VNiVERSiDAD D SALAMANCA

CAMPUS DE EXCELENCIA INTERNACIONAL

FACULTAD DE CIENCIAS

DEPARTAMENTO DE INFORMÁTICA Y AUTOMÁTICA

Tesis Doctoral:

MULTIARQUITECTURA DISTRIBUIDA PARA EL DESARROLLO DE MISIONES MULTI-ROBOT

Francisco Javier Serrano Rodríguez

Dirigida por:

Vidal Moreno Rodilla

Belén Curto Diego

Salamanca, abril 2017

Dedico esta Tesis a toda mi familia y amigos, entre los que felizmente creo que puedo incluir a todos mis compañeros y directores.

Quiero agradecer especialmente su apoyo a Julia, mi mujer, y a mi hija que está en camino, que ya antes de nacer me ha dado fuerzas para terminar este trabajo. Agradezco también la insistencia de mis padres y el apoyo de mi hermano, sé que serán de las personas que más se alegrarán de que termine. Gracias a Belén y a Vidal por su confianza, su ayuda y todos estos años felices que he pasado en la Facultad con grandes compañeros y amigos. Entre ellos, agradezco su ayuda diaria a Jesús, un ejemplo a seguir en todos los sentidos, a Teny, por sus buenos consejos en momentos difíciles, a Doc y su Amiga 500, a Chaky y sus robos de balón, a Licus por su detección de puertas, a Vicen por esos buenos ratos de descanso, al resto de los amigos con los que trabajo, por hacerme la vida más agradable, y a los amigos con los que no trabajo, por seguir siéndolo a pesar de no haber tenido casi tiempo para veros.

Índice general

1. Introducción **1**

1.1. Objetivos de este trabajo 8

1.2. Estructura de la memoria 9

2. Estado del Arte **11**

2.1. Evaluación de RDEs existentes 12

2.1.1. MissionLab 14

2.1.2. CARMEN 22

2.1.3. RIDE 26

2.1.4. ROS 28

2.1.5. Player/Stage 31

2.1.6. OROCOS 35

2.2. Localización global con filtros de partículas 37

2.2.1. Filtros Bayesianos 37

2.2.2. Filtros de partículas 39

2.2.3. Revisión de los efectos y el manejo de errores 41

2.3. Sistemas de Información Geográfica (GIS) 44

2.3.1. Modelo de datos espaciales OGC 46

2.3.2. Consultas espaciales PostGIS 48

3. MissionLab-CARMEN: nuestro RDE integrado **51**

3.1. Diseño conceptual de la arquitectura 52

3.2. Retos de implementación 55

3.2.1. Compatibilidad con distribuciones Linux recientes 55

3.2.2. El módulo puente IPC-Adapter 56

3.2.3. Mejoras en la librería IPC 58
3.2.4. Control de robots de CARMEN 59
3.3. Integración de MissionLab y Carmen 60
3.3.1. Integración a bajo nivel 60
3.3.2. Integración a alto nivel 62

4. Método propuesto de localización global 65
4.1. Arquitectura de nuestro filtro de partículas 66
4.2. Detección de puertas 69
4.3. Mapas GIS del entorno 71
4.4. Clustering 72
4.5. Mejoras en las fases estándar de los filtros de partículas 73
4.5.1. Fase de inicialización 74
4.5.2. Fase de corrección 76
4.5.3. Fase de muestreo 76
4.6. Filtro de partículas concurrente 78

5. Resultados 83
5.1. Evaluación de MissionLab-CARMEN 83
5.1.1. Misión multi-robot utilizando MissionLab 84
5.1.2. Misión de navegación en CARMEN 85
5.1.3. Misión multi-robot utilizando MissionLab-CARMEN . . . 86
5.1.4. Misión en MissionLab-CARMEN con robots reales 88
5.1.5. Misión en MissionLab-CARMEN con carretilla industrial 89
5.2. Análisis del procedimiento de localización 96
5.2.1. Entorno de pruebas 96
5.2.2. Descripción del robot 98
5.2.3. Despliegue del software de control 99
5.2.4. Caso de estudio 100
5.2.5. Número de partículas 105
5.2.6. Problema del robot secuestrado 107
5.2.7. Tasa de acierto 108

6. Conclusiones 115

1

Introducción

La robótica autónoma es un campo en rápido crecimiento relacionado con un extenso grupo de líneas de investigación, como localización, *tracking*, *mapping*, planificación de caminos, algoritmos IA, flujo óptico, etc., e interesantes retos para la industria, como nuevas plataformas robóticas, sensores y actuadores. Todo ello enfocado a la realización de tareas cada vez más complejas (búsqueda y desactivación de explosivos, localización de víctimas de catástrofes, asistencia e interacción con personas, transportes...) por medio de robots, o grupos de robots autónomos, trabajando de forma coordinada.

El desarrollo y la implementación de robots autónomos capaces de realizar estas tareas, entraña una gran complejidad para investigadores y desarrolladores. Incluso para la realización de tareas en las que participa un solo robot, es necesaria la interacción de un gran número de elementos (sensores, actuadores, drivers, algoritmos de localización, algoritmos de navegación, etc.).

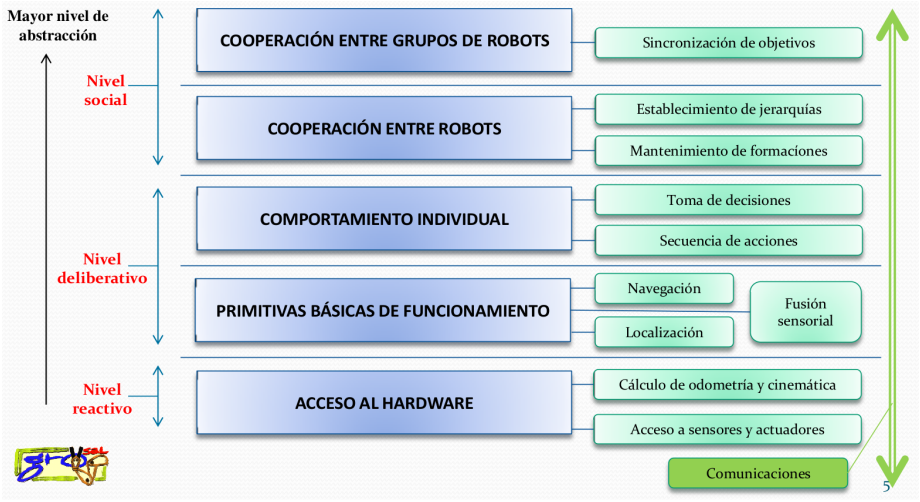


Figura 1.1: Organización de los módulos software para el desarrollo de una misión con robots autónomos

Cuando las tareas requieren de la intervención de varios robots, la comunicación entre ellos, su organización y su coordinación aumentan aún más el número de variables a tener en cuenta y la complejidad del desarrollo.

En general, el desarrollo de misiones llevadas a cabo por robots, comprende un conjunto de módulos software que van desde el acceso a hardware, hasta la coordinación entre distintos grupos de robots para conseguir objetivos. De acuerdo a su nivel de abstracción, podemos organizar todos los módulos en tres niveles: reactivo, deliberativo y social. En la figura 1.1 aparece esta organización, según su nivel de abstracción,

El nivel reactivo comprendería los módulos más básicos necesarios para el funcionamiento de cualquier robot, como el acceso a sus sensores, actuadores, el cálculo de la odometría y su cinemática. Sobre estas primitivas básicas, se apoyan las capacidades del nivel deliberativo. En este nivel de abstracción podemos situar todas aquellas capacidades que le dan una mayor o menor autonomía o *inteligencia* a un robot, como la navegación, el cálculo de su localización o la navegación. Gracias al uso de estas capacidades, los robots pueden alcanzar sus objetivos individuales. Según la complejidad de estos objetivos, puede ser necesario que el robot sea capaz de ejecutar una secuencia de acciones o reaccionar ante estímulos de su entorno cambiando su comportamiento.

Misiones más complejas requieren la colaboración entre varios robots. Para ello, es necesario que puedan organizarse de acuerdo a sus objetivos, estableciendo formaciones, comportándose de acuerdo a distintos roles, o respetando jerarquías. Finalmente, en el nivel de abstracción más alto se encontrarían las misiones que requieren la colaboración y la sincronización de los objetivos entre distintos grupos de robots.

Este aumento en la complejidad de las tareas llevadas a cabo por robots y de los módulos software implicados hace necesaria la evolución hacia entornos de desarrollo que nos faciliten su implementación. Estas plataformas reciben el nombre de Entornos de Desarrollo de Robots (RDE - *Robot Development Environments*) [1] y su objetivo consiste principalmente en ayudarnos a gestionar esta complejidad, proporcionando una plataforma modular, facilitando las comunicaciones entre distintos módulos y robots, la reutilización de funcionalidades, su asociación para la realización de tareas complejas, la puesta en marcha de las misiones, y su monitorización y control.

Así, la implementación de tareas complejas se puede realizar a un gran nivel de abstracción, simplemente combinando adecuadamente módulos de control, localización, navegación o comunicaciones, sin necesidad de conocer al detalle cada uno de ellos. Gracias a los RDEs, no es necesario (o no debería) programar cada uno de los módulos de acuerdo a nuestras necesidades concretas, programar las comunicaciones entre los distintos módulos, desplegar y arrancar el software resultante y programar nuestras propias herramientas de monitorización. De esta manera, los investigadores pueden centrarse en la definición del comportamiento de los robots ante las distintas circunstancias a las que tendrán que hacer frente.

Una misión de ejemplo podría ser la realizada por un grupo de robots móviles (un líder y varios esclavos) que exploran distintas habitaciones de un edificio en busca de un objetivo (una persona herida, material explosivo...). El robot líder posee unas herramientas más avanzadas capaces de ayudar al hombre herido, o desactivar el material explosivo, mientras que los robots esclavos cuentan con un hardware más sencillo que simplemente les permite explorar su entorno. Los robots esclavos tienen que realizar su exploración de forma coordinada e informar al resto una vez que encuentran el objetivo, para que el líder se dirija al lugar correcto y complete la misión con éxito.

Partiendo de cero, o simplemente reutilizando y combinando manualmente distintos algoritmos, es evidente que el desarrollo de una misión así puede suponer varios meses de trabajo. Contando con un entorno de desarrollo adecuado que ya proporcione funcionalidades necesarias como la localización, cálculo de rutas, etc., sólo es necesario definir una máquina de estados para cada robot, utilizando comportamientos de alto nivel como *"encontrar a la víctima / explosivo"* o *"informar de mi posición al resto de robots"*. De esta manera, el trabajo de los desarrolladores se simplifica enormemente y el tiempo de desarrollo disminuye.

Durante la realización de este trabajo de Tesis Doctoral, hemos revisado los entornos de desarrollo de robots de código abierto más importantes, como veremos en la sección 2.1. Utilizando MissionLab, la mayoría de los alumnos de Ingeniería Informática son capaces de implementar la misión de ejemplo en una sola sesión, sin tener ningún conocimiento previo de las herramientas. Sin embargo, el resto de los RDEs estudiados no permiten el desarrollo de una misión de estas características tan fácil y rápidamente, utilizando interfaces gráficas y sin escribir ni una sola línea de código.

Los usuarios en MissionLab pueden utilizar los comportamientos ya existentes para construir misiones complejas con varios robots utilizando un editor gráfico (*CfgEdit*). Esta herramienta permite configurar misiones gráficamente con varios robots, o incluso grupos de robots, donde cada uno está guiado por una máquina de estados finitos (FSM - *Finite State Machine*) y se puede comunicar con el resto. Adicionalmente, MissionLab proporciona un servidor CBR (*Case Based Reasoning*), que puede generar automáticamente planes para la misión, y permite a los robots recibir órdenes de alto nivel en tiempo de ejecución utilizando un lenguaje denominado CMDL (*Command Mission Definition Language*).

Para desarrollar la misión utilizando *CfgEdit*, sólo tenemos que construir gráficamente la máquina de estados de cada robot. Para ejecutarla, tenemos que arrancar el servidor de comunicaciones *iptserver*, los procesos que controlan el hardware del robot (*hserver*) y pulsar el botón de arrancar la misión en *CfgEdit*. Si no vamos a utilizar robots reales y simplemente queremos ejecutar la misión con robots simulados, no es necesario ejecutar los procesos controladores de hardware (*hserver*).

MissionLab ha mostrado sus fortalezas en publicaciones de varias áreas como aprendizaje [2], comportamiento jerárquico [3], formaciones de múltiples robots [4], asignación de tareas en sistemas multi-robot [5] y SLAM [6]. Ademas, MissionLab se ha utilizado también como base para la investigación en el desarrollo de una arquitectura de control para la gestión del comportamiento afectivo, emocional y ético de robots [7] [8] [9]. Su usabilidad se ha puesto a prueba en varios trabajos [1] [10] [11] [12] y el sistema se utiliza en proyectos como MAST (*Micro Autonomous Systems and Technology*) [13] [14] o misiones robóticas para contrarrestar las armas de destrucción masiva (c-WMD) para la DTRA (*Defense Threat Reduction Agency*) [15] [16] [17] [18] [19] en las que el software generado por MissionLab se verifica utilizando álgebra de procesos [20] [21]. Publicaciones recientes [22] [23] indican que hay trabajos en curso centrados en el análisis formal de las misiones generadas para tener mayores garantías de éxito.

Sin embargo, si queremos trabajar con MissionLab en lugar de con otros RDEs, debemos conocer también sus inconvenientes. Las dos principales debilidades de MissionLab son que su última versión oficial pública funciona sobre una distribución Linux sin soporte desde 2006 (Fedora Core 4) y que tiene muchas limitaciones en la creación y el uso de mapas para localización y navegación. Para solucionar las limitaciones de MissionLab manejando mapas, decidimos integrar esas funcionalidades de otro RDE de código abierto.

Tras analizar los diferentes RDEs, comprobamos que una de sus principales carencias se encuentra en el sistema de localización global en interiores, dado que el sensor GPS no funciona en estos entornos y existen muchas dificultades cuando no se desea o no se puede intervenir en el entorno de trabajo, por ejemplo, colocando marcas. Además, la localización es la pieza fundamental que sustenta la navegación y la construcción de mapas dentro del marco de los robots móviles autónomos. Varios algoritmos abordan el problema de la localización global en interiores utilizando filtros de Markov [24] o filtros de Kalman multi-hipótesis [25], pero los algoritmos más estudiados y probados se basan en filtros de partículas [26]. Los filtros de partículas, al igual que otros algoritmos de localización global, utilizan el modelo de movimiento del robot y medidas externas para encontrar la posición del robot.

Para que estos algoritmos tengan éxito, es necesario que el movimiento del robot esté modelado correctamente, y que las medidas externas proporcionen información suficiente. Comportamientos no modelados del robot o medidas erróneas pueden afectar en gran medida a los resultados obtenidos.

En muchos casos, no es posible modelar todos los posibles movimientos del robot o, al menos, es muy complicado hacerlo porque dependen de factores difíciles de controlar (choques, deslizamientos...). Esto sucede a menudo, por ejemplo, con robots que participan en competiciones deportivas, robots con mucha aceleración, irregularidades en la superficie del movimiento, etc. En estas situaciones, al igual que sucede con el problema del *robot secuestrado*, es posible que el filtro de partículas falle porque todas las partículas se encuentren en lugares erróneos, y es necesario utilizar soluciones adicionales para obtener resultados correctos.

Varios autores han propuesto modificaciones para el algoritmo de localización de Monte-Carlo (MCL) [27] intentando resolver este problema, generalmente introduciendo partículas en el filtro de forma independiente al modelo del movimiento del robot o utilizando técnicas de *clustering* [28].

La solución más simple, que en algunos casos puede proporcionar buenos resultados, es la introducción de partículas en posiciones aleatorias. Cuando alguna de esas partículas está lo suficientemente cerca de la situación real del robot, las posibilidades de que el filtro termine proporcionando un resultado correcto aumentan. Uno de los problemas que tiene este método es que requiere que el número de partículas añadidas sea suficientemente grande como para tener una probabilidad razonable de que alguna de las partículas añadidas se sitúe cerca de la posición real del robot. Si usamos este método para localización global en un área muy extensa, la cantidad de partículas necesarias puede llegar a ser demasiado alta. Esto puede hacer imposible su actualización del filtro en tiempo real o, al menos, retrasar mucho su convergencia y la obtención de resultados.

Una mejora a este método es el algoritmo SRL (*Sensor Resetting Localization*) [29], que consiste en la introducción de partículas adicionales de acuerdo a la distribución de probabilidad proporcionada por las lecturas de los sensores. En este algoritmo, el número de partículas añadidas depende de la estimación del error en el filtro, basada en el estado del robot y las lecturas de los sensores.

Aunque este método puede mejorar la convergencia de los filtros de partículas y solucionar el problema del *robot secuestrado* de forma muy eficiente, también puede causar resultados no deseados cuando se trabaja con errores persistentes en las medidas [30]. Como este método se basa en la introducción de partículas según la distribución de probabilidad instantánea proporcionada por los sensores, si las medidas de los sensores son incorrectas, la posición de esas partículas añadidas también lo será.

En el algoritmo SRL, las partículas adicionales se evalúan junto con las demás, utilizando las medidas de los sensores. Esto puede causar serios problemas porque medidas incorrectas de los sensores provocan que las partículas añadidas se coloquen en lugares incorrectos del mapa. Además, estas nuevas partículas tendrán ventaja sobre el resto durante la siguiente fase de corrección. Como se situaron en los puntos en los que es más fácil obtenerlas, esas mismas medidas u otras muy similares al estar recogidas instantes después, harán que el filtro les asigne una probabilidad más alta que al resto. En este caso, los resultados obtenidos podrían ser peores que utilizando un filtro de partículas estándar.

Las medidas externas que necesitan los algoritmos de localización se obtienen normalmente utilizando sensores láser [31] [32], transceptores inalámbricos [33] [34] o cámaras utilizando algoritmos de detección de patrones [35]. En la mayoría de los casos, la implementación de estas soluciones requiere un estudio previo del entorno del robot (tomando imágenes, creando mapas basados en las lecturas láser) o la inserción de balizas. Por esa razón, hemos intentado también encontrar una solución que no presente estos problemas para complementar nuestro RDE integrado. Para ello, decidimos utilizar la información proporcionada por los planos de los edificios en los que se mueve el robot (habitaciones, ventanas, puertas, muros), algoritmos para detectar esos elementos, y realizar mejoras sobre los algoritmos de localización existentes, de manera que nuestro método sea eficiente en entornos muy extensos y ante la presencia de errores en los sensores o en el mapa.

1.1. Objetivos de este trabajo

El principal objetivo de la Tesis es la creación de un nuevo RDE, resultante de la actualización y mejora de MissionLab, así como de su integración con otro RDE existente que le aporte, principalmente, funcionalidades de localización y navegación basadas en mapas. El RDE resultante tendrá que cumplir con los siguientes objetivos:

- Debe ser compatible con desarrollos que dependan de las versiones originales de los RDEs integrados.
- Debe conservar las capacidades multi-robot de MissionLab
- Debe proporcionar funcionalidades de localización y navegación basadas en mapas
- Debe permitir usar las funcionalidades existentes y las nuevas en el editor gráfico de misiones, sin se necesiten conocimientos avanzados de programación
- Debe ser compatible con versiones recientes de las principales distribuciones Linux

Dentro de nuestro trabajo de Tesis Doctoral, se ha desarrollado también un nuevo algoritmo de localización global en interiores que es capaz de trabajar en entornos extensos, como puede ser cualquiera de los edificios de la Universidad de Salamanca, tomando como entradas un mapa topológico del entorno, la odometría del robot e información semántica proporcionada por un algoritmo de detección de puertas, que ha sido desarrollado recientemente por el Grupo de "Robótica y Sociedad" de la Universidad de Salamanca.

Se demostrarán la validez del algoritmo propuesto y sus ventajas respecto a los algoritmos de localización global existentes en la actualidad, en cuanto a que permite trabajar en entornos extensos con un tiempo de respuesta razonable.

Además, se han validado el nuevo RDE y el algoritmo de localización sobre distintas plataformas robóticas:

- El robot *MORLACO*, que cuenta con un láser de rango y una cámara,

además de sensores propioceptivos, construido íntegramente dentro del GIR "Robótica y Sociedad".

- Una carretilla paletizadora con un láser de rango, diversos sensores sonar y una cámara, además de sensores propioceptivos, robotizada dentro del GIR "Robótica y Sociedad".

1.2. Estructura de la memoria

Esta tesis se organiza de la siguiente manera. En el capítulo 2 analizamos los principales entornos de desarrollo de robots de código abierto disponibles en la actualidad (apartado 2.1), para mostrar que casi todos ellos requieren conocimientos avanzados de programación (en oposición con los objetivos de nuestra propuesta). Además, estudiamos los principales conceptos relacionados con la localización global en interiores mediante filtros de partículas en la sección 2.2 y los relacionados con los Sistemas de Información Geográfica (sección 2.3).

En el capítulo 3 exponemos el diseño, desarrollo, características y funcionamiento de nuestro nuevo RDE integrado. En la sección 3.1, mostramos las especificaciones y el diseño que han guiado el desarrollo. En el apartado 3.2 presentamos las principales tareas que hemos tenido que llevar a cabo para dar soporte a nuestro diseño. Dentro de estas tareas se encuentran la preparación de MissionLab para su uso en distribuciones Linux actuales (apartado 3.2.1), la creación del módulo *IPCAdapter* para la gestión de las comunicaciones (apartado 3.2.2), las mejoras necesarias en la biblioteca IPC (apartado 3.2.3) y la nueva funcionalidad que permite a MissionLab tomar el control de los módulos de CARMEN interceptando sus mensajes (apartado 3.2.4). Finalmente, en la sección 3.3, explicamos el funcionamiento de la arquitectura, tanto a bajo nivel (uso de drivers, sensores y actuadores) en el apartado 3.3.1, como a alto nivel (uso de funciones de localización y navegación) en el apartado 3.3.2.

Posteriormente, en el capítulo 4 presentamos nuestra propuesta de localización global en interiores, analizando cada una de las mejoras que hemos implementado sobre los filtros de partículas estándar y comentando su funcionamiento.

En el capítulo 5 evaluamos nuestro RDE integrado (apartado 5.1) y nuestro nuevo algoritmo de localización global (apartado 5.2). La evaluación de nuestro RDE integrado se ha realizado mediante una serie de pruebas en los que se comprueba la compatibilidad con las versiones oficiales de MissionLab y CARMEN (apartados 5.1.1 y 5.1.2) y otras en las que se comprueba la integración entre ambos RDE trabajando en misiones multi-robot, tanto con robots simulados como con robots reales (apartados 5.1.3 y 5.1.4). La evaluación del sistema finaliza poniendo a prueba su fiabilidad en una misión con una carretilla elevadora industrial, descrita en el apartado 5.1.5.

Para evaluar nuestro nuevo algoritmo de localización global, en el apartado 5.2.4 exponemos un caso de uso realizado en condiciones adversas, en el que se comprueba que el método funciona correctamente en entornos de grandes dimensiones y es resistente ante fallos o inexactitudes de los sensores y errores en el mapa. Adicionalmente, analizamos el comportamiento del algoritmo según el número de partículas utilizado (sección 5.2.5), su comportamiento ante el problema del *robot secuestrado* (sección 5.2.6) y su tasa de acierto (sección 5.2.7).

En la sección 6 exponemos nuestras principales conclusiones y finalmente presentamos la bibliografía.

2

Estado del Arte

En este capitulo vamos a analizar los antecedentes relacionados con los objetivos de investigación que se abordan en esta Tesis Doctoral: los Entornos de Desarrollo de Robots (RDEs) y la localización global en interiores mediante filtros de partículas. Por ello, se realizará una evaluación y comparación de los RDEs de código abierto disponibles en la actualidad. Posteriormente, presentaremos los principales conceptos sobre filtros de partículas junto con el análisis de los trabajos más sobresalientes de su aplicación al problema de la localización global en interiores. Como complemento, se hará una revisión sobre los Sistemas de Información Geográfica (GIS), pues sirven de base a nuestra propuesta de solución para la localización global en interiores.

2.1. Evaluación de RDEs existentes

Los RDEs son plataformas modulares que ayudan a los desarrolladores de misiones para robots o grupos de robots a implementar tareas complejas de una forma más sencilla. Facilitan las comunicaciones entre distintos módulos y robots, la reutilización de funcionalidades, su asociación para la realización de tareas complejas, la puesta en marcha de las misiones y su monitorización y control.

Actualmente, existen RDEs muy populares [36], como ROS [37], OROCOS [38], Player/Stage/Gazebo [39], CARMEN [40], etc. que están basados en diseños altamente modulares. Por lo general, los módulos en los que se basan estos entornos de desarrollo tienen una interfaz de entrada, un interfaz de salida y parámetros de configuración. De esta manera se pueden combinar unos módulos con otros, aumentando la escalabilidad para añadir nuevos algoritmos y drivers.

Gracias a los RDEs, tenemos una gran cantidad de funcionalidades disponibles para ayudarnos a realizar nuestros desarrollos robóticos. Sin embargo, aunque un robot sea capaz de realizar una gran cantidad de tareas individuales (detectar patrones, coger objetos, abrir puertas, etc.), esto no garantiza que sea más autónomo o más *inteligente*. Para serlo, necesita combinar de forma adecuada estas capacidades individuales y utilizarlas cuando sea conveniente.

Si tenemos un robot que puede hacer decenas o cientos de pequeñas tareas y queremos que haga algo inteligente para aprovecharlas, parece obvio que necesitaremos herramientas que nos ayuden a modelar el comportamiento del robot. Estas herramientas pueden estar basadas en máquinas de estados finitos, razonamiento basado en casos o cualquier otra técnica de Inteligencia Artificial. Esta misma necesidad también surge cuando se construyen misiones donde están implicados decenas o cientos de robots. Cuando se trabaja con equipos de robots en lugar de con un único robot, nos encontramos con el mismo reto pero a un nivel de abstracción más alto. Si queremos que un equipo de robots realice una misión de forma eficiente, necesitamos sincronizar y acomodar el comportamiento de cada robot para que ayude a los demás a lograr los objetivos de la misión.

Sin embargo, utilizando la mayoría de los RDEs disponibles actualmente, para desarrollar una misión como la propuesta en la introducción (con varios robots con roles diferentes) se necesita escribir el código para las comunicaciones entre los distintos robots, el código que los sincronice y los coordine, o incluso implementar la máquina de estados completa de cada robot. Además, para poner en marcha estas misiones puede ser necesario arrancar manualmente decenas de módulos o crear un fichero de despliegue personalizado. Esto supone una gran cantidad de trabajo extra y la necesidad de contar con desarrolladores con suficientes conocimientos de programación y de detalles internos específicos del entorno de desarrollo utilizado. Sin estos obstáculos, sería posible centrarse únicamente en el diseño del comportamiento de los robots y la lógica de la misión, y el trabajo lo podrían desarrollar personas especialistas en estos campos, que no necesariamente tienen por qué ser programadores expertos.

El principal punto fuerte de MissionLab es precisamente la gestión de esta complejidad mediante un editor de misiones gráfico, generadores de código, el soporte multi-robot, la gestión automática de los distintos módulos y sus comunicaciones y la puesta en marcha de las misiones y su monitorización de una forma sencilla. Por ejemplo, es posible añadir más robots a una misión con una simple operación de *copiar y pegar* en su editor de misiones *CfgEdit* y dándole el nombre del servidor de hardware del nuevo robot al iniciar la misión. También es posible compartir información entre los robots gracias a comportamientos predefinidos.

Otros RDEs como ROS, OROCOS o Player no disponen de estas ventajas (figura 2.1). Revisando su documentación oficial, es obvio que son RDEs para desarrolladores con conocimientos de programación y de la arquitectura subyacente; como por ejemplo, las interfaces de programación de los distintos módulos, su sistema de comunicaciones, etc.

Pensamos que, para conseguir un mayor desarrollo de la robótica y la implementación de robots con comportamientos realmente complejos y de misiones multi-robot avanzadas, se necesita un nivel de abstracción más alto. No es realista pensar en el desarrollo de robots inteligentes que sean capaces de realizar cientos de tareas e interaccionar con su entorno como lo hacen los humanos, o el desarrollo de misiones complejas con cientos de robots, si tenemos que tra-

	Editor gráfico de misiones multi-robot	Creación y manejo de mapas para localización y navegación
MissionLab	Sí	No (muy limitado)
CARMEN	No	Sí
ROS	No (sólo editores de archivos roslaunch)	Sí
Player/Stage	No	Localización y navegación pero no creación de mapas
OROCOS	No	No

Cuadro 2.1: Tabla comparativa de características de los RDEs evaluados

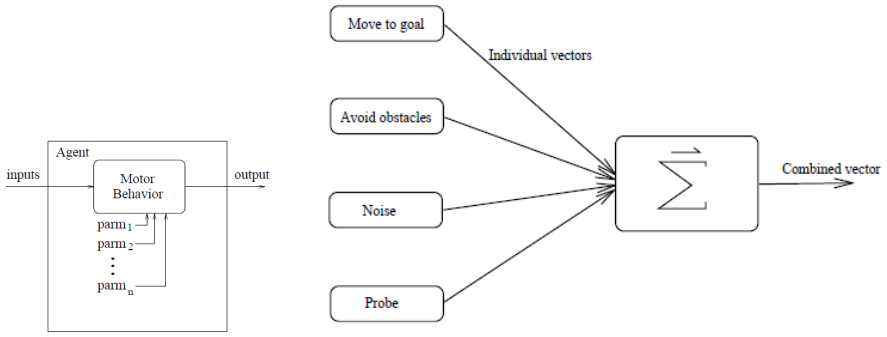
tar de forma manual con aspectos de tan bajo nivel como la programación y compilación independiente del software de cada comportamiento y cada robot, programar cada envío y recepción de mensajes, o gestionar el arranque, la parada y la coordinación de la enorme red de módulos software necesaria.

A continuación, vamos a hacer un análisis de los principales RDEs de código abierto disponibles en la actualidad. Veremos el grado de funcionalidad que nos proporcionan y su funcionamiento básico.

2.1.1. MissionLab

MissionLab es un conjunto de herramientas software para desarrollar y probar comportamientos para un único robot o grupos de robots basados en la teoría de Agentes Sociales [41]. Esta teoría define los agentes básicos o atómicos como comportamientos que reciben una serie de datos de entrada, están configurados con una serie de parámetros y generan una salida. También define la posible asociación entre varios de estos agentes, mediante distintos operadores, para la obtención de comportamientos más complejos. En la figura 2.1 aparece la representación de un agente básico (2.1(a)) y de un agente compuesto (2.1(b)) utilizando el operador de sumatorio.

MissionLab cuenta con cinco componentes principales: *Mlab*, *CfgEdit*, *Robot Executable* (en adelante simplemente *ejecutables*), *HServer* y *CBRServer*.



(a) (b)

Figura 2.1: a) Agente atómico b) Agente compuesto

Mlab permite a los usuarios monitorizar misiones, teleoperar robots y es capaz de generar datos simulados para probar misiones en simulación. En la figura 2.2 aparace la interfaz de monitorización después de haber cargado el mapa de la segunda planta de la Facultad de Ciencias de la Universidad de Salamanca. Podemos ver también su interfaz de teleoperación activa, con controles para manejar el robot mediante un joystick o un ratón, o interpretar las órdenes de movimiento como relativas al mapa o al robot, entre otras opciones.

CfgEdit es una herramienta gráfica para construir misiones complejas con varios robots, haciendo uso de los comportamientos existentes. En la figura 2.3 se observa su interfaz en el momento de añadir un nuevo comportamiento a una misión. Las misiones se representan mediante una máquina de estados, en la que los comportamientos (estados) se representan mediante círculos y los disparadores (las condiciones de transición entre distintos estados) se representan mediante rectángulos. Tanto los parámetros de los comportamientos como los de los disparadores se pueden configurar de forma gráfica. Una vez que se ha finalizado el diseño de la misión, se utiliza el botón *Compilar* a la izquierda de la imagen, y posteriormente el botón *Run* para ejecutarla. De esta forma, gráfica y simple, se puede definir una misión de un robot individual o un grupo, sin que el operador de la misión tenga que tener conocimientos de programación, y así puede centrar sus esfuerzos en refinar la misión.

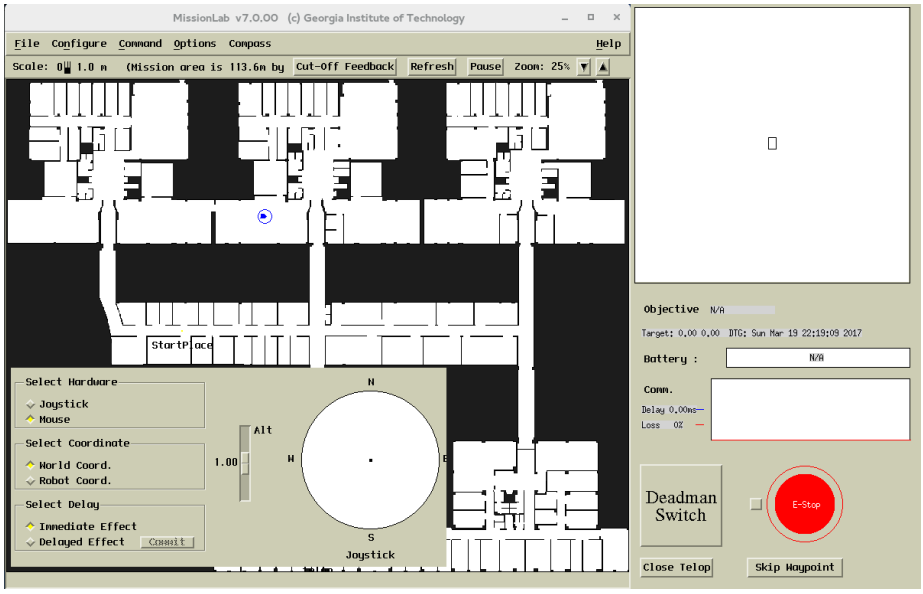


Figura 2.2: Visualizador de misiones MLab

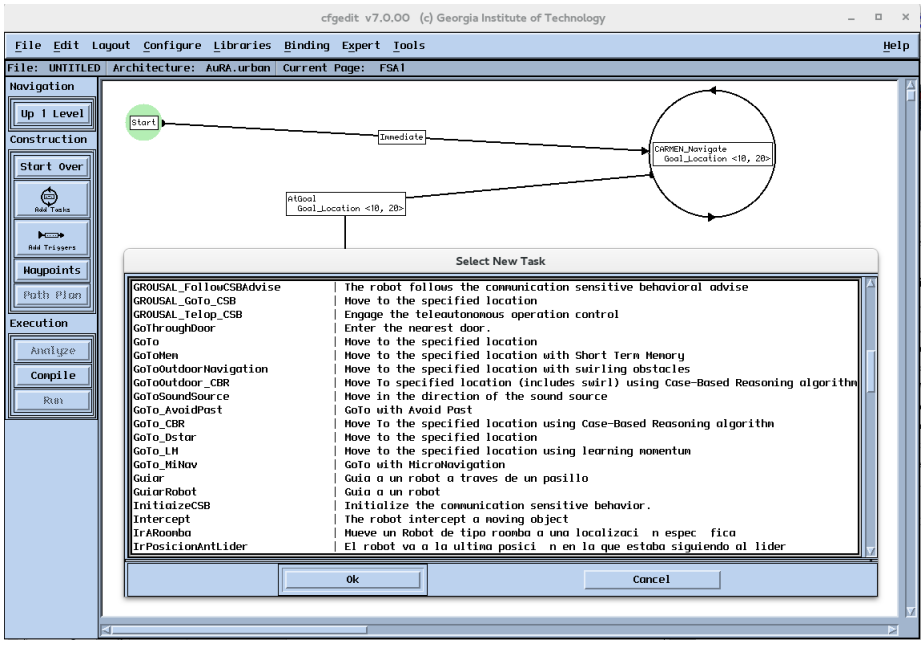


Figura 2.3: Herramienta de creación de misiones CfgEdit

```
GROUSAL_GoTo:[
 %Goal_Location = {1.0, 1.0},
 %move_to_location_gain = {1.0},
 %avoid_obstacle_gain = {1.0},
 %avoid_obstacle_sphere = {3.0},
 %avoid_obstacle_safety_margin  = {0.5},
 COOP(
  %Goal_Location = {^}, %classes = {0}, %avoid_obstacle_sphere = {^},
  %avoid_obstacle_safety_margin  = {^}, %max_sensor_range = {1000},
  members[A] = $MoveToGoal, weight[A] = {^Range_01 %move_to_location_gain},
  members[B] = $Avoid_Obstacles, weight[B] = {^Range_01 %avoid_obstacle_gain},
  members[C] = $GROUSAL_Telop,   weight[C] = {1.0}
 )|GROUSAL go to|
]|Move to the specified location using advanced teleoperation|

$GROUSAL_Telop:[
 %max_sensor_range = {^},
 %classes = {^},
 GROUSAL_TELOP(
  telop_mode = DATABASE_INT(key = {"telop_mode"}, initial = {0} ),
  robot_heading = GET_HEADING(cur_pos = $RobotLocation),
  joystick_mag = DATABASE_DOUBLE(key = {"joystick_magnitude"}, initial = {0.0}),
  joystick_x   = DATABASE_DOUBLE(key = {"joystick_x"},  initial = {0.0}),
  joystick_y   = DATABASE_DOUBLE(key = {"joystick_y"},  initial = {0.0}),
  joystick2_x  = DATABASE_DOUBLE(key = {"joystick2_x"}, initial = {0.0}),
  joystick2_y  = DATABASE_DOUBLE(key = {"joystick2_y"}, initial = {0.0}),
  slider_1     = DATABASE_DOUBLE(key = {"slider_1"},    initial = {0.0}),
  slider_2     = DATABASE_DOUBLE(key = {"slider_2"},    initial = {0.0}),
  joy_buttons  = DATABASE_INT   (key = {"joy_buttons"}, initial = {0})
 )
]|move the robot towards the joystic direction|
```

Figura 2.4: Definición de comportamientos CDL

Los *ejecutables* son generados automáticamente por *CfgEdit* mediante una compilación en tres fases, utilizando lenguajes intermedios denominados CDL y CNL. CDL se utiliza para definir de forma recursiva una sociedad de agentes y CNL para modelar los módulos que integran una misión y el flujo de datos entre ellos [42]. Sin embargo, estos lenguajes sólo se usan directamente cuando se quieren añadir nuevos comportamientos a MissionLab. Su uso es transparente para el usuario cuando se trabaja con comportamientos predefinidos.

En la figura 2.4 se muestra la definición de dos comportamientos CDL creados durante el desarrollo de este trabajo, que permiten la navegación autónoma de un robot y también su teleoperación.

El editor de misiones CfgEdit muestra de forma gráfica y permite configurar estos comportamientos CDL. Una vez generada una misión, su compilación a lenguaje CNL da lugar a la creación de múltiples comportamientos individuales con sus entradas y salidas enlazadas automáticamente. El aspecto de este código CNL autogenerado se muestras en la figura 2.5.

```
node $AN_3136 is DRIVE_W_SPIN |The Wheels Binding Point| with
 v = FSA1; max_vel = {5.0}; base_vel = {2.5};
 cautious_vel = {0.5}; cautious_mode = {false};
nend

node FSA1 is FSA1_proc |The State Machine| with
 triggers = $AN_1683; // INMEDIATE
 triggers = $AN_3259; // ATGOAL
 society = $AN_1669;  // START
 society = $AN_1577;  // GROUSAL_GOTO (COOP)
 society = $AN_1669;  // STOP
nend

//GROUSAL_GOTO
node $AN_1577 is COOP |go to| with
 members = $AN_3179; //MoveToGoal
 members = $AN_3189; //AvoidObstacles
 members = $AN_3235; //GROUSAL_TELOP
 weight = {0.4};
 weight = {0.6};
 weight = {1.0};
nend

node $AN_3235 is GROUSAL_TELOP with
 telop_mode = $AN_3200;    //DATABASE_INT
 robot_heading = $AN_3205; //GET_HEADING
 joystick_mag = $AN_3234;  //DATABASE_DOUBLE
 joystick_x  = $AN_3209; joystick_y  = $AN_3213; //DATABASE_DOUBLE
 joystick2_x = $AN_3217; joystick2_y = $AN_3221; //DATABASE_DOUBLE
 slider_1    = $AN_3225; slider_2    = $AN_3229; //DATABASE_DOUBLE
 joy_buttons = $AN_3233;   //DATABASE_INT
nend
```

Figura 2.5: Código CNL autogenerado

Los comportamientos CNL básicos cuentan con una implementación en C++ embebida en código CNL. Puesto que estos comportamientos se ejecutan repetidas veces en un bucle, su implementación especifica las instrucciones a ejecutar en cada iteración. En la figura 2.6 podemos ver la implementación de uno de estos comportamientos básicos.

Cuando, en la segunda fase de compilación de una misión, el código CNL de los comportamientos básicos y el autogenerado en la fase anterior se compilan en código C++, se obtiene, para cada comportamiento, un código C++ como el que se muestra en la figura 2.7. Como se puede apreciar, el propio compilador CNL inserta el código del comportamiento básico en un bucle infinito, añade el código necesario para la obtención de parámetros, para la devolución del resultado, y mecanismos de sincronización que permiten al sistema ejecutar el comportamiento sólo cuando es necesario.

```
procedure Vector MOVE_TO_GOAL with
   Vector  goal_relative_loc;
   double  success_radius;
header
   // optional user initialization code
body
   // user C++ code
   if( len_2d(goal_relative_loc) > success_radius )
   {  // generate a vector towards the goal
      output = goal_relative_loc;
      unit_2d(output);
   }
   else
   {  // return a zero vector if within the success circle
      VECTOR_ZERO(output);
   }
pend
```

Figura 2.6: Implementación de un comportamiento básico CNL

```
void MOVE_TO_GOAL(int parm)
{
   Vector  output;
   struct T_MOVE_TO_GOAL *parms = (struct T_MOVE_TO_GOAL *)parm;
   double success_radius;
   Vector goal_rel_loc;
   /******** start of user header ********/
   /******** end of user header ********/
   while(1)
   {
      if( parms->success_radius_chk)
      {
         while( parms->success_radius_last_seq ==
        *parms->success_radius_seq)
            cthread_yield();
         parms->success_radius_last_seq = *parms->success_radius_seq;
      }
      success_radius = *parms->success_radius;
      if( parms->goal_rel_loc_chk)
      {
         while(parms->goal_rel_loc_last_seq == *parms->goal_rel_loc_seq)
            cthread_yield();
         parms->goal_rel_loc_last_seq = *parms->goal_rel_loc_seq;
      }
      goal_rel_loc = *parms->goal_rel_loc;
      /******** start of user body ********/

      if( len_2d(goal_relative_loc) > success_radius )
      { // generate a vector towards the goal
        output = goal_relative_loc;
        unit_2d(output);
      } else   // return a zero vector if within the success circle
        VECTOR_ZERO(output);

     /******** end of user code ********/
     parms->output = output;
     parms->seq++;
     mutex_lock(_out_count_lock);
      condition_wait(_new_cycle,_out_count_lock);
      mutex_unlock(_out_count_lock);
   }
}
```

Figura 2.7: Código C++ de un comportamiento generado por MissionLab

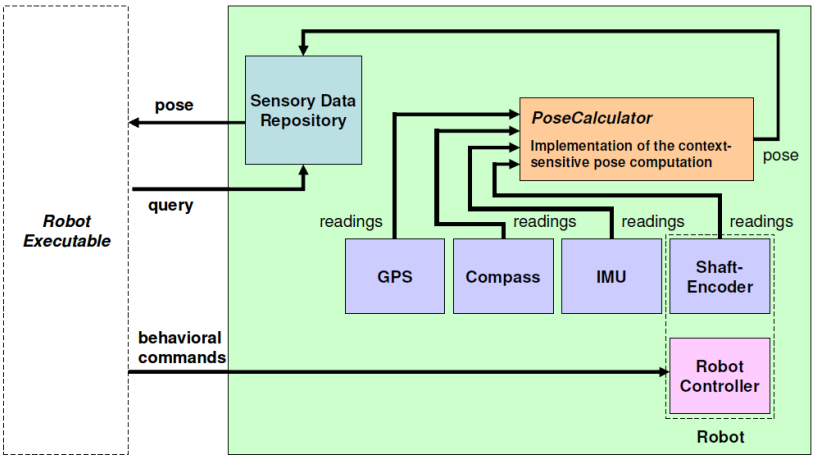


Figura 2.8: Componentes de *HServer*

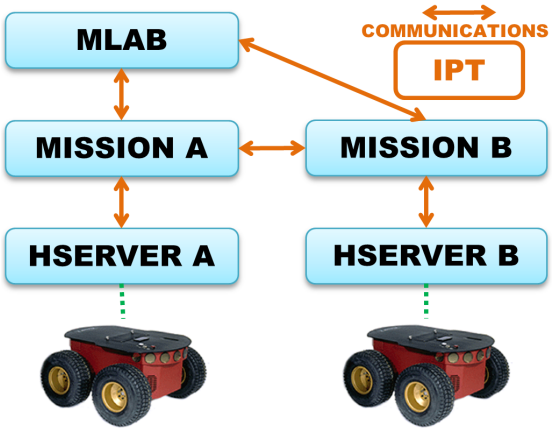


Figura 2.9: Modelo de ejecución de MissionLab

HServer (*Hardware Server*) controla directamente todo el hardware de los robots y proporciona una interfaz estándar para cada robot y cada sensor. También estima la posición del robot, integrando información de distintas fuentes por medio de varios algoritmos, como filtros de Kalman y de partículas. En la figura 2.8 se pueden ver los principales componentes de HServer [43].

CBRServer (*Case Base Reasoning Server*) genera planes de misión basándose en especificaciones dadas por los usuarios mediante la recuperación y el ensamblaje de componentes de planes previamente almacenados de misiones que tuvieron éxito. Se utiliza un método extendido [14] de razonamiento basado en casos [44] que, incluso, permite reparar las misiones generadas [11].

En el modelo de ejecución de MissionLab (mostrado en la figura 2.9), cada *ejecutable* dirige su propio robot utilizando una instancia de *HServer*. Los *ejecutables* (*MISSION* en la figura) pueden comunicarse entre sí y con *Mlab*, por medio de IPT (*InterProcess communications Toolkit*), para informar sobre su estado o recibir nuevas órdenes.

Consideraciones finales sobre la evaluación de MissionLab

Los puntos fuertes de MissionLab son sus características de alto nivel (creación automática de misiones, generación de código), su herramienta *CfgEdit*, y la integración de la información de posicionamiento desde distintas fuentes llevada a cabo por *HServer*. Sin embargo, no utiliza mapas para la localización, y el uso que hace de los mapas para la navegación es muy limitado.

En entornos exteriores, podemos utilizar un GPS para la localización global, pero en interiores, la localización de MissionLab diverge. Esto limita la precisión, la duración y la complejidad de las misiones que podemos implementar. La navegación basada en mapas tampoco es una fortaleza de MissionLab. Tiene la posibilidad de calcular rutas *offline* utilizando un algoritmo A*, pero necesita un archivo especial para el mapa. Esta función está desactivada por defecto en *CfgEdit*. Una vez que se inicia la misión, el camino es fijo y no se recalcula bajo ninguna circunstancia.

MissionLab tiene otra función para la navegación *online* que se basa en el algoritmo D* *Lite*. Se utiliza sólo en un comportamiento de MissionLab (*GoTo_DStar*) y sus datos no se publican a través del servidor de mensajes (*iptserver*). Por tanto, otros comportamientos no pueden aprovechar la información de navegación que proporciona. Cuando se trabaja en entornos conocidos, su utilidad es limitada, especialmente en interiores, puesto que no está respaldado por una localización precisa basada en mapas [45].

No existe ningún tipo de herramienta para crear, editar o visualizar los mapas utilizados por esta funcionalidad o las rutas de navegación generadas por esta funcionalidad (ni tampoco la que utiliza el algoritmo A*). Por ello, su presencia en MissionLab parece deberse más a la resolución un problema concreto y puntual, que a ser una característica para uso general proporcionada por el sistema.

Junto con las herramientas gráficas y los generadores automáticos de código para desarrollar comportamientos complejos del robot, el manejo de misiones multi-robot es también una fortaleza clave de MissionLab. En otros RDEs, como ROS, OROCOS, CARMEN o Player, los usuarios tienen que hacer frente a posibles conflictos, tienen que definir mensajes y tienen que implementar la lógica de comunicaciones de la misión dentro del código fuente, mientras que MissionLab hace todo este trabajo de forma transparente para el usuario.

2.1.2. CARMEN

CARMEN (*Carnegie Mellon Robot Navigation Toolkit)* es una colección de herramientas de código abierto desarrollada por la Universidad de Carnegie Mellon. Se distribuye bajo licencia GPL. Es un software modular, diseñado para proporcionar primitivas básicas de navegación incluyendo: control de la base robótica y de los sensores, registro de *logs*, evitación de obstáculos, localización, planificación de caminos y *mapping*. Los módulos de CARMEN están organizados siguiendo una arquitectura de tres capas [40]: acceso a hardware, control e interfaces.

Los módulos de la capa inferior interaccionan directamente con el hardware del robot, proporcionando interfaces abstractas para la base y los sensores. Además, calculan la odometría y pueden ejecutar movimientos simples de rotación y línea recta. Esta capa incluye *drivers* para una amplia gama de bases comerciales de robots, y una plataforma de simulación para todos ellos. Los módulos de la segunda capa implementan las primitivas de navegación y localización del robot. La tercera capa se reserva para tareas de nivel de usuario que pueden utilizar los servicios proporcionados por los módulos de la segunda capa.

Las comunicaciones entre los módulos de CARMEN se realizan usando un paquete separado denominado IPC (*Inter-Process Communication System*). A pesar de que IPC se distribuye junto con CARMEN, se trata de un software desarrollado de forma independiente. La madurez y la estabilidad de IPC hacen que CARMEN sea un sistema muy fiable. IPC soporta entornos multi-hilo (*multithread*) y conexiones con varios servidores IPC, aunque no soporta ambas cosas a la vez. Este ha sido uno de los problemas que hemos tenido que resolver para conseguir la integración completa de MissionLab y CARMEN.

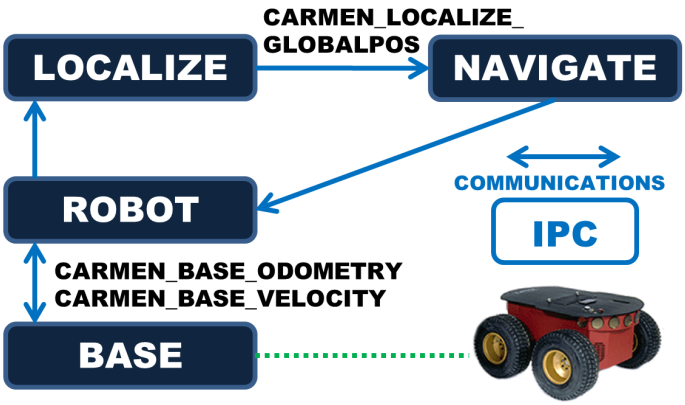


Figura 2.10: Modelo de ejecución de CARMEN

La Figura 2.10 muestra el modelo de ejecución básico de CARMEN, en el que los distintos módulos (*base*, *robot*, *localize* y *navigate*) cooperan comunicándose mediante mensajes IPC. El módulo *base* accede directamente al hardware del robot, envía mensajes IPC con información sobre los sensores y recibe los mensajes para controlar los actuadores. El módulo *robot* envía estos mensajes de control, recibe los mensajes de *base* y proporciona una interfaz común para todo tipo de robots. Además, recibe instrucciones de otros módulos como *robotgui* (cuando el robot está siendo teleoperado) o *navigator* (cuando el robot se mueve de forma autónoma) y los transmite al módulo *base* usando el mensaje *CARMEN_BASE_VELOCITY* de IPC. Dicho módulo proporciona datos de odometría al resto usando el mensaje *CARMEN_BASE_ODOMETRY*, y también implementa un detector elemental de colisiones que es capaz de detener el robot en la proximidad de los obstáculos. El módulo *localize* recibe datos de odometría y los datos del láser que genera el módulo *robot* y envía la posición global del robot que ha estimado utilizando el mensaje *CARMEN_LOCALIZE_GLOBALPOS*. En CARMEN se considera que esta localización es la más fiable y se usa en el resto de los módulos cuando tienen que tomar cualquier decisión. Por ejemplo, el módulo *navigate* recibe esta posición, calcula el camino a seguir y envía las órdenes de movimiento necesarias al módulo *robot*.

CARMEN proporciona herramientas gráficas muy intuitivas para crear mapas (*vasco*) (figura 2.12), editar los mapas resultantes (*map_editor*) y usarlos en tareas de navegación (*navigatorgui*) (figura 2.11).

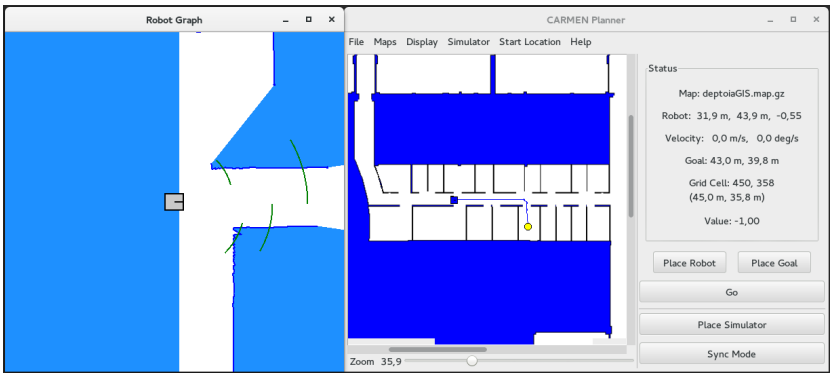


Figura 2.11: Herramientas gráficas *robotgui* y *navigatorgui* de CARMEN

Su manejo de mapas es también muy notable. Contiene una implementación del constructor de mapas de Hähnel [46] para crear mapas con información sobre zonas libres y ocupadas (permitiendo utilizar probabilidades intermedias), soporta límites virtuales para la navegación y puede identificar lugares usando sus nombres. También, permite asociar diferentes mapas utilizando puertas o ascensores. CARMEN puede localizar los robots usando datos procedentes de un láser de rango y un filtro de partículas. La navegación en CARMEN utiliza sus mapas para calcular caminos en tiempo de ejecución, y puede cambiarlos si detecta que están bloqueados. Para ello utiliza el planificador de *gradiente descendiente* de Konolige [47], propuesto por Thrun et al. (2001) [27] en combinación con Fox et al. (1997) [48], que ha sustituido al método previo porque tenía problemas de fiabilidad.

En CARMEN, toda la información sobre el mapa, localización y navegación es accesible desde cualquier otro módulo conectado al sistema, utilizando un mecanismo de suscripción proporcionado por IPC. La herramienta para hacer mapas que proporciona CARMEN (*vasco*) utiliza *logs* con datos de odometría y láser para generar mapas precisos, gracias a su algoritmo de *scan matching*. Para corregir los pocos fallos existentes en el mapa generado, *vasco* permite al usuario hacer cambios como descartar datos inválidos o rotar/trasladar el mapa. La herramienta *map_ editor* permite hacer cambios más avanzados. Los usuarios pueden cambiar las probabilidades asociadas a cada punto, especificar nombres de lugares, definir barreras virtuales o incluso crear un mapa desde cero.

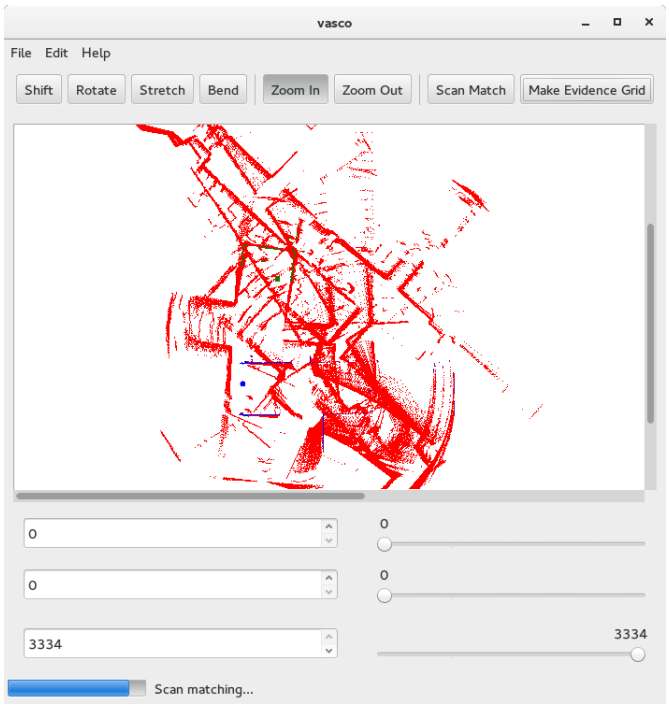


Figura 2.12: Herramientas de generación de mapas de CARMEN (*vasco*)

Consideraciones finales sobre la evaluación de CARMEN

Los principales puntos fuertes de CARMEN son sus funcionalidades relacionadas con mapas (localización, navegación y *mapping*) y su fácil usabilidad e instalación, con muy pocas dependencias. Sin embargo, para utilizar CARMEN en proyectos complejos, sería necesario un mejor soporte multi-robot, la posibilidad de utilizar un filtro Kalman para estimar la posición del robot, la capacidad de combinar varios comportamientos del robot, y el disponer de interfaces gráficas de usuario para ayudar a los desarrolladores a crear software complejo para el robot, sin tener que implementar completamente su modelo de comportamiento mediante programación.

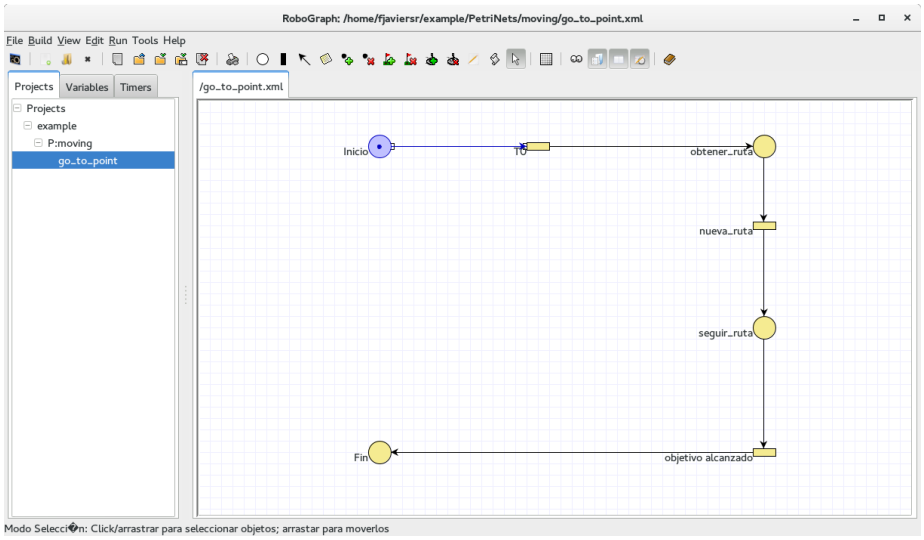


Figura 2.13: Herramienta *RoboGraph*

2.1.3. RIDE

RIDE (Robotics Integrated Development Environment) es un RDE creado en la Universidad de Vigo. Está basado en CARMEN y lo amplía proporcionando varios módulos y herramientas que aumentan sus posibilidades. Entre los módulos añadidos se encuentran JIPC, *RoboGraph*, *RoboCAN* y otras herramientas.

Los módulos más interesantes de cara a la realización de tareas de alto nivel, que además se encuentran disponibles en la página web del proyecto[1], son JIPC y *RoboGraph*. JIPC es un servidor de mensajes con una interfaz similar a IPC, pero basado en Java RMI y diseñado para mejorar el aprovechamiento del ancho de banda disponible. Adicionalmente, cuenta con funciones que permiten una monitorización más sencilla de los módulos conectados. Gracias a JIPC, RIDE puede manejar de una forma más adecuada y eficiente misiones con múltiples robots. Para ello, la gestión de las misiones se realiza mediante mensajes JIPC, mientras que los módulos de cada robot se comunican mediante IPC. Cuando es necesario, mensajes de IPC pasan al servidor JIPC y viceversa gracias al módulo *RobotWeb*.

[1] http://joaquin.webs.uvigo.es/ride.php?content=downloads&lang=es

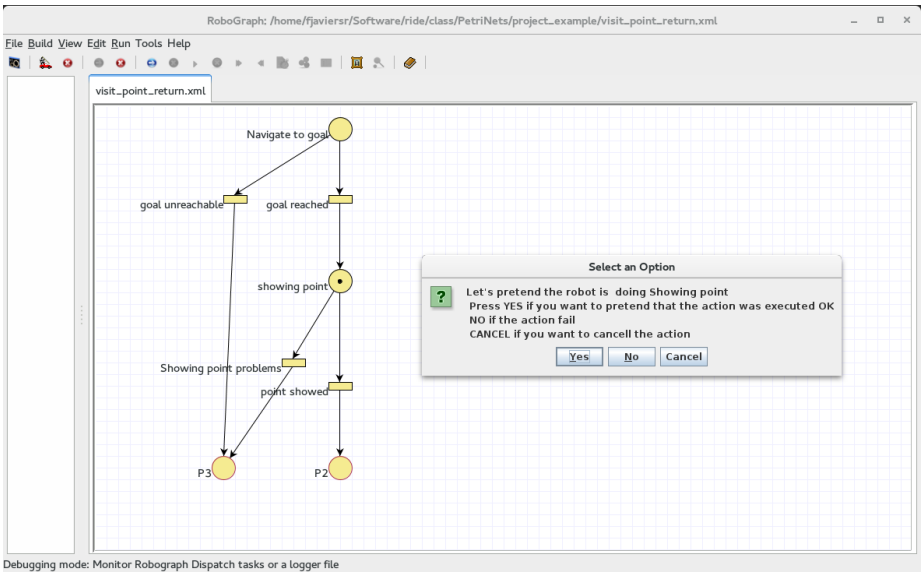


Figura 2.14: *RoboGraph* monitorizando el desarrollo de una misión

RoboGraph es un editor de misiones con algunas similitudes con el editor CfgEdit de MissionLab. Utilizando *RoboGraph*, se puede modelar el comportamiento de los robots utilizando redes de Petri. En la figura 2.13 se puede ver la interfaz gráfica de esta herramienta. Se pueden asociar los distintos estados con el envío de mensajes IPC para controlar la ejecución de la misión. Las transiciones entre estados se producen en base a la recepción de mensajes IPC o la finalización de un temporizador. *RoboGraph* permite además lanzar la ejecución de las misiones y monitorizarlas. En la figura 2.14 podemos ver a *RoboGraph* monitorizando el desarrollo de la misión de demostración incluida en el propio entorno de desarrollo.

Adicionalmente, RIDE cuenta con otras herramientas como RoboCAN, que le permite acceder a sensores y actuadores a través de un bus CAN, un programa de control de módulos IPC, que permite lanzarlos, pararlos y monitorizarlos, y un programa de configuración para *RobotWeb* en los que se pueden controlar la interacción entre IPC y JIPC.

Consideraciones finales sobre la evaluación de RIDE

Al estar basado en CARMEN, RIDE cuenta con todas sus ventajas como su diseño modular, su plataforma de comunicaciones, su manejo de mapas para localización y navegación, y su facilidad de instalación y uso. Adicionalmente, RIDE incluye módulos para mejorar los mayores puntos débiles de CARMEN: la gestión de sus módulos, el desarrollo de misiones multi-robot, y la creación de tareas complejas a partir de comportamientos simples sin necesidad de escribir manualmente el código fuente de cada misión.

La arquitectura de MissionLab y su editor de misiones CfgEdit tienen algunas ventajas sobre RIDE y RoboGraph, como que permiten la agrupación de comportamientos simples de forma que pueden estar activos a la vez de forma colaborativa, o su gestión automática de las comunicaciones en misiones multi-robot. Sin embargo, parece claro que su objetivo es solucionar los mismos inconvenientes que hemos visto en CARMEN y lo hace de una forma efectiva, aunque distinta a la que hemos decidido implementar durante el desarrollo de este trabajo.

2.1.4. ROS

ROS (*Robot Operating System*) es un framework para la escritura de software para robots. Aunque el desarrollo del núcleo del sistema lo lleva a cabo principalmente *Willow Garage*[2], una gran cantidad de instituciones colaboran aportando sus propios paquetes que añaden distintas funcionalidades.

Entre los paquetes disponibles podemos encontrar drivers para robots comerciales, simuladores, bibliotecas de navegación, localización, uso de mapas, visualizadores, etc. ROS proporciona un gestor de paquetes que permite la creación, búsqueda, descarga e instalación de los distintos paquetes disponibles, así como un sistema de comunicaciones propio.

La herramienta más básica para el funcionamiento del sistema es *roscore*, que se encarga de iniciar el servidor de mensajes, el servidor de parámetros y el sistema de creación de logs.

[2] http://www.willowgarage.com/

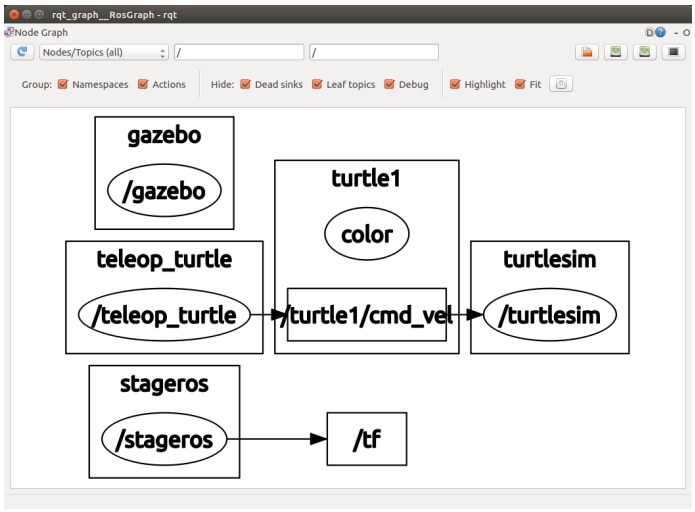


Figura 2.15: Herramienta *rqt_ graph* en funcionamiento

Al igual que otros sistemas de paso de mensajes, ROS permite el envío de mensajes *broadcast* asociados a un nombre determinado o *topic*. Los módulos pueden declarar sus propios *topics* y enviar mensajes a través de esos canales. Cualquier otro módulo interesado en esa información sólo tiene que suscribirse a esos *topics* para recibirla. Para enviar mensajes que requieren una respuesta, ROS proporciona otro mecanismo de comunicación que permite crear servicios. De esta forma, un módulo puede proporcionar un servicio y otros pueden utilizarlo enviándole mensajes y recibiendo sus respuestas correspondientes.

Para monitorizar las conexiones entre los distintos módulos en ejecución y ver el tipo de mensajes que se intercambian, puede utilizarse la herramienta *rqt_ graph*, que muestra esta información en forma de grafo (figura 2.15).

Para la gestión de la configuración de los módulos, ROS proporciona un servidor de parámetros en los que se pueden almacenar parámetros de distintos tipos, tanto públicos como privados.

Para monitorizar las misiones se utiliza la herramienta *rviz* (figura 2.16), que permite visualizar en 3D la mayoría de los datos con los que trabaja ROS.

Su manejo de mapas también es avanzado, pudiendo utilizar desde simples imágenes hasta complejos mapas con información semántica gracias a editores como *Semantic Map Editor*. ROS también tiene paquetes que permiten crear mapas (como *slam_ gmapping*) y navegar (como el software *navigation*).

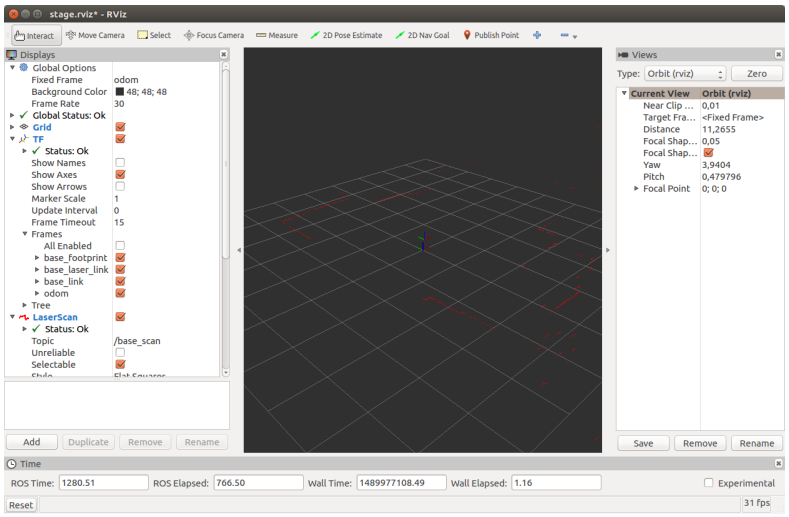


Figura 2.16: Visualizador 3D *rviz*

En cuanto a la puesta en marcha y la creación de misiones complejas, ROS dispone de un componente denominado *roslaunch* que lee ficheros XML para automatizar el proceso de lanzar los nodos ROS, establecer los parámetros y gestionar sus mensajes. En los ficheros *roslaunch* podemos definir el despliegue de los módulos y las comunicaciones entre ellos, pero no se pueden modelar cambios que se producen durante la misión. No se pueden especificar máquinas de estados finitos para dirigir el comportamiento de los robots, el sistema no permite arrancar y parar comportamientos individuales dependiendo de si se necesita o no su salida, y hay que asignar manualmente los nombres de los mensajes y la asociación entre los módulos para evitar conflictos al lanzar varias instancias de alguno de ellos.

ROS proporciona algunos editores gráficos para manejar ficheros *roslaunch* como *rxDeveloper* o *node_manager_fkie* (figura 2.17), pero éstos sufren las misma limitaciones que *roslaunch* y no evitan que el usuario tenga que especificar manualmente las conexiones entre nodos, nombres de mensaje, etc. Permiten monitorizar el estado del sistema y hacen más fácil la edición de ficheros *roslaunch*, pero no alcanzan el nivel de funcionalidad proporcionado por el editor *CfgEdit* de MissionLab, que genera automáticamente todo el código necesario, lo compila, maneja las comunicaciones entre todos los comportamientos de los robots y es capaz de lanzar la misión.

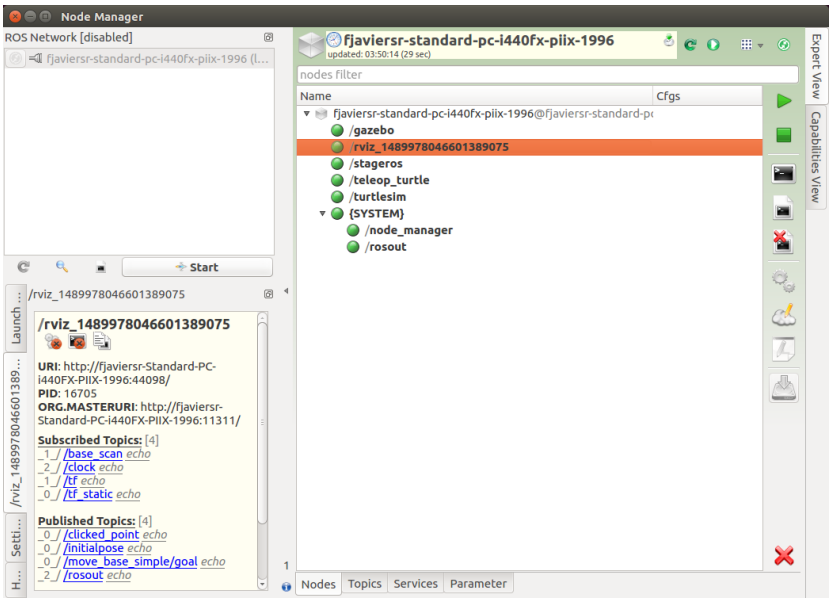


Figura 2.17: Herramienta *node_manager_fkie*

Consideraciones finales sobre la evaluación de ROS

Indudablemente, ROS es un RDE ampliamente aceptado por la comunidad científica, con una gran cantidad de colaboradores y funcionalidades disponibles. Sin embargo, no cuenta con una herramienta de creación de misiones multi-robot como *CfgEdit*. Su gran cantidad de módulos lo hacen muy potente, pero también poco portable entre distintas distribuciones Linux. La instalación en Ubuntu es muy sencilla, pero no sucede lo mismo en otras distribuciones como Fedora. Su potencia también hace que sea un sistema más complejo que otras alternativas y, por lo tanto, que requiera un tiempo de aprendizaje elevado en comparación con otros RDEs, incluso para ejecutar un ejemplo sencillo en simulación.

2.1.5. Player/Stage

El proyecto Player/Stage [49] fue iniciado en al año 2000 por Brian Gerkey, Richard Vaughan y Andrew Howard en la Universidad de California del Sur (Los Ángeles). Se distribuye bajo la licencia GPL. Proporciona principalmente interfaces con robot y sensores, y permite simularlos. Sus componentes principales son el servidor *Player* y el simulador *Stage*. El simulador 3D *Gazebo* también

```
driver
(
  name "stage"
  provides [ "simulation:0" ]
  plugin "stageplugin"
  worldfile "simple.world"
)
```

Figura 2.18: Ejemplo de configuración de un módulo de Player

ha sido una pieza importante de Player/Stage durante años, aunque en estos momentos evoluciona de forma independiente.

Player se inicia utilizando un archivo de configuración en el que se define cada módulo a cargar, tal y como podemos ver en la figura 2.18. Para cada módulo a cargar se especifica su nombre (en el campo *name*), la lista de interfaces que proporciona (en el campo *provides*), la lista de interfaces que requiere (en el campo *requires*), el nombre del archivo a cargar cuando el driver no viene incluido en *Player* (en el campo *plugin*), si queremos que el módulo se cargue desde el inicio o al recibir la primera conexión (en el campo *alwayson*) y otros parámetros de configuración específicos de cada módulo.

El simulador Stage es capaz de simular entornos en 2D a partir de un archivo de configuración (*.world*) en el que se describen tanto los robots a simular como el entorno en el que se mueven. En la figura 2.18 podemos ver cómo se configura la carga de *stage* en un archivo de configuración de *Player*, y cómo se especifica el archivo de configuración que *Stage* se encarga de simular.

Una vez lanzado *player*, si la configuración incluye la carga de *stage*, veremos una ventana similar a la de la figura 2.19. Esta ventana muestra el mundo virtual simulado por *Stage*, con los obstáculos y los robots definidos en su archivo de configuración. *Stage* da la posibilidad de aumentar o disminuir la velocidad de simulación y permite mostrar u ocultar distintos elementos.

Por otra parte, *player* acepta conexiones de programas cliente a las interfaces proporcionadas por los módulos cargados. Así, podemos utilizar, por ejemplo, la herramienta *playerv* para conectar a un robot y obtener las medidas proporcionadas por sus sensores, como podemos ver en la figura 2.20. Estas medidas pueden ser simuladas por *Stage* o pueden proceder de un robot real. En ambos casos, las interfaces de acceso serán iguales.

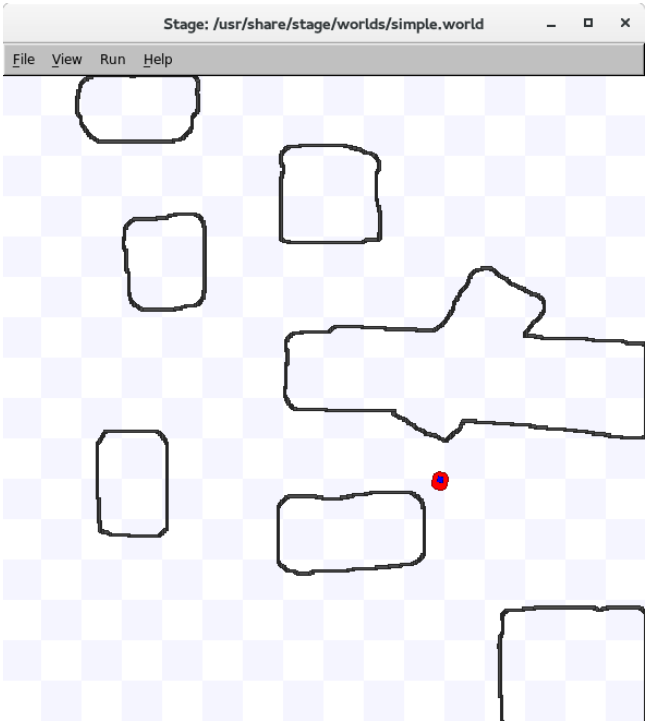


Figura 2.19: Interfaz de usuario de la herramienta *Stage*

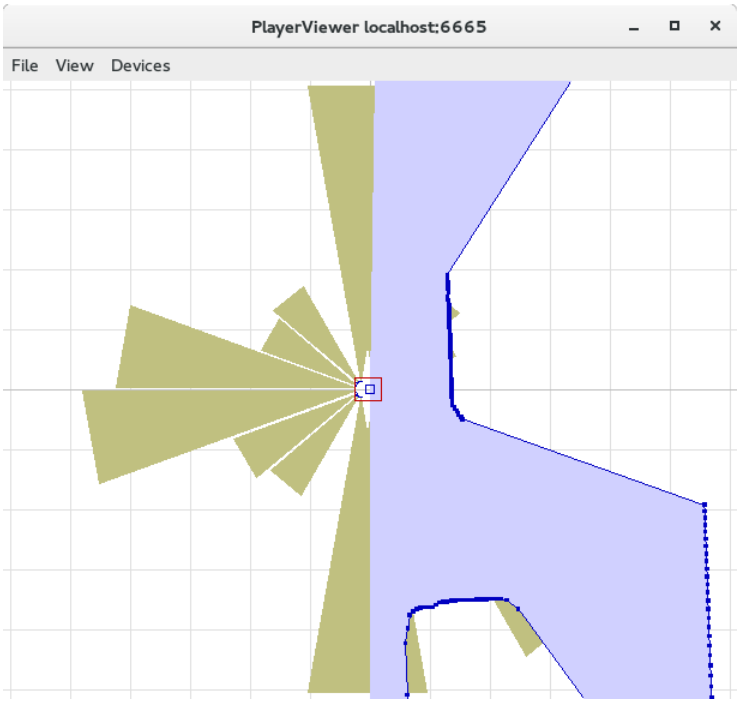


Figura 2.20: Herramienta cliente de player *playerv*

Podemos crear nuestro propio cliente de Player y acceder a los datos disponibles o controlar los robots utilizando algunas de las bibliotecas de programación disponibles. El proyecto Player/Stage proporciona bibliotecas para C, C++ y Python, pero existen una gran cantidad de bibliotecas creadas y distribuidas independientemente para crear clientes *player* en distintos lenguajes de programación.

Aunque no es su principal objetivo, entre la gran cantidad de drivers que proporciona el proyecto Player/Stage, algunos implementan funcionalidades como localización (drivers *amcl* and *ekfvmap*) y navegación (driver *wavefon*). La funcionalidad de navegación se puede controlar mediante una herramienta gráfica llamada *playernav*.

Otra herramienta que históricamente ha tenido una gran importancia en el proyecto Player/Stage es Gazebo. Gazebo es un simulador físico que cuenta con un visor 3D, utiliza varios motores de simulación de sistemas físicos y proporciona modelos de varios robots comerciales. Hoy en día, Gazebo evoluciona de forma independiente a Player/Stage. Su desarrollo continúa por parte de la *Open Source Robotics Foundation*, está integrado con ROS y es una de sus herramientas más importantes. En nuestro trabajo de Tesis Doctoral, hemos simulado en Gazebo la carretilla paletizadora industrial, tal como puede observarse en la figura 2.21.

Consideraciones finales sobre la evaluación de Player/Stage

De cara a la implementación de misiones complejas o misiones multi-robot, el entorno Player/Stage nos proporciona un entorno de pruebas muy completo y las primitivas básicas de funcionamiento de los robots, pero no permite crear mapas basados en las lecturas de los sensores, tampoco tiene ninguna herramienta para editar mapas, ni dispone de ningún componente para gestionar o sincronizar comportamientos de robots o grupos de robots. Para conseguir objetivos de más alto nivel, tenemos que programar nuestro propio cliente de *player*, conectar con las interfaces que nos proporciona e implementar manualmente toda la lógica de nuestra misión y las posibles comunicaciones con otros robots.

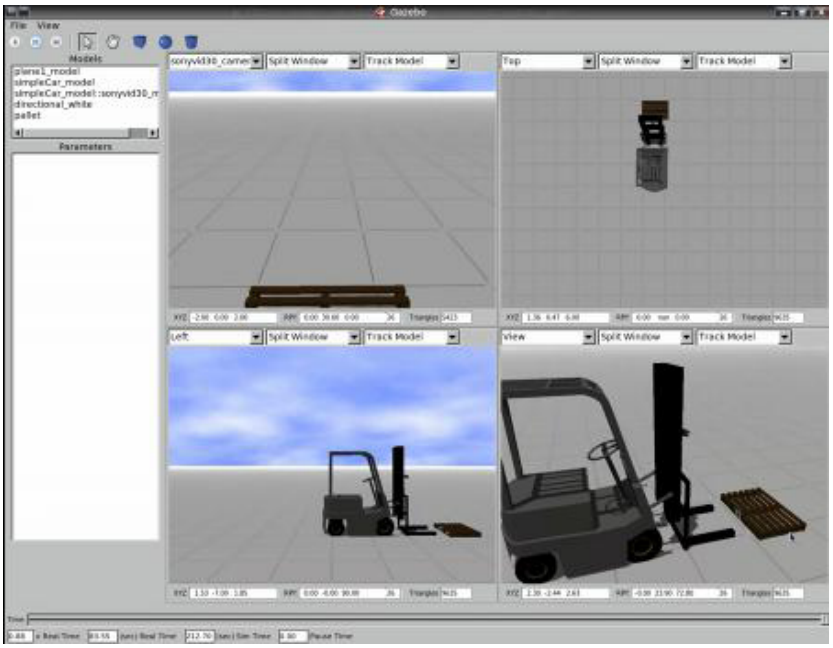


Figura 2.21: Simulador 3D *Gazebo*

2.1.6. OROCOS

El proyecto OROCOS (*Open RObot COntrol Software*) empezó su desarrollo en el año 2000, financiado por la Unión Europea en colaboración con varias empresas. La primera versión fue terminada por Peter Soetens en 2002. OROCOS nació como un framework de control de robots en tiempo real fuerte. Posteriormente, a las herramientas de control en tiempo real se añadieron dos bibliotecas complementarias: *Bayesian Filtering Library* (BFL), que implementa distintos filtros *Bayesianos* (filtro de Kalman, filtro de partículas, etc.), y *Kinematics and Dynamics Library* (KDL), que permite modelar y hacer cálculos cinemáticos sobre estructuras robóticas complejas.

El planteamiento de OROCOS para la implementación de comportamientos complejos tiene algunas similitudes con MissionLab. OROCOS permite crear ficheros XML o scripts, donde se especifican las relaciones entre diferentes componentes, que sirven de entrada para la herramienta de despliegue (*deployer*). En la figura 2.22 pse muestra uno de estos scripts.

```
// Start this script by using:
// deployer-gnulinux -s start.ops -linfo

// For ROS builds, we could also import using the package name.
import("hello-6-scripting")

loadComponent("hello","Example::Hello")
loadComponent("world","Example::World")

var double period = 0.5
setActivity("hello", period, LowestPriority , ORO_SCHED_OTHER )
setActivity("world", period, LowestPriority , ORO_SCHED_OTHER )

connectPeers("hello","world")

hello.configure()
hello.start()
```

Figura 2.22: Script para la herramienta *deployer*

En estos ficheros se nos permite definir máquinas de estados finitos y asociar componentes de forma jerárquica. El propio sistema ejecuta los componentes cuando es necesario. Además, OROCOS proporciona un lenguaje denominado *osd* que permite definir de forma fácil máquinas de estados finitos. En la figura 2.23 podemos ver un fragmento de código *osd*.

Consideraciones finales sobre la evaluación de OROCOS

Sus capacidades de control en tiempo real y su manejo de robots con múltiples grados de libertad hacen a OROCOS un framework muy adecuado para el desarrollo de software de control de maquinaria industrial. Sin embargo, al no contar con herramientas gráficas que ayuden a escribir el código necesario, su curva de aprendizaje es elevada. No es posible escribir comportamientos complejos hasta que no se conocen en profundidad y se domina la sintaxis XML aceptada por sus herramientas, su interfaz C++ y el formato *osd*, por lo que es un entorno poco adecuado para su uso educativo en robótica. Tampoco proporciona funcionalidades presentes en otros entornos de desarrollo de robots, como algoritmos de localización, navegación, gestión de mapas, ni drivers para robots comerciales. De esta forma la creación de una misión relativamente sencilla de implementar con otros RDEs, como mover un robot de un punto a otro en un mapa, requiere un gran esfuerzo de desarrollo si sólo utilizamos las herramientas proporcionadas por OROCOS.

```
StateMachine Simple_nAxes_Test
{
     var bool calibrate_offsets = true
     var bool move_to = true
     var bool stop = true

     const double pi = 3.14159265358979
     var array pos = array(6,0.0)

     initial state StartRobotState {
        entry {
            Robot.prepareForUse()
        }
        exit {
            Robot.unlockAllAxes()
            Robot.startAllAxes()
        }
        transitions {
            select CalibrateOffsetsState
        }
     }

     ...
}
```

Figura 2.23: Estado inicial de una máquina de estados definida en código OSD

2.2. Localización global con filtros de partículas

En esta sección, revisaremos los conceptos básicos sobre los filtros de partículas aplicados a localización global. Presentaremos su justificación teórica (el Teorema de Bayes), sus principales fases, y las limitaciones que hemos encontrado en los trabajos que aplican el filtro de partículas en situaciones reales.

2.2.1. Filtros Bayesianos

Los filtros Bayesianos se utilizan para estimar recursivamente el estado de un sistema dinámico en base a observaciones y a un modelo matemático del sistema. Cuando aplicamos estos filtros para localizar robots, intentamos estimar la densidad de probabilidad a posteriori de la posición del robot en el espacio de estados, condicionada por los datos disponibles.

$$bel\left(x_t\right) = p\left(x_t | z_{1:t}, u_{1:t}\right) \tag{2.1}$$

En la expresión 2.1, representamos la densidad de probabilidad a posteriori como *bel*, debido al término (*belief*) que suele usarse en inglés para referirse a ella. Los datos disponibles se diferencian en dos tipos: datos perceptivos (z) y datos de odometría (u). Los datos perceptivos contienen información sobre factores externos, como medidas de láser, ultrasonidos o imágenes. Los datos de odometría proporcionan información sobre el movimiento del robot utilizando sensores como encoders, acelerómetros, brújulas y las señales de control.

Aplicando el teorema de Bayes y la asunción de Markov en la expresión 2.1, obtenemos la fórmula conocida como filtro Bayesiano [50]

$$bel\left(x_t\right) = \eta p\left(z_t|x_t\right) \int p\left(x_t|x_{t-1}, u_t\right) bel\left(x_{t-1}\right) dx_{t-1} \tag{2.2}$$

Podemos ver cómo la probabilidad a posteriori en t depende recursivamente de la distribución de probabilidad a priori $bel\left(x_{t-1}\right)$.

El término $p\left(x_t|x_{t-1}, u_t\right)$ representa la distribución de probabilidad actual, condicionada por la distribución de probabilidad previa y los datos de odometría. Por lo general, nos referimos a este factor como el modelo de movimiento o el modelo de predicción.

$p\left(z_t|x_t\right)$ representa la probabilidad de las observaciones actuales del robot, condicionada por la estimación de su posición. Este factor se corresponde con el llamado modelo de observación o de corrección.

η es un factor normalizador que asegura que la suma de todas las probabilidades es 1 y, por lo tanto, $bel\left(x\right)$ es realmente una distribución de probabilidad.

En algunos casos, es útil definir el término $\overline{bel}\left(x\right)$ como la distribución de probabilidad antes de aplicar la corrección:

$$\overline{bel}\left(x\right) = \int p\left(x_t|x_{t-1}, u_t\right) bel\left(x_{t-1}\right) dx_{t-1} \tag{2.3}$$

Este término se denomina a menudo distribución propuesta o predicción.

2.2.2. Filtros de partículas

Los filtros de partículas son algoritmos que estiman el estado de sistemas dinámicos basándose en filtros Bayesianos. Tienen algunas ventajas sobre los filtros de Kalman extendidos. Las más importantes son que pueden representar distribuciones de probabilidad multimodales y pueden manejar modelos de movimiento o ruido no Gaussianos o no lineales [51]. Por ello, son muy adecuados para la implementación de algoritmos de localización global [52] [53] [54].

Cuando trabajamos con filtros de partículas, tratamos de representar la distribución a posteriori mediante un conjunto de n partículas ponderadas.

$$bel\left(x_t\right) \approx \left\{x_t^{(i)}, w_t^{(i)}\right\}_{i=1,\ldots,n} \tag{2.4}$$

Como podemos ver, cada partícula representa uno de los posibles estados (x) y tiene un peso (w) que indica la importancia de esa estimación en el conjunto de partículas. Puesto que los filtros de partículas, como el filtro Bayesiano, son recursivos, necesitamos seleccionar un conjunto inicial de partículas. Este conjunto inicial se escoge durante la fase de inicialización. Posteriormente, se aplica el resto de las fases del filtro de forma cíclica: predicción, corrección y muestreo.

Inicialización

Típicamente, si no conocemos el estado del robot, el conjunto inicial de n partículas se inicializa de forma uniforme, tratando de cubrir todo el espacio de estados. Si conocemos la posición inicial del robot, el conjunto inicial de partículas suele escogerse en base a una distribución Gaussiana en torno a la posición del robot. En cualquier caso, todas las partículas se inician con el mismo peso inicial.

$$w_0^{(i)} = 1/n \tag{2.5}$$

Predicción

En esta fase, generamos un conjunto de partículas que representa el término mostrado en la regla de Bayes (2.2). Para ello, aplicamos el modelo de movimiento al conjunto de partículas resultante en la iteración previa. Podemos hacer esto añadiendo la odometría (u_t) y una variable aleatoria (e_t), de acuerdo con el error estimado, a la posición de cada partícula utilizando el operador estándar de composición [55]

$$x_t^{(i)} = x_{t-1}^{(i)} \oplus (u_t + e_t) \tag{2.6}$$

Corrección

En esta fase, se corrige el error en la predicción obtenida en el paso anterior utilizando el modelo de observación. De esta forma, la distribución de probabilidad resultante aproxima mejor $bel\,(x_t)$. Para hacer esto, calculamos los pesos de cada partícula en base a la probabilidad de obtener las observaciones desde la posición de cada una.

$$w_t^{(i)} = p\left(z_t | x_t^{(i)}\right) \tag{2.7}$$

Si tenemos varias observaciones condicionalmente independientes, multiplicamos sus probabilidades.

$$p\left(z_t | x_t^{(i)}\right) = \prod_j p\left(z_t^{(j)} | x_t^{(i)}\right) \tag{2.8}$$

Una vez calculado el peso de cada partícula, se aplica una normalización al igual que en el filtro Bayesiano.

Muestreo

En esta fase, elegimos un nuevo conjunto de partículas a partir del conjunto ponderado de partículas resultante de la fase de corrección. La probabilidad de una partícula de ser seleccionada para pertenecer a este nuevo conjunto debe ser la misma que el peso normalizado de la partícula. Una vez finalizada la fase de muestreo, los pesos se reinicializan.

El objetivo de esta fase es resolver el problema que supone tener demasiadas partículas en lugares con una probabilidad baja y muy pocas en lugares con una alta posibilidad. Distribuyendo el número de partículas en cada lugar de acuerdo a su probabilidad, se consigue explorar el espacio de estados de una forma más eficiente.

2.2.3. Revisión de los efectos y el manejo de errores

En muchos casos, los sensores nos dan medidas inesperadas que no se encuentran reflejadas en nuestros mapas. Cuando creamos mapas basados en medidas de sensores láser, objetos como el mobiliario, papeleras, etc. aparecerán en el mapa generado. Si posteriormente se mueven esos objetos, las medidas obtenidas desde determinados lugares, cuando el robot navega, no serán las esperadas y, por lo tanto, los algoritmos de localización tendrán un peor comportamiento. Esto puede ocurrir también cuando usamos, por ejemplo, la intensidad de una señal inalámbrica como medida externa. Dependiendo de la forma de las habitaciones, la decoración, la orientación del robot o el estado de las puertas, la intensidad de la señal recibida puede ser significativamente mayor o menor de lo esperado.

También se pueden utilizar imágenes del entorno capturadas mediante cámaras y después procesarlas mediante técnicas de visión artificial para extraer información del entorno con distinto nivel de abstracción, como líneas, pasillos, intersecciones, o de mayor nivel como puertas, etc. Estas técnicas son especialmente propensas a generar detecciones incorrectas o inexactas. Dependiendo de factores como la iluminación, la decoración, los obstáculos o la perspectiva de la cámara, es posible que ciertos objetos sean incorrectamente identificados (en el caso de las puertas) como puertas, o que puertas existentes no puedan ser detectadas, dando lugar a falsos positivos y falsos negativos. Dentro de cada uno de estos dos tipos de errores podemos diferenciar dos subtipos: errores persistentes y errores puntuales.

Consideramos errores puntuales a aquellos que ocurren durante cortos periodos de tiempo (pocos segundos) debido a circunstancias muy específicas de luminosidad, reflejos, perspectiva, etc. que confunden el algoritmo de detección del robot.

Los errores persistentes son aquellos causados principalmente por objetos de características muy similares a los que queremos detectar. Estos objetos pueden provocar falsos positivos persistentes que no desaparecen hasta que salen del campo de visión del robot.

El efecto de estos errores de detección en la distribución de probabilidad será generalmente una disminución de la probabilidad en el lugar en el que realmente se encuentra el robot, y un incremento en los lugares del mapa que maximizan $p\left(z_t|x_t^{(i)}\right)$. Una vez que los errores de detección desaparecen, el filtro tenderá otra vez a converger hacia la posición correcta. Sin embargo, debemos tener en cuenta que los filtros de partículas cuentan con un número finito de partículas, y que la fase de muestreo las redistribuye hacia los lugares con mayor probabilidad. Si durante el tiempo en el que está presente una detección errónea todas las partículas desaparecen de la localización correcta, el filtro no podrá recuperarse y fallará.

Si los falsos positivos o falsos negativos aparecen durante la fase de inicialización del filtro y ésta se basa en la distribución de probabilidad proporcionada por las mediciones externas, las consecuencias pueden ser fatales. Puede suceder que, desde el principio, no haya partículas colocadas cerca de la localización real del robot, o que haya muy pocas y desaparezcan durante las primeras iteraciones del filtro, dada la baja probabilidad en el lugar correcto de acuerdo a las medidas de los sensores. Por consiguiente, el estado correcto no se muestrearía e inevitablemente el filtro no podría proporcionarnos la localización real del robot.

Los casos menos severos en los que las partículas desaparecen del lugar correcto debido a su baja probabilidad, normalmente pueden solucionarse aumentando el número de partículas, haciendo más lenta la convergencia del filtro o ejecutando la fase de muestreo sólo cada cierto número de iteraciones. Sin embargo, esas soluciones no son siempre factibles.

Incrementar significativamente el número de partículas no es posible cuando la potencia de computación con la que contamos es limitada, como suele ocurrir en sistemas embebidos, o cuando la localización debe realizarse en un entorno muy extenso. En algunos casos, hay optimizaciones que pueden ayudarnos a mitigar este problema, como aplicar mejoras al rendimiento del filtro [56], o utilizar conjuntos de muestreo adaptativos [57].

Retrasar la convergencia del filtro no es la solución más idónea si queremos obtener una estimación de la posición lo más pronto posible. En muchos casos, puede ser más conveniente obtener una estimación no muy exacta pero muy rápida, que una estimación muy exacta después de un tiempo muy largo. Tan pronto como tenemos una estimación relativamente precisa de la posición del robot, podemos activar los algoritmos de navegación y el robot puede empezar a moverse más fácilmente. Mientras se mueve, el algoritmo de localización puede continuar trabajando para detectar y corregir los posibles errores que haya podido cometer en su primera estimación.

La omisión de la fase de muestreo en alguna de las iteraciones del filtro tiene un efecto doble [58]. Por un lado, puede prevenir la desaparición de partículas en el área en la que se encuentra el robot pero, por otro lado, puede provocar que el número de partículas a seguir para cada una de las posibles posiciones del robot sea demasiado pequeño como para seguir adecuadamente el movimiento del robot. Por lo tanto, si aplicamos esta mejora, tenemos que aumentar el número de partículas de acuerdo al tiempo durante el que omitimos la fase de muestreo y la dispersión de la distribución de probabilidad inicial del estado del robot.

Cuando este tipo de soluciones no son suficientes, tenemos que reinicializar el filtro o añadir partículas artificialmente, de forma independiente de su ejecución normal. Un buen momento para reinicializar el filtro es cuando detectamos que el resultado de la fase de corrección, antes de la normalización, produce unos pesos anormalmente bajos, es decir, que la suma de los pesos de las partículas antes de la normalización se encuentra por debajo de un cierto umbral. Sin embargo, cuando utilizamos sensores de poca precisión o entornos con una gran cantidad de marcas, esta medida es muy difícil de implementar.

Si elegimos un umbral muy bajo, el tiempo necesario para detectar el error del filtro puede ser demasiado alto, debido a coincidencias casuales entre marcas presentes en el mapa y marcas detectadas por el robot. Por el contrario, si el umbral es demasiado bajo, es posible que el filtro se reinicialice debido a errores en los sensores, incluso aunque ya se encuentre muestreando la posición real del robot.

La introducción de partículas en el filtro al margen de su ejecución normal se lleva a cabo en métodos como SRL (*Sensor Resetting Localization*) o Mixture-MCL [27] pero, como hemos visto anteriormente, puede provocar problemas cuando esas partículas extra se introducen en base a medidas externas incorrectas o no contempladas en nuestro mapa. Nuestra propuesta de localización global basada en el filtro de partículas tiene en cuenta estas ideas.

2.3. Sistemas de Información Geográfica (GIS)

Los Sistemas de Información Geográfica se introdujeron en la Robótica Autónoma hace aproximadamente unos quince años. Uno de los primeros sistemas propuestos fue una silla de ruedas desarrollada por Mori and Kotani [59], con la intención de ayudar a personas con problemas de visión. Otros investigadores han utilizado también GIS como una fuente de información para la navegación autónoma en exteriores.

Por ejemplo, Bonnifait et al. [60] presentaron un coche autónomo guiado por laser utilizando un modelo de carretera almacenado en un GIS. Cappelle et al. [61] utilizaron un modelo de ciudad en 3D guardado en un GIS para realizar funciones de localización con un filtro de Kalman extendido e información procedente de GPS, laser y vision artificial. Mirats Tur et al. [62] propuso un sistema multi-robot basado en GIS y localización mediante filtros de partículas en entornos urbanos. Yoshida et al. [63] también presentó un sistema multi-robot, pero enfocado en las comunicaciones inter-robot a través de Daruma [64], un prototipo GIS para gestionar operaciones de rescate.

GIS se ha utilizado también en investigaciones militares. Por ejemplo, Rackliffe et al. [65] propuso el uso de GIS para permitir a vehículos aéreos (UAV) y terrestres (UGV) encontrar áreas seguras para aterrizar o realizar operaciones.

Para abordar problemas relacionados con la robótica autónoma que requieren del uso de información espacial, pensamos que una de las mejores opciones es utilizar un Sistema de Información Geográfico (GIS).

Desde los años 90, GIS se empezó a integrar con los sistemas de bases de datos espaciales. Como resultado, actualmente los sistemas GIS están enfocados en la recolección de datos espaciales, su edición, análisis y visualización, donde las bases de datos espaciales se encargan del almacenamiento de datos, las consultas, indexación, optimización y la integridad [66].

Los sistemas GIS destacan por su interoperabilidad, que se ha conseguido en gran parte gracias a la iniciativa de OGC, una organización colaborativa de estándares compuesta por más de 400 organizaciones en todo el mundo y de todo tipo (comerciales, gubernamentales, sin ánimo de lucro, dedicadas a la investigación...). Por ejemplo, los modelos IFC se pueden transformar al dominio GIS [67]. Los modelos GML también están soportados por GIS, puesto que el GML Simple Features Profile [68] y el Simple Features for SQL [69] (utilizado en bases de datos espaciales) tienen una estructura similar y describen geometrías también muy similares. Muchos otros formatos pueden importarse también en GIS, incluyendo el conocido Keyhole Markup Language (KML), o ESRI's shapefiles (SHP), por ejemplo. La infomación espacial es, en ocasiones, difícil de obtener, y la interoperabilidad permite a GIS reutilizar la mayoría de los modelos espaciales existentes. GIS tiene otras funcionalidades que lo hacen muy interesante, como su escalabilidad desde la descripción de un pequeño mueble en una habitación hasta el modelado a escala global de superficies de cientos o miles de kilómetros. Esto hace posible la gestión de grandes volúmenes de datos sin afectar al rendimiento del sistema. GIS también soporta el manejo de información semántica a través de capas. Por ejemplo, una capa podría usarse para almacenar todas las habitaciones de un edificio, mientras que otra puede usarse para almacenar todas las puertas. En definitiva, GIS es muy intuitivo, puesto que fue diseñado enfocándose en la usabilidad, el análisis y la visualización de datos geoespaciales.

El crecimiento de las necesidades de gestión y análisis de datos en los sistemas GIS se ha solucionado gracias a las bases de datos espaciales, que han sido un área activa de investigación y desarrollo durante más de tres décadas [70]. Este esfuerzo de investigación ha conseguido muchos logros, como los tipos de datos y operadores espaciales, lenguajes de consulta espacial, estrategias de procesamiento, o técnicas de indexación y *clustering* espacial [71].

Por ello, las bases de datos espaciales se pueden considerar superiores si las comparamos con el manejo directo de archivos GML o IFC. Además, la tecnología de bases de datos espaciales permite a GIS ser escalable y gestionar grandes volúmenes de datos sin penalizar el rendimiento.

Las bases de datos espaciales soportan consultas espaciales, que son un tipo de consulta exclusivo de este tipo de base de datos. Estas consultas son más complejas que las convencionales, puesto que permiten hacer uso de tipos de datos geométricos (como puntos o polígonos), y porque la relación entre los objetos implicados en una consulta espacial tiene que ser tenida en cuenta.

Para hacer frente a estas dificultades añadidas, las bases de datos espaciales utilizan índices espaciales y uniones (o *joins*) espaciales. Las consultas espaciales se crean combinando uno o varios operadores espaciales. La OGC define y agrupa los operadores espaciales en tres categorías: básicos, topológicos y operadores de análisis espacial [72].

El primer grupo puede acceder a propiedades de la geometría tales como su forma o su localización. Los operadores topológicos pueden expresar relaciones espaciales entre geometrías, permitiendo al usuario conocer si hay una intersección entre distintos objetos o uno está dentro de otro, por ejemplo. La tercera categoría de operadores permite el usuario construir consultas más avanzadas, como la unión de varias geometrías o la distancia entre ellas.

2.3.1. Modelo de datos espaciales OGC

Los datos espaciales son la suma de nuestras interpretaciones de fenómenos geográficos y se pueden referir a cualquier información que asocia a un objeto con una localización geográfica específica. En interiores, los datos espaciales pueden restringirse a los objetos que esperamos encontrar en un edificio, tales como puertas, mesas, muros, lámparas o incluso robots o personas. Los datos espaciales pueden ser recolectados y almacenados en formato raster o vectorial.

Estos formatos consideran diferentes elementos como la unidad básica espacial. En formato raster, la unidad espacial básica es un píxel, mientras que en formato vectorial, es un objeto geométrico. La representación vectorial tiene muchas ventajas sobre el la representación mediante píxels, como un menor tiempo de procesamiento, la abstracción de las geometrías y la escalabilidad.

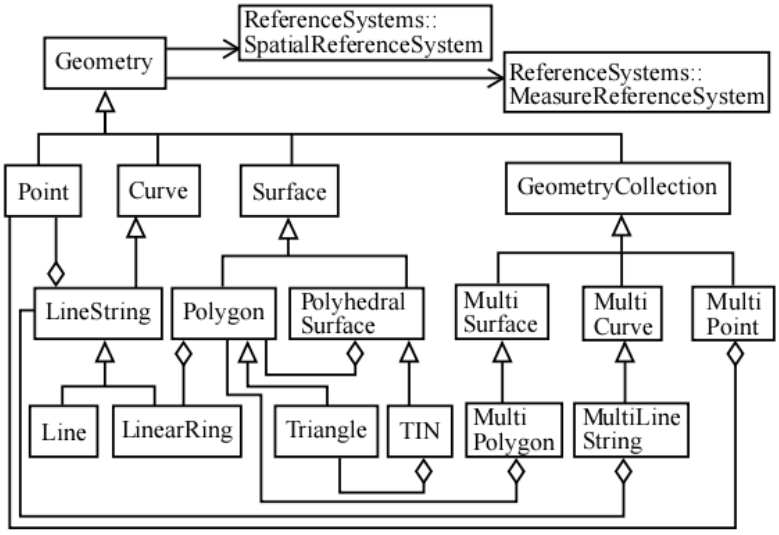


Figura 2.24: Diagrama de clases UML para el Modelo de Objetos Geométricos OGC

Puesto que los conjuntos de datos espaciales y sus relaciones se pueden modelar de muchas formas diferentes, la OGC propuso el Modelo de Objetos Geométricos [72], una jerarquía de tipos de datos espaciales que permite representar características espaciales simples en una base de datos en forma vectorial. Este modelo está limitado a características de dos dimensiones, es independiente a la plataforma utilizada y está implementado en más de 700 productos. Es importante destacar que, probablemente, la mayoría de los planos de interiores existentes que soportan información semántica se pueden transformar en GIS, puesto que GIS y las bases de datos espaciales están soportadas por la OGC.

La figura 2.24 muestra el diagrama de clases UML del *Modelo de Objetos Geométricos*. El OGC utiliza la clase *Geometry* para representar una característica espacial como un objeto que tiene al menos un atributo de un tipo geométrico en la base de datos. Geometry es la clase base de la jerarquía. Es abstracta, pero tiene cuatro subclases instanciables: *Point*, *Curve*, *Surface* y *GeometryCollection*. El resto de tipos geométricos en el modelo hacen posible la creación de objetos más complejos. Por ejemplo, *MultiPoint* y *MultiLine* pueden guardar una colección de puntos o líneas respectivamente. Cada *Geometry* está asociado con un sistema de referencia espacial, permitiéndonos conocer la posición del objeto en el mundo.

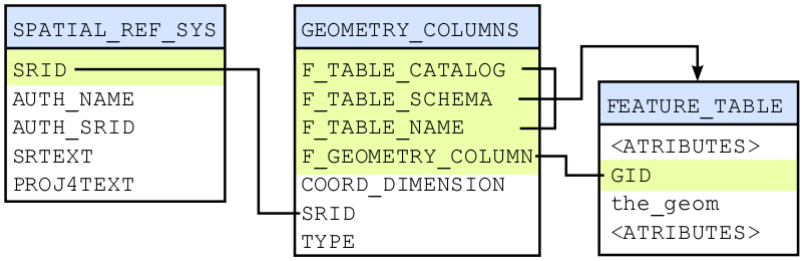


Figura 2.25: Esquema PostGIS para extender PostgreSQL con tipos geométricos

Es importante tener en cuenta que el modelo es jerárquico y que las geometrías se pueden agrupar en capas, lo cual es muy útil para manejar información simbólica de forma sencilla. Por ejemplo, una capa puede contener todas las puertas en un edificio, de manera que un objeto *Geometry* en esa capa representaría una de esas puertas. La puerta puede estar formada por una colección de primitivas básicas como puntos, líneas o polígonos. Finalmente, cada primitiva básica consiste en un conjunto de coordenadas.

Para representar un tipo geométrico, la OGC ha definido dos estandars: una representación textual (WTK o *Well-Known Text*) y una representación binaria (WKB o *Well-Known Binary*), utilizada principalmente para transferir datos entre distintos sistemas [72].

La representación WKT es un lenguaje simple de marcado definido en BNF (forma BackusNaur) donde, por ejemplo, un punto puede describirse como *"Point (10, 15)"* o una polilínea con tres puntos como *"Linestring (10 10, 20 20, 30 40)"*.

2.3.2. Consultas espaciales PostGIS

PostGIS utiliza el esquema representado en la figura 2.25 para extender PostgreSQL con soporte para datos espaciales siguiendo la especificación *Simple Features Access for SQL* del Open Geospatial Consortium.

SPATIAL_REF_SYS y *GEOMETRY_COLUMNS* son tablas de metadatos. La primera guarda información que describe completamente cada sistema de referencia espacial soportado por la base de datos (actualmente más de 3000).

GID	LAYER	THE_GEOM
5	Doors	0103000000010000000500000004D0A65A2···
72	Doors	0103000000010000000500000CA099294···
234	Rooms	0103000000010000000500000DE7A0207···
292	Rooms	0103000000010000000500000C97C2F54···
472	Wind.	0103000000010000000D0000009999EDD4···
577	Wind.	0103000000010000000D000000D2FEAE59···

Figura 2.26: Ejemplo de *FEATURE_ TABLE* con datos de nuestro mapa

```
SELECT ST_AsText(ST_Centroid(the_geom))
  FROM secondfloor
  WHERE
    ST_DWithin(the_geom,
          ST_GeomFromText('POINT(x̂_k  ŷ_k)'),
          R)
  AND "LAYER" = 'DOORS';
```

Figura 2.27: Ejemplo de consulta espacial

El propósito de la segunda es listar qué atributos y tablas de la base de datos contienen información espacial. Así, cada instancia de *F_ TABLE_ NAME* identifica una tabla de características geométricas que contiene objetos *Geometry*, donde la columna *F_ GEOMETRY_ COLUMN* indica qué campos de la tabla de características contiene objetos geométricos.

Es importante saber que un objeto *Geometry* está asociado con un sistema de coordenadas a través del campo *SRID*. Una *FEATURE_ TABLE* en PostGIS tiene al menos dos campos: un identificador único (*GID*), y un campo que almacena la información geométrica (*the_ geom*) de un objeto. El resto de atributos de la tabla son opcionales.

La figura 2.26 muestra un ejemplo de *FEATURE_ TABLE* tomado de nuestra base de datos espacial PostGIS. La tabla se llama *segundaplanta*, y cada registro representa un objeto *Geometry* con un identificador único. Los objetos están separados en capas, como puertas, habitaciones o ventanas (entre otros).

El campo *THE_ GEOM* está codificado en un formato binario de PostGIS. Para hacerlo legible, es necesario convertirlo al formato WTK utilizando la función *ST_ AsText()*.

Nuestro planteamiento para resolver el problema de la localización global utilizando filtros de partículas tiene como entrada las consultas espaciales a las tablas de nuestra base de datos espacial PostGIS. En la figura 2.27 podemos ver un ejemplo de consulta espacial que permite conocer las posiciones de las puertas situadas en un radio R alrededor de una posición (x, y).

3

MissionLab-CARMEN: nuestro RDE integrado

En base al análisis de todos los RDEs que hemos estudiado, ROS y CARMEN son los mejores candidatos para proporcionar las funcionalidades que le faltan a MissionLab. Finalmente, seleccionamos CARMEN porque sus puntos fuertes son precisamente las debilidades de MissionLab: proporciona los módulos de localización, navegación y *mapping* necesarios, el uso de sus herramientas es muy intuitivo y es adecuado para usuarios no expertos. A la vez es un RDE más simple, con menos dependencias y más portable que ROS. La similitud entre los servidores de comunicaciones de MissionLab y CARMEN es una ventaja añadida, ya que permite el uso de un sistema de mensajes común para el RDE integrado.

3.1. Diseño conceptual de la arquitectura

Basándonos en las similitudes, ventajas y desventajas de ambos RDEs, decidimos llevar a cabo la integración de MissionLab y CARMEN siguiendo las siguientes especificaciones:

1. El RDE resultante debe ser totalmente compatible con cualquier desarrollo que dependa de las versiones originales de MissionLab y CARMEN. En consecuencia, ambos sistemas deben poder ejecutarse de forma separada.

2. El RDE resultante debe ser multi-robot, permitiendo la participación de varios robots de CARMEN en una misma misión.

3. El RDE resultante debe ser capaz de usar drivers de MissionLab y CARMEN indistintamente.

4. MissionLab debe mantener el control total de todos los robots y las misiones. Para ello, tendrá la última palabra sobre la posición estimada del robot y las órdenes de movimiento que se envían a los drivers. De esta manera, se aprovechará su avanzada gestión de comportamientos y habilidad para fusionar la información procedente de distintos algoritmos y dispositivos de localización.

5. La información de localización generada por CARMEN debe poder aprovecharse en MissionLab para mejorar la estimación de la posición del robot.

6. Las lecturas de los sensores realizadas por drivers de CARMEN deben estar disponibles para los módulos de MissionLab de la misma manera que las lecturas realizadas directamente por drivers de MissionLab.

7. La funcionalidad de navegación proporcionada por CARMEN debe estar disponible como un comportamiento nativo de MissionLab, de manera que se pueda usar y combinar con el resto de comportamientos utilizando *CfgEdit*.

8. El sistema resultante debe poder ejecutarse de forma nativa en versiones recientes de Linux.

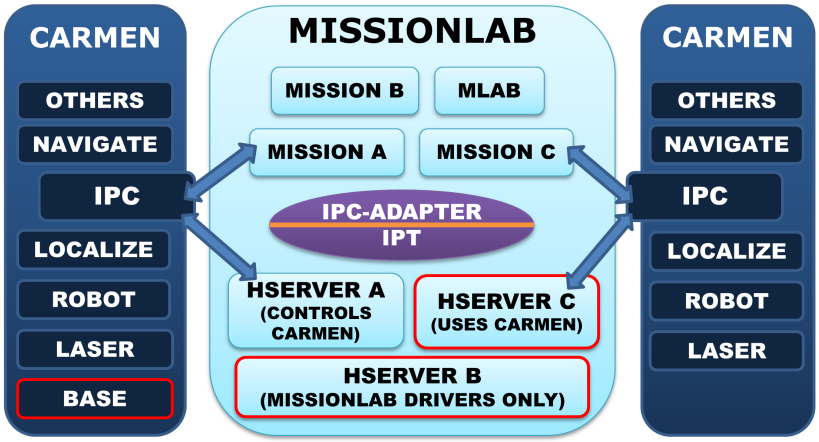


Figura 3.1: Diseño de la arquitectura propuesta

Para abordar la integración, hemos tenido en cuenta que hay dos puntos clave en el modelo de ejecución de CARMEN (Figura 2.10): el mensaje *CARMEN_LOCALIZE_GLOBALPOS* con la posición del robot, y las comunicaciones (*CARMEN_BASE_VELOCITY*, *CARMEN_BASE_ODOMETRY*) entre los módulos *robot* y *base*. Interceptando estos mensajes, es posible tomar el control del robot, porque controlamos su posición estimada, recibimos la información de los sensores y la decisión final de movimiento que ha dado CARMEN, y podemos enviar al robot nuestras propias órdenes de movimiento, independientemente de la decisión que haya tomado CARMEN. A grandes rasgos, ésta es la base del diseño que hemos seguido para permitir a MissionLab tomar el control de los robots de CARMEN.

El modelo de ejecución del RDE que hemos desarrollado permite que, dentro de una misión multi-robot, un robot pueda estar controlado por CARMEN, por MissionLab o por ambos simultáneamente. En la figura 3.1 se muestran los módulos implicados de los RDEs que representan cada una de las tres situaciones posibles:

- un robot de CARMEN controlado por los módulos a la izquierda de la imagen y *HServer A*

- un robot de MissionLab controlado por *HServer B*

- un robot controlado por MissionLab y CARMEN, usando los módulos de CARMEN a la derecha de la imagen, pero controlado directamente por *HServer C*, por lo que no necesita el módulo *base* de CARMEN.

Todas las comunicaciones entre los módulos de CARMEN para cada robot se llevan a cabo mediante su propio servidor IPC, como es habitual en CARMEN. De forma que, si una misión contiene dos robots de CARMEN, debe de haber dos servidores IPC diferentes. Esto puede tener lugar en la misma máquina pero, en este caso, cada uno debe ejecutarse en un puerto distinto. Respecto a MissionLab, las comunicaciones se pueden realizar usando el servidor por defecto IPT, como en una ejecución normal de MissionLab. Sin embargo, pero se recomienda usar la nueva librería *IPC-Adapter* que hemos desarrollado en su lugar, ya que proporciona una completa compatibilidad con las últimas distribuciones de Linux. Los detalles sobre esta librería se explicarán en la sección 3.2.2.

Cada robot puro de MissionLab (que sólo utiliza drivers de MissionLab) tiene su propio proceso *HServer* (HSERVER B en la Figura 3.1. Esto no supone ningún cambio respecto a la versión oficial de MissionLab. Cada robot integrado de CARMEN en una misión de MissionLab debe tener su propio proceso *HServer* asociado (HSERVER A y HSERVER C en la Figura 3.1). En esta asociación, el control del hardware del robot se podría hacer bien por medio de un driver *base* de CARMEN (HSERVER A) o por un driver de MissionLab (HSERVER C).

Cuando se inicia un proceso *HServer* que tiene que controlar un robot de CARMEN, éste intercepta sus comunicaciones utilizando una nueva funcionalidad que hemos implementado para CARMEN (sección 3.2.4). De esta manera, *HServer* toma el control del robot, puede utilizar la información de localización de CARMEN, reenviar toda la información de odometría y lecturas de sensores a MissionLab y transmitir las órdenes de movimiento de MissionLab al robot. Esto se explica con más detalle en la sección 3.3.1.

Las capacidades de navegación de CARMEN se integran en MissionLab a un nivel superior de abstracción. Nuevos comportamientos CDL de MissionLab son capaces de enviar comandos de navegación a CARMEN y recibir de CARMEN las órdenes de movimiento. Así, se pueden fusionar con la salida de otros comportamientos de MissionLab, si se desea. Esto se explica en la sección 3.3.2.

Para sustentar este diseño, tuvimos que hacer algunas mejoras en IPC, implementamos un nuevo mecanismo para permitir la intercepción de los mensajes de CARMEN, creamos nuevos drivers de *HServer* capaces de comunicarse con los módulos *base*, laser y *localize* de CARMEN, y dos nuevos comportamientos CDL para integrar las capacidades de navegación de CARMEN con el editor gráfico de MissionLab (*CfgEdit*).

Distinguiremos entre lo que hemos denominado *integración a bajo nivel*, que incluye la integración de las lecturas de los sensores, el movimiento del robot y la localización; y la *integración a alto nivel*, que incluye la navegación y la combinación de las decisiones de movimiento de CARMEN con otros comportamientos de MissionLab.

3.2. Retos de implementación

En esta sección vamos a explicar los cuatro grandes tareas que hemos desarrollado para sustentar nuestro diseño: la migración de MissionLab a las nuevas distribuciones de Linux, la nueva librería *IPC_ Adapter* que reemplaza IPT en MissionLab usando IPC, el soporte para conectar varios servidores en aplicaciones multihilo en IPC y un nuevo mecanismo que permite interceptar los mensajes de CARMEN.

3.2.1. Compatibilidad con distribuciones Linux recientes

Para poder cumplir con nuestros objetivos, tuvimos que portar MissionLab a las distribuciones más recientes de Linux. Esto requirió resolver muchos pequeños problemas debidos a la evolución de librerías de terceros, corregir *bugs*, fugas de memoria, pero también algunos problemas no triviales.

En primer lugar, fue necesario reemplazar la librería de hilos utilizada por MissionLab (*cthreads*) porque desde hace años que no tiene soporte y utiliza algunos *hacks* que ya no son válidos en las distribuciones recientes de Linux. Para reemplazarla elegimos *pthread*, ya que actualmente es el estándar más extendido.

A continuación, tuvimos que reemplazar la librería de comunicación usada por MissionLab (IPT) porque no es *thread-safe*. Funciona de forma correcta con la librería de hilos que utiliza la versión original de MissionLab (*cthreads*) porque esta librería es a nivel de usuario y los cambios entre hilos sólo se producen en las llamadas a esta librería, todas ellas fuera del código IPT. En realidad, esta biblioteca simula la existencia de varios hilos de proceso utilizando sólo uno, de manera que nunca pueden ejecutarse de forma concurrente. Sin embargo cuando se utiliza cualquier biblioteca como *pthread*, con hilos de proceso reales que pueden ejecutarse concurrentemente en distintos procesadores y parar o arrancar en cualquier momento, los problemas de IPT se ponen al descubierto y MissionLab deja de funcionar correctamente.

Para reemplazar IPT, seleccionamos la librería de comunicaciones que utiliza CARMEN (IPC) por varias razones: soporta aplicaciones multi-hilo, permite que los módulos se conecten a la vez a varios servidores IPC, y tiene muchas similitudes con IPT porque ambas son ramas (*forks*) del mismo proyecto (TCA). Entre estas similitudes se encuentra el uso del mismo formato para definir los mensajes (XDR [73]). Además, se ha utilizado y probado en proyectos importantes (en CMU, NASA, DARPA), y se distribuye bajo licencia BSD simplificada, que nos permite modificar y redistribuir el código dentro de proyectos como éste.

3.2.2. El módulo puente IPC-Adapter

IPC-Adapter es un componente nuevo que hemos creado para poder reemplazar IPT por IPC en MissionLab. Este componente (ver figura 3.2) presenta la misma interfaz externa de IPT (por eso el recubrimiento en azul claro), pero implementada internamente mediante IPC.

IPT proporciona más posibilidades que IPC, pero MissionLab no utiliza todas ellas. Puesto que nuestro objetivo no era reimplementar IPT completamente, sino sustituirlo por IPC en MissionLab, antes de llevar a cabo la sustitución reducimos la librería IPT a lo esencial, dejando solamente el conjunto de funcionalidades utilizado por MissionLab y descartando el resto. A continuación, procedimos a implementar la interfaz resultante mediante IPC.

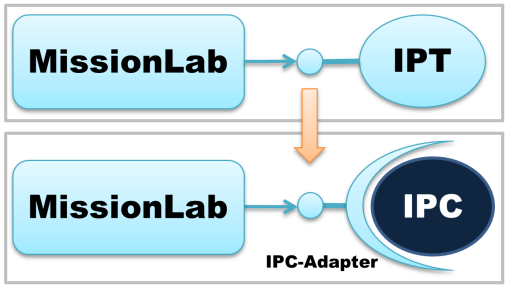


Figura 3.2: Reemplazo de IPT por IPC gracias a (*IPC-Adapter*)

Implementamos el registro y envío de mensajes *broadcast* de forma sencilla, puesto que ambas librerías definen los mensajes utilizando el mismo formato y ambas permiten difundir los mensajes a todos los módulos suscritos a ellos. Sin embargo, por lo general MissionLab no utiliza este tipo de mensajes. Principalmente usa mensajes privados, dirigidos únicamente de un módulo a otro. De esta forma, MissionLab puede gestionar sin problemas decenas o cientos de módulos y misiones multi-robot sin que se produzcan conflictos ni interferencias entre mensajes del mismo tipo enviados por distintos módulos o robots. Este comportamiento fue un problema para nuestro nuevo componente IPC-Adapter, ya que IPC no tiene la habilidad de enviar mensajes privados.

Para resolver este problema, diseñamos una política que permite a IPC-Adapter enviar mensajes sólo a los destinatarios deseados. Cada vez que un módulo de MissionLab registra un mensaje usando la interfaz de IPT, el módulo IPC-Adapter registra dos mensajes: uno con el mismo nombre para manejar los mensajes de *broadcast* y las respuestas a los mensajes (por ejemplo *NombreMensaje*); y otro que concatena el nombre del módulo y el nombre del mensaje para manejar los mensajes directos (por ejemplo *NombreModulo_NombreMensaje*). De esta manera, cuando Missionab quiere enviar mensaje directos, IPC-Adapter envía un mensaje de *broadcast* utilizando el nombre original, aunque con el nombre del módulo concatenado con el nombre del mensaje. Esto asegura que sólo los módulos correctos reciben estos mensajes, ya que sólo ellos los han registrado con este nombre. Esta solución tiene una ventaja adicional. Permite depurar el funcionamiento de MissionLab de forma más fácil y compartir información, puesto que finalmente todos los mensajes son compartidos con todos los procesos conectados y es posible recibirlos si se desea.

Otro punto importante con respecto a la implementación de IPC-Adapter son las funciones receptoras de mensajes (*callbacks*). Puesto que las funciones que reciben los mensajes en IPC e IPT no son compatibles, IPC-Adapter implementa un *callback* genérico con el formato utilizado en IPC, que recibe los mensajes, realiza las conversiones de datos necesarias y los redirige a *callbacks* con formato IPT.

El componente IPC-Adapter resultante genera un librería que MissionLab puede enlazar sin cambiar una sola línea de código.

3.2.3. Mejoras en la librería IPC

El principal reto para utilizar IPC como el sistema de mensajes de nuestro RDE integrado fue conseguir que fuese capaz de gestionar conexiones con varios servidores en un entorno multi-hilo. La versión oficial de IPC permite su uso en aplicaciones multi-hilo, y también la conexión con varios servidores de mensajes; sin embargo, no funciona correctamente cuando se hacen ambas cosas a la vez. Debido a problemas de implementación, cuando un hilo envía un mensaje o una respuesta a un servidor, en determinadas circunstancias los mensajes se envían al servidor equivocado.

Puesto que MissionLab es un software altamente multi-hilo (cada comportamiento se ejecuta en su propio hilo y una misión relativamente sencilla puede tener decenas o cientos de comportamientos simples asociados) es un gran entorno de pruebas para IPC y hace que sus problemas aparezcan rápidamente.

Esto nos ha permitido solucionar los problemas, arreglando algunos fallos en la gestión multi-hilo que provocaban interbloqueos y pérdidas importantes de rendimiento y reimplementando parte de la gestión de conexiones con distintos servidores a la vez.

El resultado es una biblioteca de comunicaciones mucho más robusta, con un rendimiento superior al de IPT, al poder aprovecharse de una ejecución multi-hilo y sin ningún tipo de efecto secundario ni inconveniente no deseado. Puesto que el código fuente de nuestro RDE está disponible públicamente y la licencia lo permite, cualquier desarrollador puede aprovecharse de estas correcciones y mejoras si lo desea.

3.2.4. Control de robots de CARMEN

Una de las especificaciones en las que se ha basado el desarrollo de nuestro RDE integrado, era que MissionLab y CARMEN mantuvieran una total compatibilidad hacia atrás. Es decir, que pudieran ejecutarse por separado de la misma forma que las versiones originales, y que fuesen compatibles con desarrollos de terceros, dependientes de estas plataformas, sin necesidad de hacer ningún tipo de modificación. Pero, al mismo tiempo, MissionLab tenía que ser capaz de manejar robots controlados por CARMEN, para poder utilizarlos en sus misiones y con sus propios comportamientos.

La forma más elegante que encontramos para conseguirlo fue que MissionLab pudiese tomar el control de los robots de CARMEN en tiempo de ejecución. Así, tanto CARMEN como MissionLab pueden ejecutarse normalmente por separado, y sólo cuando van a trabajar juntos, MissionLab toma el control.

Para permitirle tomar el control de los robots, hemos implementado una nueva funcionalidad, denominada *IPC_hook*, que permite interceptar los mensajes enviados a través de IPC. Así, MissionLab puede utilizar esta funcionalidad para interceptar mensajes claves de CARMEN (como la posición estimada o las órdenes de movimiento) procesarlos y reenviarlos modificando la información que sea necesaria.

La nueva funcionalidad *IPC_hook* se ha integrado junto al resto de funciones de IPC. Su uso es sencillo. Sólo hay que especificar el nombre del mensaje que se quiere redirigir, y el nombre con el que se enviarán realmente estos mensajes. Una vez utilizada esta función, IPC reconoce los mensajes con el nombre especificado y les cambia el nombre antes de enviarlos.

Contando con esta funcionalidad en IPC, la redirección de mensajes de CARMEN se hizo más sencilla. Como todos los módulos de CARMEN se conectan a los servidores IPC utilizando la función *carmen_ipc_initialize*, la modificamos para registrar un nuevo mensaje (*CARMEN_GLOBAL_HOOK_MSG*). En el manejador del mensaje, recibimos exactamente los parámetros necesarios para usar *IPC_hook* (nombre del mensaje original y nombre del mensaje redirigido) y los utilizamos para llamar a la nueva función. Así, MissionLab puede enviar este tipo de mensajes al servidor IPC y forzar a todos los módulos de CARMEN conectados a redirigir sus propios mensajes.

3.3. Integración de MissionLab y Carmen

En esta sección, vamos a indicar cómo se ha conseguido la integración entre MissionLab y CARMEN. Dividiremos el trabajo en dos partes que llamaremos *integración a bajo nivel* e *integración a alto nivel*. Cuando hablamos de integración a bajo nivel, nos referimos a la compatibilidad entre los drivers de ambas plataformas, tanto de bases de robot como de sensores sonar o láser. Respecto a la integración a alto nivel, comprende el desarrollo de nuevos comportamientos para MissionLab que pueden servirse de los módulos de navegación de CARMEN.

3.3.1. Integración a bajo nivel

Dos de los objetivos de nuestro RDE integrado eran permitir utilizar indistintamente drivers de CARMEN o MissionLab, y aprovechar las mejores funcionalidades de ambos RDEs. Para ello, es necesario que toda la información (odometría, láser, sonar) se publique de forma que sea accesible en ambos sistemas.

Si usamos drivers de CARMEN en nuestro robot, es necesario que la información se propague hasta los ejecutables del robot (instancias de *HServer*) y a la herramienta de monitorización *Mlab*, tal y como hacen los drivers nativos de MissionLab. Cuando usamos drivers de MissionLab, es necesario que la información obtenida se publique a través de IPC, de la misma forma que lo hacen los drivers de CARMEN, para que pueda ser aprovechada por los algoritmos de navegación y localización. En la figura 3.3 se muestra una visión general de esta integración.

Para poder usar drivers de CARMEN en MissionLab, hemos desarrollado tres nuevos drivers en *HServer*, que en la figura 3.3 se muestran en color magenta (dentro de *HServer* en azul claro):

- El nuevo driver llamado *CARMEN BASE DRIVER* recibe los datos de odometría y sonar procedentes del módulo *base* de CARMEN.
- Otro nuevo driver llamado *CARMEN LASER DRIVER* obtiene los datos procedentes del módulo *laser* de CARMEN

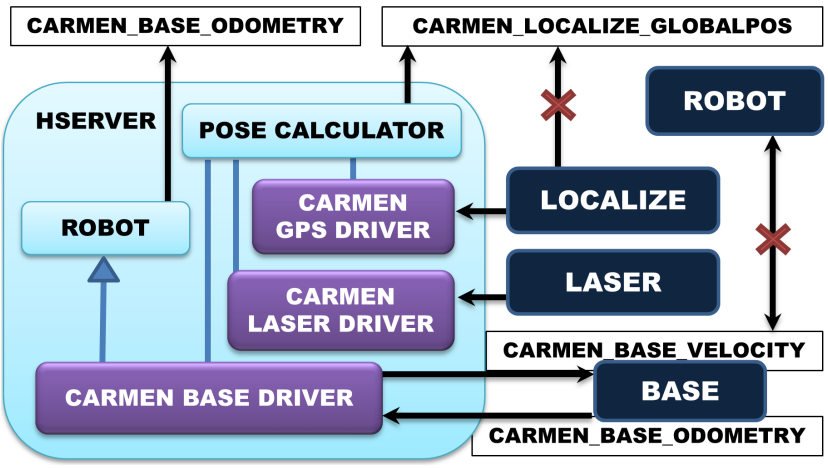


Figura 3.3: Integración de la información de localización y lecturas de sensores

- Un tercer driver de GPS (*CARMEN GPS DRIVER*) obtiene la localización global calculada por CARMEN (procedente del módulo *localize*) y se la transmite al algoritmo de *HServer* encargado de procesar todas las informaciones relativas a la posición del robot.

Para realizar su trabajo, los nuevos drivers sólo necesitan conocer la IP y el puerto que utilizan los módulos de CARMEN asociados para comunicarse. Los usuarios de MissionLab pueden aportar esta información de forma interactiva en la consola de *HServer*, o la pueden introducir en el propio archivo de configuración de *HServer* (*.hserverrc*), para poder arrancarlos de forma automática.

Para poder utilizar drivers de MissionLab en CARMEN, hemos modificado la clase *Robot* de *HServer* (en la figura 3.3 aparece en azul claro), de la que heredan todos los drivers. La modificación consiste en publicar la odometría y los datos del sonar cuando éstos son obtenidos directamente por *HServer* del hardware del robot. Esta funcionalidad se puede activar al arrancar *HServer* (utilizando la opción *-s* seguido del *host* y el puerto en el que se encuentra el servidor IPC).

Otro punto clave de la integración es que necesitamos mantener una decisión final única sobre la posición del robot en MissionLab y CARMEN para que trabajen de forma coordinada. Cuando los dos RDEs colaboran, MissionLab siempre tiene la última palabra sobre la posición del robot. Hemos tomado esta decisión de diseño porque *HServer* permite fusionar la información de posición a

partir de múltiples fuentes, utilizando su mecanismo de fusión (*PoseCalculator* en Figura 2.9) basado en un filtro Kalman o un filtro de partículas (dependiendo de la configuración). En algunos escenarios, la localización de CARMEN puede perder precisión y, en estos casos, resulta más interesante dar mayor peso a otras fuentes, tales como odometría y otros sensores que puede incorporar el robot, como un GPS, una brújula o un acelerómetro. En nuestra implementación hemos utilizado los mensajes *CARMEN_LOCALIZE_GLOBALPOS* (que proporcionan la mejor estimación de la posición del robot en CARMEN) como una entrada GPS al algoritmo de fusión de información (*PoseCalculator*) de *HServer* por medio del nuevo driver *CARMEN GPS DRIVER*, y se tiene en cuenta la precisión de cada estimación en *HServer* para fusionarlo con otras fuentes de datos de posición de la mejor manera posible.

Para conseguir controlar completamente los robots de CARMEN, *HServer* intercepta dos mensajes clave: *CARMEN_LOCALIZE_GLOBALPOS* y *CARMEN_BASE_VELOCITY* que son generados por los módulos *localize* y *robot* de CARMEN respectivamente. El primero es interceptado por el nuevo driver de GPS (representado en la figura 3.3 por una *X* en rojo) y después se reenvía, con la posición fusionada que es calculada por *HServer*, para que se pueda utilizar en el resto de módulos de CARMEN. El segundo (*CARMEN_BASE_VELOCITY*) se reenvía de forma que los comportamientos en la misión de MissionLab puedan aprovechar esa información (ver sección 3.3.2) y es finalmente reenviado por *HServer* con la decisión final de movimiento tomada por MissionLab

3.3.2. Integración a alto nivel

Las capacidades de navegación de CARMEN se han integrado con el resto de comportamientos de MissionLab (figura 3.4). Para ello, hemos implementado un nuevo comportamiento denominado *CARMEN_NAVIGATE*, que utiliza los mensajes *CARMEN_NAVIGATOR* para controlar el módulo *navigator* de CARMEN. Este comportamiento también se mantiene a la escucha de las instrucciones de movimiento generadas por CARMEN, suscribiéndose al mensaje *CARMEN_BASE_VELOCITY* que es enviado por el módulo *robot* de CARMEN. Para probar la integración de este nuevo comportamiento sim-

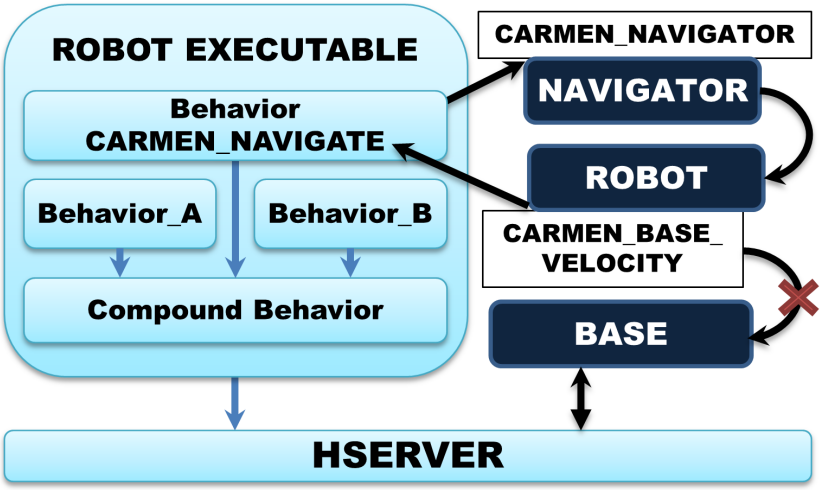


Figura 3.4: Integración de las capacidades de navegación de CARMEN con MissionLab

ple con el resto, se han creado dos tareas que pueden utilizarse en *CfgEdit*: *CARMEN_ GoTo* y *CARMEN_ Navigate*. El primero combina la navegación de CARMEN con el comportamiento para evitar obstáculos de MissionLab y con el que permite teleoperar el robot, mientras que el segundo se limita a utilizar directamente las órdenes de movimiento proporcionadas por CARMEN.

Como es habitual en MissionLab, podemos utilizar *CfgEdit* para generar misiones complejas con muchos estados y *triggers* usando estos nuevos comportamientos. Esto nos permite, por ejemplo, crear de forma fácil una misión donde un robot se mueve por varios lugares en un mapa gracias a la navegación proporcionada por CARMEN, y a la vez realiza otras actividades, como coger objetos en esos lugares, atender a su nivel de batería para ir al punto de recarga si es necesario, etc.

De esta forma, hemos conseguido integrar con éxito dos de los RDEs más importantes disponibles de código abierto, y ambos mantienen una total compatibilidad hacia atrás para utilizarlos tanto de forma separada como conjunta. Nuestro RDE integrado permite definir misiones multi-robot utilizando *CfgEdit*, donde cada uno de los robots puede estar controlado por CARMEN, por MissionLab o por ambos. Así, se consigue aprovechar las fortalezas de ambos RDEs.

4

Método propuesto de localización global

En este capítulo presentamos un método de localización global para robots en interiores que está basado en un filtro de partículas concurrente y la información sobre el entorno que se encuentra en los planos de los edificios. El método propuesto presenta novedosas contribuciones, entre las que se encuentran: utilizar un GIS donde se almacena información de alto nivel sobre el entorno; la ejecución de un segundo filtro de partículas respondiendo a estímulos ambientales; y colaboración entre dos filtros de partículas (un filtro de corto plazo y un filtro de largo plazo) que se ejecutan de forma concurrente. Con ello se consigue que sea especialmente idóneo para entornos extensos, que tenga una alta tolerancia a errores en los sensores, en el modelo de movimiento del robot o en el mapa, y que tenga una alta tasa de acierto en un tiempo de respuesta corto.

4.1. Arquitectura de nuestro filtro de partículas

La primera contribución de nuestra propuesta es que la información del entorno necesaria la proporciona un GIS mediante consultas espaciales. En nuestra institución, como en la mayoría, existe un servicio que gestiona y mantiene los planos de sus edificios. Además, existen diferentes herramientas para importar automáticamente planos de edificios almacenados en los formatos más comunes (AutoCAD, SHP, etc.) en un GIS, por lo que el esfuerzo necesario para estos sistemas es muy pequeño en comparación con los beneficios que proporciona.

Si somos capaces de utilizar esta información para alimentar algoritmos de localización y disponemos de robots capaces de detectar hechos característicos como habitaciones, puertas, ventanas o cualquier otro elemento de esos mapas, evitamos la necesidad de crear mapas específicos para nuestro propósito. En otros planteamientos, se necesita una fase previa en la que se pasea el robot por el entorno para poder disponer de un mapa con los hechos característicos utilizados. Sin embargo, reutilizando los mapas preexistentes, podemos disponer de un servicio de localización global para todos los edificios de nuestra organización o de cualquier otra con un esfuerzo mínimo.

Entre los distintos hechos característicos de alto nivel que aparecen en estos mapas, decidimos usar las puertas para nuestro algoritmo de localización global en interiores. Su principal ventaja es que normalmente son bastante numerosas, están presentes en todas las habitaciones y son más fáciles de detectar que otros objetos, al estar bastante estandarizadas. El tamaño, la forma y el color de las puertas suele ser bastante uniforme, especialmente dentro del mismo edificio. Normalmente, están al nivel del suelo o sobre un peldaño, suelen tener un marco, etc. Otros hechos, como las ventanas, suelen tener formas, tamaños y posiciones más variadas. Las salas y los pasillos también se utilizan habitualmente para tareas de localización, utilizando como sistema de detección un láser de rango.

Para detectar puertas, utilizamos un método desarrollado en nuestro grupo de investigación [74], basado principalmente en técnicas de visión artificial, que puede funcionar en tiempo real en hardware de computación estándar (más de 20 iteraciones por segundo) y no requiere ninguna modificación o intervención del entorno.

Cuando consideramos el uso de filtros de partículas para localización global en entornos muy extensos, unos de los principales problemas es que el número de partículas necesario puede llegar a ser demasiado alto y no permitir la ejecución del algoritmo en tiempo real. Si no se cuenta con un método adecuado para inicializar la posición de las partículas y simplemente se inicializan de forma aleatoria, el número de partículas necesario para muestrear satisfactoriamente el espacio de estados aumenta con el área a muestrear. Por lo tanto, pueden ser necesarias muchos miles de partículas incluso para entornos relativamente pequeños (pocos cientos de metros cuadrados). La extensión del conjunto de edificios de nuestra Universidad (u otra organización similar) es de varios miles de metros cuadrados, por lo que el número de partículas necesarias para localizar un robot en este entorno sería demasiado alto como para ejecutar el algoritmo en tiempo real, tal como demuestran las pruebas que mostramos en el apartado 5.2.5.

La figura 4.5 muestra nuestro escenario experimental, que consiste en uno de los pisos de nuestra facultad (con capacidad para más de 4000 estudiantes) con una superficie de más de 5000 m^2. Podemos ver las partículas (en color rojo y verde), representando las posibles posiciones del robot, y en color negro las puertas que detectamos principalmente mediante técnicas de visión artificial. Estas técnicas son especialmente propensas a generar detecciones incorrectas o inexactas. Dependiendo de factores como la iluminación, la decoración, los obstáculos o la perspectiva de la cámara, es posible que ciertos objetos sean incorrectamente identificados como puertas, o que puertas existentes no puedan ser detectadas, dando lugar a falsos positivos y falsos negativos.

Como contribución novedosa que soluciona estos problemas, proponemos un filtro de partículas que responde a estímulos del entorno y que requiere, en conjunto, un número muy bajo de partículas. Al mismo tiempo, es robusto ante errores de los sensores y movimientos no modelados del robot.

Se ejecutan de forma concurrente dos filtros de partículas (figura 4.1), utilizando los mismos datos de entrada: el filtro primario, principal o de largo plazo, y un filtro secundario o de corto plazo. Ambos se inicializan en base a la salida de nuestro algoritmo de detección de puertas y la información procedente de nuestro mapa GIS para reducir el número de partículas necesario.

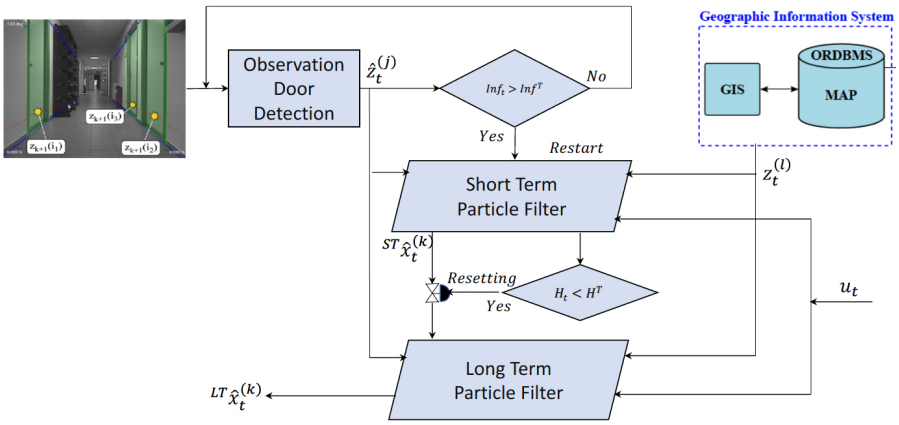


Figura 4.1: Arquitectura de nuestro filtro de partículas concurrente, que responde a estímulos ambientales

Utilizando técnicas de *clustering*, estimamos el estado de ambos filtros y proporcionamos una única salida (la localización estimada del robot).

Para que la salida sea robusta ante errores de los sensores o movimientos no modelados, el filtro secundario tiene un ciclo de vida corto, con sus propias condiciones de disparo y de finalización. Cuando se detecta información suficiente en el entorno, el filtro secundario se inicializa. Cuando converge, sus partículas participan en la fase de muestreo del filtro primario, obteniendo un pequeño porcentaje de su probabilidad total. Por tanto, nuestro filtro responde a estímulos del entorno.

En las siguientes secciones, detallamos en profundidad cómo con los dos filtros de partículas ejecutándose de forma concurrente se reduce el impacto de lecturas de sensores erróneas y movimientos no modelados del robot (sección 4.6). Utilizamos técnicas de clustering para estimar el estado de ambos filtros. Además, aplicamos optimizaciones adicionales (sección 4.5) a las fases de inicialización y muestreo para reducir en gran medida el número de partículas necesarias para conseguir resultados satisfactorios.

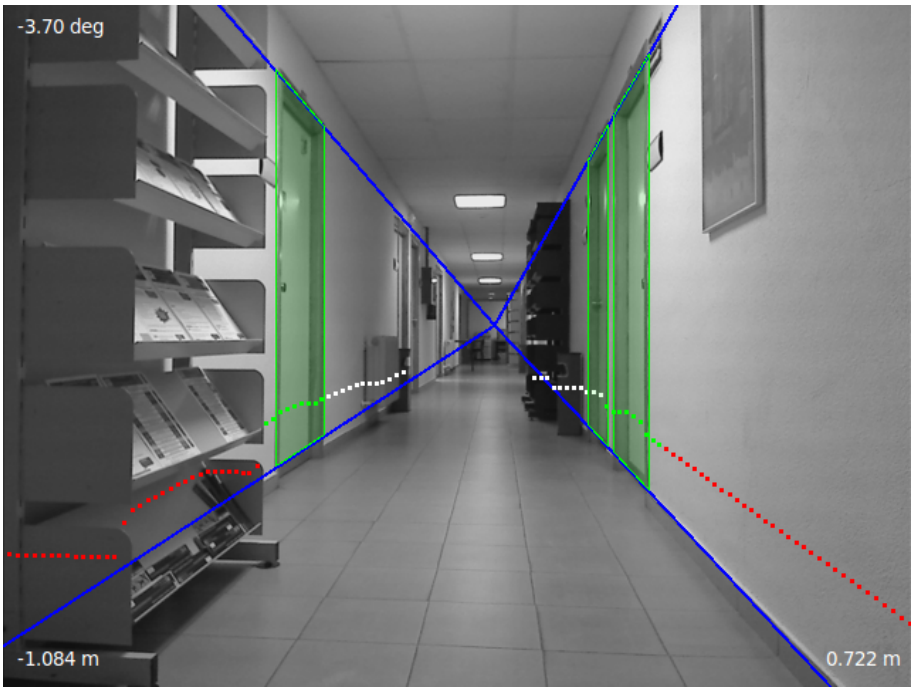
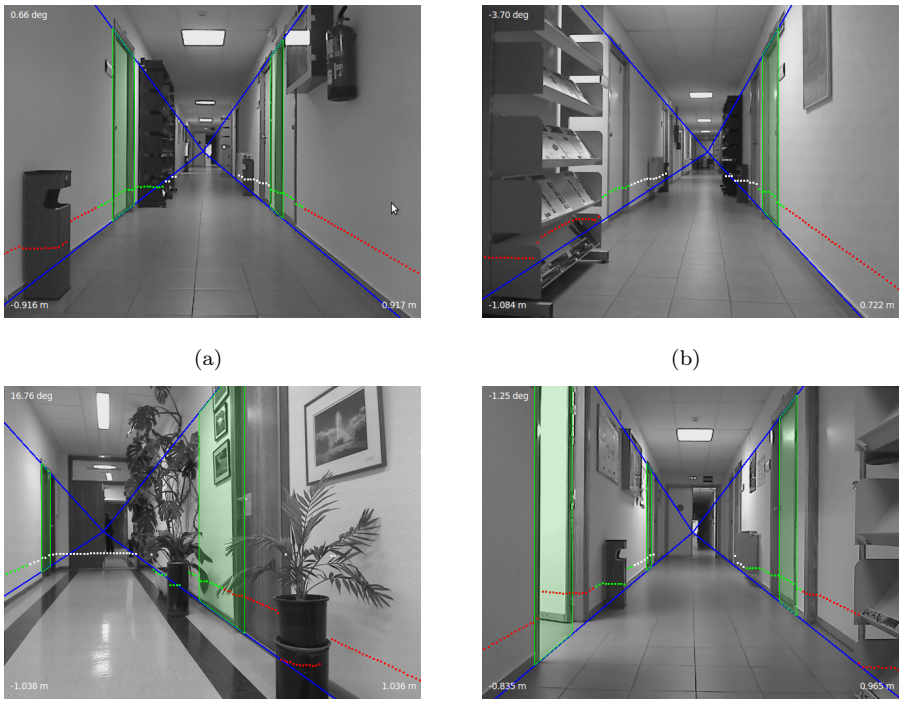


Figura 4.2: Visor del software de detección de puertas (*doorDetectionGui*)

4.2. Detección de puertas

El algoritmo de detección de puertas que utilizamos para nuestro método de localización global fue desarrollado recientemente por Fernández-Caramés, C. et al. [74] y se basa en técnicas de fusión sensorial: imágenes tomadas desde una cámara y lecturas láser en el robot. Este método utiliza la proyección de las lecturas láser sobre las imágenes, cálculos geométricos y técnicas de detección de líneas mediante características Haar. Es capaz de proporcionar la posición y orientación, relativas al sistema de referencia del robot, del punto medio de las puertas que se encuentran en el campo de visión del robot hasta una distancia de siete metros. En la figura 4.2 podemos ver de forma gráfica los resultados de la detección, gracias a la herramienta de visualización *doorDetectionGui*. El proceso de detección de puertas es muy rápido (con una velocidad de 40 iteraciones por segundo), no necesita un escenario preparado de antemano y contempla un amplio conjunto de puertas de diferentes características (color, forma, tamaño), bajo diferentes condiciones de iluminación y abiertas o cerradas.

Utilizamos la salida de este método para inicializar nuestros filtros y para evaluar sus partículas en cada fase de corrección. Desafortunadamente, como cualquier otro método de detección, no es perfecto y en algunos casos proporciona falsos positivos, falsos negativos y detecciones imprecisas, con una tasa media de aciertos obtenida en torno al 75 % en distancias inferiores a 7 metros. En la figura 4.3 podemos ver varios de estos problemas de detección.



(a) (b)

(c) (d)

Figura 4.3: Problemas en la detección de puertas

En la figura 4.3(a), debido a la cercanía de dos puertas en el lado derecho, no se han detectado correctamente y, en su lugar, se ha detectado una puerta inexistente entre las dos. En la figura 4.3(b), no se detecta correctamente una puerta en el lado izquierdo del pasillo. En la figura 4.3(c), la iluminación y la presencia de una planta hace que no se detecte la puerta en el lugar correcto. El error en la figura 4.3(d) es más sutil. Simplemente no se detectan con precisión los marcos de las puertas. Es un error de menor importancia, pero su presencia provoca importantes pérdidas de precisión en la detección, por lo que es necesario trabajar con unos márgenes de error elevados.

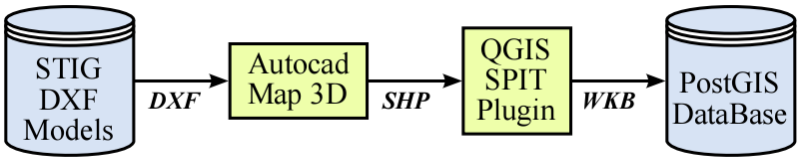


Figura 4.4: Proceso de conversión de mapas DFX a PostGIS

Para mitigar los efectos negativos de estos errores, hemos implementado varias mejoras en distintas fases estándar de los filtros de partículas (sección 4.5) y un doble filtro de partículas comentado en la sección 4.6.

4.3. Mapas GIS del entorno

Toda la información sobre el entorno necesaria para nuestro algoritmo de localización global (aparte de la percepción del propio robot) se almacena en un GIS y se extrae en tiempo de ejecución. Nuestro GIS está basado en una base de datos PostgreSQL extendida con PostGIS, una extensión espacial para la base de datos PostgreSQL. Esta configuración le permite a nuestro algoritmo obtener información sobre el entorno, determinados hechos característicos, utilizando SQL y consultas geométricas. Aprovechando estas funcionalidades, somos capaces de obtener la localización de todas las puertas en el entorno del robot, buscar posibles puertas candidatas para compararlas con las puertas detectadas por el robot durante la inicialización y crear un mapa de probabilidad para evaluar las partículas en la fase de corrección.

Actualmente, los planos de cualquier edificio moderno están disponibles en formatos CAD o BIM. Además, es bastante probable que, en un futuro cercano, el número de planos de interiores disponibles de forma gratuita experimente un gran crecimiento. Por el momento, las grandes instituciones o empresas normalmente cuentan con algún tipo de servicio o departamento responsable de los planos de las distintas plantas de cada edificio. Por ejemplo, en la Universidad de Salamanca, los planos CAD los mantiene y actualiza de forma regular el departamento STIG (Servicio Transfronterizo de Información Geográfica). Dicho departamento nos proporcionó archivos en formato DFX (AutoCAD Drawing Interchange Format) con los planos de la Facultad de Ciencias.

Para importar archivos DXF en un sistema GIS, utilizamos el proceso de conversión representado en la figura 4.4. Exportamos los archivos originales en formato DFX (*AutoCAD Drawing Interchange Format*) a formato SHP (*shapefile*), y finalmente importamos el archivo resultante en una base de datos PostgreSQL utilizando el plugin *SPIT* del software *Quantum GIS* (QGIS) [75]. De esta forma, no fue necesario crear un nuevo mapa para nuestro propósito, ni obtener manualmente información del entorno, ni utilizar el robot para hacer *mapping*.

4.4. Clustering

Utilizamos un algoritmo de clustering simple y rápido para agrupar las partículas. Empezando con un conjunto vacío de clusters, iteramos sobre el conjunto de partículas. Cuando una partícula está más cerca de un cierto umbral del centro de un cluster existente, añadimos la partícula al cluster, añadimos la probabilidad de la partícula a la probabilidad del cluster, y modificamos el centro del cluster para que coincida con la media ponderada de las posiciones de las partículas que contiene. De lo contrario, creamos un nuevo cluster con centro en la posición de la partícula y con su misma probabilidad. Los clusters resultantes se pueden ver en un instante de la ejecución de nuestro caso de estudio en la figura 4.5. Los círculos con fondo verde son los clusters del filtro secundario, mientras que los rojos son los del filtro primario.

Utilizamos estos clusters para medir la convergencia de los filtros. Representando la probabilidad de cada cluster como $\hat{p}_t^C$, consideramos la *entropía* H_t definida como

$$H_t = -\sum_{cluster\ C} \hat{p}_t^C log_2\left(\hat{p}_t^C\right) \tag{4.1}$$

Cuando H_t toma un valor alto, el filtro se encuentra muy disperso y necesita continuar trabajando para proporcionar una salida fiable. Cuando H_t tiene un valor bajo, las partículas se encuentran muy agrupadas en unos pocos lugares que acumulan la mayor parte de la probabilidad.

Este valor es esencial para la interacción entre el filtro primario y el secundario en nuestro método y también para proporcionar una única salida fiable.

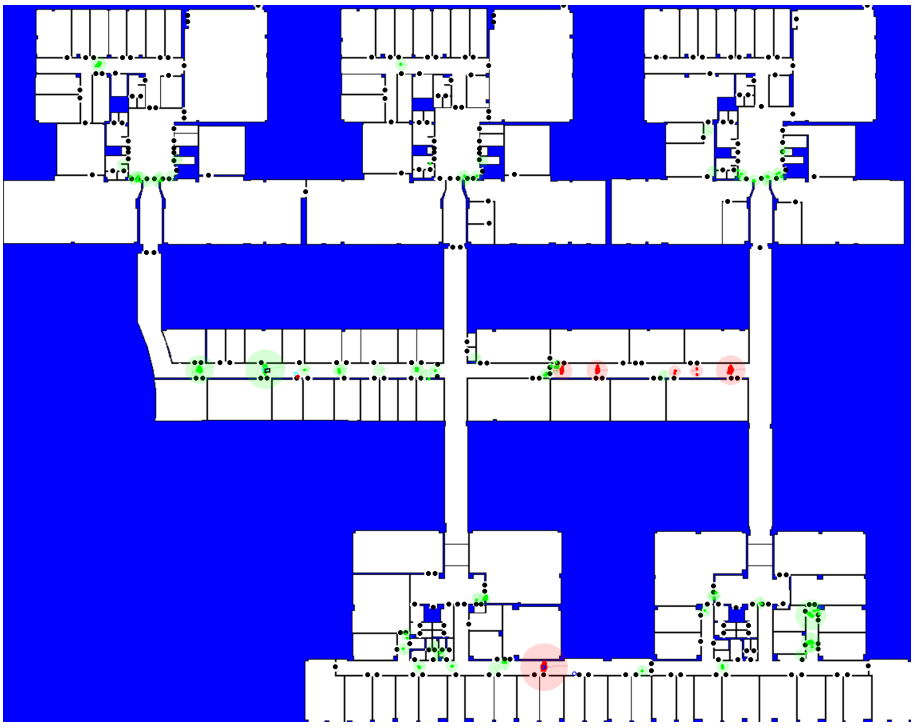


Figura 4.5: Escenario real de nuestros experimentos de localización global. La superficie habitable es de más de 5.000 m^2.

La posición del cluster con mayor probabilidad de todos en el filtro primario se considera la salida de nuestro algoritmo, cuando el valor de H_t se encuentra por debajo de un cierto umbral.

4.5. Mejoras en las fases estándar de los filtros de partículas

Las mejoras se han introducido principalmente la fase de inicialización para reducir el número de partículas necesarias para obtener resultados correctos. Además, hemos implementado nuestra fase de corrección en base a la información procedente de nuestro GIS, y omitimos la fase de muestreo en algunas iteraciones del filtro para reducir el efecto adverso de detecciones de puertas incorrectas o poco precisas. En las siguientes secciones explicamos detalladamente estas mejoras.

4.5.1. Fase de inicialización

El número de partículas que requiere un filtro de partículas para realizar adecuadamente la localización global de robots móviles en general aumenta con las dimensiones del mapa. De la misma forma, la potencia de cálculo necesaria para ejecutar el filtro en tiempo real también crece. Cuando el hardware disponible no puede satisfacer los requisitos, es necesario realizar mejoras para conseguir buenos resultados con un conjunto reducido de partículas.

La mejora más utilizada en este sentido es seleccionar el conjunto inicial de partículas en base a la distribución de probabilidad generada por las lecturas de los sensores ($p\left(x_t|z_t\right)$). Esta mejora se propuso como parte del algoritmo SRL, en el que las partículas se sitúan en los lugares que maximizan $p\left(x_t|z_t\right)$. Más adelante, como parte del algoritmo Mixture-MCL, se propuso seleccionar las partículas muestreando esta misma distribución de probabilidad.

Como contamos con un mapa GIS como fuente de información, lo utilizamos para inicializar el filtro de acuerdo a la densidad de probabilidad generada por las medidas externas del robot, en nuestro caso, las puertas que es capaz de detectar. Para conseguirlo, agrupamos las puertas detectadas en todos los posibles pares, calculamos la distancia entre cada par de puertas y buscamos puertas en el mapa GIS que estén a esa misma distancia entre ellas.

Una posible mejora para reducir el impacto de los errores en el algoritmo de detección de puertas sería inicializar el filtro cuando el robot es capaz de percibir al menos tres puertas a la vez. Así, incluso si una de las puertas detectadas no existe realmente, otros pares de puertas situarían partículas en el lugar correcto. Puesto que no es muy común que el robot detecte más de dos puertas a la vez y buscábamos comprobar la eficiencia de nuestro doble filtro de partículas frente a errores en la inicialización, no aplicamos esta mejora, pero hacerlo sería trivial.

El algoritmo de detección de puertas que utilizamos proporciona las coordenadas cartesianas de los centros de las puertas ((Xn, Yn) por cada puerta n) respecto a la posición del robot. Mediante una conversión a coordenadas polares, se determina la distancia de cada puerta al robot, su orientación y la distancia entre cada par de puertas i y j:

$$Dij = \sqrt{(Xi - Xj)^2 + (Yi - Yj)^2} \tag{4.2}$$

Para buscar pares de puertas a esa distancia en nuestro entorno, realizamos una consulta espacial a nuestra base de datos PostGIS (figura 4.6).

```
SELECT ST_Centroid(g1.the_geom), ST_Centroid(g2.the_geom)
    FROM secondfloor as g1, secondfloor as g2
    WHERE g1."LAYER" = 'DOORS' and g2."LAYER" = 'DOORS'
        and ST_X(ST_Centroid(g1.the_geom)) <= ST_X(ST_Centroid(g2.the_geom))
        and ST_Distance(ST_Centroid(g1.the_geom), ST_Centroid(g2.the_geom))
            < (distance + error)
        and ST_Distance(ST_Centroid(g1.the_geom), ST_Centroid(g2.the_geom))
            > (distance - error);
```

Figura 4.6: Consulta espacial para buscar pares de puertas candidatas

donde *distance* es la distancia (4.2) entre el par de puertas detectadas por el robot y *error* es el error estimado en la medida.

Una vez que hemos obtenido nuestras puertas candidatas, las coordenadas (Xn, Yn) proporcionadas por nuestro algoritmo de detección se aplican de forma inversa para buscar las posibles localizaciones del robot. Así, cada par de puertas proporciona dos posibles posiciones del robot. En la figura 4.7 podemos ver dos de las posibles posiciones candidatas generadas a partir de la detección de dos puertas.

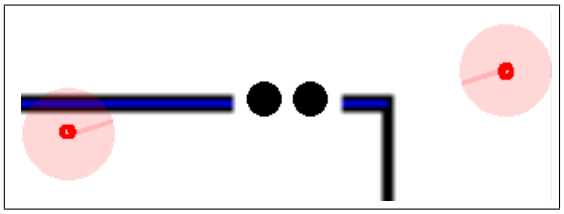


Figura 4.7: Posiciones candidatas a partir de la detección de dos puertas

Para finalizar la inicialización, las partículas del filtro de distribuyen entre esas localizaciones siguiendo una distribución Gaussiana con centro en las posiciones candidatas y una varianza acorde al error estimado en la detección de puertas. De esta forma, el número de partículas necesarias para explorar de forma efectiva el espacio de estados no depende de la extensión de nuestro entorno. Depende del número de posibles posiciones candidatas, que está limitado por el número de puertas en el mapa y su distribución, y que es mucho menor que su superficie.

4.5.2. Fase de corrección

En la fase de corrección de nuestro filtro de partículas, evaluamos cada una de ellas de acuerdo a las medidas externas (puertas detectadas por nuestro robot) y el resultado esperado desde la posición de cada partícula.

Para hacer la evaluación de la forma más eficiente posible, durante la inicialización de nuestro programa realizamos una consulta a nuestro GIS para obtener la posición de todas las puertas y generar un mapa de probabilidad en tiempo de ejecución. En dicho mapa, cada celda contiene el valor de la probabilidad de detectar una puerta en ese punto a una determinada distancia.

Como se puede observar en la representación del mapa de probabilidad en la figura 4.8, sólo utilizamos tres colores (distintos valores de probabilidad). Un valor de *acierto* con la probabilidad más alta (el color más oscuro que indica la posición de las puertas), un valor de *fallo* con una probabilidad intermedia (el color gris oscuro que indica la superficie habitable), y un valor de *error* (el color gris muy claro) con una probabilidad muy baja.

Jugando con la diferencia de los valores de *acierto* y *fallo*, podemos hacer el filtro más "agresivo" (para acelerar su convergencia y obtener resultados de forma más rápida) o más conservador (para ralentizar su convergencia y obtener resultados más fiables a costa de necesitar más tiempo). El valor más bajo de probabilidad se lo asignamos a todos los puntos que están fuera de la superficie habitable del mapa, con la intención de eliminar rápidamente las partículas que se encuentran en lugares completamente erróneos.

Utilizando el mapa de probabilidad, sólo tenemos que sumar las coordenadas relativas de cada puerta detectada a la posición de cada partícula y multiplicar su probabilidad por el valor que se encuentra en la celda correspondiente.

4.5.3. Fase de muestreo

La fase de muestreo de los filtros de partículas tiene la intencionalidad de explorar el espacio de estados de forma eficiente, situando más partículas en las zonas con más probabilidad de ser la posición real del robot. Sin embargo, también puede eliminar todas las partículas de áreas que tienen una probabilidad baja, aunque no nula, de ser la posición verdadera del robot.

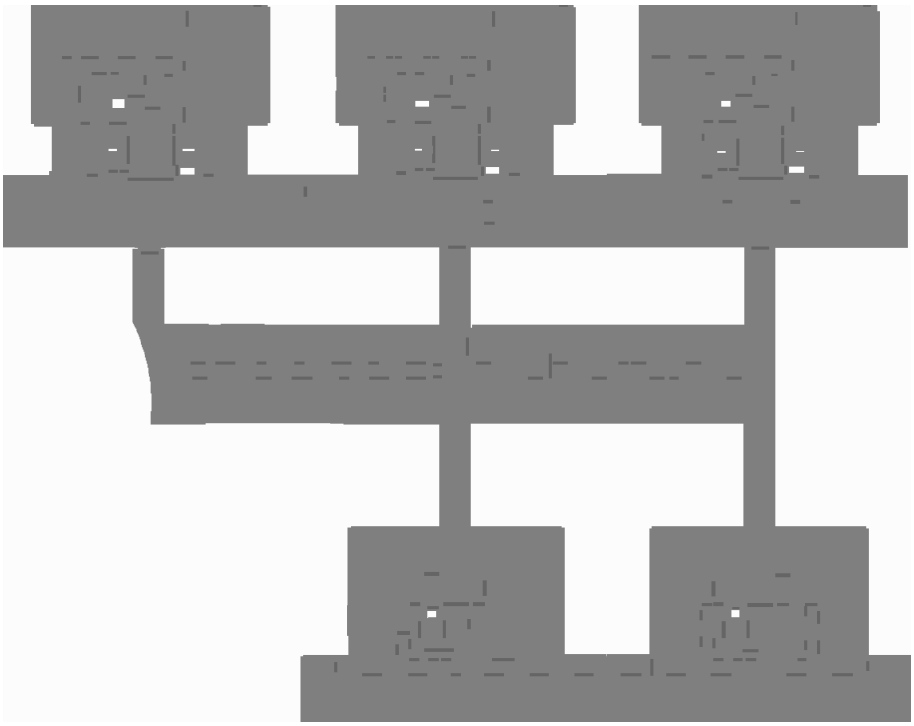


Figura 4.8: Mapa de probabilidad creado en tiempo de ejecución con información procedente de nuestro GIS

Si todas las partículas cercanas a esta posición del robot desaparecen, el filtro fallará, a no ser que se aplique algún tipo de solución para volver a muestrear esa posición.

Una solución para reducir este efecto no deseado es omitir la fase de muestreo en algunas iteraciones del filtro. Al hacer esto, debemos propagar los pesos (probabilidad) de las partículas a la siguiente iteración, para no perder la información obtenida durante la fase de corrección.

$$w_t^{(i)} = w_{t-1}^{(i)} p\left(z_t | x_t^{(i)}\right) \tag{4.3}$$

Esta modificación ya fue propuesta [58] para prevenir la pérdida de diversidad en las posiciones de las partículas y está implementada en el algoritmo de localización de CARMEN, proporcionando buenos resultados. En nuestro algoritmo, ha probado ser útil para reducir los efectos adversos de los errores puntuales en la detección de puertas.

4.6. Filtro de partículas concurrente

El enfoque que proponemos está inspirado en el comportamiento natural de los humanos cuando tratamos de localizarnos. En cualquier momento, una persona puede tener una certeza razonable de encontrarse en un determinado lugar. Sin embargo, cuando se percibe un hecho significativo en el entorno que entra en conflicto con esa idea (quizá un monumento o cualquier otro hecho característico que resulte familiar para la persona), si es suficientemente importante, puede crear una duda y la persona puede empezar a considerar una segunda hipótesis sobre su posición. Ambas, la idea inicial y esta segunda hipótesis, pueden evaluarse en paralelo y, si la duda se hace suficientemente grande en base a las siguientes percepciones del entorno, la persona puede cambiar de opinión.

Aplicando estas ideas al ámbito de la robótica, nuestro enfoque se describe en la figura 4.1. Para localizar el robot en un periodo de tiempo corto, el filtro debe muestrear la posición en la que realmente se encuentra y eso normalmente requiere el uso de una enorme cantidad de partículas cuando se quieren explorar superficies muy grandes (en nuestro caso, de más de 5000 m^2). En nuestra propuesta, el número de partículas es significativamente menor gracias a nuestra inicialización basada en las puertas detectadas por el robot y datos procedentes de nuestro GIS (sección 4.5.1).

Cuando consideramos la información que el robot percibe del entorno (las puertas cercanas delante de nuestro robot) está claro que, en la mayor parte de los casos, dadas las similitudes y simetrías que aparecen frecuentemente en escenarios reales y a la limitada precisión de nuestras percepciones, no es posible determinar la posición del robot en base únicamente a las primeras puertas detectadas. En estos casos, si el número de partículas del filtro es bajo y la precisión de los sensores no es lo suficientemente alta, es probable que el filtro converja hacia una posición errónea. Por esa razón, es muy importante aprovechar la información significativa que puede aparecer frente a nuestro robot en cualquier momento. En nuestro caso, esa información significativa puede ser una acumulación de puertas en pocos metros, o una disposición de las puertas muy distinta a la del resto de nuestro entorno. Debido a este comportamiento, decimos que nuestro método *responde a estímulos del entorno.*

Puesto que esa información significativa puede aparecer frente a nuestro robot en cualquier momento (que puede no ser en el estado inicial) proponemos el uso de un filtro de partículas secundario, que se inicia cada vez que el entorno ofrece información sobre la posición del robot, es decir, cuando el número de medidas externas es superior a un cierto umbral. A este evento lo llamaremos el *evento de disparo*, que sucede cuando

$$Inf_t > Inf^T$$

donde Inf^T es el umbral seleccionado y Inf_t está definido como:

$$Inf_t = card\left\{\hat{z}_t^{(j)}\right\}$$

En nuestro caso, las medidas externas son puertas en frente del robot $z_t^{(j)}$ detectadas por el método de observación desarrollado por Fernandez-Carames, C. et al. [74], por lo que Inf_t es el número de puertas detectadas.

Durante nuestra fase experimental, notamos que, en nuestro entorno, es posible detectar varias puertas a la vez en varios lugares, a pesar de que la mayor parte del tiempo el robot detecta sólo una o ninguna puerta. En esos momentos, inicializamos el filtro secundario de acuerdo a la distribución de probabilidad obtenida en base a las puertas detectadas (sección 4.5.1).

En este sentido, nuestra propuesta es similar a SRL o Mixture-MCL. La principal diferencia es que, en dichos métodos, las nuevas partículas se introducen inmediatamente en el filtro de partículas y, como hemos comentado anteriormente, eso causa problemas graves en presencia de errores de los sensores. En nuestro planteamiento, las nuevas partículas se evalúan en el filtro secundario, sin afectar inicialmente a la estabilidad del filtro primario ni a la salida de nuestro algoritmo.

Para evaluar la calidad de las partículas del filtro secundario, utilizamos la entropía del filtro (H_t) (fórmula 4.1), como una medida de la dispersión de las partículas. Cuando el valor de la entropía es suficientemente bajo, las partículas en el filtro secundario participan en la fase de muestreo del filtro primario (obteniendo un pequeño porcentaje de su probabilidad). Justo después de esa fase híbrida de muestreo, el filtro secundario se *resetea* y se mantiene parado para inicializarse en el siguiente *evento de disparo*.

Por ello, definimos la *condición de reseteo* como

$$H_t < H^T \tag{4.4}$$

donde H^T indica la madurez del filtro secundario. Este valor debe ser suficientemente bajo como para asegurar que el filtro ha recibido suficiente información externa para poder agrupar la mayor parte de sus partículas en unos pocos clusters con una alta probabilidad. En nuestro caso, determinamos este valor de forma experimental.

Gracias a esta fase de muestreo híbrida, algunas partículas de esos clusters de alta probabilidad del filtro secundario empiezan a evaluarse en el filtro primario. Esto marca una gran diferencia respecto a los algoritmos SRL y Mixture-MCL. Mientras que en dichos métodos las partículas se introducen en el filtro en base a una sola observación (que podría ser errónea), con nuestro método, las partículas introducidas ya han sido evaluadas durante cierto tiempo y son el resultado de la convergencia del filtro secundario. Eso reduce el ruido introducido en el filtro primario y lo protege frente a partículas generadas en base a medidas externas erróneas.

De hecho, con nuestro método, todavía es posible introducir partículas mal colocadas generadas por medidas externas erróneas durante la inicialización del filtro secundario. Sin embargo, en los métodos SRL y Mixture-MCL, las nuevas partículas se evalúan utilizando las mismas medidas o, al menos, con una fuerte dependencia condicional, dándole a estas nuevas partículas ventaja frente al resto y empeorando los efectos negativos de los errores de detección. Con nuestro método, esas partículas erróneas permanecen aisladas en el filtro secundario sin afectar a la estabilidad del primario. Cuando finalmente algunas de ellas pasan al filtro primario, las medidas externas que las evaluarán son mucho más independientes de las que las generaron, por lo que es muy improbable que obtengan una mayor probabilidad que las partículas situadas correctamente.

Por lo tanto, cuando nuestro filtro primario está equivocado y un pequeño número de partículas colocadas correctamente se añaden procedentes del filtro secundario, es probable que provoquen un cambio en el filtro primario y éste termine convergiendo hacia la posición correcta, mientras que, cuando el filtro secundario añade partículas erróneas, es improbable que afecten a la estabilidad del filtro primario y lo hagan fallar.

En nuestros resultados, veremos como nuestra propuesta se ha implementado correctamente en un entorno de grandes dimensiones, proporcionando un tiempo de respuesta corto y demostrando su robustez frente a errores o imprecisiones en la detección.

5

Resultados

En esta sección, presentaremos los resultados de la evaluación del RDE integrado y el algoritmo de localización global propuesto. El nuevo RDE se ha evaluado mediante la implementación de una serie de tests en simulación y con robots reales. Debido a su mayor complejidad, explicaremos con más detalle la prueba final, en la que se utiliza la arquitectura para mover de forma autónoma una carretilla industrial automatizada durante más de veinte minutos por un parking al aire libre. Finalmente, presentamos también los resultados obtenidos con el algoritmo de localización global propuesto.

5.1. Evaluación de MissionLab-CARMEN

Hemos utilizado cinco escenarios de prueba con el fin de garantizar que nuestra arquitectura software de control multi-robot cumple con las especificaciones y funciona como esperamos. Hemos comprobado su capacidad de trabajar con

múltiples robots, que aparece de forma natural en MissionLab, y las capacidades de localización y navegación basadas en mapas que proporciona CARMEN.

Mediante los dos primeros tests, hemos examinado su compatibilidad y comparado los resultados obtenidos utilizando las últimas versiones oficiales de MissionLab y CARMEN. El resto de tests no podrían hacerse con las versiones originales de estos sistemas, puesto que necesitan las nuevas funcionalidades que hemos añadido. Todos los tests se han ejecutado en las versiones más recientes de Ubuntu, Fedora, Debian, OpenSUSE y CentOS.

Para facilitar la repetición de nuestros resultados, hemos creado scripts que permiten ejecutar los tests de forma automática. Dichos scripts pueden encontrarse en la carpeta */usr/demos/grousal_demos/*, una vez instalado en el sistema el software que hemos desarrollado.

5.1.1. Misión multi-robot utilizando MissionLab

Este primer escenario está pensado para demostrar la compatibilidad con la versión oficial de MissionLab[1] (primer punto de nuestras especificaciones). Además, comparamos los resultados obtenidos con la misma misión ejecutada utilizando la versión oficial de MissionLab.

La misión consta de tres robots que se mueven a lo largo del perímetro de un cuadrado utilizando el comportamiento *GoTo* de MissionLab. Estos robots son simulados utilizando el simulador por defecto de *HServer*.

Gracias a las mejoras que hemos hecho (comentadas en la secciones 3.2.1 y 3.2.2) el uso de CPU se ha reducido en un 40 %. El consumo de memoria en el arranque (alrededor de 80 MB) es casi el mismo en ambas versiones de MissionLab. Sin embargo, en la versión oficial el uso de memoria aumenta hasta cerca de 800 MB después de cuatro horas de funcionamiento, mientras que en nuestra versión modificada se mantiene estable, gracias a las fugas de memoria que hemos corregido. Esto permite la ejecución de misiones durante días o meses.

El desarrollo de la misión es el mismo en ambos sistemas, sin que se haya apreciado ninguna diferencia significativa. En la figura 5.1 se muestra la ventana de *MLab* y una de las consolas de *HServer* durante la misión.

[1] http://arce2.fis.usal.es/MissionLab.mp4

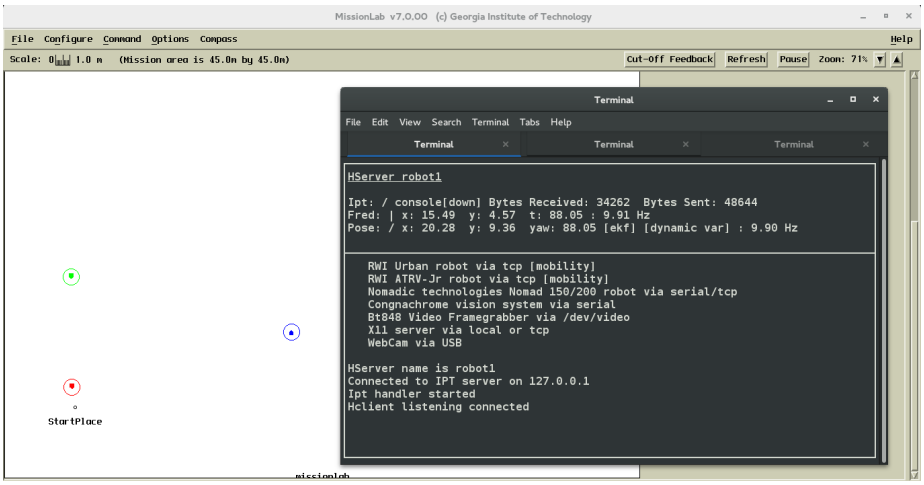


Figura 5.1: Misión multi-robot utilizando MissionLab

5.1.2. Misión de navegación en CARMEN

Este test tiene la finalidad de comprobar la compatibilidad con la versión oficial de CARMEN[2] y comparar el rendimiento de nuestra versión frente al de la versión oficial.

La prueba comienza con el arranque de todos los módulos necesarios: *ipc*, *param_ daemon*, *simulator*, *robot*, *navigator* y *localize*. Esta inicialización se realiza automáticamente gracias a un script que hemos creado para ello. Posteriormente, utilizando la herramienta *navigatorgui*, colocamos al robot simulado en el mapa, seleccionamos el destino y esperamos hasta que el robot lo alcanza.

El comportamiento del robot, al igual que el consumo de memoria, el uso de la CPU y el rendimiento de la misión son prácticamente idénticos que con la versión oficial. Esto se debe a que las mejoras que hemos hecho en IPC solamente están relacionados con las conexiones desde varios hilos a varios servidores, y los cambios que hemos hecho en CARMEN para interceptar mensajes nunca se utilizan en misiones en las que no interviene MissionLab. En la figura 5.2 pueden verse las herramientas *robotgui* y *navigatorgui* durante la misión.

[2]http://arce2.fis.usal.es/CARMEN.mp4

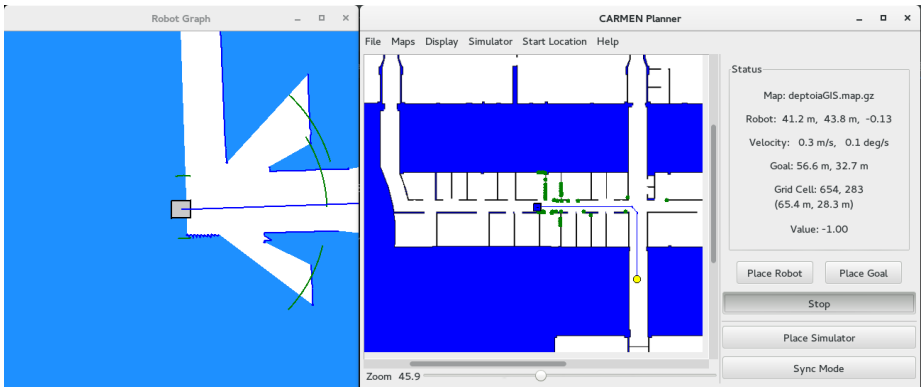


Figura 5.2: Misión utilizando CARMEN

5.1.3. Misión multi-robot utilizando MissionLab-CARMEN

En esta prueba se utiliza la plataforma integrada MissionLab-CARMEN[3] para ejecutar una misión en la que participan juntos tres robots simulados por ambos sistemas.

El primero robot se simula mediante el simulador por defecto de *HServer* y utiliza el comportamiento *GoTo* de MissionLab para desplazarse entre dos posiciones del mapa.

El segundo robot es un *Pioneer-I* simulado completamente por CARMEN que navega de un lado a otro el mapa utilizando el nuevo comportamiento *CARMEN_ GoTo* que hemos desarrollado para nuestro RDE integrado.

El tercer robot, al igual que el primero, se simula mediante el simulador por defecto de *HServer*, pero también genera lecturas de láser simuladas por CARMEN y usa sus capacidades de localización y navegación. Gracias a ello, puede utilizar el comportamiento *CARMEN_ GoTo* para navegar entre distintos puntos del mapa.

Este test demuestra la integración entre MissionLab y CARMEN a distintos niveles. Por una lado, el comportamiento *CARMEN_ GoTo* está compuesto por un subcomportamiento que accede a módulos de CARMEN (*CARMEN_ Navigate*) y otros subcomportamientos propios de la versión oficial de MissionLab (teleoperación y evitación de obstáculos). Esto demuestra que el comportamiento de los robots puede modelarse de forma cooperativa utilizando

[3]http://arce2.fis.usal.es/MissionLabCARMEN.mp4

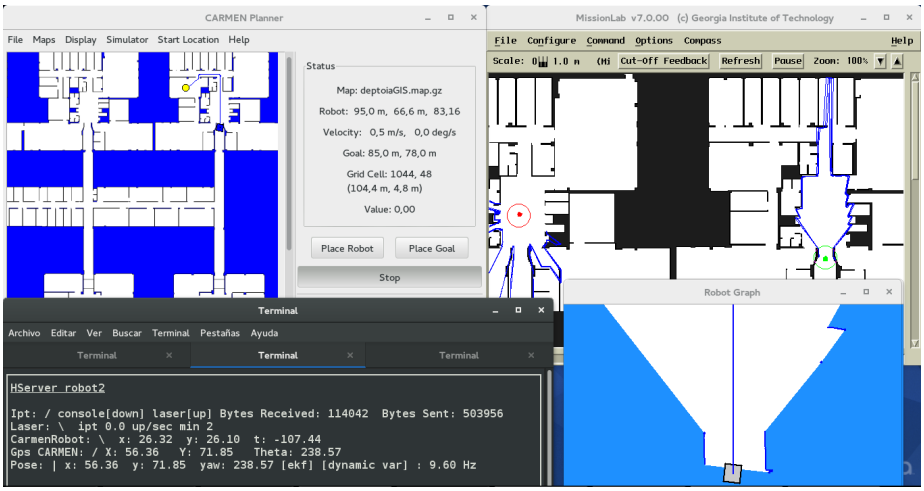


Figura 5.3: Test con robots simulados de CARMEN y MissionLab

funcionalidades de ambos entornos. Por otra parte, se demuestra cómo robots controlados por CARMEN y por MissionLab pueden convivir en una misma misión, pueden controlarse con los mismos comportamientos independientemente del sistema que proporcione sus drivers, e incluso es posible que un mismo robot tenga drivers proporcionados por distintos entornos para controlar distintos sensores o actuadores.

La mayor parte de las especificaciones del entorno de desarrollo integrado pueden validarse gracias a esta misión de ejemplo. La información de localización generada por CARMEN para el segundo robot se integra en MissionLab a través de *HServer*, MissionLab tiene el control de la misión, las lecturas de sensores generadas por CARMEN se propagan hacia MissionLab, y la misión utiliza nuestro nuevo comportamiento *CARMEN_GoTo*, que integra la navegación de CARMEN con MissionLab.

La figura 5.3 muestra la consola de *HServer* (en la esquina inferior izquierda) y las herramientas gráficas de MissionLab y CARMEN durante la misión: (*navigatorgui* en la esquina superior izquierda, *mlab* en la esquina superior derecha y *robotgui* en la esquina inferior derecha).

El resultado es el esperado, con los tres robots moviéndose sobre el recorrido programado y la ventana de *MLab* mostrando la localización y las lecturas de sensores (sonar, láser) independientemente de que las genere CARMEN o MissionLab.

Figura 5.4: Misión multi-robot con robots reales de CARMEN y MissionLab

5.1.4. Misión en MissionLab-CARMEN con robots reales

En esta prueba introducimos robots reales en una misión conjunta MissionLab-CARMEN que hemos creado utilizando *CfgEdit*[4]. El desarrollo de esta misión se llevó a cabo durante el desarrollo de un proyecto de investigación con el objetivo de implementar una plataforma de limpieza multi-robot.

Nuestro robot MORLACO, equipado con un láser actúa como el líder de la misión, mientras que un robot *Roomba* se limita a seguir al robot líder hasta que éste detecta una puerta abierta con su láser. Al detectar la puerta, el robot líder le envía la posición al *Roomba*, para que éste entre en la habitación y después salga para volver a seguir al líder en busca de la siguiente puerta.

El robot MORLACO se controla mediante un driver de CARMEN, obtiene los datos de su sensor láser gracias al módulo *laser* de CARMEN, y utiliza las funcionalidades de navegación y localización de CARMEN. Por otra parte, el robot Roomba se controla mediante un driver de MissionLab. La colaboración entre ambos se produce gracias a la misión generada por la herramienta CfgEdit de MissionLab.

[4]http://arce2.fis.usal.es/CleaningSystem.mp4

Como el robot *Roomba* no puede estimar de forma precisa su posición porque no dispone de sensores avanzados, ni puede percibir la distancia a la puerta abierta, se guía mediante un comportamiento de *seguir pared*, golpeando repetidamente la pared con sus *bumpers* hasta que éstos dejan de chocar.

En este escenario, al igual que en el anterior, no podemos comparar el rendimiento con la versiones oficiales de MissionLab y CARMEN, puesto que para que se pueda ejecutar es necesaria la integración que hemos hecho. En la figura 5.4 pueden verse ambos robots durante la misión.

5.1.5. Misión en MissionLab-CARMEN con carretilla industrial

Una vez que nos aseguramos de que nuestro RDE funcionaba correctamente, probamos su fiabilidad en un entorno más complicado. Creamos una misión que usa la localización y la navegación de CARMEN, nuestro nuevo comportamiento para MissionLab *CARMEN_ Navigate*, y drivers de *HServer* para conseguir la navegacion autónoma de una carretilla industrial entre distintos puntos de un parking al aire libre.

Entorno de pruebas

El entorno utilizado ha sido el parking de la Facultad de Ciencias de la Universidad de Salamanca (figura 5.5). Se trata de un parking al aire libre, situado en la parte baja de las instalaciones.

Para poder hacer pruebas en este entorno, tuvimos que crear previamente un mapa. Utilizamos el propio entorno desarrollado para crear un log con las lecturas de odometría y láser obtenidas durante un recorrido de la carretilla por todo el parking. El recorrido se realizó conduciendo manualmente la carretilla y asegurándose de que el láser captaba todas las paredes.

Para conseguir un mapa lo más correcto posible sin necesidad de modificarlo manualmente, la obtención de los logs se realizó cuando el parking se encontraba casi vacío. Una vez generado el log con la herramienta *log_ carmen* de CARMEN, utilizamos la herramienta *vasco* para obtener el mapa. En la figura 5.6 se muestra el programa *vasco* mostrando los datos de uno de estos logs antes de procesarlos para construir el mapa.

Figura 5.5: Parking utilizado para las pruebas

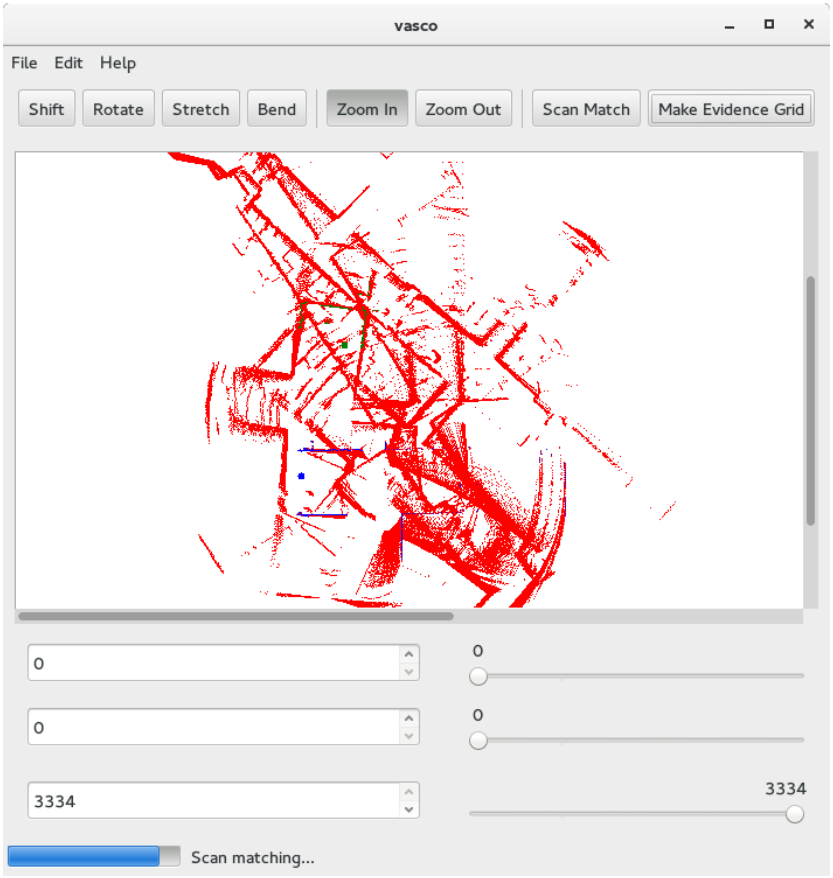


Figura 5.6: Datos brutos de odometría y láser

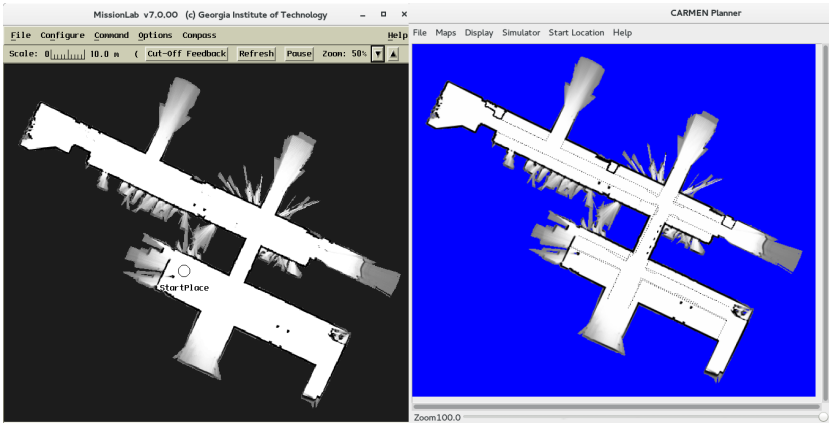


Figura 5.7: Mapa cargado en *MLab* y *navigator_gui*

Tras sucesivas iteraciones ajustando los parámetros de configuración de *vasco*, obtuvimos un mapa muy ajustado a la realidad. Sobre este resultado inicial, se añadieron manualmente, gracias a la herramienta *map_editor*, varias escaleras de acceso al edificio que el algoritmo de *mapping* no fue capaz de situar correctamente. Con la misma herramienta, se añadieron también límites virtuales para la navegación, protegiendo los lugares de aparcamiento y las zonas de giro, para evitar que el algoritmo de navegación generara rutas que pudieran estar cubiertas por coches estacionados, o giros demasiado cercanos a las paredes.

Para refinar estas modificaciones, se realizaron diversas pruebas de navegación en simulación, comprobando las rutas y resultados obtenidos variando ligeramente los elementos modificados. Una vez que se consiguieron resultados satisfactorios en simulación, se realizaron pruebas con la carretilla real, con trayectos cortos completamente controlados. Se puso especial atención a las distancias de seguridad con las paredes y los coches aparcados, así como a la correcta generación y ejecución de rutas con curvas.

Una vez obtenido el mapa final, se realizó una conversión para que pudiera cargarse con la herramienta de visualización de misiones de MissionLab (*MLab*). En la figura 5.7 podemos ver el mapa final tanto en la herramienta *navigatorgui* de CARMEN como en *MLab*, con unas dimensiones de 122.4 metros de largo por 106.8 metros de ancho, y una resolución de 20 cm por píxel.

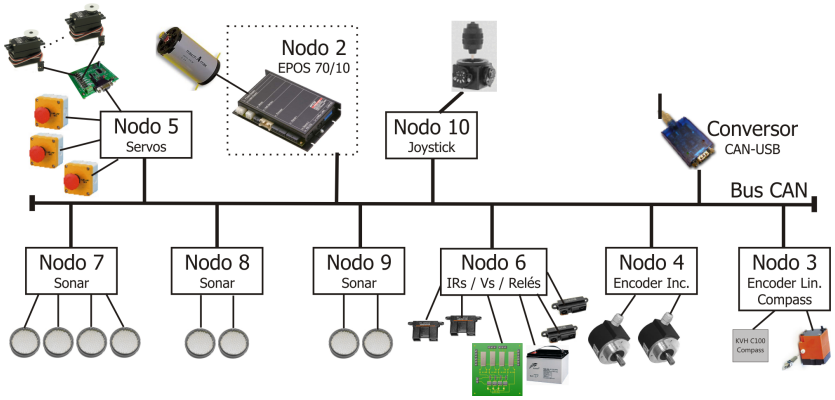


Figura 5.8: Nodos conectados al bus CAN de la carretilla

Descripción del robot

El robot utilizado ha sido una carretilla industrial completamente automatizada gracias al trabajo del Grupo de *Robótica y Sociedad* de la Universidad de Salamanca. La carretilla cuenta con un motor eléctrico alimentado por un conjunto de baterías que se encuentran en la parte trasera. Se controla mediante un volante, un pedal de aceleración, uno de freno y dos palancas que permiten controlar la altura y la inclinación de la pinza delantera.

Para su automatización, se colocaron actuadores en el volante, los pedales y las palancas, sensores sonar en los laterales y la parte trasera, sensores IR para medir la altura y la inclinación de la pinza, un láser de rango en la parte delantera, encoders en las ruedas delanteras y en el volante, una brújula y un GPS en la parte superior, tres setas de emergencia en los laterales y la parte trasera capaces de activarse manualmente o mediante un mando inalámbrico, relés para accionar el claxon y las luces, un módulo de control conectado directamente a un joystick, y un adaptador USB-CAN para las comunicaciones con software de control externo. En la figura 5.8 se pueden ver esquemáticamente todos estos elementos conectados al bus CAN.

Varios de los actuadores cuentan con su propio bucle de control a bajo nivel, de manera que es posible, por ejemplo, no sólo controlar la fuerza que se ejerce sobre los pedales, si no también configurar la velocidad deseada y dejar que el los actuadores de los pedales funcionen automáticamente gracias a un controlador PID realimentado con las lecturas de los encoders.

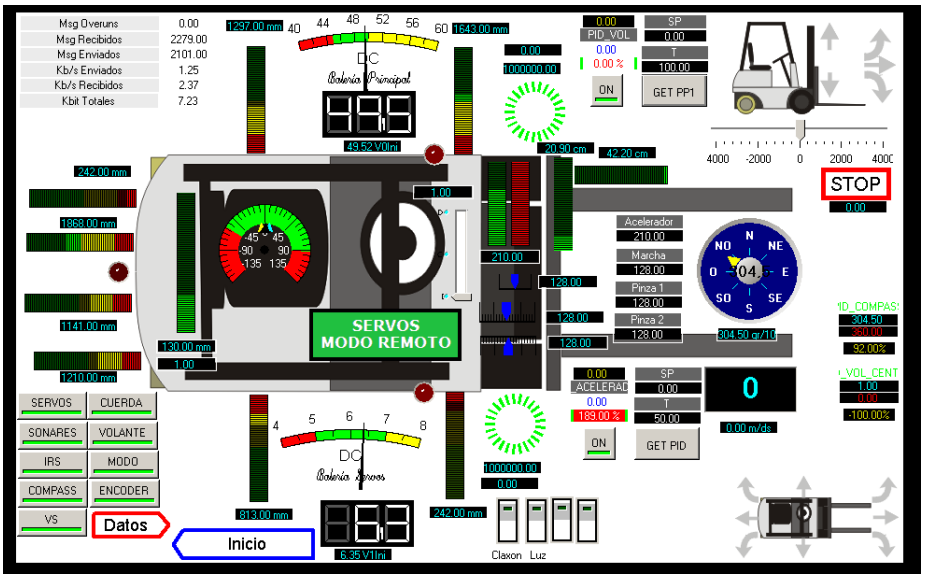


Figura 5.9: Programa SCADA para la depuración a bajo nivel de la carretilla

Para la depuración y el manejo a bajo nivel de los sensores y actuadores, se utiliza un programa SCADA que muestra de forma gráfica los distintos elementos de la carretilla. En la figura 5.9 podemos ver una imagen de este programa en funcionamiento.

El manejo de la carretilla, desde nuestra nueva arquitectura de control integrada, se consigue gracias a una biblioteca de comunicaciones de desarrollo propio (*cdomotic*) inspirada en la biblioteca de comunicaciones *domotic* que ayudé a desarrollar durante mi estancia en el Departamento de Ingeniería de Sistemas y Automática de la Universidad de Vigo.

Gracias a un archivo de configuración, la biblioteca *cdomotic* conoce todos los nodos del bus CAN, lee regularmente el estado de los sensores, con un periodo de muestreo configurable por cada valor, y permite modificar los valores de las variables de control. La biblioteca *cdomotic* se utiliza dentro del driver de la carretilla implementado para *HServer*. Gracias a la integración realizada entre MissionLab y CARMEN, esto es suficiente para poder acceder a toda la información y permitir la realización de tareas de alto nivel (localización, navegación, evitación de obstáculos) desde distintos módulos de MissionLab o CARMEN indistintamente.

Despliegue del software de control

Aparte del software de control a bajo nivel que se encuentra en las distintas placas electrónicas conectadas al bus CAN y a los sensores y actuadores, el despliegue de los distintos componentes software que controlan la carretilla automatizada se realiza sobre dos ordenadores situados en la parte trasera, y un ordenador portátil, generalmente situado al lado del asiento del conductor. Todos ellos, al igual que el resto de componentes electrónicos, obtienen la alimentación de las propias baterías de la carretilla, gracias a distintos conversores de corriente. El adaptador USB-CAN que da acceso a todos los sensores y actuadores se conecta a uno de los ordenadores de la parte trasera, que ejecuta también *HServer* y tiene acceso directo a las lecturas del láser a través de un adaptador USB-Serie. Este ordenador se comunica con los otros dos a través de un router WiFi, también en la parte trasera de la carretilla. El segundo ordenador fijo ejecuta todos los módulos de CARMEN necesarios para el desarrollo de las misiones (*localize*, *navigator*, *param_ daemon*, *central*). El ordenador portátil se utiliza principalmente para ejecutar las herramientas gráficas de MissionLab y CARMEN (*MLab* y *navigatorgui*) y acceder por *ssh* a los otros dos cuando es necesario. Gracias al router WiFi el portátil puede utilizarse fuera de la carretilla, o se pueden conectar dispositivos móviles adicionales (portátiles, móviles, tablets) para monitorizar las misiones.

Desarrollo de la misión

La misión consiste en hacer navegar la carretilla de forma autónoma por distintos lugares del parking. El entorno no fue acondicionado para la ejecución de la misión, por lo que están presentes objetos como coches, bicicletas, plantas, etc., que no estaban mapeados en el plano original. Para la creación de la misión, se ha utilizado el editor de misiones *CfgEdit* de MissionLab. Una máquina de estados, basada en nuestro nuevo comportamiento *CARMEN_ Navigate*, hace que la carretilla se mueva entre distintos lugares del mapa identificados por sus coordenadas. El nuevo comportamiento *CARMEN_ Navigate* permite al módulo *navigate* de CARMEN crear rutas para la carretilla, y, al mismo tiempo, deja actuar al algoritmo de prevención de colisiones de CARMEN, para que la carretilla se detenga al detectar un obstáculo demasiado cercano.

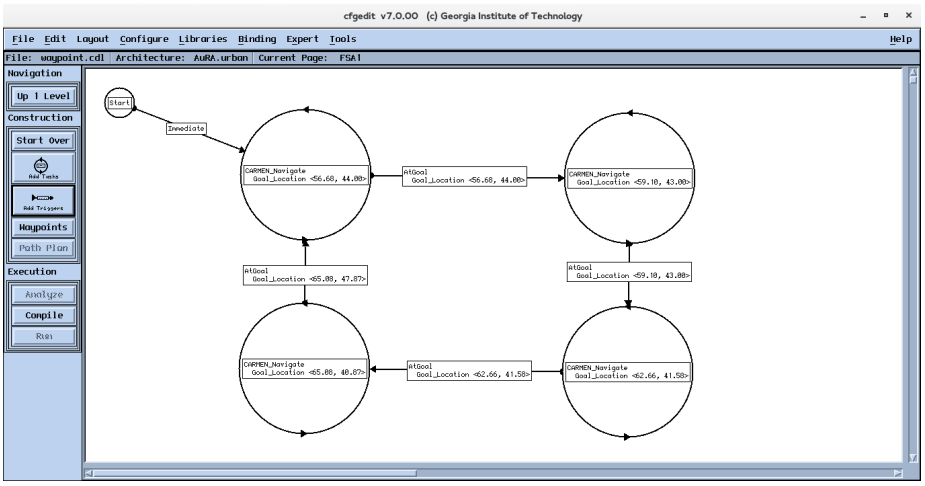


Figura 5.10: Misión desarrollada para la carretilla

La misión (figura 5.10) cuenta con sólo cinco estados: uno inicial y otros cuatro conectados mediante un ciclo cerrado que hacen que la carretilla se mueva entre distintos puntos del mapa gracias al comportamiento *CARMEN_ Navigate*. Los puntos seleccionados permiten que la carretilla se mueva por casi todo el parking, con trayectos en los que se mueve tanto hacia adelante como hacia atrás. Durante el transcurso de la misión, se obstaculizó a la carretilla varias veces, para comprobar que se detiene correctamente ante un obstáculo que le impide el paso y continúa su marcha cuando el obstáculo desaparece.

El el vídeo de la misión[5] se puede apreciar que el terreno presenta muchas irregularidades, baches, pendientes, zonas bloqueadas, etc., por lo que las condiciones de navegación presentan mucha variabilidad. A pesar de la dificultad que implica la dirección *ackerman* de la carretilla industrial, la baja exactitud de la odometría debido a su peso, a sus dimensiones y a las irregularidades del terreno, además de la presencia de obstáculos en el entorno, el robot completó la misión con éxito. La carretilla se movió de forma autónoma durante más de 20 minutos, recorriendo los lugares de forma cíclica y recorriendo marcha atrás trayectos de la misión donde el robot no podía girar. Finalmente, paramos la misión de forma manual. Para la navegación el robot no dispuso ni de GPS, ni de giróscopo, ni de brújula, porque fue suficiente con la localización de CARMEN en base a la información proporcionada por el láser de rango y los *encoders*.

[5] http://arce2.fis.usal.es/AutonomousForklift.mp4

Figura 5.11: Carretilla navegando de forma autónoma con nuestro RDE integrado

Esta prueba no sólo valida la correcta integración entre MissionLab y CARMEN, si no que también demuestra que nuestro RDE es suficientemente fiable como para ser utilizado en ambientes complejos y delicados, en los que un error podría causar daños significativos. La figura 5.11 muestra la carretilla industrial durante la prueba.

5.2. Análisis del procedimiento de localización

En esta sección, comentaremos las pruebas realizadas para probar nuestro algoritmo de localización global. Veremos el entorno en el que se han hecho los tests, el robot que utilizamos y el software necesario para su ejecución. Finalmente, analizaremos detalladamente un caso de estudio, estudiaremos el comportamiento de nuestro algoritmo utilizando distintos números de partículas, veremos cómo se enfrenta al problema del robot secuestrado y analizaremos su porcentaje de acierto.

5.2.1. Entorno de pruebas

Las pruebas de nuestro algoritmo de localización global se han desarrollado en la segunda planta de la Facultad de Ciencias de la Universidad de Salamanca, que tiene una extensión de alrededor de 5.000 m^2 y una cantidad significativa de puertas.

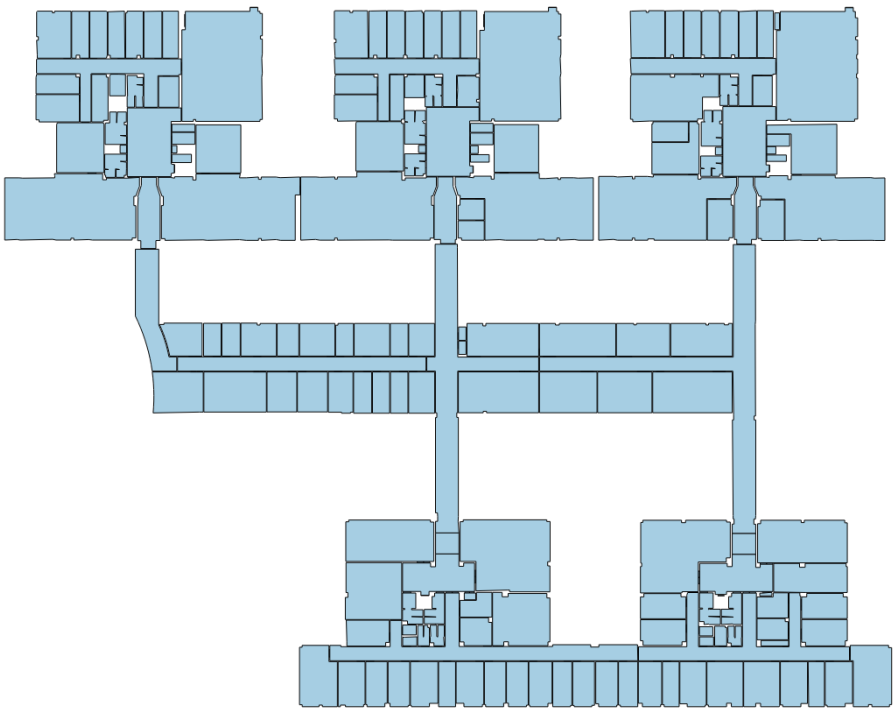


Figura 5.12: Escenario de pruebas para nuestro algoritmo de localización global

En la figura 5.12, se puede observar una representación de dicho entorno. Se trata de una superficie extensa en la que otros métodos de localización global mediante filtros de partículas fallan debido a la gran cantidad de partículas necesaria para explorer el espacio de estados.

Para poder hacer una gran cantidad de tests, hemos movido el robot entre diferentes lugares de la planta, utilizando el nuevo comportamiento CARMEN_Navigate que hemos desarrollado, y hemos salvado todas sus percepciones y datos de odometría mediante la herramienta de creación de logs de CARMEN. En la figura 5.13 se muestra la representación de una de estas misiones de prueba. Además, le proporcionamos la posición inicial a CARMEN para permitirle seguir correctamente el movimiento del robot y tener una estimación bastante precisa de la posición real durante las pruebas.

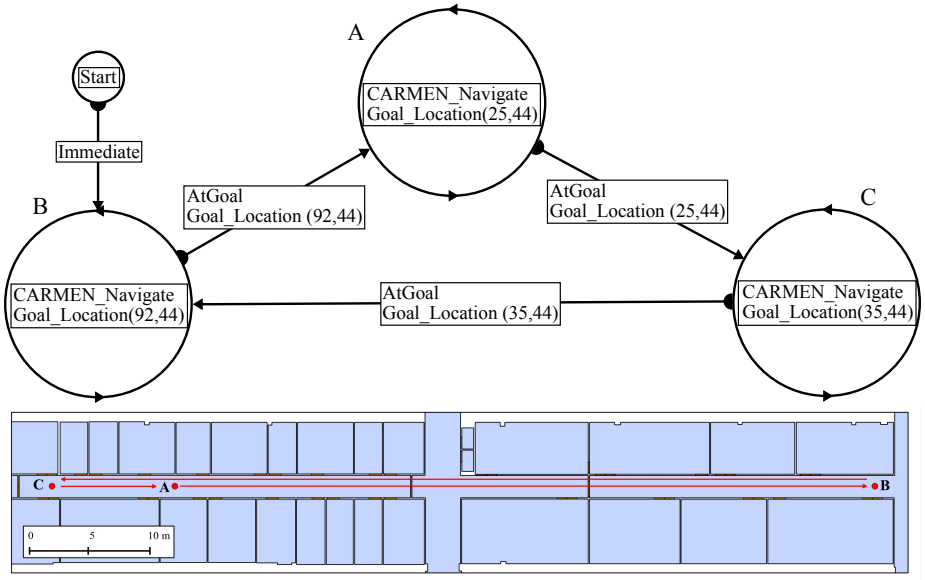
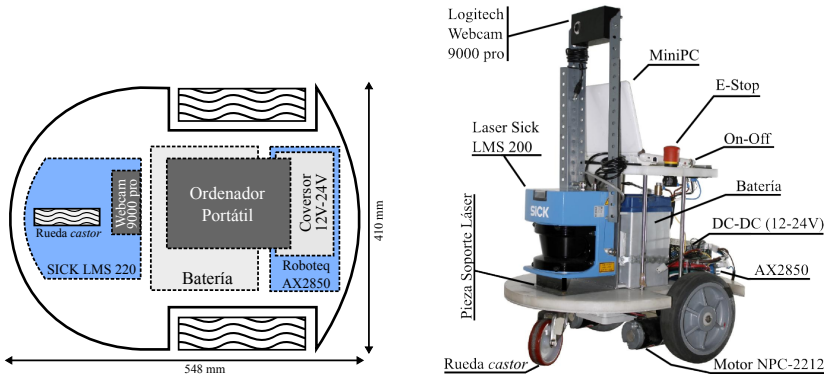


Figura 5.13: Representación de una de las misiones utilizadas para obtener datos de prueba

5.2.2. Descripción del robot

Para nuestros pruebas, hemos utilizado el robot *MORLACO* (***Mo**bile **R**obot – **La**ser **Co**ntrolled*) que se encuentra completamente integrado con la arquitectura que hemos desarrollado. *MORLACO* tiene una configuración diferencial, con dos ruedas motorizadas diametralmente opuestas ubicadas en un eje perpendicular a la dirección de avance del robot, y una rueda castor que le sirve de apoyo y le permite girar libremente en cualquier dirección.

Tanto el diseño CAD del robot como su aspecto se muestran en la figura 5.14, junto con la distribución de los principales componentes. ruedas, motores, controladora de motores, batería, sensor láser y cámara, además de otros elementos de menor tamaño como un conversor de 12 a 24 V, *encoders* para medir el desplazamiento del robot, y un mini-ordenador portátil. Su sistema sensorial está compuesto por un láser de medición de distancias (Sick LMS 200) y una cámara de visión monocular (Logitech Webcam 9000 Pro), así como un par de *encoders* ópticos incrementales (E3 de USdigital). Gracias a sus motores, *MORLACO* cuenta con una velocidad máxima de 2,5 m/s y una aceleración máxima de 2 m/s^2.



(a) (b)

Figura 5.14: a): Diseño CAD de la base del robot b) Aspecto final del robot

5.2.3. Despliegue del software de control

El módulo *camera9000_pro* se encarga de la comunicación por USB con la cámara de *MORLACO*. Para obtener datos del láser, se utiliza el módulo existente en CARMEN, que consigue obtener datos a la máxima velocidad posible para nuestro modelo (500KB/s). El módulo de CARMEN llamado *morlaco* se comunica con la controladora AX2850 a través de un puerto serie e implementa funciones para consultar el desplazamiento relativo de los *encoders* y para enviar comandos de actuación a los motores.

Gracias al módulo *localize* de CARMEN, tenemos una estimación fiable de la posición real del robot para compararla con nuestros resultados y utilizarla como entrada para la navegación durante la obtención de datos. El módulo *navigator* nos permite utilizar el nuevo comportamiento que hemos desarrollado para MissionLab (CARMEN_Navigate) para mover al robot y hacer pruebas, mientras que otro módulo de CARMEN llamado llamado *doorDetection* implementa el algoritmo de detección de puertas que utilizamos. Este módulo ha sido desarrollado por el Grupo de Robótica y Sociedad de la Universidad de Salamanca y está integrado con el módulo *log_carmen*, de manera que podemos crear logs almacenando todos los datos de odometría, de localización, y también de detección de puertas para su uso posterior. Gracias a la herramienta *play_carmen*, podemos reutilizar estos logs y probar el comportamiento de nuestro método variando diferentes valores de configuración y comparando los resultados.

Sobre esta configuración, nuestra módulo de localización global se conecta por un lado a todos los mensajes de CARMEN que necesita (odometría, detección de puertas) y, por otra parte, a una base de datos PostgreSQL con nuestro GIS.

5.2.4. Caso de estudio

El caso de estudio elegido es especialmente interesante porque muestra el comportamiento de nuestro doble filtro de partículas en circunstancias difíciles. Sería inútil evaluar el algoritmo de localización en un caso ideal en el que el mapa es perfecto y las mediciones externas son muy precisas porque cualquier método debería tener éxito en esas circunstancias. En su lugar, en nuestro caso de estudio, percepciones incorrectas del robot hacen que los filtros de partículas fallen durante las primeras iteraciones. Así forzamos a nuestro doble filtro de partículas a demostrar que puede proporcionar buenos resultados en escenarios adversos. Después del error inicial, el filtro de partículas secundario se reinicia varias veces de acuerdo a nuestro método. Debido a inexactitudes en las medidas externas, aunque en la mayor parte de los casos es capaz de situar y mantener partículas en el lugar correcto hasta el final de su ciclo de vida, en algunos casos falla, pero incluso en esos casos no causa problemas en el filtro primario, que se mantiene estable una vez que detecta la posición correcta del robot.

La figura 4.5 muestra el estado de nuestro doble filtro de partículas justo antes de la primera fase de muestreo en la que participan los dos filtros conjuntamente. Los círculos grandes con fondo semitransparente y diferentes tamaños y colores son los clusters de partículas de cada filtro. El radio del círculo está relacionado con la suma de las probabilidades de las partículas en el cluster ($\hat{p}_C$). El color rojo está asociado al filtro principal, mientras que el color verde se utiliza para colorear los clusters del filtro de partículas secundario.

La posición real del robot en ese momento se encontraba en el pasillo horizontal en el medio de la figura (se muestra como un pequeño rectángulo negro cerca de la cuarta oficina empezando por el lado izquierdo del pasillo en la imagen). El cluster con mayor probabilidad del filtro de primario se encuentra en la parte inferior de la figura, muy alejado de la posición real del robot (más de 50 m), por lo que es claro que el filtro primario falló en su intento de obtener la

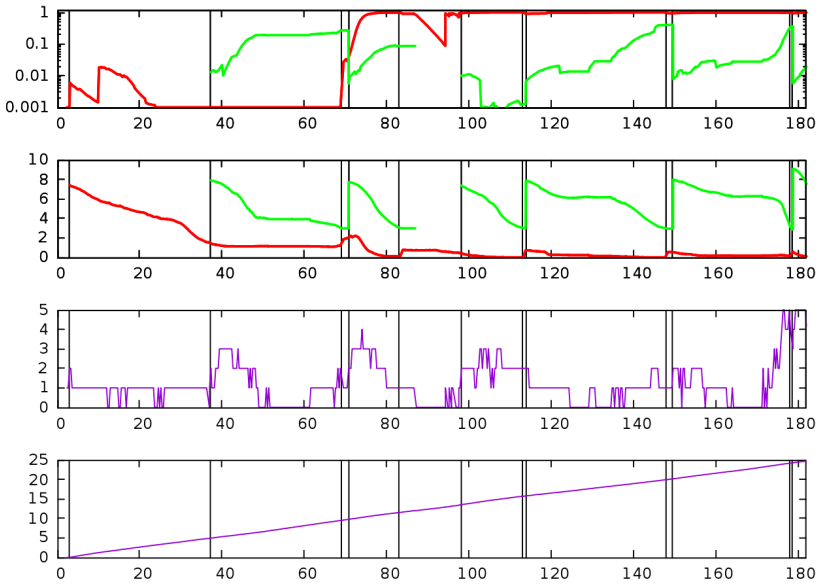


Figura 5.15: Evolución del doble filtro de partículas en el tiempo

posición del robot. Sin embargo, el filtro secundario, que se inicializó más tarde durante la primera *condición de disparo* sí fue capaz de situar una cantidad significativa de partículas en el lugar correcto.

En la figura 5.15, podemos ver la evolución temporal de este mismo caso de estudio. El gráfico superior muestra la precisión de ambos filtros (la probabilidad situada en el lugar correcto). La línea roja está asociada al filtro de partículas primario mientras que la verde corresponde al secundario. La segunda gráfica muestra la entropía H_t de ambos filtros a lo largo del tiempo. La tercera gráfica muestra las puertas detectadas en cada momento. Se puede observar como la mayor parte del tiempo se detecta una pequeña cantidad de puertas (una o cero), pero en ciertos instantes se cumple la *condición de disparo* del filtro secundario. El último gráfico muestra la distancia recorrida por el robot, de manera que podemos ver tanto el tiempo como la distancia entre distintos eventos. Como se puede observar, una vez recorridos 10 metros nuestro método ya proporciona la localización correcta del robot, a pesar de encontrarse ante unos de los peores escenarios posibles. También es relevante que la salida correcta se mantiene a lo largo del tiempo, incluso cuando el entorno no proporciona información o el filtro secundario se equivoca.

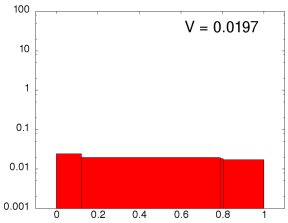
Como puede verse en el primer gráfico, el filtro de partículas principal sitúa algunas en la posición correcta al inicio de la prueba, pero desaparecen sobre el segundo 22 (tras unos 3 metros recorridos). La tercera gráfica muestra como, alrededor del segundo 38 (después de 5 metros recorridos desde el inicio) aparece la primera *condición de disparo* que provoca la inicialización del filtro secundario. Tras otros 5 metros recorridos, la entropía del filtro secundario dispara la *condición de reseteo*, que experimentalmente determinamos con el valor $H^T = 3$. El momento de esa primera *condición de reseteo* está representado espacialmente en la figura 4.5.

Cuando las partículas del filtro secundario participan en la fase de muestreo del filtro primario (con una probabilidad reservada del 20 %) la entropía del filtro primario se incrementa ligeramente y, de forma muy rápida, el filtro converge hacia la posición correcta del robot. Después, la siguiente condición de disparo aparece casi de forma instantánea, por lo que el filtro secundario se inicializa de nuevo. Las condiciones de *disparo* y *reseteo* están marcadas con líneas verticales en la gráfica (65 segundos para la primera, 80 segundos para la segunda, 117 segundos para la tercera, etc.). Como se puede observar en la figura 5.15, en la segunda *condición de reseteo* el filtro secundario tiene menos de un 10 % de partículas cerca del lugar correcto y, en la tercera, prácticamente no hay ninguna partícula bien situada. El muestreo híbrido incrementa la entropía del filtro primario ligeramente pero, en todos los casos, se mantiene estable en su estimación y descarta las partículas erróneas procedentes del filtro secundario rápidamente

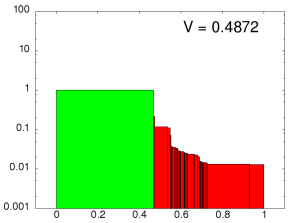
La figura 5.16 muestra el estado de los clusters (*perdido* o no) de ambos filtros cuando la *condición de reseteo* se cumple y el filtro primario muestrea partículas del secundario. Cada cluster se representa mediante su probabilidad (en el eje horizontal) y su precisión N_C (la inversa de la distancia del centro del cluster a la posición correcta limitada a la precisión que consideramos como acertada) en el eje vertical. Siendo

$$A_C = \frac{1}{abs\,(D_C + \epsilon)}$$

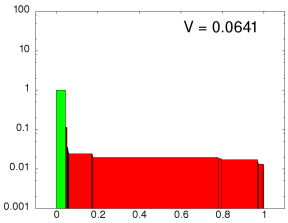
donde D_C es la distancia desde la posición real del robot (almacenada por el localizador de CARMEN y supervisada durante las pruebas) hasta el centro del cluster, ϵ un valor arbitrariamente pequeño para evitar que el denominador



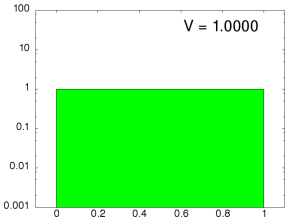
(a) Verosimilitud del filtro primario antes de TS=1



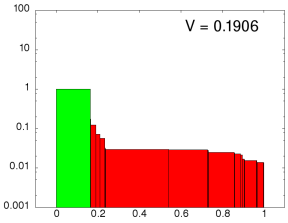
(b) Verosimilitud del filtro secundario antes de TS=1



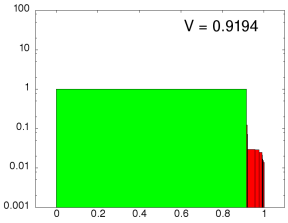
(c) Verosimilitud del filtro primario después de TS=1



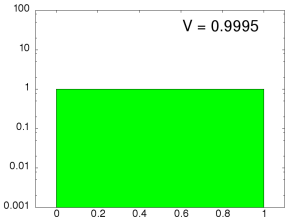
(d) Verosimilitud del filtro primario antes de TS=2



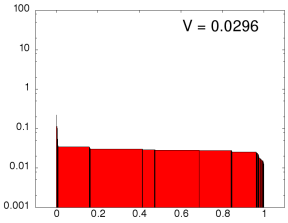
(e) Verosimilitud del filtro secundario antes de TS=2



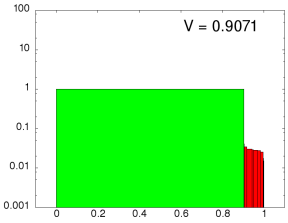
(f) Verosimilitud del filtro primario después de TS=2



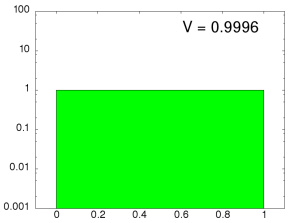
(g) Verosimilitud del filtro primario antes de TS=3



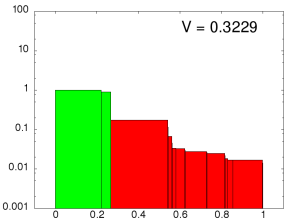
(h) Verosimilitud del filtro secundario antes de TS=3



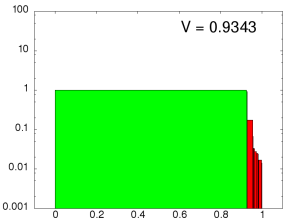
(i) Verosimilitud del filtro primario después de TS=3



(j) Verosimilitud del filtro primario antes de TS=4



(k) Verosimilitud del filtro secundario antes de TS=4



(l) Verosimilitud del filtro primario después de TS=4

Figura 5.16: Verosimilitud de los filtros de partículas inmediatamente antes y después de las fases de muestreo híbrido. Puede observarse como la robustez del procedimiento permanece en el tiempo.

sea cero y

$$N_C = \min_{cluster\ C} \{A_C, A_T\}$$

donde A_T es la precisión que consideramos suficiente para decir que una partículas está colocada en el lugar correcto. En nuestro caso, hemos considerado $A_T = 1$, de manera que cualquier partícula situada a menos de un metro de distancia de la localización proporcionada por CARMEN se considera bien situada.

En base a este valor de precisión y a la probabilidad de cada cluster, definimos la verosimilitud V del filtro en cada gráfica como el área total de los clusters en la gráfica:

$$V = \sum_{cluster\ C} \{N_C\, \hat{p}_C\} \tag{5.1}$$

Para hacer los gráficos más legibles a simple vista, los clusters que se encuentran en el lugar correcto se han coloreado en verde y el resto en rojo. Cada fila de gráficas se corresponde con una *condición de reseteo* distinta para el filtro secundario. Dentro de cada fila, el primer gráfico representa el estado del filtro primario antes del muestreo híbrido, la segunda gráfica el estado del filtro secundario en ese mismo momento, y la tercera gráfica el estado del filtro primario después del muestreo.

Como se puede observar durante la primera condición de reseteo ($TS = 1$), el valor de V para el filtro primario es muy bajo y todos los clusters son rojos, es decir, el filtro se encuentra completamente perdido. Esta situación puede apreciarse también en las figuras 5.15 y 4.5. En el mismo momento, el filtro secundario ya ha recogido suficiente información del entorno como para situar partículas en el lugar correcto (un gran cluster verde correctamente colocado en la figura 4.5, aunque sin la mayoría de la probabilidad del filtro).

El resultado del primer muestreo híbrido se muestra en el gráfico de la figura 5.16(c). Se puede apreciar como algunas partículas se encuentran en el lugar correcto, gracias al muestreo de partículas del filtro secundario.

En la siguiente *condición de reseteo* ($TS = 2$), las pocas partículas colocadas correctamente en el instante $TS = 1$ ya han evolucionado y conseguido colocar toda la probabilidad del filtro en la posición correcta.

A pesar de que el filtro secundario añade una mayoría de partículas mal colocadas al primario en la segunda y tercera *condición de reseteo*, el filtro primario las descarta rápidamente antes de la siguiente iteración. Así, podemos ver como pocas partículas bien colocadas añadidas al filtro primario procedentes del secundario son capaces de corregir su estado en un tiempo muy corto, mientras que la introducción de partículas erróneas por parte del filtro secundario durante la *condición de reseteo* no causa ningún problema al primario. Principalmente, porque es muy difícil que partículas mal colocadas del filtro secundario, que fueron añadidas en base a errores o imprecisiones en la detección, compitan con partículas bien colocadas en el filtro primario al ser evaluadas con mediciones externas independientes.

En el vídeo que muestra la evolución de nuestro método en este caso de estudio[6], se puede apreciar su robustez durante momentos en los que no se detectan puertas o incluso cuando se detectan puertas que no existen en el mapa (las puertas están señaladas como pequeños círculos negros).

5.2.5. Número de partículas

En base a nuestro caso de estudio, hemos hecho pruebas con distintos números de partículas para determinar cuántas son necesarias para obtener resultados satisfactorios y cuántas para obtener resultados repetibles.

Durante las sucesivas inicializaciones de los filtros primario y secundario en nuestro caso de estudio, el número de posiciones candidatas para ser la posición real del robot, calculadas a partir de las puertas detectadas, se encuentra entre 199 y 375. Por ello, comenzamos nuestras pruebas utilizando 400 partículas en cada filtro. Con un número menor a 375, no sería posible muestrear cada posición candidata ni siquiera con una sola partícula.

El conjunto de pruebas con nuestra inicialización basada en las puertas detectadas y la información procedente de mapas GIS ha consistido en la ejecución de 10 tests con 400, 800, 1200, 2500 y 5000 partículas por filtro, utilizando en todos los casos los mismos datos que en nuestro caso de estudio y permitiendo el movimiento del robot hasta 12 metros.

[6] http://arce2.fis.usal.es/testparticles.mkv

Con 400 partículas, obtuvimos un 70 % de acierto, aunque en el 20 % del total el desarrollo fue distinto. Es decir, en ese 20 % el filtro obtuvo la localización correcta del robot en menos de 15 metros, pero hubo variaciones importantes en el proceso, como que se obtuvo la posición correcta en la segunda inicialización del filtro secundario en vez de la primera, o que otra posición candidata obtuvo una probabilidad mayor durante más tiempo.

Con 800 partículas, el porcentaje de acierto subió al 90 %, aunque en un 20 % de los casos todavía se apreciaron diferencias significativas en el desarrollo de la prueba respecto a la descripción presentada en la sección 5.2.4.

Con 1200 partículas, el resultado al final de los 15 metros fue correcto en el 100 % de los casos. No se apreciaron diferencias muy significativas en el desarrollo de las pruebas, aunque sí ligeras variaciones como la desaparición de las partículas en algunas posiciones candidatas un poco antes o después que en la descripción de nuestro caso de estudio.

Con 2500 partículas, el resultado fue acertado en todos los casos y, a simple vista, el desarrollo de la misión también fue prácticamente igual en todos los casos. Una vez determinado el número de partículas que nos permite obtener resultados correctos y repetibles, decidimos utilizar el doble (5000 partículas por filtro) para los casos de estudios descritos en este documento, tanto en la sección 5.2.4 como en las secciones 5.2.6 y 5.2.7, con el fin de asegurar que los resultados sean repetibles y tratar de que el acierto o error dependa lo mínimo posible del número de partículas utilizado.

Para determinar la mejora conseguida gracias a nuestro método frente a la inicialización de las partículas de forma aleatoria sobre la superficie habitable del mapa, ejecutamos también varias pruebas con distintos números de partículas. Inicializando los filtros con 5000 partículas situadas de forma aleatoria sobre la superficie habitable del mapa, no obtuvimos un resultado correcto en ninguna de nuestras 10 pruebas. Utilizando 50000 partículas por filtro, obtuvimos la localización correcta del robot tan solo en el 20 % de los casos. Con 100000 partículas por filtro, nuestra CPU se encontraba ya al 100 % de uso. Pese a ello, obtuvimos un 60 % de acierto, y un comportamiento comparable al obtenido usando 400 partículas con nuestra inicialización.

Utilizando 200000 partículas, el retardo en el procesamiento de los datos ya fue claramente perceptible. Probablemente por esa razón el porcentaje de acierto se redujo al 30 %. Pruebas con 500000 partículas corroboraron que el algoritmo ya no podía ejecutarse a una velocidad suficiente y proporcionaron resultados incorrectos en todos los casos.

Comparando los resultados, podemos concluir que para la obtención de unos resultados aceptables (60-70 % de acierto) fueron necesarias sólo 400 partículas por filtro con nuestra inicialización y 100000 con una inicialización aleatoria. Es decir, nuestra inicialización redujo en 250 veces el número de partículas necesarias. La obtención de resultados correctos y repetibles no fue posible inicializando las partículas aleatoriamente, puesto que el número de partículas necesarias excede la capacidad de nuestro hardware para actualizarlas en tiempo real.

5.2.6. Problema del robot secuestrado

Para poner a prueba la robustez de nuestro algoritmo, modificamos el caso de estudio descrito anteriormente para *secuestrar* el robot una vez que ya estaba correctamente localizado y ver la reacción de nuestros filtros de partículas.

En la figura 5.17 podemos ver la evolución temporal de la prueba. Hasta el segundo 95 (marcado con la primera línea vertical en la figura) todo evoluciona como en nuestro caso de estudio anterior. En ese momento, configuramos la herramienta de reproducción de logs de CARMEN para que rebobinara las últimas 4500 líneas del log, lo que equivale a unos 8 metros de desplazamiento. En la figura 5.18 se puede apreciar la posición en la que se encontraba el robot, correctamente localizada por la mayoría de las partículas, y a su izquierda la posición a la que movimos el robot de forma instantánea, mediante la manipulación de la reproducción de los logs, señalada por un pequeño rectángulo negro.

Dos segundos después del movimiento, aparece la primera condición de disparo para el filtro secundario (marcada por la segunda línea vertical de la figura 5.17). Durante su ejecución, el filtro secundario consigue colocar partículas en la nueva posición correcta y, cuando algunas de ellas pasan al filtro primario en la siguiente fase de reseteo (señalada en la imagen con la tercera línea vertical), el filtro primario empieza a converger hacia la nueva posición del robot.

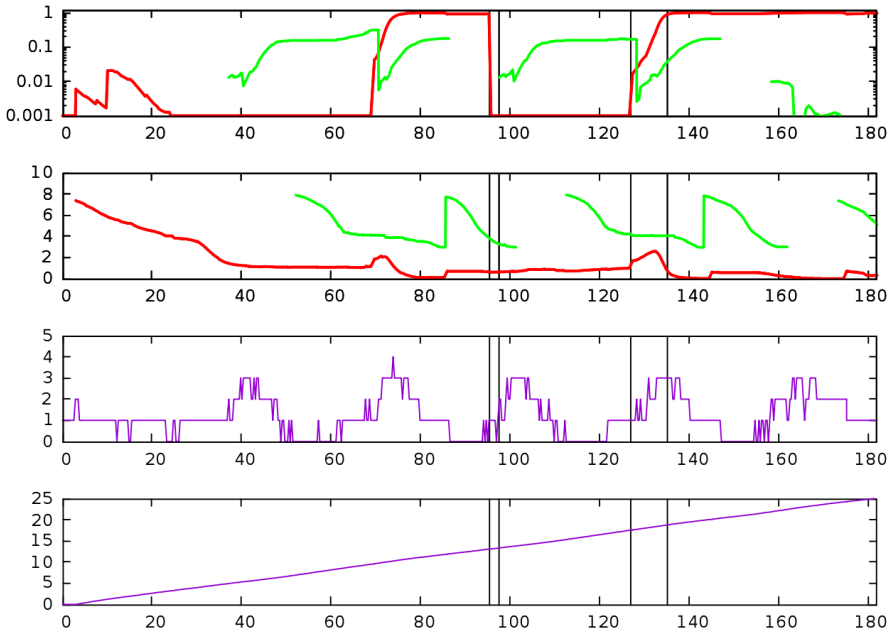


Figura 5.17: Evolución del caso de estudio manejando el problema del robot secuestrado. La primera gráfica muestra la probabilidad de cada filtro (rojo para el primario, verde para el secundario) colocada en el lugar correcto respecto al tiempo (en segundos). La segunda gráfica muestra la entropía de cada filtro con respecto al tiempo. La tercera gráfica muestra el número de puertas detectadas en cada momento, y la última el desplazamiento del robot (en metros).

Unos segundos más tarde, la salida del filtro primario ya ofrece la posición correcta (cuarta línea vertical en la figura 5.17). Desde el momento del cambio de posición hasta que el filtro la encuentra y proporciona el resultado correcto transcurren aproximadamente 40 segundos en los que el robot recorre unos 5 metros y medio.

5.2.7. Tasa de acierto

Además de detallar el funcionamiento de nuestro doble filtro de partículas en un caso particularmente complicado, hemos realizado otras 50 pruebas iniciando la ejecución de nuestro algoritmo de localización global en diferentes lugares a lo largo de todo el pasillo central, permitiendo un desplazamiento del robot de hasta 12 metros.

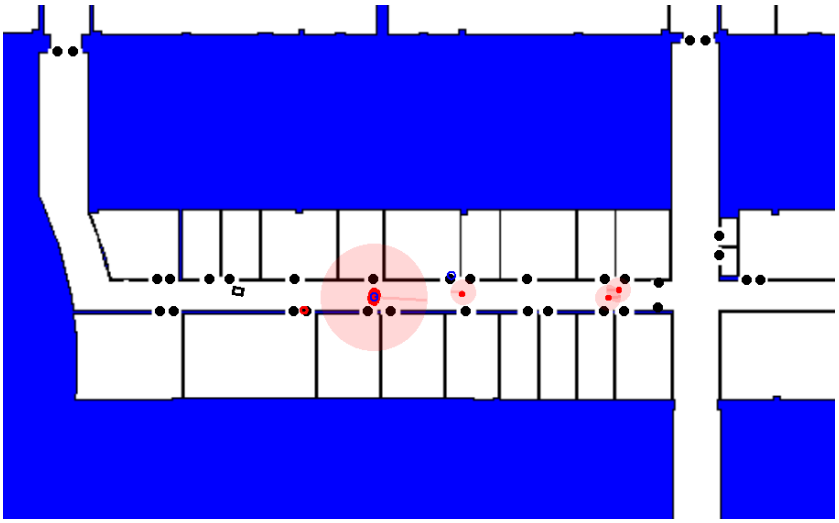


Figura 5.18: Evolución del caso de estudio con el problema del robot secuestrado

Durante estas pruebas, hemos almacenado el error en la salida de nuestro algoritmo y el desplazamiento del robot en cada momento. Cabe destacar que no hemos detenido ninguna de las pruebas hasta los 12 metros, por lo que, en caso de que el filtro se volviera inestable y empezara a proporcionar una salida errónea después de haber encontrado la posición correcta, el error se recogería en los datos. Para evaluar también que la orientación del robot proporcionada por el filtro y evitar que se considere válida la salida si la orientación del robot es muy distinta a la real, hemos valorado cada 20 grados de desviación sobre la posición del robot como un metro de distancia.

En la figura 5.19, podemos ver todos los datos recogidos en la misma gráfica. Se puede apreciar que según aumenta la distancia recorrida por el robot, el número de pruebas que ofrecen una salida errónea se reduce. En la figura 5.20 se han procesado los mismos datos para mostrar el número de pruebas que ofrecen una salida correcta (menos de 2 metros de distancia a la posición proporcionada por CARMEN) según el desplazamiento del robot.

En tan solo 4 metros de distancia, el 50 % de las pruebas ya habían localizado correctamente al robot. Una vez recorridos 9 metros, en el 90 % de los casos el filtro proporcionaba ya la situación correcta del robot, mientras que, pasados los 12 metros, el 98 % de los tests tuvieron éxito.

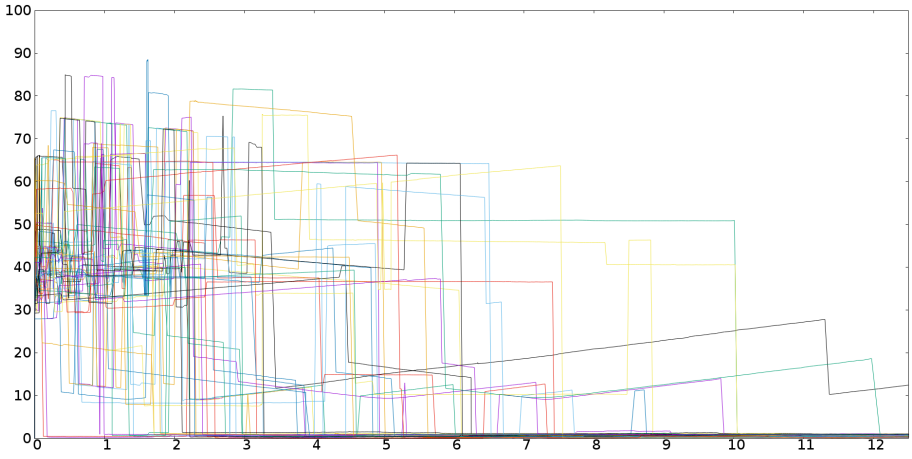


Figura 5.19: Evolución del error en el filtro (en metros) respecto al tiempo (en segundos) durante nuestras pruebas

Debemos tener en cuenta que los datos no fueron preprocesados de ninguna forma y los lugares de inicio no se escogieron manualmente. En algunos casos, el robot pasa por lugares en los que no ve puertas durante varios metros, lo cual retrasa la convergencia de los filtros aunque el robot esté en movimiento.

Puesto que sólo falló un caso, decidimos investigarlo a fondo para ver por qué el algoritmo no fue capaz de determinar la posición correcta del robot durante los primeros 12 metros de desplazamiento.

En la figura 5.21 podemos ver la representación del estado del filtro en distintos instantes de tiempo durante nuestra prueba fallida. Como podemos apreciar en la figura 5.21(a), la inicialización falla. Ninguna partícula se coloca en el lugar correcto porque la detección de puertas está proporcionando resultados erróneos en ese momento. Se detectan tres puertas, una de ellas correctamente, otra donde en realidad hay una pared, y una tercera en el medio de una puerta doble.

Segundos después de la inicialización, el robot gira 180 grados sin que haya ninguna partícula en el filtro primario que esté muestreando su posición real. Tras varios metros sin detectar ninguna puerta, aparecen dos juntas dentro del campo de visión del robot, lo que permite que el filtro secundario se inicialice y coloque partículas en el lugar correcto (figura 5.21(b)).

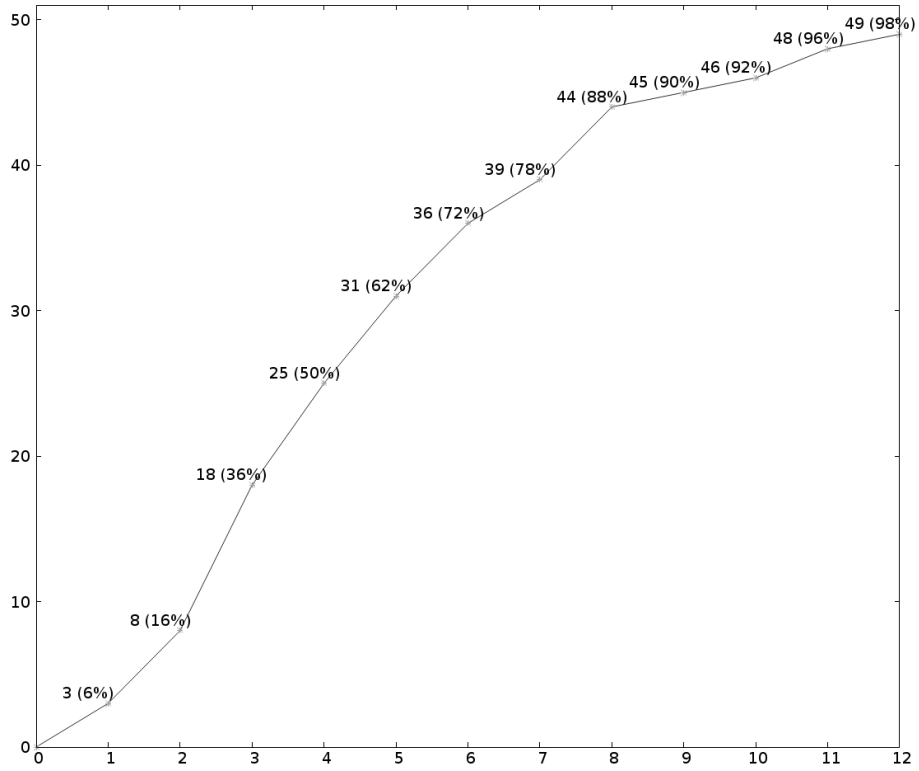
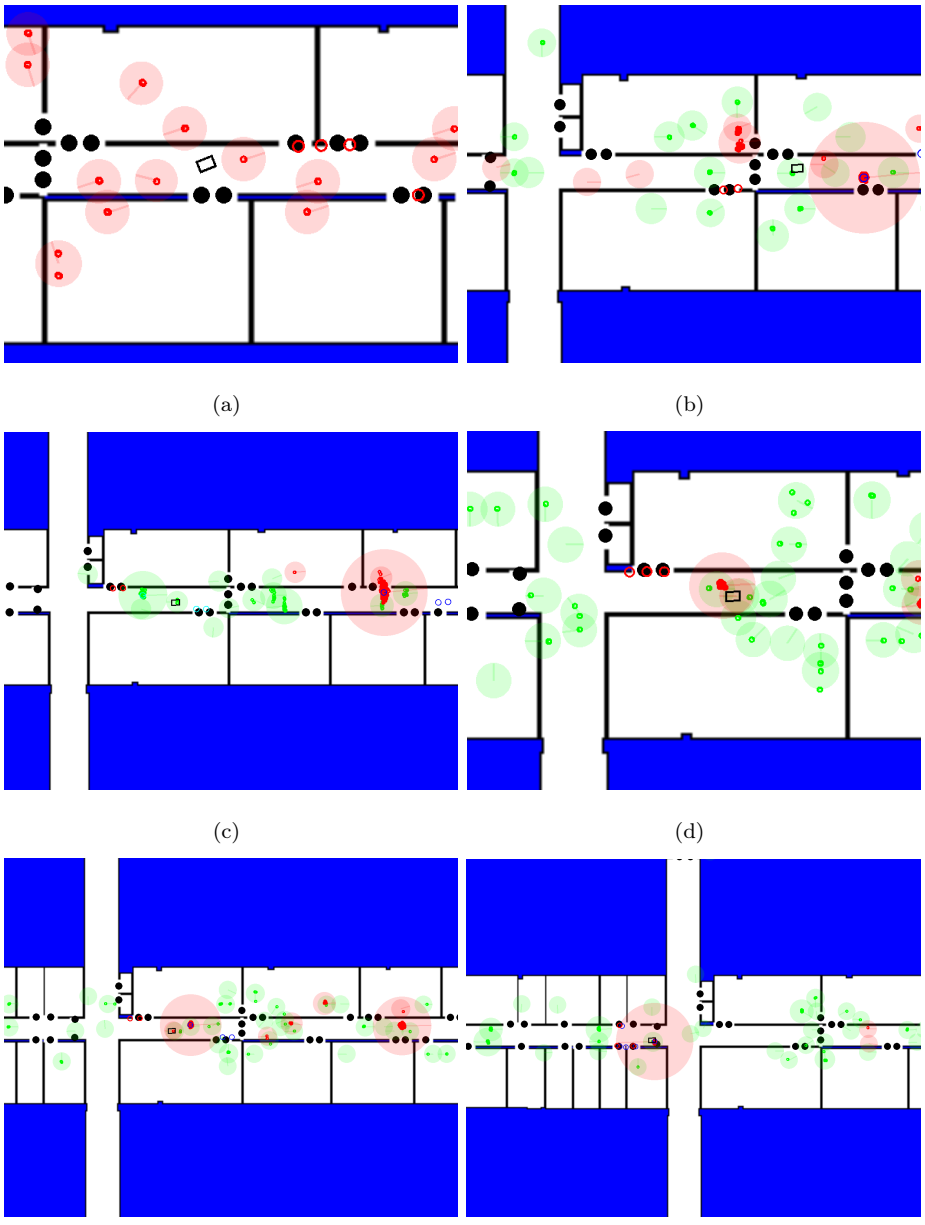


Figura 5.20: Número de pruebas en las que nuestro algoritmo proporciona la posición real del robot en función de su desplazamiento desde el inicio de la prueba

Unos metros más adelante, el filtro primario sigue sin conocer la posición del robot, mientras que el secundario evoluciona con varias partículas en el lugar correcto, pero un número mayor en lugares incorrectos debido a simetrías del entorno e imprecisiones en la detección de puertas (figura 5.21(c)).

Cuando se produce la primera *condición de reseteo*, el filtro secundario añade partículas en el lugar correcto al filtro primario (figura 5.21(d)). Desafortunadamente, justo en ese instante hay otra posición desde la que podrían verse las puertas detectadas en ese momento de la misma forma y que cuenta con una probabilidad más alta, casi en el mismo lugar en el que se encuentra el robot pero en la dirección opuesta.

Un par de metros más adelante, se siguen viendo sólo las dos puertas que también podrían verse desde la otra posición candidata, así que el filtro no puede evolucionar hacia la posición real del robot (figura 5.21(e)).



(a) (b)

(c) (d)

(e) (f)

Figura 5.21: Evolución de la prueba fallida.

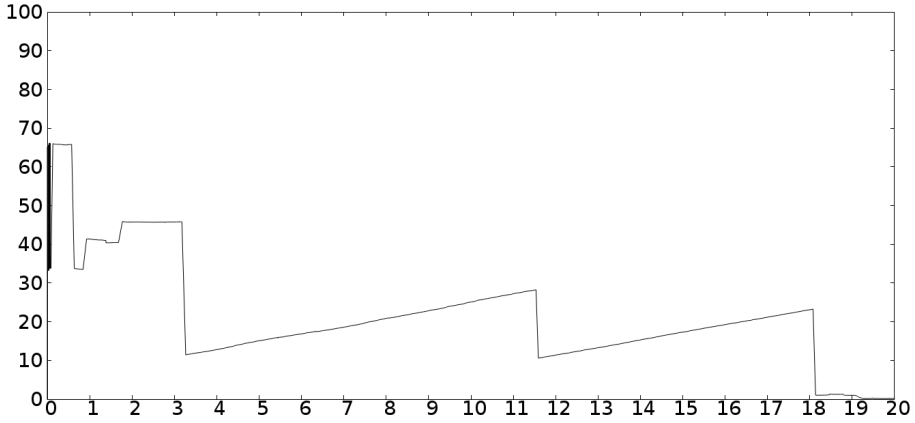


Figura 5.22: Evolución de la salida del filtro primario respecto al espacio recorrido

El robot sigue avanzando entre los dos pasillos principales, sin ver ninguna puerta durante varios metros, por lo que el filtro sigue equivocado cuando terminan los 12 metros de recorrido de la prueba.

En este caso, dejamos la prueba funcionando durante más tiempo para ver si finalmente el filtro era capaz de recuperarse de su error y, tan pronto como el robot empezó a percibir puertas otra vez, el filtro encontró la posición real del robot (figura 5.21(f)).

En la figura 5.22 se representa la evolución del error en la salida del filtro primario respecto al espacio recorrido. Como se puede apreciar, pasados 12 metros desde el inicio de la prueba la salida todavía es errónea. En este caso, fueron necesarios 18 metros de desplazamiento para encontrar la posición real. Podemos ver que, en este caso, han coincidido una gran cantidad de circunstancias adversas: una inicialización errónea debido a errores de detección, falta de puertas durante muchos metros que no permiten al filtro secundario inicializarse varias veces, y puertas que no ofrecen mucha información porque las simetrías del entorno no permiten decidir cuál es la posición correcta. Pero aún en esas circunstancias, el algoritmo termina encontrando la posición correcta en cuanto recibe información significativa.

6

Conclusiones

En este trabajo, hemos conseguido desarrollar con éxito un entorno de desarrollo de robots que integra dos de los mejores *RDEs* de código abierto que están disponibles públicamente: MissionLab y CARMEN.

El nuevo sistema cumple todas las especificaciones que habíamos definido y, tanto MissionLab como CARMEN, mantienen una compatibilidad hacia atrás completa. Además, el sistema permite el desarrollo de misiones multi-robot en las que todos los robots pueden aprovecharse de las mejores funcionalidades de ambos.

El rendimiento en términos de consumo de memoria y uso de la CPU ha mejorado respecto a la versión oficial de MissionLab. La plataforma resultante ha sido incluso reconocida por los creadores de MissionLab, que ofrecen un link hacia nuestro trabajo en su propia web oficial[1].

[1] http://www.cc.gatech.edu/ai/robot-lab/research/MissionLab/

Junto con el código fuente completo del nuevo entorno de desarrollo, proporcionamos paquetes *deb* y *rpm* precompilados para las últimas versiones de las distribuciones de Linux más extendidas (Fedora, Ubuntu, OpenSUSE, Debian, CentOS) y arquitecturas (x86, x86_64). De esta forma, facilitamos a los investigadores, docentes, o incluso a los usuarios ocasionales, la oportunidad de utilizar este entorno para sus proyectos. También proporcionamos instrucciones para reproducir nuestros misiones de demostración que no dependen de hardware específico[2] para facilitar la reproducción de nuestros resultados. Esperamos que este entorno de desarrollo sea útil en ambientes educativos y fomente la integración, o la implementación de las grandes funcionalidades proporcionadas por MissionLab (como creación gráfica de misiones, la generación automática de código, los comportamientos compuestos, etc.) en los entornos de desarrollo de robots más extendidos en la actualidad.

En cuanto al algoritmo de localización global desarrollado, hay varias razones que explican y justifican los resultados obtenidos. En primer lugar, la importancia de la inicialización de los filtros de partículas. Si queremos utilizar un número de partículas reducido en un entorno muy extenso, tenemos que confiar en las medidas externas de los sensores para colocarlas en los lugares con una mayor probabilidad. Si estas medidas son incorrectas, la inicialización también lo será y el filtro probablemente fallará. El arranque y parada cíclicos de nuestro filtro secundario de acuerdo a las condiciones de disparo y reseteo, le da al filtro primario más oportunidades de obtener la posición del robot.

En segundo lugar, la independencia entre las medidas externas en el momento de la inicialización del filtro secundario y en la fase de corrección del filtro primario después del muestreo híbrido, previene que las partículas añadidas procedentes del filtro secundario desestabilicen el primario. En métodos como SRL o Mixture-MCL, las nuevas partículas se añaden directamente al filtro y se corrigen utilizando los mismos datos que las generaron o, al menos, condicionalmente dependientes en gran medida. Esto da ventaja a las nuevas partículas frente al resto, empeorando los efectos negativos de los errores de detección. Con nuestro método, incluso si las partículas añadidas se generaron en base a detecciones incorrectas, no confunden al filtro primario.

[2] http://gro.usal.es/web/research.php?name=missionlab&section=index&lang=en

Bibliografía

[1] D. C. MacKenzie and R. C. Arkin, "Evaluating the usability of robot programming toolsets," *The International Journal of Robotics Research*, vol. 17, p. 381, apr 1998.

[2] R. C. Arkin, Y. Endo, B. Lee, D. MacKenzie, and E. Martinson, "Multistrategy learning methods for multirobot systems.," *Star*, vol. 44, no. Lm, pp. 137–150, 2006.

[3] D. C. MacKenzie and R. C. Arkin, "Formal specification for behavior-based mobile robots," in *Mobile Robots VIII* (W. J. Wolfe and W. H. Chun, eds.), pp. 94–104, feb 1994.

[4] T. Balch and R. C. Arkin, "Behavior-based formation control for multirobot teams," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 6, pp. 926–939, 1998.

[5] R. C. Arkin, T. R. Collins, and Y. Endo, "Tactical mobile robot mission specification and execution," in *Mobile Robots XIV*, pp. 150–163, 1999.

[6] S. Jiang and R. C. Arkin, "SLAM-Based Spatial Memory for Behavior-Based Robots," *IFAC-PapersOnLine*, vol. 48, no. 19, pp. 195–202, 2015.

[7] L. Moshkina, S. Park, R. C. Arkin, J. K. Lee, and H. Jung, "TAME: Time-Varying Affective Response for Humanoid Robots," *International Journal of Social Robotics*, vol. 3, pp. 207–221, aug 2011.

[8] R. C. Arkin, M. Scheutz, and L. Tickle-Degnen, "Preserving dignity in patient caregiver relationships using moral emotions and robots," *2014 IEEE*

International Symposium on Ethics in Science, Technology and Engineering, ETHICS 2014, 2014.

[9] R. C. Arkin and L. Moshkina, "Affect in Human-Robot Interaction," in *The Oxford Handbook of Affective Computing*, Oxford University Press, jan 2015.

[10] Y. Endo, D. MacKenzie, and R. Arkin, "Usability Evaluation of High-Level User Assistance for Robot Mission Specification," *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)*, vol. 34, pp. 168–180, may 2004.

[11] L. Moshkina, Y. Endo, and R. C. Arkin, "Usability evaluation of an automated mission repair mechanism for mobile robot mission specification," in *Proceeding of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction - HRI '06*, (New York, New York, USA), p. 57, ACM Press, 2006.

[12] L. Moshkina, Y. Endo, and R. C. Arkin, "Usability evaluation of an automated mission repair mechanism for mobile robot mission specification," in *Proceeding of the 1st ACM SIGCHI/SIGART conference on Human-robot interaction - HRI '06*, (New York, New York, USA), p. 57, ACM Press, 2006.

[13] Z. Kira, R. C. Arkin, and T. R. Collins, "A design process for robot capabilities and missions applied to micro-autonomous platforms," in *Nanotechnology*, vol. 7679, pp. 767911–767911–12, apr 2010.

[14] P. Ulam, Y. Endo, A. Wagner, and R. Arkin, "Integrated mission specification and task allocation for robot teams - Design and implementation," in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 4428–4435, IEEE, apr 2007.

[15] D. Lyons, R. Arkin, P. Nirmal, S. Jiang, and T. Liu, "A Software Tool for the Design of Critical Robot Missions with Performance Guarantees," *Procedia Computer Science*, vol. 16, pp. 888–897, 2013.

[16] D. Lyons, R. Arkin, P. Nirmal, and S. Jiang, "Designing autonomous robot missions with performance guarantees," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2583–2590, IEEE, oct 2012.

[17] R. C. Arkin, D. Lyons, J. Shu, P. Nirmal, and M. Zafar, "Getting it right the first time: predicted performance guarantees from the analysis of emergent behavior in autonomous and semi-autonomous systems," in *Proc. SPIE 8387, Unmanned Systems Technology XIV*, p. 83871F, may 2012.

[18] D. Lyons, S. Jiang, R. Arkin, P. Nirmal, S. Fox, and M. Zafar, "Characterizing performance guarantees for multiagent, real-time systems operating in noisy and uncertain environments," *Proceedings of the Workshop on Performance Metrics for Intelligent Systems - PerMIS '12*, p. 114, 2012.

[19] S. Jiang, R. C. Arkin, D. M. Lyons, T. M. Liu, and D. Harrington, "Performance guarantees for C-WMD robot missions," *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics, SSRR 2013*, 2013.

[20] M. O'Brien, R. Arkin, and D. Harrington, "Automatic Verification of Autonomous Robot Missions," *... , and Programming for ...*, vol. 8810, pp. 462–473, 2014.

[21] D. M. Lyons, R. C. Arkin, S. Jiang, D. Harrington, F. Tang, and P. Tang, "Probabilistic Verification of Multi-robot Missions in Uncertain Environments," in *2015 IEEE 27th International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 56–63, IEEE, nov 2015.

[22] D. M. Lyons, R. C. Arkin, S. Jiang, D. Harrington, and T. Liu, "Verifying and validating multirobot missions," *IEEE International Conference on Intelligent Robots and Systems*, pp. 1495–1502, 2014.

[23] D. M. Lyons, R. C. Arkin, S. Jiang, T. M. Liu, and P. Nirmal, "Performance Verification for Behavior-Based Robot Missions," *IEEE Transactions on Robotics*, vol. 31, pp. 619–636, jun 2015.

[24] W. Burgard, D. Fox, D. Hennig, and T. Schmidt, "Estimating the absolute position of a mobile robot using position probability grids," *Proc. of the Fourteenth National Conference on Artificial Intelligence*, 1996.

[25] Y. Bar-Shalom and T. E. Fortmann, "Tracking and Data Association," *Academic Press*, 1988.

[26] A. Doucet, "On sequential simulation-based methods for Bayesian filtering," *Technical report CUED/F-INFENG/TR.310*, 1998.

[27] S. Thrun, D. Fox, W. Burgard, and F. Dellaert, "Robust Monte Carlo Localization for Mobile Robots," *Artificial Intelligence*, vol. 128, pp. 99–141, may 2001.

[28] A. Milstein, J. Sánchez, and E. Williamson, "Robust global localization using clustered particle filtering," *Proceedings of the National ...*, 2002.

[29] S. Lenser and M. Veloso, "Sensor resetting localization for poorly modelled mobile robots," in *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No.00CH37065)*, vol. 2, pp. 1225–1232, IEEE, 2000.

[30] L. Marchetti, G. Grisetti, and L. Iocchi, "A comparative analysis of particle filter based localization methods," *RoboCup 2006: Robot Soccer World Cup ...*, 2007.

[31] D. Zhuohua and C. Zixing, "Evolutionary Particle Filter for Robust Simultaneous Localization and Map Building with Laser Range Finder," 2008.

[32] F. Lu and E. Milios, "Robot Pose Estimation in Unknown Environments by Matching 2D Range Scans," *Journal of Intelligent and Robotic Systems*, vol. 18, no. 3, pp. 249–275, 1997.

[33] M. Babu and A. Ramprasad, "Energy Aware Adaptative Monte Carlo Localization algorithm for WSN based on Antithetic Markov chain (AMCAM)," *International Journal of Computer Engineering & Technology*, vol. 3, no. 1, pp. 180–190, 2012.

[34] K. Yun and D. Kim, "Robust location tracking using a dual layer particle filter," *Pervasive and Mobile Computing*, vol. 3, pp. 209–232, jun 2007.

[35] N. Bhat and T. Vashisth, "Robot Localization by Particle Filter using Visual Database," *IRNet Transactions on Computer Science and Engineering*, no. 1, pp. 22–27, 2012.

[36] J. Kramer and M. Scheutz, "Development environments for autonomous mobile robots: A survey," *Autonomous Robots*, vol. 22, pp. 101–132, dec 2007.

[37] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, vol. 32 of *Open-Source Software workshop*, pp. 151–170, 2009.

[38] H. Bruyninckx, "Open robot control software: the OROCOS project," in *Proceedings 2001 ICRA. IEEE International Conference on Robotics and Automation (Cat. No.01CH37164)*, vol. 3, pp. 2523–2528, IEEE, IEEE, 2001.

[39] B. P. Gerkey, R. T. Vaughan, and A. Howard, "The Player / Stage Project : Tools for Multi-Robot and Distributed Sensor Systems," *Proceedings of the International Conference on Advanced Robotics (ICAR 2003)*, vol. 1, pp. 317–323, 2003.

[40] M. Montemerlo, N. Roy, and S. Thrun, "Perspectives on standardization in mobile robot programming: the Carnegie Mellon Navigation (CARMEN) Toolkit," in *Proceedings 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2003) (Cat. No.03CH37453)*, vol. 3, pp. 2436–2441, IEEE, 2003.

[41] D. C. Mackenzie, *A design methodology for the configuration of behavior-based mobile robots*. PhD thesis, Georgia Institute of Technology, 1996.

[42] D. C. MacKenzie, R. Arkin, and J. M. Cameron, "Multiagent Mission Specification and Execution," *Autonomous Robots*, vol. 4, no. 1, pp. 29–52, 1997.

[43] Y. Endo, P. D. Ulam, R. C. Arkin, T. R. Balch, and M. D. Powers, "An Empirical Evaluation of Context-Sensitive Pose Estimators in an Urban Outdoor Environment," tech. rep., Georgia Tech Mobile Robot Laboratory, Atlanta, Georgia, 2005.

[44] J. Main, T. Dillon, and S. Shiu, "A tutorial on case based reasoning," in *Soft computing in case based reasoning*, pp. 1–28, London: Springer London, 2001.

[45] S. Koenig and M. Likhachev, "Fast replanning for navigation in unknown terrain," in *IEEE Transactions on Robotics*, vol. 21, pp. 354–363, IEEE, 2005.

[46] D. Hahnel, D. Schulz, and W. Burgard, "Map building with mobile robots in populated environments," in *Robots and Systems,*, vol. 17, pp. 579–597, IEEE, 2002.

[47] K. Konolige, "A gradient method for realtime robot control," in *Iros*, vol. 1, pp. 639–646, IEEE, 2000.

[48] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robotics and Automation Magazine*, vol. 4, pp. 23–33, mar 1997.

[49] G. Biggs, R. B. Rusu, T. Collett, B. Gerkey, and R. Vaughan, "All the robots merely players: History of player and stage software," *IEEE Robotics and Automation Magazine*, vol. 20, pp. 82–90, sep 2013.

[50] D. Fox, S. Thrun, W. Burgard, and F. Dellaert, "Particle filters for mobile robot localization," in *Sequential Monte Carlo Methods in Practice* (A. Doucet, N. de Freitas, and N. Gordon, eds.), Springer Verlag, 2000.

[51] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," *Radar and Signal Processing IEE Proceedings F*, vol. 140, no. 2, pp. 107–113, 1993.

[52] F. Dellaert, D. Fox, W. Burgard, and S. Thrun, "Monte Carlo localization for mobile robots," *IEEE International Conference on Robotics and Automation (ICRA99)*, may 1999.

[53] A. Milstein, *Improved particle filter based localization and mapping techniques*. PhD thesis, University of Waterloo, 2008.

[54] L. Zhang, R. Zapata, and P. Lepinay, "Self-adaptive Monte Carlo localization for mobile robots using range sensors," in *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 1541–1546, IEEE, oct 2009.

[55] R. Smith, M. Self, and P. Cheeseman, "Estimating uncertain spatial relationships in robotics," in *Autonomous robot vehicles* (I. J. Cox and G. T. Wilfong, eds.), ch. Estimating, pp. 167–193, Springer-Verlag New York, Inc., 1990.

[56] J. Carpenter, P. Clifford, and P. Fearnhead, "Improved particle filter for nonlinear problems," *IEE Proceedings - Radar, Sonar and Navigation*, vol. 146, no. 1, p. 2, 1999.

[57] D. Fox, "Adapting the Sample Size in Particle Filters Through KLD-Sampling," *The International Journal of Robotics Research*, vol. 22, no. 12, pp. 985–1003, 2003.

[58] S. Thrun, W. Burgard, and D. Fox, *Probabilistic robotics*. 2005.

[59] H. Mori and S. Kotani, "Robotic Travel Aid for the Blind: HARUNOBU-6," in *Proceedings of the Second European Conference on Disability Virtual Reality and Associated Technologies*, pp. 193–202, 1998.

[60] P. Bonnifait, M. Jabbour, and V. Cherfaoui, "Autonomous Navigation in Urban Areas using {GIS}-Managed Information," *International Journal of Vehicle Autonomous Systems*, vol. 6, no. 1-2, pp. 83–103, 2008.

[61] C. Cappelle, M. El Badaoui El Najjar, D. Pomorski, and F. Charpillet, "Intelligent Geolocalization in Urban Areas Using Global Positioning Systems, Three-Dimensional Geographic Information Systems, and Vision," *Journal of Intelligent Transportation Systems*, vol. 14, no. 1, pp. 3–12, 2010.

[62] J. M. Mirats Tur, C. Zinggerling, and A. Corominas Murtra, "Geographical information systems for map based navigation in urban environments," *Robotics and Autonomous Systems*, vol. 57, pp. 922–930, 2009.

[63] T. Yoshida, K. Nagatani, E. Koyanagi, Y. Hada, K. Ohno, S. Maeyama, H. Akiyama, K. Yoshida, and S. Tadokoro, "Field Experiment on Multiple

Mobile Robots Conducted in an Underground Mall," in *Field and Service Robotics* (A. Howard, K. Iagnemma, and A. Kelly, eds.), vol. 62 of *Springer Tracts in Advanced Robotics*, pp. 365–375, Springer Berlin / Heidelberg, 2010.

[64] I. Noda and J.-i. Meguro, "Data Representation for Information Sharing and Integration among Rescue Robot and Simulation," in *Proceedings of SICE-ICASE International Joint Conference*, pp. 3449–3454, oct 2006.

[65] N. Rackliffe, H. A. Yanco, and J. Casper, "Using {G}eographic {I}nformation {S}ystems ({GIS}) for {UAV} landings and {UGV} navigation," in *IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pp. 145–150, apr 2011.

[66] A. K. W. Yeung and G. B. Hall, *Spatial Database Systems*, vol. 87 of *GeoJournal Library*. Springer, 2007.

[67] U. Isikdag, J. Underwood, and G. Aouad, "An investigation into the applicability of building information models in geospatial environment in support of site selection and fire response management processes," *Advanced Engineering Informatics*, vol. 22, pp. 504–519, oct 2008.

[68] L. van den Brink, C. Portele, and P. A. Vretanos, "Geography Markup Language (GML) simple features profile (Corrigendum)," *Open Geospatial Consortium Inc.*, 2012.

[69] OpenGIS Geospatial Consortium, "OpenGIS implementation Standard for Geographic Information - Simple Feature Access - Part 2: SQL option," *2011*.

[70] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, Xuan Liu, and Chang-Tien Lu, "Spatial databases-accomplishments and research needs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 11, no. 1, pp. 45–55, 1999.

[71] V. Gandhi, J. M. Kang, and S. Shekhar, *Spatial Databases*. Hoboken, NJ, USA: John Wiley & Sons, Inc., mar 2008.

[72] Open Geospatial Consortium, "OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 1: Common architecture," *OpenGIS Implementation Standard*, 2011.

[73] M. Eisler and Network Appliance, "XDR: External Data Representation Standard," tech. rep., jun 2006.

[74] C. Fernández-Caramés, V. Moreno, B. Curto, J. F. Rodríguez-Aragón, and F. J. Serrano, "A Real-time Door Detection System for Domestic Robotic Navigation," *Journal of Intelligent & Robotic Systems*, vol. 76, pp. 119–136, sep 2014.

[75] C. Fernández-Caramés, F. J. Serrano, V. Moreno, B. Curto, J. Rodríguez-Aragón, and R. Alves, "A real-time indoor localization approach integrated with a Geographic Information System (GIS)," *Robotics and Autonomous Systems*, vol. 75, pp. 475–489, jan 2016.