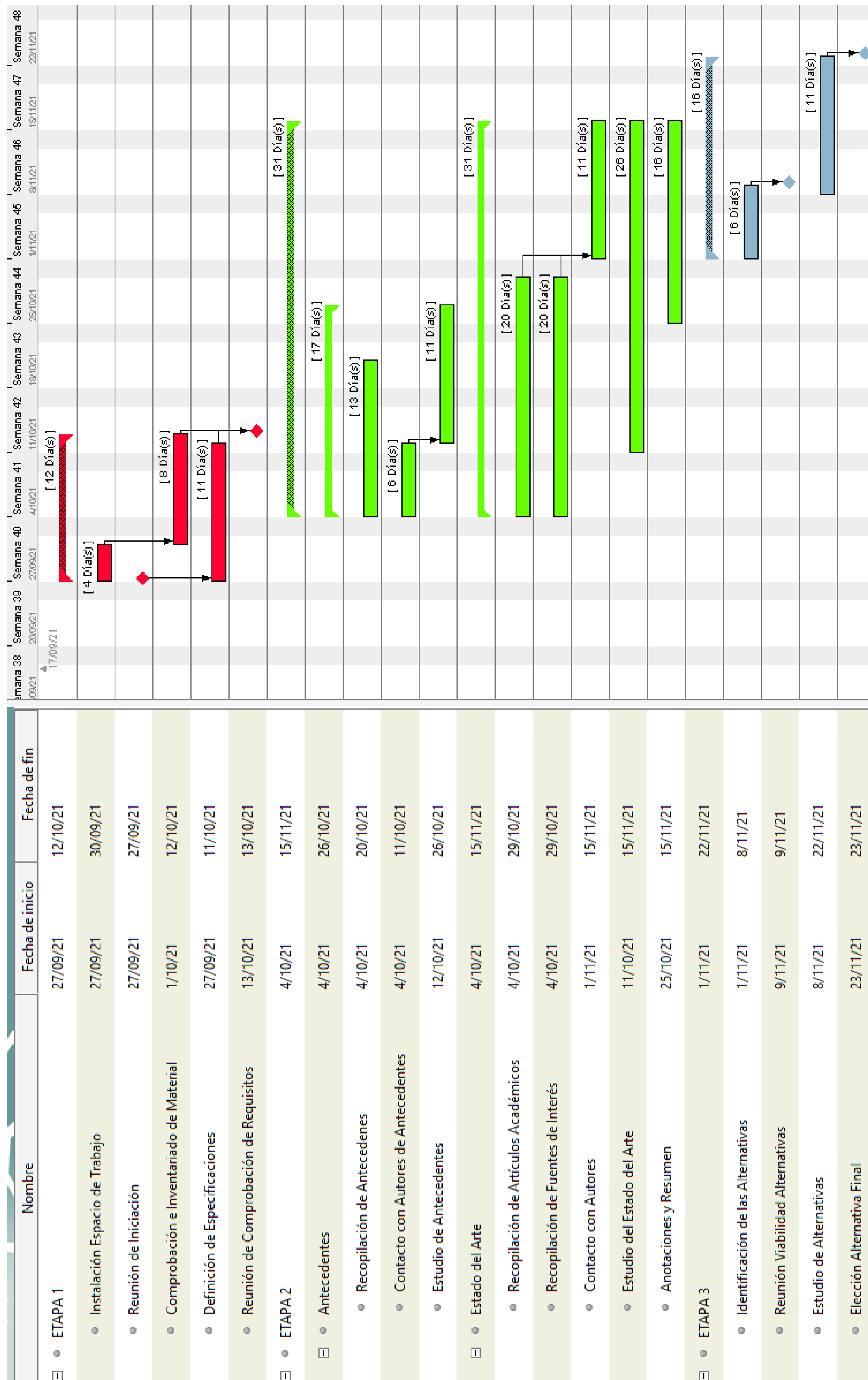


Figura 8.1 Planificación Etapas 1-3 de la Fase de Viabilidad

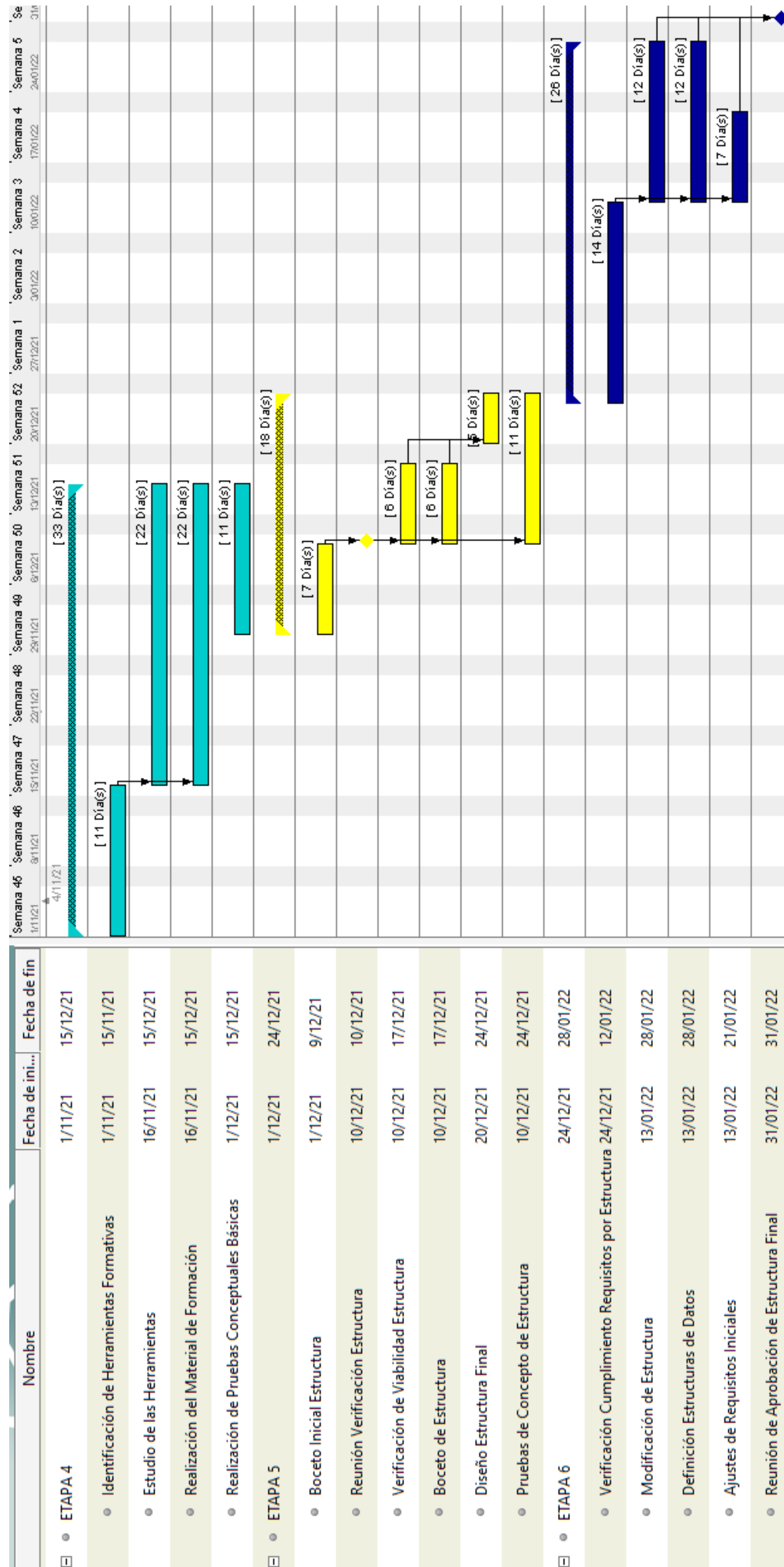


Figura 8.2 Planificación Etapas 4-6 de la Fase de Diseño

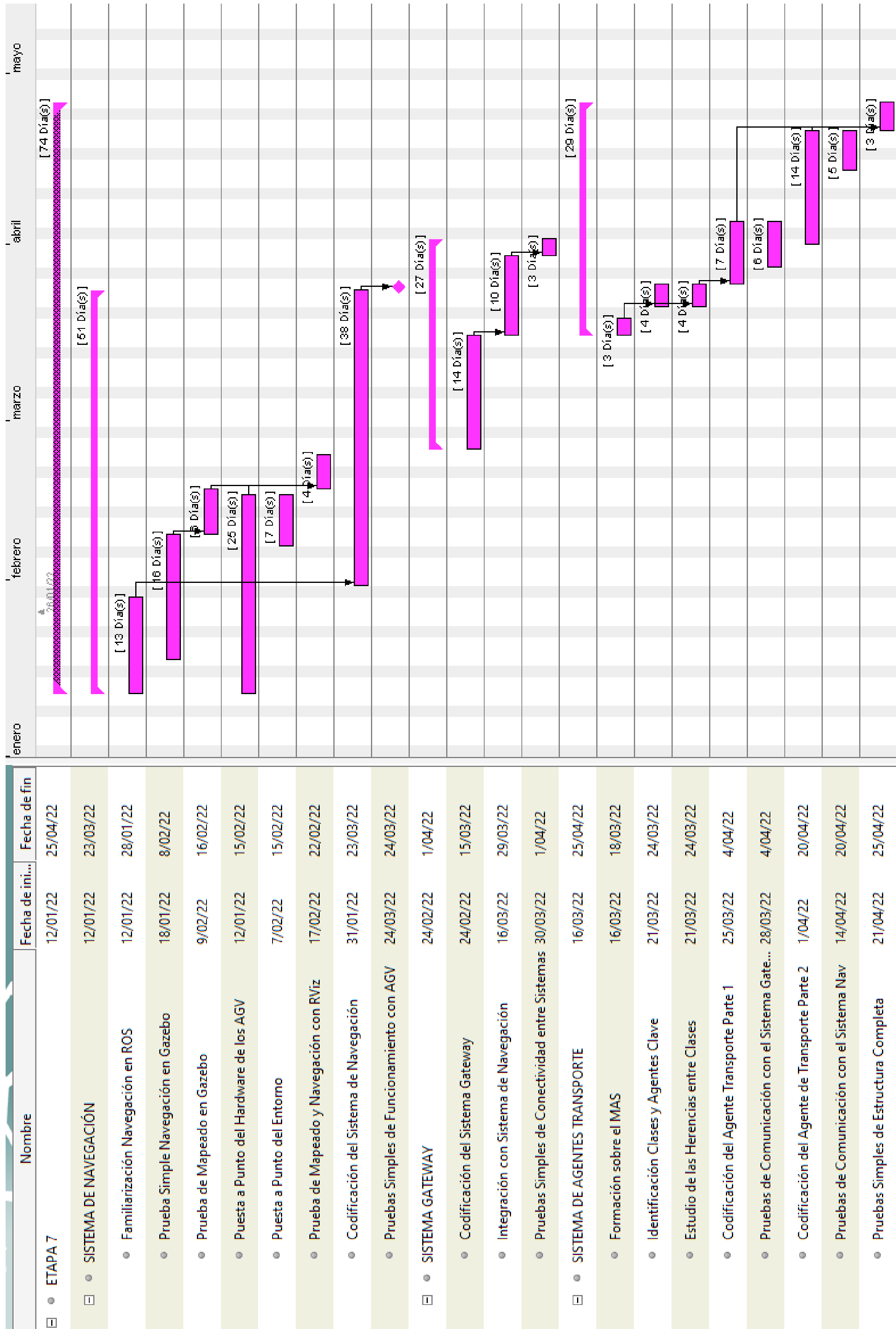


Figura 8.3 Planificación Etapa 7 de la Fase de Desarrollo

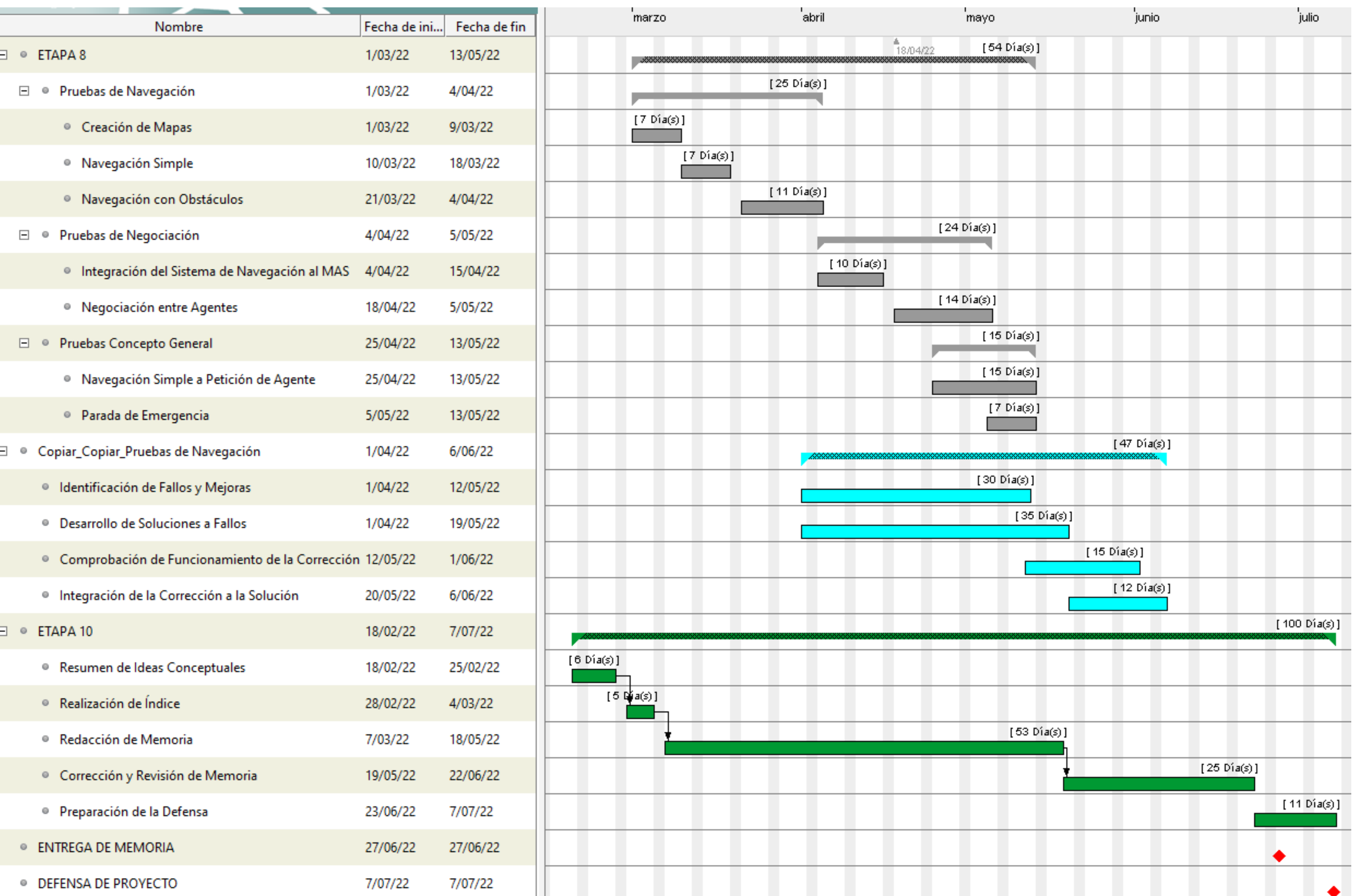


Figura 8.4 Planificación Etapas 8-10 de la Fase de Validación

8.2 Presupuesto

En lo que concierne al presupuesto del proyecto, se mostrará en dos partes: en la primera se desglosarán los precios unitarios de cada elemento necesario para la realización de este; en la segunda, donde se mostrarán los precios totales. Del mismo modo, se identificarán tres partes importantes: los gastos referentes a la fabricación y compras, las inversiones realizadas de las amortizaciones y el coste del recurso humano necesario.

8.2.1 Cuadro de Precios Unitarios

Para llevar a cabo el presupuesto referente a los componentes y consumibles mínimos, así como las amortizaciones y recursos humanos, se han tenido en cuenta las siguientes consideraciones:

- El precio de todos los elementos figura considerando tanto el IVA (Impuesto sobre el Valor Añadido) como los gastos de envío.
- Para el inventariado de todos los componentes se ha considerado las limitaciones y requerimientos de compra mínima por parte de los proveedores.
- Algunos elementos necesarios para la finalización del trabajo, así como los cargadores de las bases robóticas o los adaptadores de los sensores LiDAR y del Brazo Robótico WidowX vienen incorporados con la compra convencional de estos elementos en el proveedor indicado. Es por ello, que no figuran como tal en la *Tabla 8.1*.Tabla 7.7
- Dado que los gastos de envío y los tiempos de llegada suponen un aspecto importante a considerar, se ha dado preferencia a los proveedores que mayor número de elementos son capaces de suministrar, de manera que se reduzca la incertidumbre referente a la logística lo menos posible.
- Las licencias recogidas dentro del apartado de amortizaciones han sido gratuitas para el desarrollo del proyecto, al contar con la opción de licencia gratuita para estudiantes de la UPV/EHU. Sin embargo, se han incluido dentro del presupuesto del proyecto.
- Dentro del concepto de “Material de Oficina” se engloba todo el material referente a papelería.
- No se ha tenido en cuenta un coeficiente de seguridad para la realización del inventariado del presupuesto.
- Se ha considerado una pequeña cantidad de gasto para un director de proyecto, el cual se encargaría de supervisar al ingeniero/a junior responsable de realizar el trabajo.

8.2.1.1 Gastos de Fabricación y Compras

En esta sección se muestran por subconjuntos, los componentes, elementos y herramientas que deben de adquirirse para que los responsables del proyecto puedan llevar a cabo todo el diseño y montaje del trabajo.

REFERENCIA	PROVEEDOR	PRECIO	CANTIDAD	TOTAL
Airport Extreme A1354	Amazon	41,40 €	1 ud	41,40 €
Cámara Kinect XBOX 360	Amazon	21,50 €	1 ud	21,50 €
Concentrador USB HUB UH720	Amazon	62,00 €	4 ud	248,00 €
Material Oficina	Amazon	16,70 €	1 lote	16,70 €
Módulo ConVertidor Buck LM2596	Amazon	1,80 €	4 ud	7,20 €
Odroid XU4	Amazon	116,54 €	4 ud	466,16 €
Router TL-WR802N	Amazon	21,79 €	4 ud	87,16 €
Brazo Robótico WidowX	ROS Components	1.800,00 €	2 ud	3.600,00 €
Kobuki iCleBot Turtlebot2	ROS Components	650,00 €	4 ud	2.600,00 €
RPLiDAR A2	ROS Components	330,00 €	4 ud	1.320,00 €
Arandelas M4 Acero	RS	3,98 €	1 lote	3,98 €
Arandelas M4 Goma	RS	4,86 €	1 lote	4,86 €
Bridas 100 mm	RS	4,15 €	1 lote	4,15 €
Cable Alpha Wire AWG 22 Negro 20 m	RS	21,37 €	1 ud	21,37 €
Cable Alpha Wire AWG 22 Rojo 20 m	RS	21,37 €	1 ud	21,37 €
Cable Ethernet Cruzado	RS	14,20 €	1 lote	14,20 €
Cinta Americana Negra 100 m	RS	25,54 €	1 ud	25,54 €
Cinta Doble Cara	RS	8,35 €	1 ud	8,35 €
Conector Molex Microfit 43025-0410	RS	0,61 €	10 ud	6,10 €
Conector Molex Microfit 43645-0200	RS	0,44 €	10 ud	4,40 €
Estaño Composición SN60Pb40	RS	22,30 €	1 ud	22,30 €
Portabridas	RS	10,72 €	1 lote	10,72 €
Tornillos M4 x 16 mm Acero	RS	18,90 €	1 lote	18,90 €
Tuercas M4 Acero	RS	4,27 €	1 lote	4,27 €
Suma Total				8772,03 €

Tabla 8.1 Presupuesto de Componentes y Consumibles

8.2.1.2 Amortizaciones

Pese a que la duración de la planificación de la línea temporal es inferior a 1 año, concretamente una duración de 9 meses se ha considerado un uso de trabajo de 1 año para el cálculo de las amortizaciones.

SUJETO	PROVEEDOR	INVERSIÓN	VIDA ÚTIL	USO EN TRABAJO	AMORTIZADO
Impresora 3D Artillery Sidewinder X2	Amazon	436,99 €	6 años	1 año	72,83 €
MackBook Air	Apple	420,00 €	10 años	1 año	42,00 €
Licencia Jetbrains Completa (IntelliJ y PyCharm)	Jetbrains	785,29 €	1 año	1 año	785,29 €
Licencia Microsoft Office 2021	Microsoft	19,09 €	1 año	1 año	19,09 €
Crimpador Molex Microfit	RS	193,40 €	10 años	1 año	19,34 €
Soldador Eléctrico de Estaño	RS	43,50 €	5 años	1 año	8,70 €
Tijeras de Electricista	RS	19,70 €	10 años	1 año	1,97 €
Suma Total					949,22 €

Tabla 8.2 Presupuesto de Amortizaciones

8.2.1.3 Recursos Humanos

Tal y como se ha comentado anteriormente, se ha reservado una pequeña parte del presupuesto para un supervisor o director del responsable junior a desarrollar el trabajo.

CONCEPTO	TIEMPO	PRECIO	PRECIO TOTAL
Graduado en Ingeniería Electrónica Industrial y Automática Junior. Máster en Ingeniería de Control, Automatización, Robótica. O similar.	720 h	12.5 € / h	9000,00 €
Director de Proyecto	25 h	35 € / h	875,00 €
Suma Total			9875,00 €

Tabla 8.3 Presupuesto de Recursos Humanos

8.2.2 Cuadro de Precios Totales

Tal y como se puede apreciar en la *Tabla 8.4* el costo total del proyecto es de 21947,81 €, estando el IVA ya considerado en cada uno de los precios individuales mostrados anteriormente.

CONCEPTO	PRECIO
Componentes & Consumibles	8772,03 €
Amortizaciones	949,22 €
Recursos Humanos	9875,00 €
Subtotal	19596,25 €
Costes Indirectos (5%)	979,82 €
Imprevistos (7%)	1371,74 €
Total	21947,81 €

Tabla 8.4 Resumen de Precios Totales

CAPITULO 9

CONCLUSIONES Y TRABAJOS FUTUROS

9 Conclusiones y Trabajos Futuros

En el presente documento se ha desarrollado una solución para la gestión y coordinación de múltiples unidades autónomas de transporte y su integración en un sistema multi-agente previamente desarrollado. Este trabajo, demuestra no solo la capacidad de integración de la arquitectura propia del proyecto MINECO al entorno de ROS, sino que ofrece y establece la posibilidad de llevar a cabo la integración tanto de unidades autónomas de transporte como otra clase de células y maquinaria a la arquitectura a través de ROS.

Tras situar el proyecto, analizar el contexto en el que se sitúa el GCIS y estudiar su alcance, beneficios y requisitos del sistema, se dio un cuna idea global de lo que el trabajo debía ser: Un sistema escalable y configurable que permitiese la integración de múltiples AGV y que abarcase las tareas más simples de navegación, negociación entre agentes y percepción del entorno. Del mismo modo, al tratarse de un trabajo llevado a cabo en un ámbito puramente académico, ha de considerarse primordial el desarrollo de una solución económica no muy exigente.

La solución final aportada por este trabajo ha dado con tres sistemas claramente diferenciados: Uno centrado en la navegación autónoma de AGV, el cual permite integrar a diferentes unidades robóticas de varias gamas; un segundo sistema que permite la integración del primero en una arquitectura previamente desarrollada y dentro de un MAS; y un último sistema, que reafirma la posibilidad de integrar tanto en el entorno de ROS como la plataforma JADE a través de un paquete desarrollado en ROSJava.

Contar con una solución tal, permitirá al GCIS no únicamente con la posibilidad de tener en su mano al primer sistema de navegación plenamente funcional e integrado en la arquitectura del proyecto MINECO, previamente basada en el proyecto FLEXMANSYS, sino que abre la puerta a nuevas soluciones basadas en esta primera versión. Dichas soluciones, podrían incluir sistemas de percepción más avanzados, un sistema de manipulación o bien plantearse la migración del proyecto a otra clase de distros y versiones de plataforma.

Sin embargo, la solución final también cuenta con una serie de inconvenientes los cuales podrían llegar a ser notorios para determinadas aplicaciones, como la no posibilidad de ejecutar el sistema en tiempo real, la necesidad de aumentar el ancho de banda en el que se ejecuta todo el entorno de ROS o situaciones de fallo no contempladas en el sistema desarrollado como colisiones en la parte superior del brazo del AGV.

Por último, se dan por alcanzados prácticamente todos los objetivos iniciales de ese trabajo, pese a que en varios será necesaria una optimización u adaptación a ciertos puntos, como la percepción mediante visión. También se califica como satisfactorio el resultado final para el marco temporal en el que se ha manejado el proyecto. Los requisitos y especificaciones del cliente y de las normativas seguidas han sido alcanzados igualmente.

9.1 Mejoras y Trabajos Futuros

En lo referente a las posibles líneas de trabajo futuras y los aspectos en los que puede mejorar la solución aportada, se establecen las siguientes:

- Desarrollo de un sistema de manipulación con percepción integrada junto con la cámara Kinect. El paquete de ar_track_alvar_indiv permite detectar objetos en 3D si sus tres

superficies son equipadas con 3 códigos AR diferentes. Estos objetos, pueden ser integrados de manera automática en el árbol de transformadas del entorno de ROS, de manera que pueda desarrollarse un sistema de manipulación considerablemente sencillo y de bajo costo.

- Del mismo modo, la cámara Kinect 360 al ser un elemento de bajo costo y cuya integración es muy sencilla en el Sistema de Navegación desarrollado, se plantea una percepción de objetos sencilla para detectar obstáculos. Esta percepción, al ser el suelo del entorno de operación monocolor y de tono grisáceo, ayudaría enormemente a su detección.
- Migración del Sistema de Navegación desde ROS a ROS2, de manera que pueda adaptarse a una distribución de ROS no obsoleta.
- Desarrollo de una interfaz más amigable que permita interactuar con la persona operaria, bien sea en LabVIEW o sobre RViz o similares.
- El entorno de trabajo debería modificarse para ser más accesible y seguro para las unidades de transporte, de manera que las partes inferiores de las mesas del laboratorio no las detectase como un espacio accesible al que entrar, puesto que, al ser equipados con el brazo robótico en su parte superior, los AGV podrían llegar a colisionar. Del mismo modo, el ventanal del laboratorio debería de poder ser detectado por el LiDAR de las unidades de transporte, por lo que debería de posicionarse una tira negra a la altura del LiDAR de manera que puedan rebotar los haces de luz láser y se pueda incorporar la Zona V al mapa empleado para la navegación.
- Automatizar el proceso de configuración inicial de redes del sistema de navegación, es decir, introducir en el archivo .bash correspondiente de cada odroid las direcciones IP correspondientes que el maestro ROS tomará y su respectiva ROS_HOSTNAME.

CAPITULO 10

REFERENCIAS BIBLIOGRÁFICAS

10 Referencias Bibliográficas

En el presente capítulo se recogen todas las referencias bibliográficas empleadas para el desarrollo y justificación de la información recogida en el presente documento.

Nótese que algunas de las referencias, sobre todo las relativas a ROS, se tratan de páginas web, al tratarse ambos entornos de plataformas abiertas donde la mayor parte de la información se recoge en webs de acceso público, por lo que se tratan de las fuentes de referencia a la hora de trabajar con estas herramientas.

[1] A. López, E. Estévez Estévez, and M. Marcos, "Trazabilidad de la producción basada en agentes industriales," *Xlii Jornadas Automática Libr. Actas*, no. August, pp. 717–723, 2021, doi: 10.17979/spudc.9788497498043.717.

[2] J. Martin, O. Casquero, B. Fortes, and M. Marcos, "A generic multi-layer architecture based on ros-jade integration for autonomous transport vehicles," *Sensors (Switzerland)*, vol. 19, no. 1, 2019, doi: 10.3390/s19010069.

[3] E. A. Oyekanlu *et al.*, "A review of recent advances in automated guided vehicle technologies: Integration challenges and research areas for 5G-based smart manufacturing applications," *IEEE Access*, vol. 8. Institute of Electrical and Electronics Engineers Inc., pp. 202312–202353, 2020, doi: 10.1109/ACCESS.2020.3035729.

[4] P. Beinschob, M. Meyer, C. Reinke, V. Digani, C. Secchi, and L. Sabattini, "Semi-automated map creation for fast deployment of AGV fleets in modern logistics," *Rob. Auton. Syst.*, vol. 87, pp. 281–295, Jan. 2017, doi: 10.1016/j.robot.2016.10.018.

[5] Y. Lian, Q. Yang, W. Xie, and L. Zhang, "Cyber-Physical System-Based Heuristic Planning and Scheduling Method for Multiple Automatic Guided Vehicles in Logistics Systems," *IEEE Trans. Ind. Informatics*, vol. 17, no. 11, pp. 7882–7893, Nov. 2021, doi: 10.1109/TII.2020.3034280.

[6] X. Deng, R. Li, L. Zhao, K. Wang, and X. Gui, "Multi-obstacle path planning and optimization for mobile robot," *Expert Syst. Appl.*, vol. 183, Nov. 2021, doi: 10.1016/j.eswa.2021.115445.

[7] G. Toffetti, T. Lötscher, S. Kenzhegulov, J. Spillner, and T. M. Bohnert, "Cloud robotics: SLAM and autonomous exploration on PaaS," in *UCC 2017 Companion - Companion Proceedings of the 10th International Conference on Utility and Cloud Computing*, Dec. 2017, pp. 65–70, doi: 10.1145/3147234.3148100.

[8] H. Hu, X. Jia, K. Liu, and B. Sun, "Self-adaptive traffic control model with Behavior Trees and Reinforcement Learning for AGV in Industry 4.0," *IEEE Trans. Ind. Informatics*, 2021, doi: 10.1109/TII.2021.3059676.

[9] D. Shi, H. Mi, E. G. Collins, and J. Wu, "An Indoor Low-Cost and High-Accuracy Localization Approach for AGVs," *IEEE Access*, vol. 8, pp. 50085–50090, 2020, doi: 10.1109/ACCESS.2020.2980364.

[10] A. Armentia, U. Gangoiti, R. Priego, E. Estévez, and M. Marcos, "Flexibility support for homecare applications based on models and multi-agent technology," *Sensors (Switzerland)*, vol. 15, no. 12, pp. 31939–31964, 2015, doi: 10.3390/s151229899.

[11] K. Aloui, A. Guizani, M. Hammadi, T. Soriano, and M. Haddar, "Integrated design methodology of automated guided vehicles based on swarm robotics," *Appl. Sci.*, vol. 11,

no. 13, Jul. 2021, doi: 10.3390/app11136187.

[12] Z. Rozsa and T. Sziranyi, "Obstacle prediction for automated guided vehicles based on point clouds measured by a tilted lidar sensor," *IEEE Trans. Intell. Transp. Syst.*, vol. 19, no. 8, pp. 2708–2720, Aug. 2018, doi: 10.1109/TITS.2018.2790264.

[13] K. Guo, J. Zhu, and L. Shen, "An Improved Acceleration Method Based on Multi-Agent System for AGVs Conflict-Free Path Planning in Automated Terminals," *IEEE Access*, vol. 9, pp. 3326–3338, 2021, doi: 10.1109/ACCESS.2020.3047916.

[14] A. Martínez-gutiérrez, J. Díez-gonzález, R. Ferrero-guillén, P. Verde, R. Álvarez, and H. Perez, "Digital twin for automatic transportation in industry 4.0," *Sensors*, vol. 21, no. 10, May 2021, doi: 10.3390/s21103344.

[15] IEEE Robotics and Automation Society and Institute of Electrical and Electronics Engineers, *Collaborative Robot Transport System Based on Edge.* 2019.

[16] W. Q. Zou, Q. K. Pan, T. Meng, L. Gao, and Y. L. Wang, "An effective discrete artificial bee colony algorithm for multi-AGVs dispatching problem in a matrix manufacturing workshop," in *Expert Systems with Applications*, Dec. 2020, vol. 161, doi: 10.1016/j.eswa.2020.113675.

[17] W. Q. Zou, Q. K. Pan, and L. Wang, "An effective multi-objective evolutionary algorithm for solving the AGV scheduling problem with pickup and delivery," *Knowledge-Based Syst.*, vol. 218, Apr. 2021, doi: 10.1016/j.knosys.2021.106881.

[18] C. Chen, D. Tran Huy, L. K. Tiong, I. M. Chen, and Y. Cai, "Optimal facility layout planning for AGV-based modular prefabricated manufacturing system," *Autom. Constr.*, vol. 98, pp. 310–321, Feb. 2019, doi: 10.1016/j.autcon.2018.08.008.

[19] K. Yan and B. Ma, "Mapless navigation based on 2D LIDAR in complex unknown environments," *Sensors (Switzerland)*, vol. 20, no. 20. MDPI AG, pp. 1–16, Oct. 02, 2020, doi: 10.3390/s20205802.

[20] A. Yilmaz and H. Temeltas, "Self-adaptive Monte Carlo method for indoor localization of smart AGVs using LIDAR data," *Rob. Auton. Syst.*, vol. 122, Dec. 2019, doi: 10.1016/j.robot.2019.103285.

[21] Q. Zou, Q. Sun, L. Chen, B. Nie, and Q. Li, "A Comparative Analysis of LiDAR SLAM-Based Indoor Navigation for Autonomous Vehicles," *IEEE Trans. Intell. Transp. Syst.*, 2021, doi: 10.1109/TITS.2021.3063477.

[22] A. : Gorka, A. A. Directora, and A. Armentia, "Puesta en Marcha del Control de Robots de Transporte," 2017.

[23] F. Novoa and B. Marcos Muñoz, "Robots colaborativos enprocesos de fabricación flexibles:Integración ROS-JADE," 2017.

[24] J. Peñalver, B. Supervisor, : Oskar, and C. Oyarzabal, "Autonomous Ground Vehicle Management for Flexible Manufacturing Systems SISTEMEN INGENIERITZA ETA AUTOMATIKA SAILA DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA," 2019.

[25] P. Iigo-Blasco, F. Diaz-Del-Rio, M. C. Romero-Ternero, D. Cagigas-Muiz, and S. Vicente-Diaz, "Robotics software frameworks for multi-agent robotic systems development," *Rob. Auton. Syst.*, vol. 60, no. 6, pp. 803–821, 2012, doi: 10.1016/j.robot.2012.02.004.

[26] E. Tsardoulias and P. Mitkas, "Robotic frameworks, architectures and middleware comparison," 2017, [Online]. Available: http://arxiv.org/abs/1711.06842.

[27] S. Kolak, A. Afzal, C. Le Goues, M. Hilton, and C. S. Timperley, "It Takes a Village to Build a Robot: An Empirical Study of the ROS Ecosystem," *Proc. - 2020 IEEE Int. Conf. Softw. Maint. Evol. ICSME 2020*, pp. 430–440, 2020, doi: 10.1109/ICSME46990.2020.00048.

[28] H. Yoshida, H. Fujimoto, D. Kawano, Y. Goto, M. Tsuchimoto, and K. Sato, "ROS: An Open-Source Robot Operating System," *IECON 2015 - 41st Annu. Conf. IEEE Ind. Electron. Soc.*, no. Figure 1, pp. 4754–4759, 2015, doi: 10.1109/IECON.2015.7392843.

[29] R. T. Miguel Angel Rodríguez, Alberto Ezquerro, "ROS Basics in 5 Days."

[30] "ROS/Tutorials - ROS Wiki." http://wiki.ros.org/ROS/Tutorials (accessed Apr. 22, 2022).

[31] "ROS/TCPROS - ROS Wiki." http://wiki.ros.org/ROS/TCPROS (accessed Apr. 22, 2022).

[32] "Packages - ROS Wiki." http://wiki.ros.org/Packages (accessed Apr. 22, 2022).

[33] "Bags - ROS Wiki." http://wiki.ros.org/Bags (accessed Apr. 22, 2022).

[34] "ROS/Patterns/Communication - ROS Wiki." http://wiki.ros.org/ROS/Patterns/Communication (accessed Apr. 22, 2022).

[35] "msg - ROS Wiki." http://wiki.ros.org/msg (accessed Apr. 22, 2022).

[36] "ROS/Tutorials/DefiningCustomMessages - ROS Wiki." http://wiki.ros.org/ROS/Tutorials/DefiningCustomMessages (accessed Apr. 22, 2022).

[37] "catkin/workspaces - ROS Wiki." http://wiki.ros.org/catkin/workspaces (accessed Apr. 22, 2022).

[38] "catkin/CMakeLists.txt - ROS Wiki." http://wiki.ros.org/catkin/CMakeLists.txt (accessed Apr. 22, 2022).

[39] "catkin/package.xml - ROS Wiki." http://wiki.ros.org/catkin/package.xml (accessed Apr. 22, 2022).

[40] "roslaunch - ROS Wiki." http://wiki.ros.org/roslaunch (accessed Apr. 22, 2022).

[41] "tf - ROS Wiki." http://wiki.ros.org/tf (accessed Apr. 22, 2022).

[42] T. Foote, "Tf: The transform library," *IEEE Conf. Technol. Pract. Robot Appl. TePRA*, 2013, doi: 10.1109/TePRA.2013.6556373.

[43] "navigation - ROS Wiki." http://wiki.ros.org/navigation (accessed Apr. 22, 2022).

[44] K. Zheng, "ROS Navigation Tuning Guide," *Stud. Comput. Intell.*, vol. 962, pp. 197–226, 2021, doi: 10.1007/978-3-030-75472-3_6.

[45] D. C. Conner and J. Willis, "Flexible Navigation: Finite state machine-based integrated navigation and control for ROS enabled robots," *Conf. Proc. - IEEE SOUTHEASTCON*, 2017, doi: 10.1109/SECON.2017.7925266.

[46] W. A. S. Norzam, H. F. Hawari, and K. Kamarudin, "Analysis of Mobile Robot Indoor Mapping using GMapping Based SLAM with Different Parameter," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 705, no. 1, 2019, doi: 10.1088/1757-899X/705/1/012037.

[47] Y. Abdelrasoul, A. B. S. H. Saman, and P. Sebastian, "A quantitative study of tuning ROS gmapping parameters and their effect on performing indoor 2D SLAM," *2016 2nd IEEE Int. Symp. Robot. Manuf. Autom. ROMA 2016*, 2017, doi: 10.1109/ROMA.2016.7847825.

[48] P. L. Bonate, "A brief introduction to Monte Carlo simulation," *Clin. Pharmacokinet.*, vol. 40, no. 1, pp. 15–22, 2001, doi: 10.2165/00003088-200140010-00002.

[49] M. Kokot, D. Miklic, and T. Petrovic, "Path Continuity for Multi-Wheeled AGVs," *IEEE Robot. Autom. Lett.*, vol. 6, no. 4, pp. 7437–7444, Oct. 2021, doi: 10.1109/LRA.2021.3099086.

[50] "rosjava - ROS Wiki." http://wiki.ros.org/rosjava (accessed Apr. 22, 2022).

[51] "Gazebo." https://gazebosim.org/home (accessed Apr. 22, 2022).

[52] W. Qian *et al.*, "Manipulation task simulation using ROS and Gazebo," *2014 IEEE Int. Conf. Robot. Biomimetics, IEEE ROBIO 2014*, no. December, pp. 2594–2598, 2014, doi: 10.1109/ROBIO.2014.7090732.

[53] "xacro - ROS Wiki." http://wiki.ros.org/xacro (accessed Apr. 22, 2022).

[54] "rviz - ROS Wiki." http://wiki.ros.org/rviz (accessed Apr. 22, 2022).

[55] H. R. Kam, S. H. Lee, T. Park, and C. H. Kim, "RViz: a toolkit for real domain data visualization," *Telecommun. Syst.*, vol. 60, no. 2, pp. 337–345, 2015, doi: 10.1007/s11235-015-0034-5.

[56] "Distributions - ROS Wiki." http://wiki.ros.org/Distributions (accessed Apr. 22, 2022).

[57] D. G. Fabio Bellifemine, Giovanni Caire, *Developing Multi-Agent Systems with JADE*. .

[58] "About — kobuki 2.0 documentation." https://iclebo-kobuki.readthedocs.io/en/latest/about.html (accessed Apr. 22, 2022).

[59] A. S. Pinto de Aguiar, M. A. Riem de Oliveira, E. F. Pedrosa, and F. B. Neves dos Santos, "A Camera to LiDAR calibration approach through the optimization of atomic transformations," *Expert Syst. Appl.*, vol. 176, Aug. 2021, doi: 10.1016/j.eswa.2021.114894.

[60] N. Zarrabi, R. Fesharakifard, and M. B. Menhaj, "Robot localization performance using different SLAM approaches in a homogeneous indoor environment," *ICRoM 2019 - 7th Int. Conf. Robot. Mechatronics*, no. ICRoM, pp. 338–344, 2019, doi: 10.1109/ICRoM48714.2019.9071902.

[61] "arbotix - ROS Wiki." http://wiki.ros.org/arbotix (accessed Apr. 22, 2022).

[62] "freenect_launch - ROS Wiki." http://wiki.ros.org/freenect_launch (accessed Apr. 22, 2022).

[63] "ar_track_alvar - ROS Wiki." http://wiki.ros.org/ar_track_alvar (accessed Apr. 22, 2022).

CAPITULO 11

ANEXO I: GLOSARIO

11 ANEXO I: Glosario

En el presente anexo se recogen todas las abreviaciones y acrónimos mencionados en este documento, así como la definición de términos propios empleados. De este modo, se identifican los siguientes:

- **2D:** *Two Dimensions.*
- **3D:** *Three Dimensions.*
- **A*:** *A-Star Algorithm.*
- **ACC:** *Agent Communication Channel.*
- **ACL:** *Agent Communication Language.*
- **AGV:** *Automated Guided Vehicles.*
- **AMCL:** *Adaptative Monte Carlo Localization.*
- **AMR:** *Autonomous Mobile Robots.*
- **AMS:** *Agent Management System.*
- **API:** *Application Programming Interface.*
- **AR:** *Augmented Reality.*
- **BA:** *Batch Agent.*
- **CAD:** *Computer Assisted Design.*
- **CD:** *Duty Cycle.*
- **CMake:** *Cross Platform Make.*
- **CPS:** *Cyber-Physical System.*
- **CPU:** *Control Process Unit.*
- **CSMA** *Carrier Sense Multiple Access.*
- **D*:** *D-Star Algorithm.*
- **DF:** *Directory Facilitator.*
- **DHCP:** *Dynamic Host Control Protocol.*
- **DNS:** *Domain Name System.*
- **DWA:** *Dynamic Window Approach.*
- **EIB:** *Escuela de Ingeniería de Bilbao.*
- **ERP:** *Enterprise Resource Planning.*
- **FIPA:** *Foundation for Intelligent Physical Agents.*

- **FLEXMANSYS:** *Flexible Manufacture System.*
- **FMS:** *Flexible Manufacturing System.*
- **FoF:** *Factory of the Future.*
- **GCIS:** *Grupo de Control e Integración de Sistemas.*
- **GPIO:** *General Purpose Inputs and Outputs.*
- **GUI:** *Graphical User Interface.*
- **IaaS:** *Infrastructure as a Service.*
- **ID:** *Internet Direction*.
- **IDE:** *Integrated Development Environment.*
- **INCAR:** *Ingeniería de Control Automatización y Robótica*
- **IP:** *Internet Protocol.*
- **JADE:** *Java Agent Development Framework.*
- **LAN:** *Local Area Network.*
- **LED:** *Light-Emitting Diode.*
- **LIDAR:** *Light Detection and Ranging.*
- **LUT:** *Lookup Table.*
- **MA:** *Machine Agent.*
- **MAC:** *Medium Access Control.*
- **MARS:** *Multi Agent Robotic Systems.*
- **MAS:** *Multi Agent System.*
- **Mbps:** *Megabits per second.*
- **MCL:** *Monte Carlo Localization.*
- **MES:** *Manufacturing Execution System.*
- **MMW:** *Matrix Manufacturing Workshop.*
- **MoA:** *Monitor of Agent.*
- **MQTT:** *Message Queue Telemetry Transport.*
- **MRS:** *Multi Robot Systems.*
- **NA:** *Node Agent.*
- **NTP:** *Network Time Protocol.*
- **OpenRAVE:** *Open Robotics Automation Virtual Environment.*
- **OpenRTM:** *Open Real Time Middleware.*

- **OR:** *Order Agent.*
- **ORCOS:** *Open Robot Control Software.*
- **P2P:** *Peer-to-Peer.*
- **PA:** *Planner Agent.*
- **PaaS:** *Platform as a Service.*
- **PSA:** *Power Station Agent.*
- **QR:** *Quick Response***.**
- **RGB:** *Red Green Blue.*
- **RMA:** *Remote Management Agent.*
- **ROS:** *Robot Operative System.*
- **RSF:** *Robotic Software Framework.*
- **SBC:** *Single Board Computer.*
- **SLAM:** *Simultaneous Localization and Mapping.*
- **SM:** *System Manager.*
- **SO:** *Sistema Operativo.*
- **SOC:** *State of Charge.*
- **TA:** *Transport Agent.*
- **TCP/IP:** *Transmission Control Protocol/Internet Protocol.*
- **TPS:** *Toyota Production Systems.*
- **TUN:** *Transport Unit Name.*
- **UAV:** *Unnamed Aerial Vehicle.*
- **UPV/EHU:** *Universidad del País Vasco / Euskal Herriko Univertsitatea.*
- **UR:** *Universal Robots.*
- **USB:** *Universal Serial Bus.*
- **UWB:** *Ultra Width Band.*
- **VDC:** *Voltage Direct Current.*
- **WAN:** *Wide Area Network.*
- **Wi-Fi:** *Wireless Fidelity.*
- **WISP:** *Wireless Internet Service Provider.*
- **XML:** *Extensible Markup Language.*
- **YARP:** *Yet Another Robot Platform.*

CAPITULO 12

ANEXO II: MANUAL DE USO DEL SISTEMA DE NAVEGACIÓN

12 ANEXO II: Manual de Uso del Sistema de Navegación

En el presente anexo se recogen todos los pasos e indicaciones que han de llevarse a cabo para poner en marcha la solución propuesta. Para ello, primeramente, se mostrará el cómo poner en marcha el Sistema de Navegación suponiendo que la persona operaria ocupará las funciones propias de los otros dos sistemas. Por otro lado, tras este primer punto, se mostrará el cómo arrancar el Sistema de Agentes de Transporte y Sistema Gateway sin la necesidad de la presencia de una unidad de transporte que esté haciendo uso de un Sistema de Navegación. Finalmente, se explicará el cómo interconectar los tres sistemas.

12.1 Puesta en Marcha del Sistema de Navegación

El Sistema de Navegación requiere de hasta diez terminales por cada unidad de transporte para su funcionamiento a plena capacidad. Estas terminales están divididas en dos grupos distintos: las del grupo "Turtlebot", las cuales se indicarán mediante "TT"; y las del grupo "Macbook" o equipo maestro, las cuales se indicarán mediante "TM". Tras abrir las diez terminales, ha de introducirse el siguiente comando en todas ellas:

1. $ source /home/borjartime/catkin_ws/devel/setup.bash

Tras ello, ha de encenderse la estación Airport Extreme y esperar hasta que su LED indicador se turne verde. Posteriormente, se comprueba entre las redes Wi-Fi disponibles la presencia de la red "Arkham Asylum", y se conecta el PC en el cual vaya a correr el maestro de ROS a dicha red mediante la contraseña "Laboratorio Automática". Se espera en torno a 30 segundos a que la odroid termine por arrancar y se procede a la realización de los siguientes pasos, que se llevarán a cabo en las tres terminales asociadas al Kobuki o Turtlebot:

2. TT1, TT2, TT3: $ ssh odroid@192.168.1.108¡ TT1, TT2, TT3: $ odroid

Con los dos comandos anteriores nos conectaremos a través del equipo maestro ROS a la odroid que gestiona a la unidad de transporte mediante terminal remota. Tras ello, ha de configurarse la IP que tomarán los nodos de la unidad de transporte en el entorno de ROS mediante la variable ROS_HOSTNAME y el lugar en el que reside el maestro ROS mediante la variable ROS_MASTER_URI. Nótese, que en función del tipo de transporte que vaya a emplearse y de la dirección que toma el equipo maestro en la red Arkham Asylum puede variar el contenido de ambas variables.

3. TT1, TT2, TT3: $ export ROS_MASTER_URI=http://192.168.1.134:11311
4. TT1, TT2, TT3: $ export ROS_HOSTNAME=192.168.1.108

Una vez configurado lo relativo a los transportes, se procede a configurar lo relativo al equipo maestro. Para ello, en las 7 terminales restantes del equipo, se introducen los siguientes comandos:

5. TM1-TM7: $ export ROS_MASTER_URI=http://192.168.1.134:11311
6. TM1-TM7: $ export ROS_HOSTNAME=192.168.1.134

Posteriormente, ha de arrancarse el nodo ROS maestro en el equipo en el que vaya a emplearse. Puede comprobarse que el entorno ROS ha sido correctamente configurado introduciendo el comando "$ rostopic list" en el equipo remoto odroid.

7. TM1: $ roscore

Antes de arrancar el Sistema de Navegación, es necesario que tanto la odroid como el equipo maestro estén sincronizados en tiempo, por lo que ha de invocarse el servicio de chrony.

8. TT1: $ service chrony restart

Se puede comprobar que ambos equipos tienen la misma hora en sus relojes internos mediante el siguiente comando:

9. TT1 y TM1: $ timedatectl

Una vez se hayan configurado ambos relojes, se procede a arrancar los nodos que gestionan la base robótica y el LiDAR mediante los siguientes comandos:

10. TT1: $ roslaunch turtlebot_bringup minimal.launch
11. TT2: $ roslaunch rplidar_ros rplidar_custom.launch

Cuando ambos elementos estén ejecutándose de manera correcta, se deberá arrancar el paquete relativo al ROS Navigation Stack, es decir:

12. TM2: $ roslaunch turtlebot_transport_flexmansys turtlebot_transport_ flexmansys_move_base.launch

Para comprobar que se reciben todos los datos de manera correcta y que el sistema puede arrancar el nodo maestro, ha de arrancarse la interfaz de RViz mediante el siguiente comando:

13. TM3: $ rviz

La configuración que permite ver todo lo necesario dentro de RViz para llevar a cabo una navegación autónoma correcta, se muestra en las siguientes figuras.

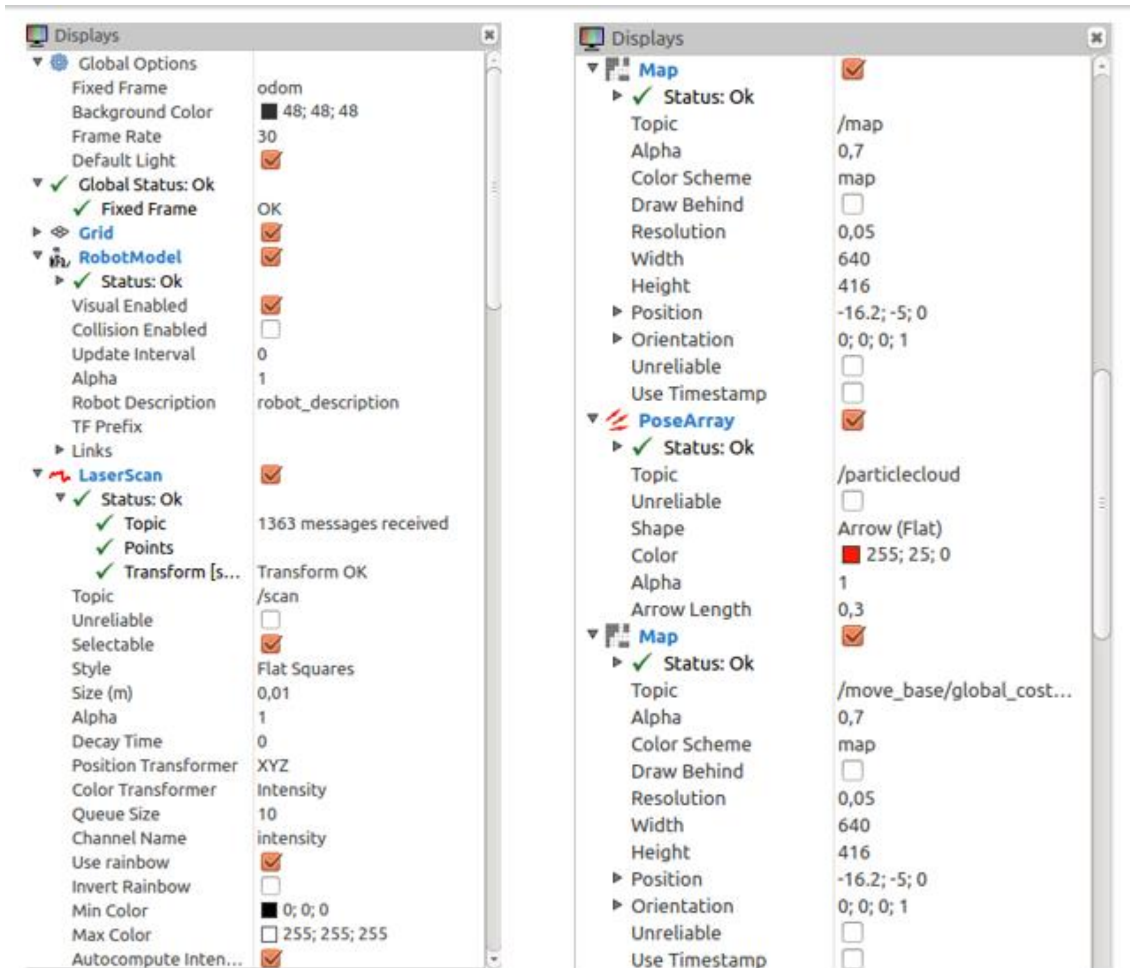


Figura 12.1 Cofiguración RViz Parte 1

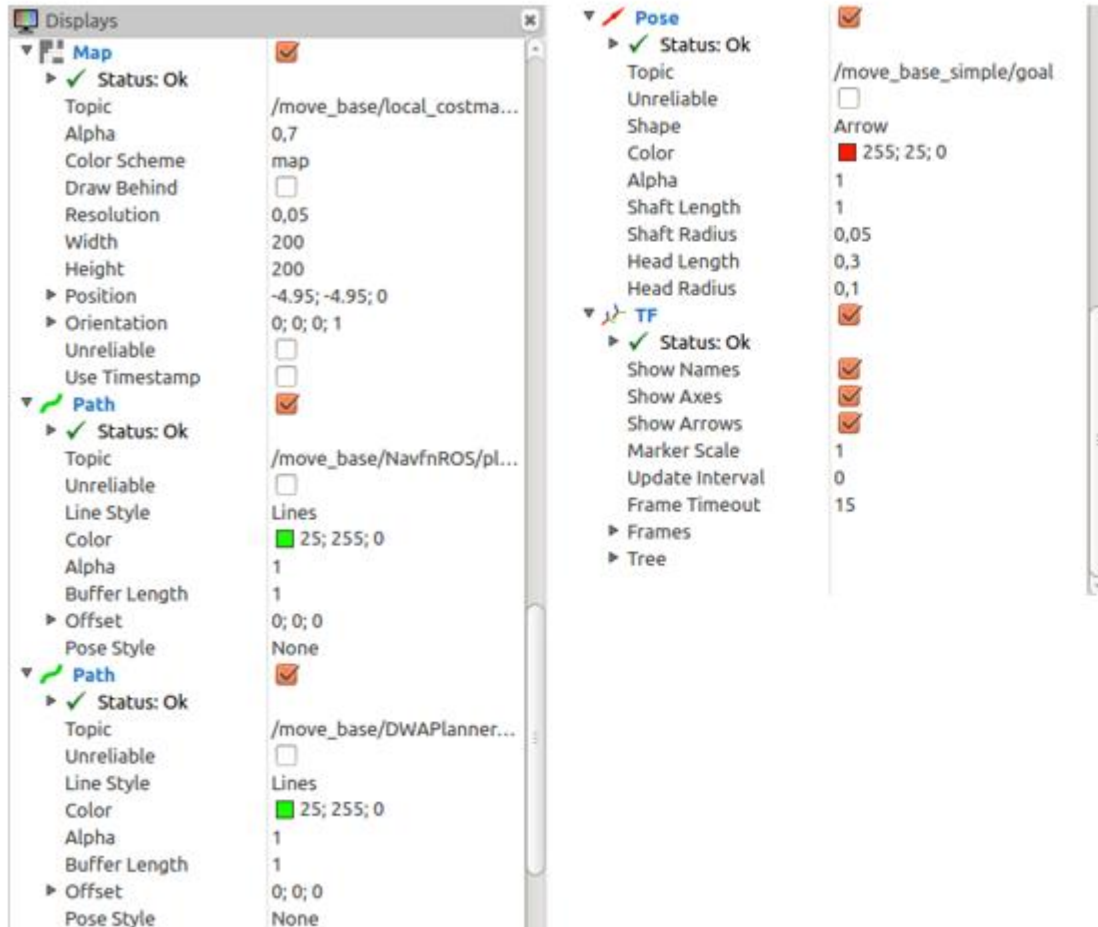


Figura 12.2 Configuración RViz Parte 2

Con RViz ya arrancado y configurado, se podrá poner en marcha el Sistema de Navegación. Para ello, ha de introducirse el siguiente comando para arrancar los cuatro nodos principales que lo forman:

14. TM4: $ roslaunch turtlebot_transport_flexmansys_

    launchfile.launch

A continuación, en la terminal en la que se arranca el Sistema de Navegación deberá de mostrarse el siguiente mensaje de la *Figura 12.3*.

```
MAC /home/borjartime/catkin_ws/src/turtlebot_transport_flexmansys/launch/turtlebot_transport_flexmansys_launchfile.laur
borjartime@borjartime-MacBookAir:~$ roslaunch turtlebot_transport_flexmansys turtlebot_transport_flexmansys_
launchfile.launch
... logging to /home/borjartime/.ros/log/91030596-cac6-11ec-bfdd-b88d12182a4a/roslaunch-borjartime-MacBookAi
r-18693.log
Checking log directory for disk usage. This may take awhile.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.

started roslaunch server http://192.168.1.134:46857/

SUMMARY
========

PARAMETERS
 * /rosdistro: kinetic
 * /rosversion: 1.12.17

NODES
  /
    flexmansys_transport_node_leonardo_state (turtlebot_transport_flexmansys/transport_flexmansys_state_publ
isher.py)
    transport_flexmansys_emergency_stop_node_leonardo (turtlebot_transport_flexmansys/transport_flexmansys_e
mergency_stop_node.py)
    transport_flexmansys_node_leonardo (turtlebot_transport_flexmansys/transport_flexmansys_main.py)
    transport_flexmansys_node_server_leonardo (turtlebot_transport_flexmansys/transport_flexmansys_send_goal
.py)

ROS_MASTER_URI=http://192.168.1.134:11311

process[transport_flexmansys_node_leonardo-1]: started with pid [18725]
process[transport_flexmansys_node_server_leonardo-2]: started with pid [18727]
process[transport_flexmansys_emergency_stop_node_leonardo-3]: started with pid [18728]
process[flexmansys_transport_node_leonardo_state-4]: started with pid [18729]
[INFO] [1651571546.343952]: *************** TRANSPORT IDLE ***************
[INFO] [1651571546.346036]: La unidad de transporte leonardo esta en modo idle
[INFO] [1651571546.347116]: Nivel de Bateria: 14.2 VDC
```

Figura 12.3 Información Mostrada por Sistema navegación

Si el sistema ha sido arrancado correctamente, deberá poder verse por terminal la interfaz de bienvenida del sistema. En caso de no contar con dicha terminal a mano, también puede comprobarse leyendo el tópico de estado del transporte. Por ejemplo, para el caso del transporte Leonardo, se introduciría el siguiente comando:

15. TM5: $ rostopic echo /flexmansys/state/Leonardo

En caso de leerse el tópico de estado anterior, deberá de mostrarse la información recogida en la siguiente *Figura 12.4.*

```
MAC    borjartime@borjartime-MacBookAir: ~ 87x23
---
kobuki_general:
  transport_unit_name: "Leonardo"
  transport_unit_state: "Idle"
  battery: 14.1000003815
kobuki_obstacle:
  detected_obstacle_bumper: False
  detected_obstacle_camera: False
kobuki_position:
  transport_in_dock: False
  recovery_point: "NONE"
  odom_x: 0.0
  odom_y: 0.0
  rotation: 0.0
odroid_date:
  year: 2022
  month: 5
  day: 3
  hour: 11
  minute: 53
  seconds: 50
```

Figura 12.4 Información Mostrada por Tópico de Estado

Posteriormente, el Sistema de Navegación ya estaría listo para ser empleado correctamente. Por ello, pueden empezar a introducirse coordenadas y comandos a través del tópico de coordenadas correspondiente a la unidad de transporte arrancada. Por terminal, pueden introducirse mediante el siguiente comando, siendo lo contenido entre las dos comillas la coordenada a introducir.

16. TM6: $ rostopic pub /flexmansys/coordenada/leonardo std_msgs/String "X"

En caso de querer arrancar la cámara Kinect y su sistema de visión y reconocimiento de códigos AR, deberá de introducirse estos dos comandos por separado, uno en la odroid o equipo externo y otro en el equipo maestro o MacBook, en este caso. Estos comandos, son los siguientes:

17. TT3: $ roslaunch freenect_launch my_freenect.launch
18. TM7: roslaunch turtlebot_transport_flexmansys_

    ar_trac_alvar_indiv.launch

Nuevamente, puede emplearse en una nueva terminal otra sesión de RViz, con una configuración especial para visualizar las imágenes que llegan desde la cámara Kinect, además de la posición relativa y número de los códigos AR detectados:

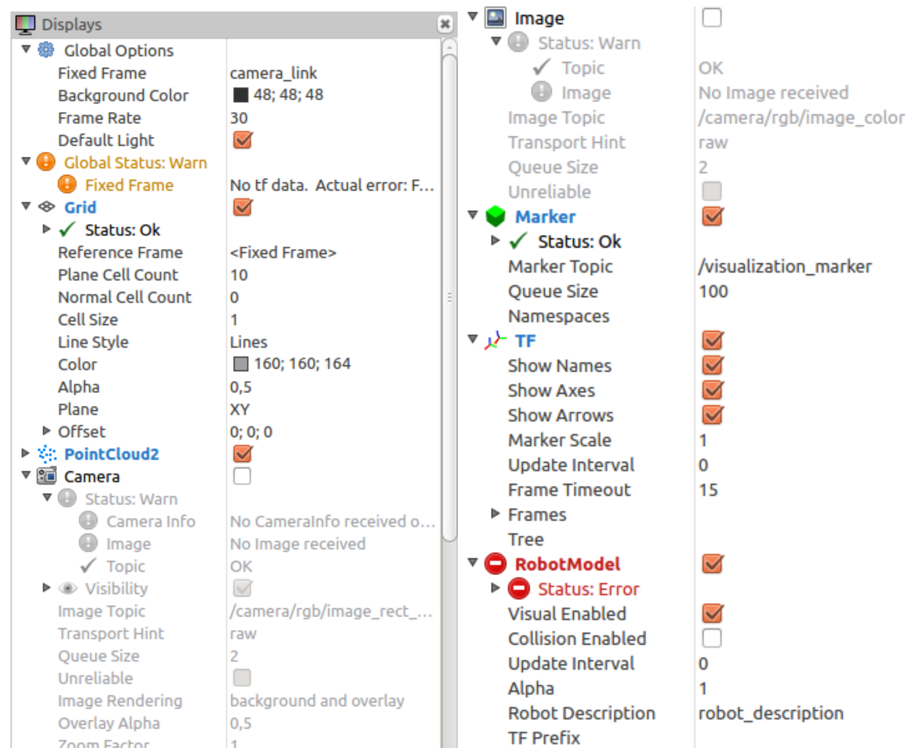


Figura 12.5 Configuración RViz para la Visualización de Cámara Kinect

Nótese, que si se trata de la primera vez que se ejecutan los paquetes respectivos al entorno de trabajo catkin_ws, al inicio, es muy probable que haya que realizarse un catkin_make, es decir, construir los paquetes. Para ello, se puede indicar que se construya únicamente el paquete respectivo al Sistema de Navegación: turtlebot_transport_flexmansys. Para ello, antes del source realizado en el primer paso, debería de introducirse el siguiente comando situándonos en el directorio donde reside el entorno de trabajo.

- $ catkin_make –only-pkg-with-deps turtlebot_transport_flexmansys

Una vez el sistema esté completamente arrancado y la unidad de transporte en marcha, la persona usuaria podrá suplir la función de los agentes software del MAS e introducir los comandos que considere necesarios a través del comando mostrado en el paso 17, es decir, a través del terminal TM6.

12.2 Puesta en Marcha del Sistema de Agentes de Transporte y Sistema Gateway

Del mismo modo al anterior ejemplo serán necesarias varias terminales para poner en marcha todos los agentes del sistema. En este caso, únicamente serán necesarias 7 terminales, de las cuales las cuatro relativas a la puesta en marcha de agentes pueden recogerse dentro de la configuración del IDE empleado para automatizar el proceso. Al igual que en el paso anterior, será necesario abrir todos los terminales e introducir en todos y cada uno de ellos el siguiente comando:

1. $ source /home/borjartime/IdeaProjects/FlexManSys/rosjava_ws/devel/setup.bash

La ejecución de este comando, como puede observarse por el directorio al que señala, implica la instalación en modo local del repositorio donde reside el MAS. A través de dicho source, se introducirán dentro del entorno de ROS todas las dependencias necesarias para la ejecución de los paquetes contenidos dentro del entorno de trabajo de rosjava_ws.

Posteriormente, dado que se va a suponer que no se cuenta con el Sistema de Navegación activo, será necesario el generar un terminal que imite los tipos de mensajes de estado que publican las unidades de transporte para poder poner en marcha los agentes de transporte. Para ello, uno de los seis terminales anteriores deberá de tener un source diferente para poder trabajar con los tipos de mensajes propios del entorno de trabajo en el que opera el Sistema de Navegación.

2. T1 $ source /home/borjartime/IdeaProjects/FlexManSys/catkin_ws/devel/setup.bash

Tras realizar este source, se deberá de arrancar en otro terminal el nodo maestro, introduciendo simplemente la siguiente línea:

3. T2 $ roscore

Con el nodo maestro ejecutándose, ya puede darse paso a la publicación del tópico de estado que suplirá a las unidades de transporte. Para ello, debe de introducirse el siguiente comando:

4. T1 $ rostopic pub /flexmansys/state/leonardo turtlebot_transport_flexmansys

   /TransportUnitState TAB

En caso de haberse direccionado en el paso dos correctamente el espacio de trabajo a dicha terminal, este "TAB" anterior representa a dos click en el tabulador del teclado para autocompletar el texto, de manera que el comando del terminal quede como se muestra en la *Figura 12.6.*

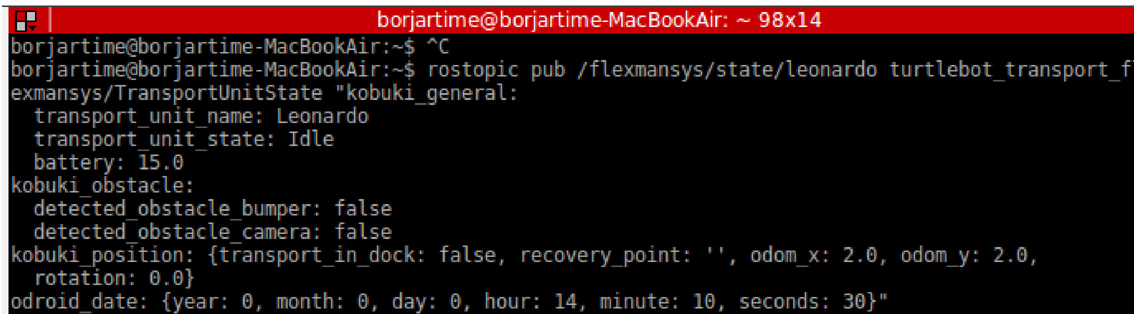


Figura 12.6 Mensaje de Estado Autocompletado

Posteriormente, hay que situarse en el directorio correspondiente en el que reside el entorno de trabajo de rosjava_ws y hacer un catkin make. Tras ello, se vuelve al directorio del respositorio FlexManSys.

5. T3 $ cd /IdeaProjects/FlexManSys/rosjava_ws
6. T3 $ catkin_make --only-pkg-with-deps fms_transp
7. T3 $ cd ..

Tras ello, se da paso a la ejecución de la clase java correspondiente a la ejecución de la GUI de JADE.

8. T4 $ java -Dlog4j.configurationFile="log4j2.xml" -cp ".\lib\jade;.\lib\log4j\log4j-api-2.3.jar;.\lib\log4j\log4j-core-2.3.jar" jade.Boot -gui

Lo siguiente, sería arrancar el agente System Model Agent, o SMA, el cual gestionará el registro de nuevos agentes en el MAS:

9. T5 $ java -Dlog4j.configurationFile="log4j2.xml" -cp ".\lib\jade;.\lib\log4j\log4j-api-2.3.jar;.\lib\log4j\log4j-core-2.3.jar;.\lib\commons\commons-codec-1.3.jar;.\lib\commons\commons-io-2.6.jar;.\classes" jade.Boot -container sa:es.ehu.SystemModelAgent

Con ello, y el tópico de estado publicando datos de manera periódica, se da paso al arranque del agente Gateway. Se sabrá que se ha arrancado correctamente al ver que en su terminal correspondiente pueden observarse los datos recogidos desde el tópico de estado. Nótese que se introducen dos argumentos de entrada: Leonardo y T_01. Es decir, el TUN o nombre de la unidad de transporte y el número que tomará en el MAS, respectivamente.

10. T3 $ java -Dlog4j.configurationFile="log4j2.xml" -cp "./lib/jade:././lib/log4j/log4j-api-2.3.jar:./lib/log4j/log4j-core-2.3.jar:./lib/commonmmons-codec-1.3.jar:./lib/Gson/gson-2.8.6.jar:./lib/commons/commons-io-2.6.jar:./lib/commons/commons-lang3-3.8.1.jar:./rosjava_ws/src/fms_transp/turtlebot2/build/install/turtlebot2/lib/*:." com.github.rosjava.fms_transp.turtlebot2.ACLGWAgentROS Leonardo T_01

El siguiente paso consistiría en arrancar al agente transporte, lo cual se hará mediante el siguiente comando que cuenta con tres argumentos de entrada:

11. T6 $ java -Dlog4j.configurationFile="log4j2.xml" -cp ".\lib\jade;.\lib\log4j\log4j-api-2.3.jar;.\lib\log4j\log4j-core-2.3.jar;.\lib\commons\commons-codec-1.3.jar;.\lib\Gson\gson-2.8.6.jar;.\lib\commons\commons-io-2.6.jar;.\lib\commons\commons-lang3-3.8.1.jar;.\classes" jade.Boot -container auxma-local:es.ehu.domain.manufacturing.agents.TransportAgent("T_01","classes/resources/ResInstances/KeyPositions1.xml","classes/resources/ResInstances/TransportPlan1.xml")

En caso de que no se arranque el agente transporte correctamente a la primera, deberá de repetirse todo el proceso desde el paso 8. En cambio, si se arranca bien el sistema, se podrá ver en el tópico de coordenadas que se están publicando comandos y coordenadas introduciendo el siguiente comando:

12. T7 $ rostopic echo /flexmansys/coordenada/leonardo

A partir de este punto, la persona usuaria será capaz de interactuar con los agentes a través de lo publicado en el tópico de estado, es decir, a través de la terminal 1 o T1.

12.3 **Puesta en Marcha General**

En los dos puntos anteriores se muestra la puesta en marcha de cada sistema abstrayendo tanto el Sistema de Agentes de Transporte y Gateway como el propio Sistema de Navegación. Para poder emplear los res sistemas en su conjunto, no hay que realizar ningún cambio considerable en lo mostrado anteriormente. Únicamente, hay que considerar que no debe de pisarse el sistema publicando en el tópico de estado de manera manual a través del terminal T1, y que previamente debe de arrancarse el Sistema de Navegación antes que el Gateway y los agentes de transporte.

CAPITULO 13

ANEXO III: ESQUEMAS, MAPAS Y GRÁFICOS

13 ANEXO II: Esquemas y Gráficos

En el presente anexo se recogen varios esquemas que pueden ayudar al entendimiento de la solución desarrolalda. Por un lado, se encuentran esquemas simples, y por otro, gráficos generados por la plataforma ROS a través de la herramienta de ROSGraph, de manera que pueda verse de manera más completa y sin simplificar todos los nodos que toman parte en la solución diseñada, además de la interacción entre ellos.

1. El primer gráfico mostrado en la *Figura 13.1* muestra la interacción y herencias entre los distintos tipos de agentes de recurso que se dan en el MAS.
2. El segundo gráfico mostrado en la *Figura 13.2* muestra todos los nodos del Sistema de Navegación, gráfico aportado por la herramienta ROSGraph.
3. El tercer gráfico mostrado en la *Figura 13.3* muestra la interacción de un agente transporte con el interfaz aportado por el sistema de navegación.

Figura 13.1 Herencias y Extensiones de los Distintos Tipos de Agentes Recuso en el MAS

Figura 13.2 Distribución Nodos del Sistema de Navegación según ROSGraph

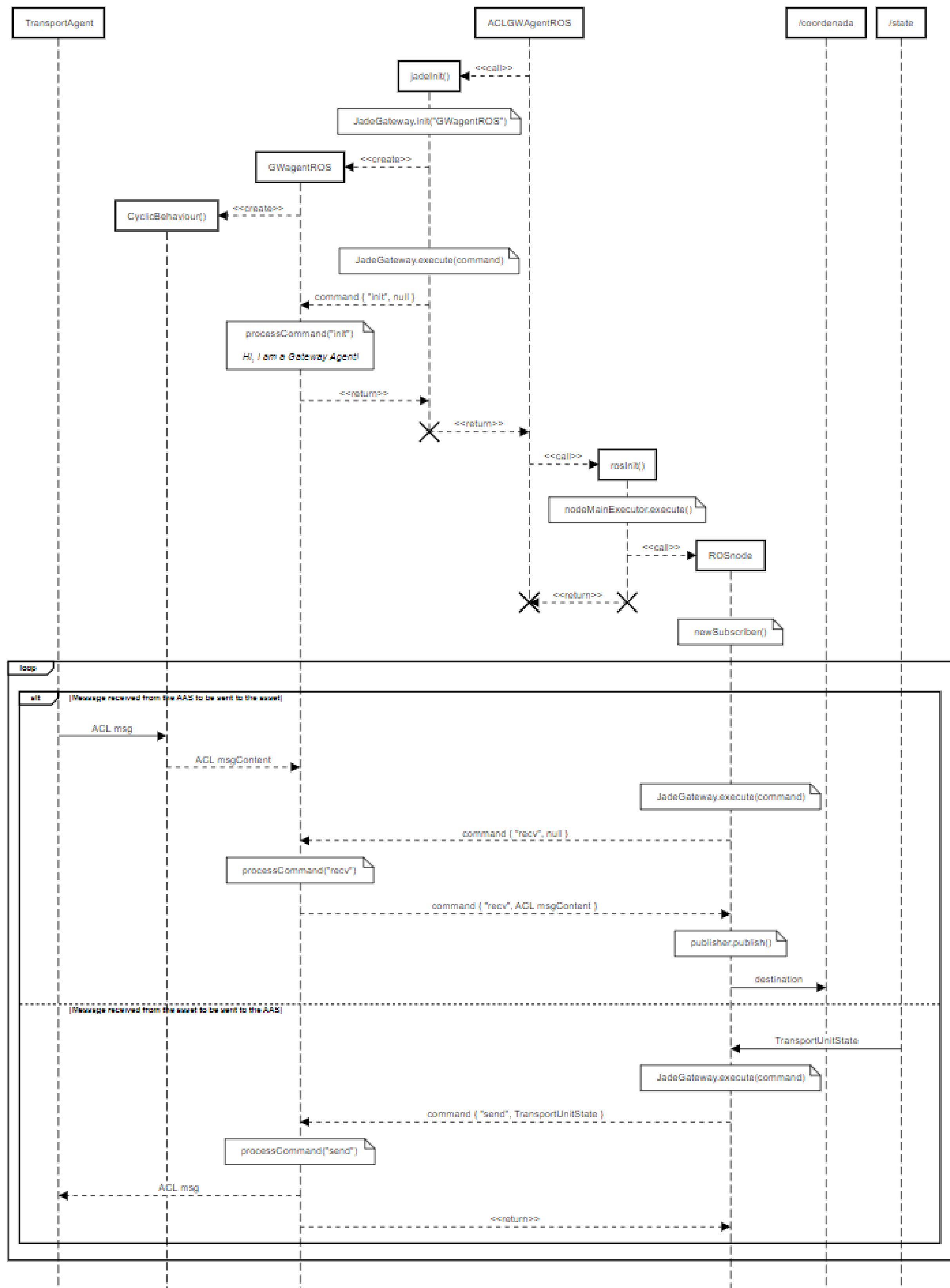


Figura 13.3 Gráfico UML de la Interacción Agente Transporte e Interfaz del Sistema de Navegación

19. **Error! Referencia de hipervínculo no válida.**