UNIVERSIDAD DE SEVILLA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

PROYECTO FINAL DE CARRERA
INGENIERO DE TELECOMUNICACIÓN

Diseño e Implementación de Interfaz Persona Máquina y Sistema de Análisis Post-Misión para robot autónomo móvil Romeo-4R

Autor:
D. DANIEL PÉREZ RODRÍGUEZ

Tutor:
Dr. D. JESÚS IVÁN MAZA ALCAÑIZ

Ponente:
Dr. D. LUIS MERINO CABAÑAS

Enero, 2013

Este documento ha sido generado con LaTeX.

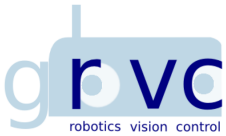
A mis padres,
por ser responsables de ser como soy ...

Proyecto realizado en el Departamento de Ingeniería de Sistemas y Automática

http://www.esi2.us.es/ISA/GAR/

En colaboración con el Grupo de Robótica, Visión y Control, dirigido por el Catedrático Dr. D. Aníbal Ollero Baturone

http://grvc.us.es/



AGRADECIMIENTOS

En primer lugar, me gustaría agradecer sinceramente a todas las personas que componen el Grupo de Robótica, Visión y Control de la Escuela Superior de Ingenieros de Sevilla, liderados por el catedrático del Departamento de Ingeniería de Sistemas y Automática, Dr. D. Aníbal Ollero Baturone, por darme la oportunidad de aprender y trabajar a su lado y hacer de todo este tiempo una maravillosa experiencia de aprendizaje, tanto profesional como personal.

Agradecer, ya de forma más específica a mi tutor en el proyecto URUS y director de este proyecto fin de carrera, el Dr. D. Luis Merino Cabañas, por sus indicaciones durante el trabajo realizado y agradecer también a mis compañeros Francisco Real Pérez, Pablo Soriano Tapia y Miguel Ángel Rodríguez, por tener paciencia conmigo y aportarme tanto durante este tiempo en URUS y en otros proyectos.

Mencionar sin ninguna duda el apoyo y conocimientos recibidos de los profesores del Departamento de Ingeniería de Sistemas y Automática de la Escuela Superior de Ingenieros de Sevilla, principalmente al Dr. D. Iván Maza Alcañiz, tutor de este proyecto fin de carrera y al Dr. D. Fernando Caballero Benítez por toda la experiencia aprendida en múltiples proyectos así como al Dr. D. Guillermo Heredia Benot, al Dr. D. Ángel Rodríguez Castaño y al resto de profesores del Grupo de Robótica, Visión y Control.

Gracias también al resto de compañeros y miembros del grupo con lo que mi trabajo y día a día ha sido más cercano, Víctor, Jesús, Roberto, José Antonio, Sabine, Adrián, ... y alguno que otro que seguro me dejo en el tintero.

Tengo que buscar un hueco los habitantes de la Palmilla, Vito, Jordi, Fran, Eugenio, ... Ellos saben por qué están aquí.

Sin duda no se me ocurre mejor forma de casi acabar esta página de agradecimientos que dándole las gracias a mi familia, a mis padres y a mi hermano, porque sin duda todo lo que soy y todo lo que este trabajo significa no habría tenido sentido sin su apoyo y cariño a lo largo de todo este tiempo, que no ha sido poco.

El último gesto de agradecimiento no podría ser para otra persona que para ti Mari Carmen. No tengo palabras para agradecerte el apoyo y cariño que durante tanto tiempo han sido el motor que ha impulsado mi vida y la brújula que siempre me ha ayudado a encontrar el camino, deseando que sigan siéndolo en las nuevas aventuras que encontraremos en nuestro camino juntos.

PREFACIO

La introducción de la robótica en la vida cotidiana de los seres humanos es un proceso que continua creciendo. Existen múltiples iniciativas y proyectos que apuestan y tratan de avanzar en este campo, como es el caso de proyecto europeo URUS [8] *Ubiquitous Networking Robotics in Urban Settings*, que trata de desarrollar nuevas formas de cooperación entre redes de robots y sensores en zonas peatonales del área urbana.

Este proyecto fin de carrera se centra en recoger el trabajo desarrollado por el autor en el contexto del proyecto URUS, desde septiembre de 2008 hasta junio de 2009, a través de una beca en el Grupo de Robótica, Visión y Control, participante activo del proyecto URUS. Dicho trabajo consistió principalmente en el diseño y realización de una interfaz hombre máquina para el control del robot Romeo-4R, así como alguna labor de soporte y apoyo en otros aspectos del mantenimiento y desarrollo de los módulos del robot, destacando en estos aspectos el desarrollo del sistema de análisis Post-Misión, o aplicación Logplayer, para la reproducción de los experimentos realizados en la realidad en un entorno de simulación.

Por ello, el primer capítulo de la memoria se dedica a introducir la robótica móvil, los problemas principales que aparecen en la navegación autónoma y específicamente en entornos urbanos; también dedicamos una introducción al diseño de interfaces hombre máquina/computador, así como otra breve descripción de los aspectos y resultados más importantes del proyecto URUS para acabar recapitulando los objetivos principales del proyecto fin de carrera, consistentes en las dos aplicaciones que ya hemos anticipado en el párrafo anterior.

El segundo capítulo se dedica a una completa descripción tanto a nivel hardware como software del robot Romeo-4R. Dicho conocimiento del sistema es básico para realizar la integración de la interfaz hombre máquina en el robot.

El tercer capítulo recoge una descripción de las principales librerías software, basadas completamente en software libre, utilizadas en los módulos del robot y por lo tanto en la propia interfaz, que como se verá, se entenderá como un módulo más del sistema completo.

El cuarto capítulo realiza una descripción completa del diseño y funcionamiento de la interfaz gráfica Romeo HMI mientras que el quinto capítulo realiza una descripción análoga del Sistema de Análisis Post-Misión, o módulo Logplayer.

El sexto capítulo recoge las principales conclusiones y resultados obtenidos en el desarrollo de la interfaz y del módulo Logplayer y de su uso posterior en los experimentos finales del proyecto URUS.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

1.1. Introducción a la robótica

En el término robot confluyen las imágenes de máquinas para la realización de trabajos productivos y de imitación de movimientos y comportamientos de seres vivos.

Los robots actuales son obras de ingeniería y como tales concebidas para producir bienes y servicios o explotar recursos naturales.

En el proceso de creación de un robot, confluyen numerosos conocimientos y ramas de la ciencia y la tecnología, entre los que caben destacar la mecánica, la electrónica, la informática, inteligencia artificial e ingeniería de control.

Por tanto, podríamos definir a los robots como máquinas en las que se integran componentes mecánicos, eléctricos, electrónicos y de comunicaciones, dotados de un sistema informático para su control en tiempo real, percepción del entorno y programación.

En la figura 1.1 se muestra el esquema básico de un robot. En él se identifican un sistema mecánico, actuadores, sensores y el sistema de control como elemento básico necesario para cerrar la cadena Actuación - Medidas - Actuación.

Los sensores, se encargan de las labores de percepción y medida que necesita el robot para realizar su tarea.

Podemos realizar una clasificación de los sensores de un robot en externos e internos.

Los sensores internos miden el estado de la estructura mecánica del robot y, en particular, giros o desplazamientos relativos entre articulaciones, velocidades,

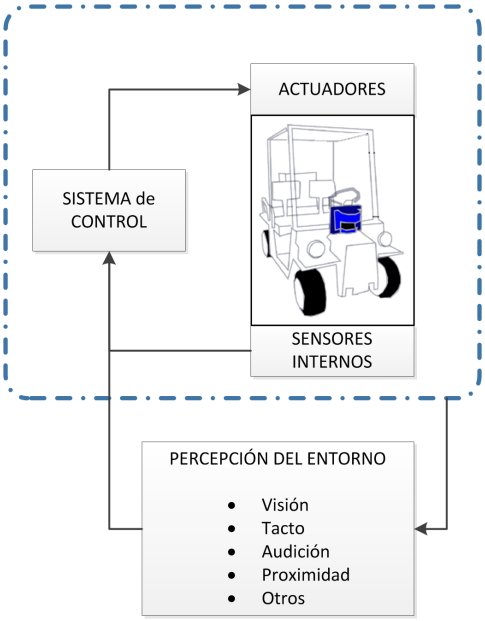


Figura 1.1: Robot y su interacción con el entorno.

fuerzas y pares.

Los sensores externos permiten dotar de sentidos al robot. La información que suministran es utilizada por el sistema de percepción para aprehender la realidad del entorno.

Los sistemas de percepción sensorial hacen posible que un robot pueda adaptar automáticamente su comportamiento en función de las variaciones que se producen en su entorno.

Es decir, los sensores permiten al robot conocer el universo a su alrededor así como conocer su propio estado y actuar en consecuencia.

Existen diversos criterios para la clasificación de los distintos robots.

Según [23], atendiendo a su grado de autonomía, los robots pueden clasificarse en teleoperados, de funcionamiento repetitivo y autónomos o inteligentes.

En los **robots teleoperados** las tareas de percepción del entorno, planificación y manipulación compleja son realizados por humanos. Es decir, el operador actúa en tiempo real cerrando un bucle de control de alto nivel. Los sistemas evolucionados suministran al operador realimentación sensorial del entorno (imágenes, fuerzas, distancias).

En manipulación se emplean brazos y manos antropomórficos con controladores automáticos que reproducen los movimientos del operador.

Alternativamente, el operador mueve una réplica a escala del manipulador, reproduciéndose los movimientos de éste.

Figura 1.2: ***Robot Konabot.***

Estos robots son interesantes para trabajos en una localización remota (acceso difícil, medios contaminados o peligroso), en tareas difíciles de automatizar y en entornos no estructurados, tales como las que se realizan en la construcción o en el mantenimiento de líneas eléctricas.

Las mayores dificultades radican en las limitaciones del hombre en la capacidad de procesamiento numérico y precisión y, sobre todo en el acoplamiento y coordinación entre el hombre y el robot. En algunas aplicaciones el retraso de transmisión de información resulta fundamental en el diseño del sistema de control. El diseño de la interfaz persona-máquina suele ser crítico. La investigación actual se dirige a hacer recaer en el operador únicamente las tareas que requieren toma de decisiones en función de información sensorial, experiencia y habilidad. No obstante existen limitaciones por el ancho de banda de la transmisión y, eventualmente por la complejidad de la tarea del operador.

Los **robots de funcionamiento repetitivo** son la mayor parte de los que se emplean en cadenas de producción industrial. Trabajan normalmente en tareas predecibles e invariantes, con una limitada percepción del entorno. Son precisos, de alta repetibilidad y relativamente rápidos; incrementan la productividad ahorrando al hombre trabajos repetitivos y, eventualmente, muy penosos o incluso peligrosos.

Los **robots autónomos o inteligentes** son los más evolucionados desde el punto de vista del procesamiento de información Son máquinas capaces de percibir, modelar el entorno, planificar y actuar para alcanzar objetivos sin la intervención, o con una intervención mínima de supervisores humanos. Pueden trabajar en en-

Figura 1.3: Robot KUKA KR150.

tornos poco estructurados y dinámicos, realizando acciones en respuesta a contingencias variadas en dicho entorno. Durante las últimas décadas se han realizado importantes esfuerzos en la aplicación de técnicas de inteligencia artificial. Se han empleado métodos simbólicos de tratamiento de la información basados en modelos geométricos del entorno.

De esta forma, se resuelven problemas basados en un modelo previo del entorno cuyas soluciones sólo son válidas si el modelo corresponde exactamente a la realidad. La técnica obvia de reducir esta incertidumbre consiste en incrementar la información de que se dispone de dicho entorno mediante realimentación sensorial. Existen métodos que permiten intercalar la formulación y ejecución de planes con la captación de la información necesaria para asegurar que el modelo que se utiliza para la planificación sea lo suficientemente fiable. Las limitaciones vienen impuestas por el sistema de percepción y por la propia arquitectura del sistema de información y control del robot.

Desde el punto de vista de la planificación, existen diferentes arquitecturas diseñadas teniendo en cuenta especificaciones sobre el tiempo que tiene el sistema para responder y la disponibilidad de información potencialmente interesante.

La solución se sitúa normalmente entre dos extremos, en uno de los cuales está la planificación puramente estratégica. En este caso, se supone que la situación en la que va a ejecutarse el plan puede ser predecida de forma suficientemente precisa durante la planificación. En el otro extremo se sitúa la planificación puramente reactiva en la que se supone que el entorno es incierto, buscándose la mayor flexibilidad posible para reaccionar en cualquier instante lo suficientemente rápido a las discrepancias entre el modelo actual y la realidad observada en el entorno.

El problema puede plantearse también en términos de un compromiso entre

eficiencia y flexibilidad. En efecto, las arquitecturas diseñadas para conseguir la mayor flexibilidad ante cualquier eventualidad del entorno son mucho menos eficientes que las que utilizan criterios de decisión basados en modelos del entorno suficientemente precisos sin tener demasiado en cuenta la posibilidad de generalizar el comportamiento. En este punto conviene poner de manifiesto el interés de las arquitecturas con capacidad de aprendizaje que combinan la planificación estratégica, basada en técnicas de búsqueda, con la planificación puramente reactiva.

Otra clasificación interesante de los robots puede realizarse atendiendo a su arquitectura. La arquitectura, que es definida por el tipo de configuración general del Robot, puede ser metamórfica. El concepto de metamorfismo, de reciente aparición, se ha introducido para incrementar la flexibilidad funcional de un Robot a través del cambio de su configuración por el propio Robot. El metamorfismo admite diversos niveles, desde los más elementales (cambio de herramienta o de efecto terminal), hasta los más complejos como el cambio o alteración de algunos de sus elementos o subsistemas estructurales. Los dispositivos y mecanismos que pueden agruparse bajo la denominación genérica del Robot, tal como se ha indicado, son muy diversos y es por tanto difícil establecer una clasificación coherente de los mismos que resista un análisis crítico y riguroso. Así pues, una subdivisión de los Robots, con base en su arquitectura, se hace en los siguientes grupos: Poliarticulados, Móviles, Androides, Zoomórficos e Híbridos.

Los *robots poliarticulados* constituyen un grupo que incluye robots de muy diversa forma y configuración cuya característica común es la de ser básicamente sedentarios (aunque excepcionalmente pueden ser guiados para efectuar desplazamientos limitados) y estar estructurados para mover sus elementos terminales en un determinado espacio de trabajo según uno o más sistemas de coordenadas y con un *número limitado de grados de libertad*. En este grupo se encuentran los manipuladores, los robots industriales, y se emplean cuando es preciso abarcar una zona de trabajo relativamente amplia o alargada, actuar sobre objetos con un plano de simetría vertical o reducir el espacio ocupado en el suelo.

Los *robots móviles* son robots con grandes capacidad de desplazamiento, basados en carros o plataformas y dotados de un sistema locomotor de tipo rodante. Siguen su camino por telemando o guiándose por la información recibida de su entorno a través de sus sensores. Estos robots, en entornos industriales aseguran el transporte de piezas de un punto a otro de una cadena de fabricación. Guiados mediante pistas materializadas a través de la radiación electromagnética de circuitos empotrados en el suelo, o a través de bandas detectadas fotoeléctricamente, pueden incluso llegar a sortear obstáculos y están dotados de un nivel relativamente elevado de inteligencia.

La robótica móvil se extiende también más allá de los entornos industriales, apareciendo robots móviles dotados de una gran autonomía e inteligencia tanto

Figura 1.4: Robots industriales Staubli RX90. Laboratorio de Sistemas y Automática.

para la navegación como para la evitación de obstáculos. A este respecto es muy interesante el campeonato y los equipos participantes en el Darpa Urban Challenge [2].

Figura 1.5: Robot móvil Romeo-4R. Experimento de evitación de obstáculo.

Los *androides* son robots que intentan reproducir total o parcialmente la forma y el comportamiento cinemático del ser humano.

Actualmente los androides son todavía dispositivos muy poco evolucionados y sin utilidad práctica real, y destinados, fundamentalmente, al estudio y experimentación. Uno de los aspectos más complejos de estos robots, y sobre el que se centra la mayoría de los trabajos, es el de la locomoción bípeda.

En este caso, el principal problema es controlar de forma dinámica y coordinadamente en tiempo real el proceso a realizar y mantener simultáneamente el equilibrio del robot.

Los *robots zoomórficos*, que considerados en sentido no restrictivo podrían incluir también a los androides, constituyen una clase caracterizada principalmente

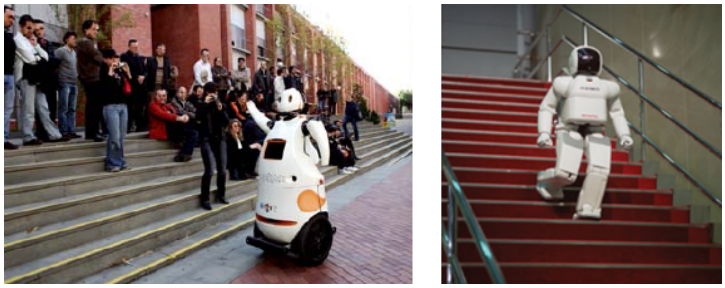


Figura 1.6: Izquierda) Robot Tibi (Universidad Politécnica de Cataluña). Derecha) Robot Asimo (Honda).

por sus sistemas de locomoción que imitan a los diversos seres vivos. A pesar de la disparidad morfológica de sus posibles sistemas de locomoción es conveniente agrupar a los robots zoomórficos en dos categorías principales: caminadores y no caminadores.

El grupo de los robots zoomórficos no caminadores está muy poco evolucionado. Los experimentados efectuados en Japón basados en segmentos cilíndricos biselados acoplados axialmente entre sí y dotados de un movimiento relativo de rotación. Los robots zoomórficos caminadores multípedos son muy numeroso y están siendo experimentados en diversos laboratorios con vistas al desarrollo posterior de verdaderos vehículos terrenos, pilotados o autónomos, capaces de evolucionar en superficies muy accidentadas. Las aplicaciones de estos robots serán interesantes en el campo de la exploración espacial o en el estudio de los volcanes.

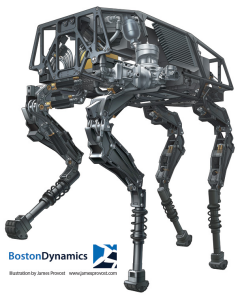


Figura 1.7: Robot Big Dog. Boston Dynamics.

Por último, los *robots híbridos* corresponden a aquellos de difícil clasificación cuya estructura se sitúa en combinación con alguna de las anteriores ya expuestas. Por ejemplo, un dispositivo segmentado articulado y con ruedas, posee al mismo tiempo atributos de los robots móviles y de los robots zoomórficos.

1.2. Robótica móvil

El desarrollo de robots móviles responde a la necesidad de extender el campo de aplicación de la robótica, restringido inicialmente al alcance de una estructura mecánica anclada en uno de sus extremos. Se trata también de incrementar la autonomía, limitando todo lo posible la intervención humana y aumentando en definitiva, las posibilidades y funcionalidades de los robots.

El objetivo final de la robótica móvil consiste en dotar al robot de la suficiente inteligencia como para reaccionar y tomar decisiones basándose en observaciones de su entorno, sin suponer que este entorno es perfectamente conocido a priori.

La autonomía de un robot móvil se basa principalmente en el sistema de navegación automática. En estos sistemas se incluyen tareas de planificación, percepción y control. En los robots móviles el problema de la planificación, en el caso más general, puede descomponerse en planificación global de la misión, de la ruta, de la trayectoria y finalmente, evitar obstáculos no esperados.

Existen numerosos métodos de planificación de caminos para robots móviles que se basan en hipótesis simplificadoras, tales como: entorno conocido y estático, robots omnidireccionales, con movimiento lento y ejecución perfecta de trayectoria. En particular hay muchos métodos que buscan caminos libres de obstáculos que minimizan la distancia recorrida en un entorno modelado mediante polígonos. En otros casos, se modela el espacio libre tratando de encontrar caminos por el centro del mismo. Para facilitar la búsqueda existen técnicas de descomposición del espacio en celdas, utilización de restricciones de varios niveles de resolución y búsqueda jerarquizada, que permiten hacer más eficiente el proceso con vistas a su aplicación en tiempo real, minimizando el coste computacional del mismo.

La planificación de la trayectoria puede realizarse también de forma dinámica, considerando la posición actual del vehículo y los puntos intermedios de paso definidos en la planificación de la ruta. La trayectoria se corrige debido a acontecimientos no considerados. La definición de la trayectoria debe tener en cuenta las características cinemáticas del vehículo en cuestión. Por ejemplo, en vehículos con ruedas y tracción convencional, interesa definir trayectorias de curvatura continua que puedan ejecutarse con el menor error posible.

Además de las características geométricas y cinemáticas, puede ser necesario tener en cuenta modelos dinámicos de comportamiento del vehículo contemplando la interacción vehículo-terreno, es decir, el comportamiento de un mismo vehículo puede ser muy diferente según el tipo de terreno por el que se desplace. Por otra parte puede plantearse también el problema de la planificación de la velocidad teniendo en cuenta las características del terreno y del camino que se pretenda seguir.

Una vez realizada la planificación de la trayectoria, es necesario planificar movimientos concretos y controlar dichos movimientos para mantener al vehículo en la trayectoria planificada. De esta forma, se plantea el problema del seguimiento de caminos, que para vehículos con ruedas se concreta en determinar el ángulo de dirección teniendo en cuenta la posición y orientación actual del vehículo con respecto a la trayectoria que debe seguir. Asimismo, es necesario resolver el problema del control y regulación de la velocidad del vehículo durante todo el recorrido de la trayectoria.

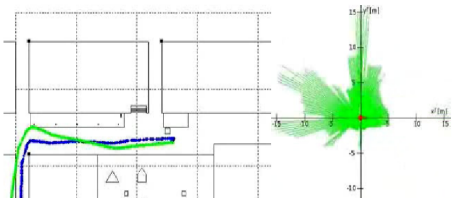


Figura 1.8: Proyecto URUS. Trayectorias de Robots.

En cualquier caso, el problema del control automático preciso de un vehículo con ruedas puede resultar más complejo que el de los manipuladores debido a la presencia de restricciones no holónomas. Los bucles de control se plantean tanto en el espacio de las variables articulares como en coordenadas del mundo, y las ecuaciones de movimiento son complejas si se considera la interacción con el terreno. Mientras en manipuladores es relativamente fácil el cálculo y medida de los pares y fuerzas que se ejercen sobre la estructura mecánica, la determinación de estos pares en vehículos con ruedas es muy difícil. En la actualidad se emplean fundamentalmente métodos geométricos y modelos cinemáticos simplificados. No obstante, la consideración de aspectos dinámicos es necesaria cuando la velocidad es alta.

Nótese también que el control del vehículo requiere disponer de medidas de su posición y orientación, a intervalos suficientemente cortos. La técnica más simple consiste en la utilización de la odometría a partir de las medidas suministradas por los sensores situados en los ejes de movimiento, típicamente codificadores ópticos. Sin embargo la acumulación del error puede ser muy grande. Se emplean también sistemas de navegación inercial incluyendo giróscopos y acelerómetros, aunque estos sistemas también acumulan error, especialmente en la determinación de la posición empleando los acelerómetros. No obstante, la combinación de las técnicas odométricas con la medida de los ángulos de orientación puede dar buenos resultados en intervalos de tiempo y distancia viajada suficientemente pequeños.

La corrección de la inevitable acumulación de error hace necesario el empleo de otros sensores. Con este fin, en aplicaciones de exteriores, en las que las distancias que recorre el vehículo autónomo son considerables, se emplean sistemas de

posicionamiento global mediante satélites (GPS).

El sistema de percepción de un robot móvil o vehículo autónomo tiene un triple objetivo: permitir una navegación segura, detectando y localizando obstáculos y situaciones peligrosas en general, modelar el entorno construyendo un mapa o representación de dicho entorno (fundamentalmente geométrica), y estimar la posición del vehículo de forma precisa. Asimismo el sistema de percepción de estos robots puede aplicarse no sólo para navegar sino también para aplicaciones tales como el control de un manipulador situado en un el robot.

Para el diseño de estos sistemas de percepción deben tenerse en cuenta diferentes criterios, algunos de los cuales son conflictivos entre sí. De esta forma, es necesario considerar la velocidad del robot, la precisión, el alcance, la posibilidad de interpretación errónea de datos y la propia estructura de la representación del entorno.

En muchas aplicaciones se requiere tener en cuenta diversas condiciones de navegación con requerimientos de percepción diferentes. De esta forma, puede ser necesario estimar de forma muy precisa, aunque relativamente lenta, la posición del robot y a la vez, detectar obstáculos lo suficientemente rápido, aunque no se necesite una gran precisión en su localización.

Existen también arquitecturas en las que el sistema de percepción se encuentra integrado en el controlador de forma que, en entornos estructurados, es posible estimar de forma muy rápida la posición para navegar a alta velocidad.

Asimismo se han aplicado redes neuronales para generar el ángulo de dirección a partir del sistema de percepción.

Conviene mencionar también el interés del empleo de técnicas de procesamiento en paralelo para el tratamiento de imágenes en el guiado autónomo de vehículos.

Con respecto a los sensores específicos, además de las características de precisión, rango, e inmunidad a la variación de condiciones del entorno, es necesario tener en cuenta su robustez ante vibraciones y otros efectos originados por el vehículo y el entorno, su tamaño, consumo, seguridad de funcionamiento y desgaste.

Las cámaras de vídeo tienen la ventaja de su amplia difusión y precio, su carácter pasivo (no se emite energía sobre el entorno) y que no es necesario, en principio, el empleo de dispositivos mecánicos para la captación de la imagen. Las desventajas son los requerimientos computacionales, la sensibilidad a las condiciones de iluminación, y los problemas de calibración y fiabilidad.

La percepción activa mediante láser es un método alternativo que ha cobrado una importante significación en robots móviles. Se utilizan dispositivos mecánicos y ópticos de barrido en el espacio obteniéndose imágenes de distancia y reflectancia a las superficies intersectadas por el haz.

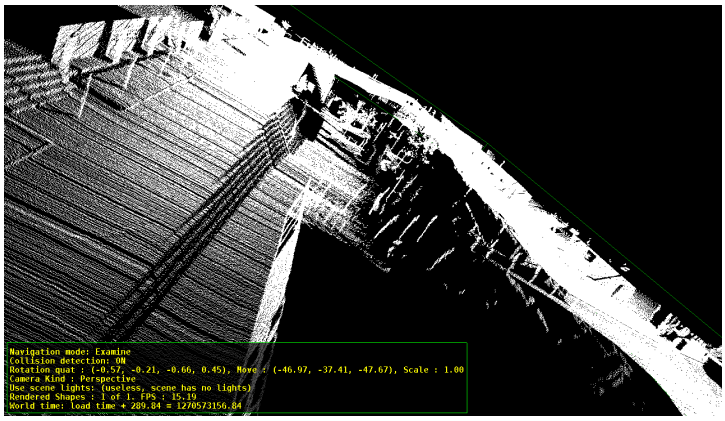


Figura 1.9: Proyecto URUS. Romeo-4R World Laser Generation.

Los sensores de ultrasonido son económicos y simples para la navegación. Se basan en la determinación del denominado tiempo de vuelo de un pulso de sonido (típicamente entre 30KHz y 1 MHz). Sin embargo, la influencia de las condiciones ambientales puede ser significativa, debiendo corregirse mediante una calibración adecuada.

Por otra parte la relación señal/ruido es normalmente muy inferior a la de otros sensores, lo que puede hacer necesario el empleo de múltiples frecuencias y técnicas de filtrado y tratamiento de la incertidumbre de mayor complejidad computacional.

Asimismo, la resolución lateral es mala, existiendo para evitarlo técnicas de enfoque mediante lentes acústicas o transmisores curvos.

1.3. Navegación en entornos urbanos

Los escenarios de aplicación de la robótica han evolucionado en las últimas décadas, desde entornos muy simples y controlados, típicamente entornos industriales, a entornos muy dinámicos en exteriores.

Al mismo tiempo, para afrontar ciertas aplicaciones, la cooperación en grupos de varios robots se ha convertido en una necesidad. Una tendencia en la actualidad es la investigación en sistemas que consideren la colaboración entre robots y

sensores heterogéneos presentes en el entorno para multitud de aplicaciones, como robótica de servicio en entornos urbanos, o monitorización de desastres.

La razón fundamental es que estas aplicaciones involucran entornos dinámicos, con condiciones cambiantes para la percepción, etc. En la mayoría de las ocasiones, un único agente (por ejemplo un robot o una cámara) no permite conseguir la robustez y eficacia necesarias. En estos casos, la cooperación de diferentes agentes (robots, sensores en el entorno) puede ser muy relevante.

Por otro lado, hay un creciente interés en la robótica de servicio por lo que cada vez son más frecuentes aplicaciones de robots en entornos urbanos, para tareas como el guiado y asistencia de personas, transporte de personas y objetos, etc. Como veremos más adelante, en el proyecto URUS se obtuvieron interesantes resultados de robótica cooperativa en este tipo de entornos, utilizando una flota de robots móviles, una red de cámaras fijas, así como una red inalámbrica de sensores.

Todos estos elementos pueden comunicarse entre sí de forma inalámbrica, y forman lo que se llama un sistema de robots en red (Network Robot System, NRS). Dicho sistema ha sido desplegado en un entorno urbano demostrando su utilidad. Por ejemplo, la fusión de la información de los distintos elementos permite un seguimiento más preciso, así como hacer frente a oclusiones, en tareas como el guiado de personas.

Una de las posibles aplicaciones de la robótica móvil en robótica de servicio es el guiado e incluso el transporte de personas y objetos en entornos urbanos.

Por ejemplo, en zonas peatonales o zonas que se convierten en peatonales en las ciudades para mejorar la calidad de vida de las ciudades. En este caso, los robots deben ser capaces de navegar a través de calles al mismo tiempo que las personas y posiblemente otros robots.

Adelantamos aquí que la navegación en entornos urbanos se enfrenta además a serios obstáculos, que no están presentes por ejemplo en un entorno industrializado. En efecto, la arquitectura y disposición de los distintos elementos en una zona urbana dista mucho de ser un entorno sencillo de modelar como una nave industrial: bordillos, aceras, tipos de adoquines, peatones, otros vehículos convencionales, rampas, escaleras, etc., situaciones que todas ellas deben ser tenidas en cuenta, desarrollando estrategias para que el robot pueda solventarlas con éxito.

1.4. Interacción humana con el robot

Según [12], podemos definir al usuario como la persona que interacciona con un sistema informático, así pues definimos interacción como todos los intercam-

bios que suceden entre el usuario y el sistema informático, es decir, entre la persona y el ordenador.

Según [17] se define la interacción persona ordenador (IPO) como la disciplina relacionada con el diseño, implementación y evaluación de sistemas informáticos interactivos para uso de seres humanos y con el estudio de los fenómenos más importantes con los que están relacionados.

En inglés la nomenclatura cambia ligeramente, llamando HCI, Human Computer Interaction, y que puede extenderse en nuestro ámbito como HMI Human Machine Interaction, dado que la frontera entre ordenador y robot es cada día más difusa. En adelante en el texto no se hará distinción alguna entre cualquiera de éstos términos, teniendo además en cuenta que hoy en día el concepto de usuario ha evolucionado mucho más allá de la imagen de una persona sentada delante de un terminal.

Los objetivos básicos de estudio de la HMI se basan en conseguir desarrollar o mejorar la seguridad, utilidad, efectividad, eficiencia y usabilidad de cualquier tipo de sistemas que incluyan ordenadores o computación.

Para hacer sistemas usables es preciso:

- Comprender los factores (psicológicos, ergonómicos, organizativos y sociales) que determinan cómo la gente trabaja y hace uso de los sistemas informáticos.
- Desarrollar herramientas y técnicas para ayudar a los diseñadores de sistemas interactivos.
- Conseguir una interacción eficiente, efectiva y segura.

En general, un buen principio de diseño se sustentaría en considerar que los usuarios no han de cambiar radicalmente su manera de ser, sino que los sistemas han de ser diseñados para satisfacer los requisitos del usuario.

La experiencia nos demuestra que la interfaz de un sistema es una parte muy importante del éxito o fracaso de una aplicación, pudiendo facilitar el calado y distribución entre los usuarios como se recoge en la figura 1.10 o bien, saturando al usuario ante una sobrecarga de información, como en la figura 1.11.

- La interfaz constituye entre el 47 % y el 60 % de las líneas de código [18].
- Un 48 % del código de la aplicación está dedicado al desarrollo de la interfaz [22].

Figura 1.10: Interfaz de uso del Buscador Google.

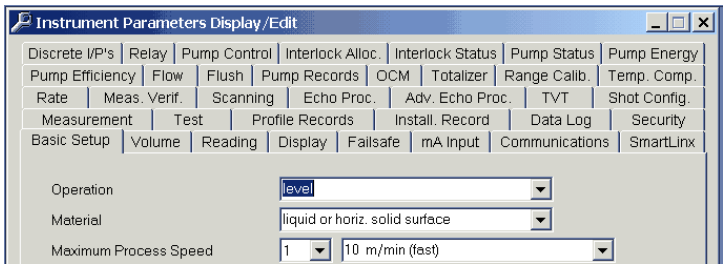


Figura 1.11: Interfaz de uso Herramienta Dolphin Plus.

Otro factor que se debe tener en consideración es que los sistemas informáticos son cada vez más utilizados por gente sin formación específica en este campo.

La Real Academia Española define, una interfaz, en el ámbito de la informática como la conexión física y funcional entre dos aparatos o sistemas independientes. Las interfaces aparecen pues entre diferentes tipos de entidades, físicas o lógicas, dentro de los sistemas informáticos.

En IPO las entidades son la persona y el ordenador. En la vida cotidiana tenemos muchos ejemplos de interfaces (el volante de un coche, la selección de programa de una lavadora, el pomo de una puerta, ...).

En el desarrollo de la IPO se dan cita, no sólo muchas ramas de la ciencia y la ingeniería, sino también otras ramas que están más relacionadas con los estudios de disciplinas sociales y psicológicas.

Para que un sistema interactivo cumpla sus objetivos tiene que ser usable y accesible a la mayor parte de la población humana. Definimos la usabilidad como la medida en la que un producto se puede usar por determinados usuarios para conseguir unos objetivos específicos con efectividad, eficiencia y satisfacción en un contexto de uso dado.

Por tanto podemos entender un software usable como aquel que es fácil de aprender (permite realizar las tareas rápidamente y sin errores) y fácil de utilizar (realiza la tarea para la que se usa).

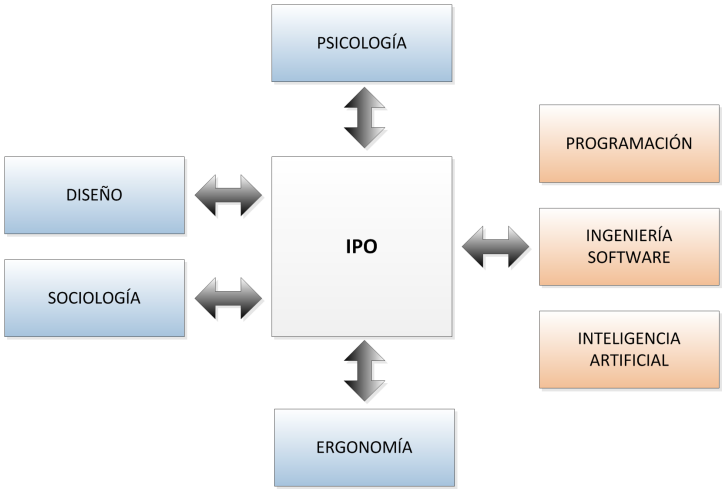


Figura 1.12: Disciplinas relacionadas con la IPO.

Una aplicación usable es la que permite al usuario centrarse en su tarea, no en la aplicación. Podríamos enumerar los principios generales de la usabilidad de una interfaz o aplicación en:

- Facilidad de aprendizaje.
- Flexibilidad.
- Consistencia.
- Robustez.
- Recuperabilidad.
- Tiempo de respuesta.
- Adecuación de las tareas.
- Disminución de la carga cognitiva.

Por tanto, el diseño de sistemas interactivos debe realizarse pensando siempre en el usuario final al que va destinado.

Para lograr que personas no relacionadas con la materia puedan interaccionar y usar las redes robóticas, se hace necesario dotar a los robots de capacidades de interacción con los seres humanos.

Los objetivos marcados en el proyecto URUS fueron definir las modalidades de interacción humano robot, la creación de una interfaz hombre robot que pudiera

ser desplegada en cualquier plataforma del proyecto URUS y el diseño de algoritmos para el reconocimiento de gestos y posiciones en las áreas urbanas mediante la red de cámaras.

Figura 1.13: Proyecto URUS. Interfaz de Usuario Robot Tibi.

Figura 1.14: Proyecto URUS. Identificación de persona realizando una petición al sistema.

1.5. Proyecto URUS

Las ciudades más antiguas de Europa se están convirtiendo en lugares molestos para vivir, debido a los problemas de ruido, polución, falta de calidad en infraestructuras y seguridad. Los ayuntamientos de las mismas comienzan a ser conscientes de estos problemas y empiezan a estudiar soluciones, como por ejemplo, reducir las áreas de libre circulación de vehículos e implementando nuevos modos de transporte, seguridad y asistencia a los ciudadanos.

URUS es un proyecto europeo que se ha dedicado a analizar y probar la viabilidad de incorporar una red de robots, formada por robots, sensores inteligentes, dispositivos portátiles inteligentes y comunicaciones, para mejorar la calidad de vida en las áreas urbanas. El proyecto URUS se ha centrado en el desarrollo de elementos tecnológicos requeridos para el trabajo de robots en áreas urbanas y en la realización de experimentos que demuestren la viabilidad de dichos elementos en tareas habituales urbanas, como el guiado, asistencia y transporte de personas y bienes.

El proyecto URUS también ha analizado los requerimientos y necesidades para el despliegue de una red de robots en entornos urbanos, analizando también los aspectos legales concernientes a la seguridad y privacidad de las personas, que pueden verse afectados por el despliegue de dichas redes de robots y sensores.

Figura 1.15: Proyecto URUS. Tracking de personas mediante red de cámaras IP.

El objetivo general del proyecto URUS es el desarrollo de nuevas formas de cooperación mediante redes de robots y seres humanos y/o el entorno en áreas urbanas, para poder realizar de manera eficiente distintos tipos de tareas habituales en nuestras ciudades, como por ejemplo, coger un taxi.

Más en concreto, el objetivo científico y tecnológico es el desarrollo de una arquitectura de red de robots adaptable, la cual integre funcionalidades como: localización y navegación cooperativa de robots, percepción cooperativa del entorno, generación cooperativa de mapas, interacción humana con los robots, asignación de tareas, comunicaciones inalámbricas mediante dispositivos móviles, redes de sensores y otros tipos de robots heterogéneos.

La arquitectura de red robótica URUS ha sido probada en un escenario urbano abierto de 10.000 m2, en el Barcelona Robot Lab (BRL), localizado en Barcelona, donde 8 diferentes robots, dos humanoides, dos vehículos robots y cuatro plataformas robóticas, fueron usadas en estos experimentos.

Los experimentos consistieron en guiado de personas, transporte de personas y materiales, y vaciado de una zona especificada.

En el apartado 6.3 de esta memoria recogeremos algunos de los resultados más interesantes que fueron obtenidos en los experimentos finales del proyecto URUS.

1.6. Objetivos del proyecto fin de carrera

Una vez introducido el contexto de trabajo, no está de más recalcar cuáles son los objetivos finales tangibles del presente Proyecto Final de Carrera.

Distinguimos como objetivos del proyecto el diseño y desarrollo de dos aplicaciones de apoyo o soporte para los investigadores miembros del Grupo de Robóti-

Figura 1.16: Equipo del Proyecto URUS.

ca, Visión y Control que han trabajado en el proyecto URUS. Las aplicaciones en cuestión son:

- **Interfaz Gráfica Romeo HMI.** Aplicación en lenguaje C++ que implementa una interfaz gráfica basada en diversas librerías de desarrollo y cuya utilidad principal es mostrar la información principal de los sensores del robot Romeo-4R en tiempo real, así como prestar un servicio para la introducción de rutas. A la descripción completa de la interfaz dedicamos el capítulo 4 de esta memoria.

- **Sistema de Análisis Post-Misión (Logplayer).** Aplicación en lenguaje C++ e interfaz en línea de comandos que sirve para reproducir en simulación los experimentos realizados, utilizando para ello la información recogida en los logs de los sensores y módulos del robot. El capítulo 5 está dedicado a una descripción completa de esta aplicación.

Para el desarrollo de estas aplicaciones (así como para otros módulos del robot Romeo-4R) se han utilizado una serie de librerías de desarrollo a las que se dedica de manera conjunta el capítulo 3 de la memoria.

Capítulo 2

Vehículo Autónomo Romeo-4R[1]

2.1. Introducción

Romeo-4R es un robot móvil que viene siendo desarrollado en el Departamento de Ingeniería de Sistemas y Automática de la Escuela Superior de Ingenieros de Sevilla desde la década de los 90.

Su nombre deriva de *Robot Móvil para Exteriores*, y es el resultado de la adaptación de un vehículo eléctrico convencional de cuatro ruedas (4R). Este tipo de vehículos se utilizaron durante la Exposición Universal de Sevilla de 1992 y fue adquirido por el Departamento de Ingeniería de Sistemas y Automática tras la finalización de la misma. Tal y como puede verse en la figura 2.1, es el equivalente a un antiguo carrito de golf.

La principal función de Romeo-4R es la experimentación de distintas técnicas de navegación en exteriores, mediante la utilización de los sensores y actuadores de los que dispone. A día de hoy, aún se puede admirar en los laboratorios a su predecesor, Romeo-3R. Pero en adelante y para el resto de este texto, cuando se hable de «Romeo», se deberá entender como una referencia a Romeo-4R.

En lo relativo a los sistemas de referencia que se utilizarán a lo largo del presente proyecto, hay que distinguir entre el sistema de coordenadas global o del mundo (WCS) y el sistema de coordenadas local solidario al robot (RCS). La utilización de al menos estos dos sistemas de referencias es muy habitual en los sistemas robóticos móviles.

Para el caso de Romeo-4R, con origen en la proyección del punto medio del eje trasero en el plano del suelo, el sistema RCS se compone de forma que el eje y se dirige hacia adelante, el eje z hacia el cielo, y el eje x de manera que resulte un sistema dextrógiro, como podemos ver en la figura 2.2.

[1] Este capítulo está basado en el trabajo [24]

Figura 2.1: Robot Romeo-4R.

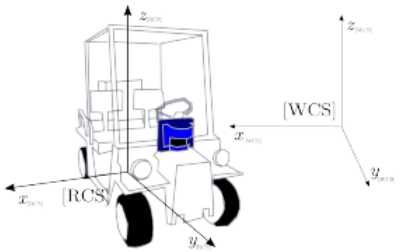


Figura 2.2: Sistemas de referencia global y local.

Aunque este tipo de definición es la mas completa, suele utilizarse poco en la robótica terrestre, donde la altura del robot viene determinada por la altura del suelo que pisa. Por ello es más habitual en estos casos utilizar un sistema de referencia plano, proyección del anterior, ver figura 2.3. En él existe un punto del mapa que se considera origen del sistema de coordenadas global, y unos ejes definidos. Como sólido rígido en movimiento en el plano, el robot dispone de tres grados de libertad que quedan definidos por la posición en el sistema global del origen de su sistema local (x, y) y por la orientación de sus ejes respecto a los globales (θ).

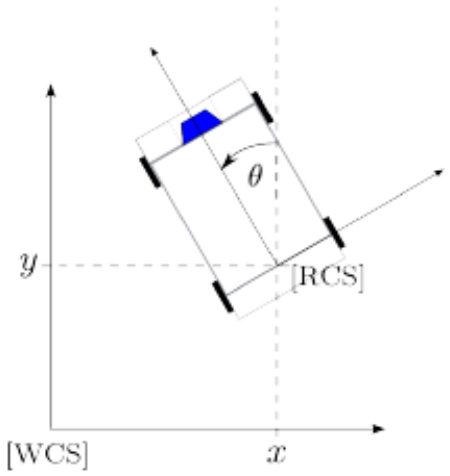


Figura 2.3: Particularización de los sistemas de referencia para el caso plano.

2.2. Equipamiento Hardware

Con unas dimensiones que rondan los 2,80m x 1,40m x 2,10m y un peso aproximado de 700 kg, podría decirse que el soporte físico de Romeo es un vehículo de cuatro ruedas convencional.

El sistema de tracción dispone de dos ruedas paralelas motrices colocadas sobre el eje transversal trasero del vehículo. Estas ruedas son movidas por un motor de corriente continua de 36 V de gran potencia (2 CV) que le permite alcanzar velocidades de hasta 3 m/s. El sistema de dirección está formado por dos ruedas directrices unidas mediante un eje rígido, utilizándose otro motor de corriente continua para controlar la dirección.

La rueda delantera interna gira un ángulo ligeramente superior a la externa para evitar el deslizamiento. Las prolongaciones de los ejes de las dos ruedas delanteras intersectan en un punto sobre la prolongación del eje de las ruedas traseras. El lugar de los puntos de contacto de cada rueda delantera con el suelo forma dos circunferencias aproximadamente concéntricas, resultando válido el modelo cinemático de la bicicleta, en la descripción de su movimiento en coordenadas globales. En estas ecuaciones x, y, θ son la posición y la orientación del vehículo respectivamente, ν es la velocidad longitudinal y γ la curvatura, o lo que es lo mismo, la inversa del radio de giro que se describirá para un ángulo concreto en las ruedas de dirección.

$$\dot{x} = -\nu \cdot \sin\theta \tag{2.1}$$

$$\dot{y} = \nu \cdot \cos\theta \tag{2.2}$$

$$\theta = \nu \cdot \gamma \tag{2.3}$$

Diseñado para que sea posible su conducción tanto manual como automática, durante el funcionamiento manual el conductor puede mover libremente el volante, acelerar y frenar, mientras que en el modo automático se bloquea el volante y el acelerador se anula.

Como controladores, monta un par de PCs industriales, uno dedicado al control y otro a la visión. Para los trabajos aquí desarrollados se ha empleado el de control, que cuenta con un procesador Intel Pentium 4 a 2,40GHz sobre el que corre GNU/Linux Debian 2.6.18. Este PC dispone de una tarjeta de control de motores DCX-PC 100, de la marca Precision MicroControl Corporation. Como indica su nombre se utiliza para el control de los motores de tracción y dirección, es decir, sirve de interfaz entre el ordenador y el propio hardware de control de motores. Consta de dos módulos MC-200 que no son más que servocontroladores de motores de corriente continua. Hay un módulo para cada motor donde cada uno utiliza un control PID programable con realimentación directa de la lectura de cada codificador. Además de estos codificadores (que permiten generar medidas de odometría) Romeo dispone de una serie de sensores y elementos que le permiten realizar medidas del entorno y aumentar su utilidad, como puede verse en las figuras 2.4 y 2.5, y que pasamos a describir a continuación.

- **Giróscopo**

  El modelo utilizado, un Autogiro Navigator Plus de KVH Industries, está construido como un interferómetro de fibra óptica de un solo eje y resulta muy adecuado para sistemas de navegación terrestre. Proporciona medidas de la velocidad angular de giro respecto al eje z del robot, perpendicular al plano del suelo. La integración de dichas medidas en el tiempo permite estimar el giro del vehículo, y calcular as la orientación del mismo. El giróscopo se comunica a través de puerto serie a una velocidad de 9600 Bd.

- **IMU**

  La unidad de medida inercial a bordo de Romeo es en realidad un compás y magnetómetro modelo EZ-COMPASS-3A de Advanced Orientation Systems Inc. Comunica su orientación 3D a través de puerto serie con una tasa de actualización de 10Hz y una resolución de 0,08°.

Figura 2.4: Robot Romeo-4R. Sensores (I).

Figura 2.5: Robot Romeo-4R. Sensores (II).

- **GPS**

  Un receptor Novatel OEM conectado a través de un puerto serie permite la estimación de la posición global del robot en aquellos lugares con cobertura GPS.

  Para conseguir una mayor precisión en las medidas puede utilizarse en modo diferencial (DGPS), en cuyo caso habrá dos dispositivos GPS, uno montado en el robot que actuará de estación remota y otro montado en una posición conocida actuando de estación base. Ambos dispositivos se comunicarán entre sí mediante radiomódems. En este modo de trabajo el error en la posición es de tan sólo unos pocos centímetros.

- **Láser SICK**

  Se trata de un escáner láser bidimensional de medida de distancias de no contacto basado en el tiempo de vuelo de pulsos láser. El modelo es LMS 220-30106 (versión para exteriores) de la marca SICK Optic Electronic. Es un láser bidimensional que mide distancias en el plano horizontal con ángulos de exploración (100°ó 180°) y resolución (0,25°-0,5°ó 1°) programables y capaz de medir distancias de hasta 80 m. Hay que destacar que el láser LMS 220 no necesita que los objetos tengan propiedades reflectoras especiales para detectarlos. No obstante, cuanto más reflectante sea el objeto mejor será la detección. Por ejemplo, los objetos de color blanco serán mejor detectados que los de color oscuro.

- **Láseres HOKUYO**

  Romeo cuenta además con un par de láseres modelo URG-04LX de HOKUYO. De dimensiones muy reducidas (50 x 50 x 50 mm) y gran ángulo de exploración de (240°con resolución de 0,36°), este sensor lanza a 10 Hz medidas entre 20 - 4000 mm con una precisión de 10 mm. A pesar de que están especificados para aplicaciones en interiores, el comportamiento en exteriores para el montaje actual es más que aceptable.

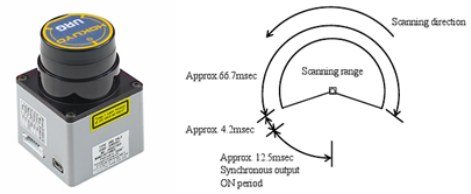


Figura 2.6: Láser Hokuyo URG-04LX.

Finalmente, se añadió un sensor láser más a Romeo, el modelo UTM-30LX de HOKUYO, instalado en el pan & tilt superior, que conocida la inclina-

ción del mismo y orientado hacia el suelo, permite la creación de mapas de elevación del entorno.

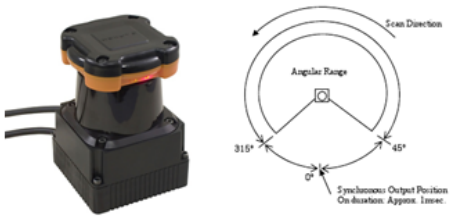


Figura 2.7: Láser Hokuyo UTM-30LX.

- **ELO Touchscreen Serie ET1515L**

  Romeo-4R cuenta con una pantalla táctil de la serie ET1515L del fabricante ELO que facilita la introducción de comandos y el manejo de la interfaz del vehículo.

- **Cámara Video DFK 21BF04**

  Romeo-4R cuenta también con una videocámara FireWire del fabricante The Imaging Source, modelo DFK 21BF04 que se utiliza para labores de reconocimiento facial y de apoyo a la navegación y localización.

Figura 2.8: Videocámara DFK 21BF04.

2.3. Arquitectura Software

La incorporación en Romeo de las aplicaciones objeto del presente trabajo ha sido posible gracias a la arquitectura software existente. Totalmente modular, en ella no se hace uso del esquema clásico cliente-servidor, sino que su modo de funcionamiento está inspirado en las redes peer-to-peer o redes de pares, donde cada uno de los nodos de la red no es cliente ni servidor fijo, sino que interactúan como iguales entre sí, algo que se adapta a las necesidades actuales de los sistemas robóticos avanzados.

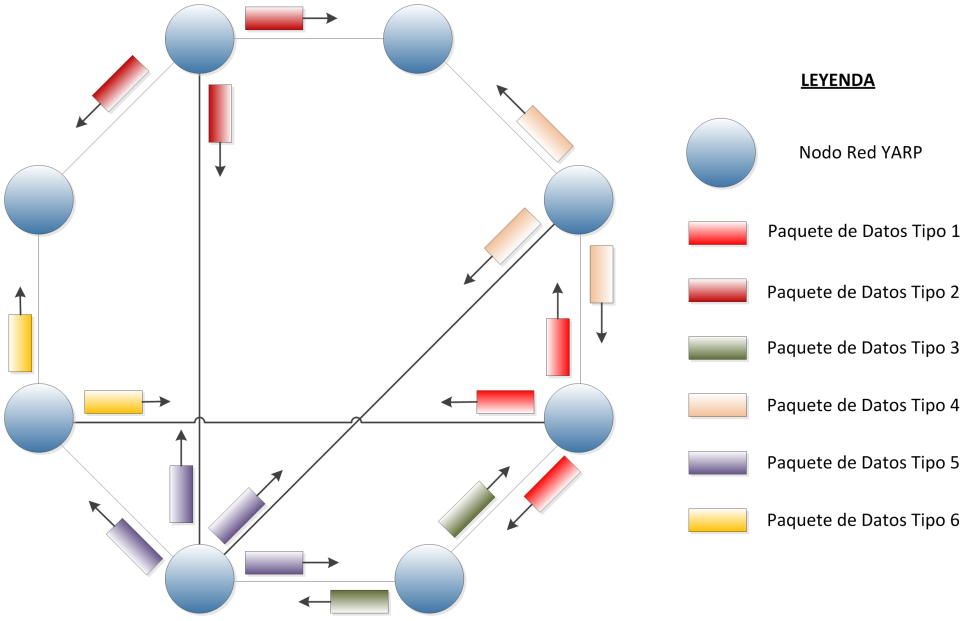


Figura 2.9: Red YARP - Modo peer to peer.

Así, cada uno de los múltiples módulos ofrece distintos servicios, entendiendo por servicio un flujo de datos de un tipo concreto, resultantes de sus distintas funcionalidades. Como resultado, se elimina cualquier tipo de jerarquía explícita, y la adición de un nuevo módulo se reduce a la resolución de nuevas relaciones de servicio, es decir, de tipado de datos.

Cada uno de los módulos, que corresponde a un proceso independiente desarrollado en C++, puede englobarse dentro de un nivel de abstracción distinto, de modo que cambiar el hardware sólo supondrá una modificación de los módulos de bajo nivel, haciendo muy portable tanto la arquitectura como las funcionalidades que implementa. Para la comunicación entre módulos, se ha hecho uso de la librería multiplataforma y de código abierto YARP [10], de la que se ofrece una descripción más completa en el Capítulo 3 de este texto.

Adelantaremos aquí que esta librería ofrece una abstracción de las comunicaciones entre procesos basada en el concepto de puerto. Cada puerto puede conectarse con otros para establecer un flujo de datos, cuyo tipado puede venir definido fácilmente por el usuario programador de C++.

Concretamente, en el proyecto URUS se ha diseñado una jerarquía de clases para los distintos módulos, partiendo de una clase base de tipo comunicaciones, cuya declaración recogemos a continuación.

```
class Comms: public yarp::os::Portable {

public:
        //! Friends:
        friend std::ostream& operator << (std::ostream& os,
            Comms& comms);
        friend std::istream& operator >> (std::istream& is,
            Comms& comms);
        friend std::ostream& operator << (std::ostream& os,
            Comms* comms);
        friend std::istream& operator >> (std::istream& is,
            Comms* comms);

        //! Data...
        double m_sec;   // Time from 1st january 1970 [s]
        double m_nsec;  // To increase time resolution [ns]
        int status;     // Status byte (system, error, ...)

        // Default constructor
        Comms(): m_sec(0.0), m_nsec(0.0), status(0) {}

        static void connect(std::string writer, std::string
            reader,  const char* defaultLocal = "tcp");
        virtual void print(PrintType); // Self-printing
           function (for debugging)
        virtual std::string logHeader();// Log-related
           function:

        // Timing functions:

        double tic();   // generates timestamp and keeps it
            in public data members
                        // m_sec, m_nsec; returns seconds
                           passed since 1970.
        double toc();   // returns seconds passed since
           last tic() function-call.

protected:

        // Log-related functions:
        virtual void logStream(std::ostream& os);
        virtual void fromStream(std::istream& is);

        // struct to get time marks
        struct timespec epochTime;
};
```

Tabla 2.1: ***Proyecto URUS – Comms.h.***

En la figura 2.10 puede verse un esquema de conexión entre los distintos módulos correspondientes al robot Romeo-4R.

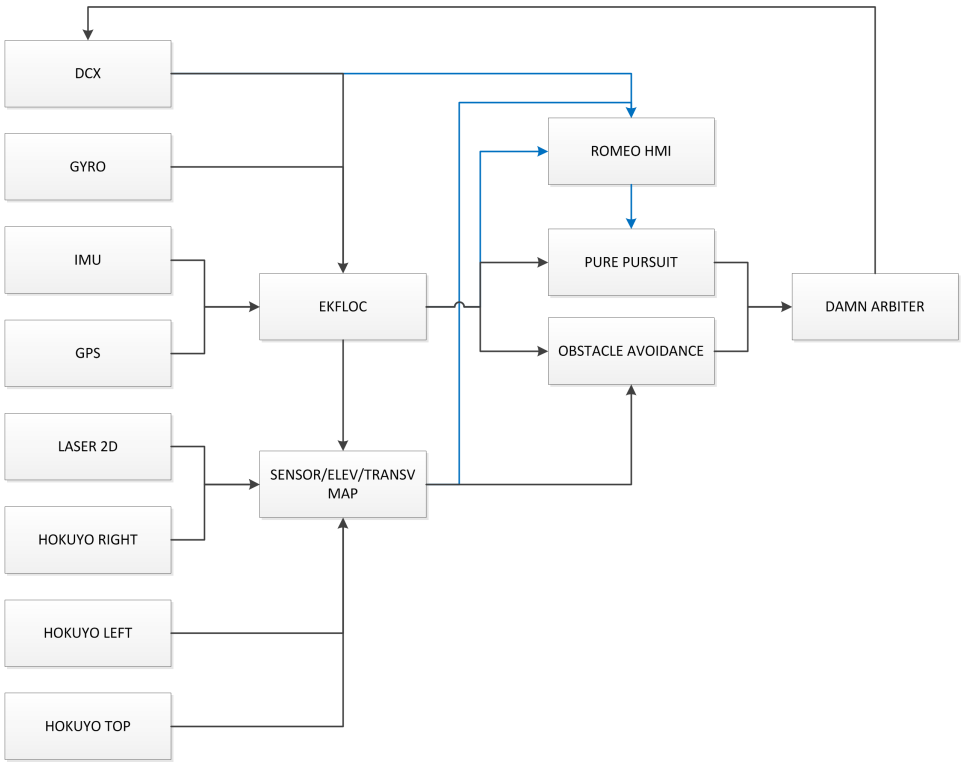


Figura 2.10: Conexiones módulos software Romeo-4R y puertos YARP.

En este esquema no se debe confundir cliente con *receptor de información* o servidor con *emisor de información*. En este sentido, un módulo llamado árbitro, que como se verá es el único que manda consignas al módulo de control de bajo nivel, ofrece un servicio que se podría definir como *consideraré los votos que reciba para la elección del comando que ejecutará finalmente el robot*. Para ello, los clientes deberán establecer la conexión, entendiéndose así que solicitan el servicio de este módulo. En este ejemplo queda claro que el servidor puede perfectamente recibir información de los clientes y prestar un servicio al mismo tiempo.

Otro ejemplo a considerar es el módulo de generación de mapas, que ofrece como servicio el mapa sensorial más actual y a su vez es cliente de los módulos de láser y del módulo de localización. Por otro lado, el módulo de evitación de obstáculos es cliente del generador de mapas, del módulo de localización y del árbitro; el servicio que ofrece se definirá como *intentará llegar al objetivo que reciba, evitando al mismo tiempo los obstáculos que haya en el camino*. Por ultimo, el módulo de tracking láser, que ofrece como servicio una lista de objetos percibidos, con sus posiciones, velocidades y firmas, podrá ser cliente tanto de módulos

láser como del generador de mapas, a la vez que podrá pasarle puntos de destino al módulo de evitación de obstáculos, resultando entonces una persecución.

La complejidad subyacente al tratamiento de imágenes tanto en la representación de mapas como en el tracking láser, se ha visto aliviada en parte gracias a la utilización de funciones incluidas dentro de otra librería de libre distribución, OpenCV (Open Source Computer Vision) [6]. Se trata de una librería de funciones programables en C/C++ dirigidas y optimizadas principalmente para la visión artificial en tiempo real, y que es independiente del hardware, sistema operativo o gestor de ventanas utilizado. En el capítulo 3 se hace una descripción más completa de esta librería.

A continuación se realizará una descripción más detallada de cada uno de los módulos actualmente implementados en Romeo. Mientras que los módulos de bajo nivel se encargan principalmente de interaccionar con los sensores y actuadores específicos de cada robot, aquellos que desarrollan funciones de más alto nivel resultan más independientes del hardware, con lo que resultan totalmente portables a otros robots con la misma arquitectura, como es el caso de otro de los robots del laboratorio llamado Monster.

- **Módulo DCX**

  Módulo que se comunica con la tarjeta de control DCX, sirve la velocidad lineal y la curvatura función del ángulo que forman las ruedas delanteras, ambas variables medidas mediante encoders, además de recibir las referencias para los motores de tracción y de dirección, que se pasan al control de bajo nivel que implementa un par de PID's.

- **Módulo GYRO**

  Interfaz con el giróscopo, este módulo, como el resto de los relacionados con sensores, se limita a empaquetar cada nuevo dato y servirlo por un puerto YARP para que cualquier otro módulo (cliente) pueda tener acceso a él. En el caso del giróscopo, este dato se refiere al ángulo y velocidad de giro en torno al eje z de Romeo.

- **Módulo IMU**

  Módulo que lee a través del puerto serie los datos provenientes de la Unidad de Medida Inercial, empaqueta los datos relativos a los ángulos de Tait-Bryan (roll, pitch, yaw) y los sirve al resto de módulos que los necesiten.

- **Módulo GPS**

  Lee a través del puerto serie la trama GGA del dispositivo GPS y a partir de ella extrae y empaqueta la posición global del robot (latitud, longitud y altura o UTM), la desviación estándar de dicha posición y otras variables de

interés como el número de satélites disponibles junto una marca de tiempo local.

- **Módulos Laser 2D**

  Servidores de la información generada por el láser SICK-LMS220, o por los URG-04LX de HOKUYO, resultan una fuente de información que por sus características resulta especialmente adecuada para la detección de obstáculos y generación de mapas. Esta información se compone básicamente de los campos relativos al número total de medidas y distancias percibidas para cada ángulo del barrido, así como del rango máximo o desviación estándar de la medida. En todo caso, la interfaz resulta común a ambos dispositivos, ya que toda la información que se puede extraer del sensor se sirve dentro de la estructura de datos correspondiente.

- **Módulo EKFLOC**

  Módulo encargado de localizar el vehículo en un sistema de coordenadas global, ya sea definido a priori como en el caso de UTM, ya sea definido por la postura inicial del robot, para ello toma los datos curvatura y velocidad (odometría), del giróscopo y del GPS y los integra mediante un Filtro de Kalman Extendido (EKF), dando como resultado una estimación de la posición y orientación del vehículo.

- **Módulo SENSOR MAP**

  La necesidad de un mapa local que integrara la información sensorial, por ejemplo para la identificación de la situación actual dentro del paradigma situación-acción que emplea el módulo de evitación de obstáculos, fue la motivación para la creación de este nuevo módulo. Si bien puede no considerarse estrictamente necesario, ya que la información sensorial está disponible en los módulos correspondientes a los sensores, en su funcionalidad cumple misiones importantes de cara a la flexibilidad de la arquitectura y a la precisión de la información:

  - *Fusión sensorial:* la información relativa a obstáculos de todos los sensores se condensa aquí ya no es necesario ser cliente de cada uno de los sensores. Esto permite entre otras cosas añadir o eliminar sensores sin que ningún otro módulo tenga que tener conocimiento de ello, además de mejorar la fiabilidad mediante tratamiento estadístico de los datos.
  - *Memoria:* sin la cual sólo se podrá disponer de la información actual. Ajustable por parámetro, especialmente para la aplicación en la evitación de obstáculos es necesaria una solución de compromiso: no conviene mantener un recuerdo perfecto de todos los obstáculos, que no tienen porqué ser fijos.

En la figura 2.11 podemos ver un mapa generado por este módulo en un experimento realizado en los laboratorios de la Escuela Superior de Ingenieros de Sevilla.

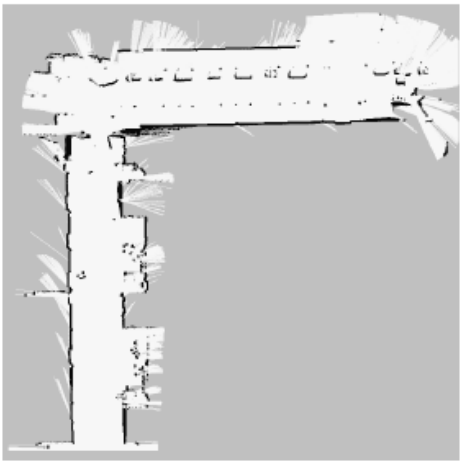


Figura 2.11: Mapa generado sensorialmente.

- **Módulo OBSTACLE AVOIDANCE**

  La funcionalidad de este módulo se basa en algoritmos clásicos de evitación de obstáculos para el cálculo de una acción que resulte en un movimiento dirigido a un objetivo y que al mismo tiempo evita obstáculos. Para ello se vale de una serie de diagramas, abstracción de toda la información relativa al problema. Inspirado por el trabajo de Minguez y Montano [21], no se trata de un planificador, sino de un algoritmo de navegación reactiva con ventajas respecto a la utilización de campos potenciales o incluso a la replanificación de trayectorias.

- **Módulo PURE PURSUIT**

  Módulo encargado de realizar un seguimiento estricto de trayectorias, tomará como entrada la estimación de la posición del robot para ejecutar el algoritmo de pure pursuit [14], un clásico en la navegación de robots móviles con direccionamiento Ackerman. Junto al módulo anterior, será cliente del módulo árbitro que veremos a continuación, que decide la acción definitiva a enviar al control de bajo nivel.

- **Módulo DAMN ARBITER**

  Este módulo estará encargado de arbitrar un numero variable de referencias en velocidad y curvatura, provenientes de diferentes módulos, de manera que las referencias finales que se envían al control de bajo nivel sean una solución de compromiso. Cada entrada tiene además un peso que indica al módulo su importancia relativa con respecto a las demás. De esta forma, la evitación

de un obstáculo próximo se antepondrá al seguimiento de una trayectoria previamente establecida, mientras que en ausencia de obstáculos se seguirá fielmente la trayectoria, tal y como se expone en [25].

- **Módulo LASER TRACKER**

  Diseñado para el tracking de objetos basado en percepción láser, parte de [15] para desarrollar un algoritmo de predicción y matching de firmas, formas percibidas, para en cada instante conocer la posición e identidad de cada objeto dentro del rango del sensor.

  Aunque se podrá usar también algoritmos de visión sobre mapas generados sensorialmente para la segmentación y extracción de los objetos, los retrasos variables hacen más adecuada en esta implementación concreta la utilización directa de los datos de sensores láser.

- **Módulo ELEVATION MAP**

  Módulo que integrando la información procedente del láser instalado en el techo del vehículo permite obtener en tiempo real un mapa de elevación del entorno y derivarlo en un mapa de transversabilidad del mismo, tal y como reflejan las imágenes de la figura 2.12.

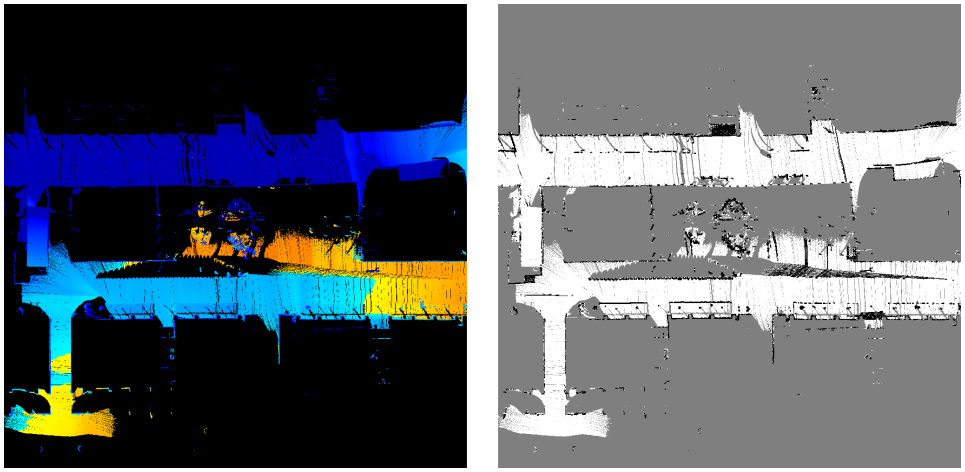


Figura 2.12: Proyecto URUS. Mapas de Elevación y Transversabilidad.

- **Módulo PORT LOGGER**

  Para la obtención de datos que permitan el posterior análisis y reproducción de los experimentos, se dispone también de este módulo, que se encarga de monitorizar y archivar en disco todo el flujo de datos que corre por la red YARP del robot.

- **Módulo LOG PLAYER**

Módulo que se utiliza para reproducir en simulación experimentos realizados previamente y cuyos datos obtenidos fueron logueados por el módulo PORT LOGGER.

Por tanto, no debe entenderse este módulo como un módulo más del sistema del robot. Se debe entender como un módulo o herramienta de apoyo al trabajo y al desarrollo del sistema completo, pues permite probar o refinar algoritmos de navegación o de filtrado de datos de sensores, sin tener que realizar un nuevo experimento real, algo que suele llevar mucho tiempo debido a la logística requerida para llevar a Romeo a un lugar donde poder realizar las pruebas reales.

El capítulo 5 de la presente memoria se dedica por completo a esta aplicación.

- **Módulo ROMEO HMI (Human Machine Interface)**

  La interfaz gráfica para el control y supervisión del funcionamiento del robot, objetivo principal del presente proyecto, se integra como un módulo más del mismo sistema, leyendo y escribiendo en diversos puertos YARP, permitiendo interactuar así con el robot. Esta aplicación fue diseñada pensando en mostrar información útil a los desarrolladores en tiempo real durante la realización de los distintos experimentos con el robot y no es en ningún caso una interfaz destinada a un posible usuario final.

Figura 2.13: Splashcreen Interfaz Romeo HMI.

El capítulo 4 de la presente memoria se dedica por completo a esta aplicación.

Capítulo 3

Librerías Software

3.1. Introducción

En la introducción de este texto ya se ha indicado que la robótica aglutina conocimientos y aplicaciones de ramas muy diversas de la ciencia y la tecnología. En este sentido, la investigación y el desarrollo de nuevos robots o técnicas de navegación se hace una tarea excesivamente compleja.

Es por ello que en en los proyectos de robótica se utilizan de forma genérica o habitual multitud de desarrollos disponibles (ya sean totalmente libres o soluciones privativas, hardware o software) que facilitan la consecución de los objetivos.

Realizamos en esta introducción un breve repaso a algunas de las librerías y middleware más utilizados en la programación de sistemas robóticos, estableciendo una clasificación según la funcionalidad de los mismos según se utilicen en comunicaciones, interfaces gráficas o visión por computador y justificando la elección concreta de cada librería utilizada en el proyecto.

3.1.1. Comunicaciones

La percepción cooperativa requiere capas de comunicación adecuados para compartir información. Algunos de los desarrollos más conocidos son:

- **MIRO [5].** Es un framework distribuido orientado a objetos para el control de robots móviles basado en la tecnología CORBA (Common Object Request Broker Architecture). Los componentes básicos de MIRO se han desarrollado en C++ para Linux. Pero debido a la independencia lenguaje de programación de componentes del sistema CORBA se puede escribir en cualquier lenguaje de programación y en cualquier plataforma que proporciona implementaciones CORBA.

- **PLAYER/STAGE [7].** Se ha convertido en un estándar de facto en la comunidad opensource robótica. El servidor Player se utiliza como capa de abstracción de hardware que se puede utilizar con muchos robots reales o simulados, e incluye servicios de comunicación a través de sockets TCP que permiten la intercomunicación.

- **YARP [10][20].** Yet Another Robot Platform es otro middleware de amplia aceptación para el desarrollo de sistemas robóticos. Escrito y mantenido en lenguaje C++ es un sistema multiplataforma, soporta computación distribuida y está orientado principalmente para el control en robots de manera eficiente.

En el proyecto URUS se optó por la utilización de YARP dada su facilidad para su integración con el código escrito en C++, su modularidad, fiabilidad probada en anteriores proyectos del GRVC así como por supuesto ser una herramienta totalmente libre. El siguiente apartado de la memoria se realizará una descripción exhaustiva de esta librería.

3.1.2. Interfaces gráficas

Las interfaces gráficas de usuario aparecen como una evolución natural de los intérpretes de comandos utilizados para interactuar con los primigenios sistemas operativos, a pesar de no haber sido capaces de eliminar completamente los intérpretes de comandos en los actuales sistemas, los cuales siguen siendo muy demandados y teniendo un peso específico realmente importante en según qué contextos.

A día de hoy, podemos distinguir varias líneas claramente establecidas, con sus cuotas de mercado correspondientes en el complejo mundo de los entornos gráficos. Actualmente podemos distinguir los entornos gráficos (típicamente llamados de escritorio) de los sistemas Microsoft Windows, el sistema X-Window utilizado en sistemas operativos GNU/Linux, e incluso el Aqua utilizado en Mac OS X de Apple, entre otros menos generalistas.

Tal y como se ha comentado en el capítulo 2, el robot Romeo-4R está soportado por un sistema operativo GNU/Linux con un entorno gráfico por tanto basado en X Window, concretamente, el entorno de escritorio KDE y por lo tanto, para el desarrollo de la interfaz gráfica de usuario las opciones se reducían inicialmente a las librerías para sistemas X Window.

Sin duda las librerías más populares para X Window System son Qt y GTK.

- **Qt [11].** Librería multiplataforma, desarrollada inicialmente por la empresa Trolltech, orientada principalmente al diseño de interfaces de usuario, aunque también admite desarrollo software sin interfaces gráficas.

- **GTK [4].** The GIMP Toolkit es un conjunto de librerías multiplataforma para desarrollar interfaces gráficas de usuario. Dichos desarrollos se integran de forma natural para los entornos gráficos GNOME, XFCE y ROX, ya que han sido implementados con estas misma librería, aunque también se puede usar en el escritorio de Windows, MacOS y otros. Inicialmente fueron creadas para desarrollar el programa de edición de imagen GIMP, sin embargo actualmente se usan bastante por muchos otros programas en los sistemas GNU/Linux. Junto a Qt es una de las librerías más populares para X Window System.

Ambas permiten programar con múltiples lenguajes como C, C++, C#, Ruby, Perl, PHP, Python, ..., ofreciendo ambas además, una serie de clases intuitiva y que incluyen la mayoría de elementos clásicos en las interfaces de usuario.

Se decidió utilizar finalmente la librería Qt ya que, a pesar de que ambas pueden utilizarse en sistemas KDE, el presente en el PC de control de Romeo-4R, la integración de aplicaciones Qt en KDE es mucho más sencilla y natural, al estar el propio entorno KDE desarrollado con esta librería. La situación o relación análoga se da entre los entornos de escritorio GNOME y XFCE con las aplicaciones escritas en GTK.

Además, el desarrollo de la aplicación sirvió como testeo y toma de contacto de las posibilidades de la propia librería Qt, ya que previos desarrollos y trabajos realizados en el GRVC habían sido realizados con GTK.

3.1.3. Visión por Computador

La visión por computador es una rama de la ciencia y de la técnica que podría definirse como las técnicas o procedimientos que nos permiten extraer información del mundo físico a partir de imágenes, utilizando para ello la ayuda de un computador.

A lo largo de los años se han desarrollado numerosas librerías software que sirven de apoyo o como punto de partida para implementar todas estas técnicas en diferentes tipos de proyectos, entre ellos los relacionados con la robótica. Algunas de estas librerías son:

- **OpenCV [6].** Es una librería libre de visión artificial originalmente desarrollada por Intel. Desde que apareció su primera versión alfa en el mes de enero de 1999, se ha utilizado en infinidad de aplicaciones, desde sistemas de seguridad con detección de movimiento, hasta aplicativos de control de procesos donde se requiere reconocimiento de objetos.
- **VXL [9].** Visión-X-Library es una colección de librerías de C++ diseñada para la investigación y la aplicación de visión por computador. Fue creado a

partir de TargetJr y el IUE, con el objetivo de hacer un sistema ligero, rápido y consistente. VXL está escrito en ANSI/ISO C++ y está diseñado para ser portado a muchas plataformas.

- **CAMELLIA [1].** Es una librería de código abierto para procesamiento de imágenes y visión por computador. Escrita en C, es multiplataforma (Unix/Linux, Windows) y robusto. Incluye una gran cantidad de funciones para el tratamiento de imagen (filtrado, matemáticas, morfología, etiquetado, deformación, etc.), la mayoría de ellos altamente optimizadas. Al utilizar la estructra CamImage/IplImage para describir las imágenes, es un buen reemplazo a la popular, pero suspendida de Intel IPL y un buen complemento a la librería OpenCV.

En el proyecto URUS se ha utilizado OpenCV por ser sin ningún tipo de dudas la librería de referencia en visión por computador actualmente. Además su integración con el resto de librerías en C++ es un trabajo relativamente sencillo.

Como hemos comentado, en los siguientes apartados haremos una descripción más exhaustiva de las principales librerías utilizadas en el proyecto URUS.

3.2. YARP

YARP [10] es un framework de código abierto, multiplataforma, escrito casi en su totalidad en C++, que soporta computación distribuida, orientado principalmente para el control en robots de manera eficiente.

Figura 3.1: Logo YARP.

Se trata de un conjunto de librerías, protocolos y herramientas para mantener los módulos y dispositivos limpiamente desacoplados.

Se trata de un middleware que básicamente da soporte a comunicaciones de datos, ya sea genéricos propios o datos típicos usados en robótica. No se trata por tanto de ningún sistema operativo. YARP está escrito por y para investigadores en robótica, en especial la robótica humanoide, en la que se debe trabajar con hardware diverso y complicado de controlar así como con una torre de módulos software igualmente complicados de manejar y comunicar.

YARP fue desarrollado como una herramienta para facilitar todo este trabajo, inicialmente a sus propios desarrolladores y posteriormente, al ser liberado al resto

de desarrolladores y trabajadores en robótica.

Los componentes principales de la librería YARP se pueden desglosar en:

- **libYARP_OS.** Interfaz con el sistema operativo para apoyar fácilmente la transmisión de datos a través de muchos hilos y/o través de muchas máquinas. YARP está escrito para ser independiente del sistema operativo utilizado, y ha sido utilizado en Linux, Windows, Mac OSX y Solaris. YARP utiliza el código abierto de la librería ACE (Adaptive Communication Environment), que es portable a través de una gama muy amplia de entornos, y de la que YARP hereda la portabilidad.

- **libYARP_sig.** Para realizar tareas comunes de procesamiento de señales (visual, auditiva) de manera abierta, facililitando la conexión con otras librerías de uso común, por ejemplo OpenCV.

- **libYARP_dev.** Interfaz con los dispositivos comunes usados en robótica: Framegrabbers, cámaras digitales, paneles de control de motores, etc.

Estos componentes se mantienen por separado. El componente básico es libYARP_OS, que deberá estar disponible antes de que otros componentes pueden ser utilizados.

Para un funcionamiento correcto en tiempo real, la sobrecarga de la red debe reducirse al mínimo, por lo que YARP está diseñado para funcionar en una red aislada o detrás de un firewall, pudiéndose obtener resultados insatisfactorios en caso contrario.

Para interactuar con el hardware, YARP depende de los sistemas operativos específicos a las que las empresas de hardware dan soporte. Actualmente pocas empresas hardware facilitan el código fuente de sus drivers, algo que limita el avance de los sistemas abiertos.

La librería libYARP_dev se estructura para interactuar fácilmente con el código proporcionado por los fabricantes de hardware, a la vez que protege el sistema de dicho código, facilitando posteriores cambios de hardware.

En consecuencia, YARP tiene tres niveles de configuración: sistema operativo, hardware, y el nivel de robot. El primer nivel de configuración debe referirse sólo si se va a compilar YARP en un sistema operativo nuevo.

El segundo nivel es el hardware. Una nueva adición en una plataforma existente o una plataforma completamente nueva puede requerir la preparación de algunos drivers YARP de dispositivos. Estos son a todos los efectos clases C++ que soportan los métodos de acceso al hardware que normalmente se implementa a través de

llamadas a función a lo proporcionado por el proveedor de hardware. Esto viene típicamente en la forma de una DLL o una librería estática.

Por último, se pueden preparar los archivos de configuración de una plataforma robótica completamente nueva.

3.2.1. Definiciones

La librería YARP soporta la transmisión de un flujo de información del usuario a través de diferentes protocolos: TCP, UDP, MCAST (multi-cast), y memoria compartida, aislando al desarrollado de los detalles intrínsecos a la tecnología y protocolos de la red usada.

Estos protocolos de bajo nivel se denominan *Carriers* o *Portadores* para distinguirlos de los protocolos de más alto nivel. Cada conexión se lleva acabo usando un protocolo y/o red física diferente. El uso de diferentes protocolos permite explotar sus mejores características en cada caso:

- *TCP:* Garantiza la recepción de los mensajes.
- *UDP:* Más rápido que TCP pero sin garantías.
- *Multicast:* Eficiente para distribuir el mismo flujo de información a múltiples destinos.
- *Memoria compartida:* empleada para conexiones locales (seleccionada automáticamente siempre que sea posible, sin necesidad de la intervención del programador).

Si los mensajes siguen las directrices YARP, entonces pueden ser convertidos a conexión en modo texto, para facilitar la monitorización humana y la intervención en el sistema.

Debemos comentar también que el uso de TCP puede presentar problemas de rendimiento en comunicaciones inalámbricas en exteriores, típicamente con los estándares IEEE 802.11 a/b/g.

Para los propósitos de YARP, la comunicación se produce a través de *conexiones* entre entidades con nombre denominadas *Puertos* (Ports). Todo junto forma un grafo dirigido llamado *YARP network* (o *Red YARP*) donde los puertos son los nodos y las conexiones son los arcos.

A cada Puerto se le asigna un nombre único (como */motor/wheels/left*). Todos los Puertos se registran por su nombre en un servidor de nombres denominado *Servidor YARP*. El objetivo es asegurar que si el nombre de un Puerto es conocido, lo

es también todo lo necesario para comunicarte con él desde cualquier máquina.

El propósito de los Puertos es mover *Contenido* (*Content*) (secuencias de Bytes que representan información de usuario) desde un hilo a otro (u otros) a través de las fronteras de los procesos y las máquinas.

El flujo de información puede ser manipulado y monitorizado externamente (por ejemplo desde la línea de comandos) en tiempo de ejecución. En otras palabras, los arcos en la red YARP son totalmente modificables. Un puerto puede enviar Contenido a una cantidad indefinida de Puertos diferentes.

Un puerto puede recibir Contenido de una cantidad indefinida de Puertos diferentes. Si un puerto ha sido configurado para enviar Contenido a otro, se dice que tienen una Conexión.

Las Conexiones pueden ser añadidas o eliminadas libremente y pueden utilizar diferentes Portadores.

El servidor de nombres YARP es un servidor que almacena la información sobre los puertos. Ordena la información por nombre, jugando un papel análogo a los servidores DNS en internet. Para comunicarse con un puerto, las propiedades de dicho puerto necesitan ser conocidas (la máquina en que está ejecutándose, el socket en el que escucha, los portadores que soporta).

El servidor de nombres YARP ofrece el lugar conveniente para almacenar estas propiedades, de manera que sólo es necesario el nombre del puerto para recuperarlas.

3.2.2. Propiedades de una red YARP

Una red YARP consiste en las siguientes entidades: un conjunto de puertos, un conjunto de conexiones, un servidor de nombres y un conjunto de registros.

- Todos los puertos tienen un nombre único.
- Toda conexión tiene un Puerto fuente y un puerto objetivo.
- Cada Puerto mantiene una lista de todas las conexiones para las cuales es el puerto objetivo.
- Cada Puerto mantiene una lista de todas las conexiones para las que es el puerto fuente.
- Sólo hay un servidor de nombres en cada red YARP.

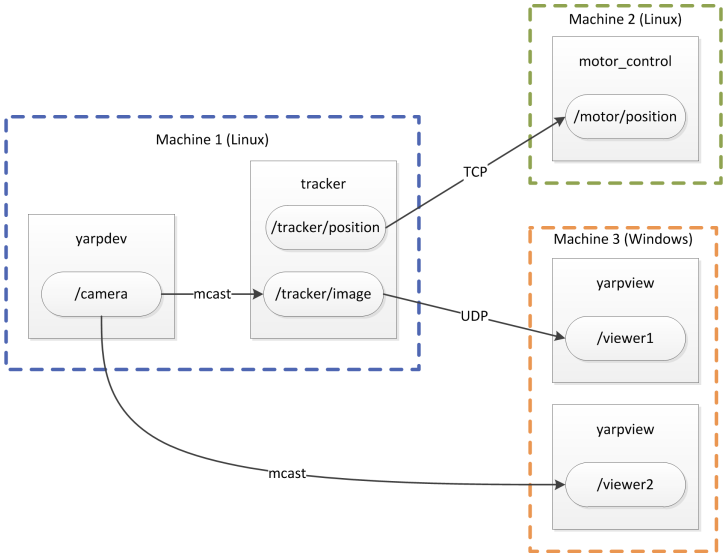


Figura 3.2: Red YARP en varias máquinas y sistemas operativos.

- El servidor de nombres mantiene una lista de registros. Cada registro contiene información sobre cada Puerto, identificados por su nombre.

- La comunicación dentro de una red YARP puede ocurrir entre dos Puertos, entre un Puerto y el servidor de nombres, entre un Puerto y una entidad exterior y entre el servidor de nombres y una entidad exterior.

- La comunicación entre dos Puertos ocurren si y sólo si hay una conexión entre ellos. Esta comunicación usa el *protocolo de conexión*.

- Las conexiones concernientes a un Puerto pueden ser creadas, destruidas o accedidas por una comunicación entre una entidad externa y ese Puerto. Esto se hace enviando *comandos de Puerto* usando el protocolo de conexión YARP.

- Los Puertos se comunican con el servidor de nombres usando el *protocolo del servidor de nombres YARP*. Esta comunicación ha de crear, borrar y acceder a registros.

- Las entidades externas también pueden usar el *protocolo de servidor de nombres YARP* para acceder al servidor de nombres.

- La utilidad YARP estándar puede ser utilizada para crear un servidor de nombres y también para actuar como una entidad externa para acceder y modificar la red YARP.

3.2.3. Gestión de los puertos

Un Puerto es un objeto que puede leer y escribir valores en objetos del mismo rango difundidos por una red de ordenadores. Pueden ser creados en un programa y añadir y eliminar conexiones tanto desde este programa como desde otro o desde la línea de comandos. Los Puertos se especializan en flujos de información, como las imágenes provenientes de una cámara o los comandos a un motor.

Además es posible cambiar los protocolos de la red sin cambiar ni una sola línea de código. Existen dos tipos de puertos:

- Puerto simple o *Port*: constituye un mini-servidor para comunicación en una red. Mantiene una colección dinámica de conexiones entrantes y salientes. Los datos provenientes de cualquier conexión entrante pueden ser recibidos llamando a la función Port::read. Las llamadas a la función Port::write hace que los datos sean enviados a todas las conexiones salientes.
- Puerto con memoria o *BufferedPort*: constituye un mini-servidor para realizar comunicación en red en background. De esta forma se puede enviar y recibir mensajes sin detener el procesamiento de órdenes.

A pesar de lo conveniente de esta funcionalidad, requiere el entendimiento del ciclo de vida de los objetos escritos y leídos de la red.

3.2.4. Empaquetamiento de la información

YARP ofrece un Contenido estándar llamado Bottle que no es más que una simple colección de objetos que pueden ser descritos y transmitidos de una forma portable. Los objetos se almacenan en una lista a la que se puede acceder y añadir elementos.

Por otra parte se pueden crear estructuras propias para almacenar información y usarlos como Contenido de manera sencilla. El objetivo en cualquier caso es encontrar formatos más eficientes de transmisión. Para la implementación de nuestra arquitectura se han implementado estructuras de datos específicos para cada uno de los módulos que envían información.

La información acerca de la herramienta YARP aquí presentada se puede extender acudiendo al sitio web de la librería: `http://eris.liralab.it/yarp`

3.3. Qt

Qt [11] es una librería multiplataforma, desarrollada inicialmente por la empresa Trolltech, orientada principalmente al diseño de interfaces de usuario, aunque

también admite desarrollo software sin interfaces gráficas.

Figura 3.3: Logo Qt.

Qt es soportado en las plataformas recogidas en la tabla 3.1.

PLATAFORMA	VERSIÓN
Microsoft Windows	98, NT, 4.0, ME, 2000, XP, ...
Unix/X11	Linux, Sun Solaris, HP-UX, HP Tru64 UNIX, ...
Mac OS X	Mac OS X 10.3+
Embedded Linux	Plataformas con soporte framebuffer

Tabla 3.1: Plataformas con soporte Qt.

Asimismo, se ofrecen diferentes ediciones de Qt:

- **Edición Comercial:** Utilizada para desarrollo de software comercial. Se permite distribución tradicional de software comercial, incluyendo actualizaciones gratuitas y soporte técnico.
- **Edición Open Source:** Utilizada para desarrollo de Software Libre y Gratuito. Se provee libre de cargo bajo los términos de las licencias Q Public License y la GNU General Public License.

Trolltech también provee de un amplio rango de componentes para la industria y plataformas específicas que hacen un complemento perfecto de Qt en el ámbito industrial. Algunas de estas soluciones se encuentran disponibles para ciertos clientes mientras que otras están disponibles para todos los usuarios de Qt.

Algunos ejemplos de programas de amplia difusión basados en Qt son:

- Adobe Photoshop Album, aplicación para organizar imagenes.
- Doxygen, API generadora de documentación.
- Google Earth, simulador de mapas en 3D.
- KDE, popular entorno de escritorio para sistemas operativos tipo-Unix.
- Texmaker y LyX, GUIs para LaTeX.
- Mathematica, la versión de Linux usa Qt para el GUI.

- Qt Creator, el entorno de desarrollo integrado, software libre y multiplataforma de Nokia.
- Quantum GIS, sistema de Información Geográfica.
- Skype, aplicación de VOIP.

Algunas de las principales características de la librería son:

- Basado en una librería de clases de C++ intuitiva.
- Portabilidad entre sistemas operativos empotrados y de escritorio.
- Herramientas integradas de desarollo multiplataforma (IDE).
- Alto rendimiento en ejecución y poco espacio en disco y memoria en sistemas empotrados.

Qt 4 se compone de diferentes módulos, donde cada uno de los cuales reside en una librería independiente. Según las funcionalidades que se quieran implementar en la aplicación se irán incluyendo dichos módulos o minilibrerías, los cuales se recogen en la tabla 3.2.

MÓDULO	**DESCRIPCIÓN**
QtCore	Clases de no GUI usadas por otros módulos
QtGui	Componentes de las interfaces gráficas de usuario
QtNetwork	Clases para programación de aplicaciones en red
QtOpenGL	Clases con soporte para OpenGL
QtSql	Clases para integración con SQL
QtScript	Clases para evaluación de Sripts Qt
QtSvg	Clases par amostrar el contenido de ficheros SVG
QtXml	Clases para el manejo de XML
QtDesigner	Clases para QtDesigner
QtUiTools	Clases para el manejo de formularios de QtDesigners en las aplicaciones
QtAssistant	Soporte para ayuda en Línea
Qt3Support	Clases con soporte de compatibilidad para Qt 3
QtTest	Clases con herramientas para pruebas

Tabla 3.2: Módulos Qt.

Los módulos de extensión disponibles en la versión comercial de Qt para sistemas Windows se recogen en la tabla 3.3.

Mientras que para sistemas Linux disponibles en todas las ediciones de Qt se incluye el módulo QtDBus, que incluye clases para el sistema de comunicaciones

MÓDULO	DESCRIPCIÓN
QaxContainer	Extensiones para el acceso controles ActiveX
QaxServer	Extensiones para el desarrollo de servidores ActiveX

Tabla 3.3: Módulos Extra Qt en Sistemas Windows.

Inter-Procesos D-Bus.

Los módulos principales en cualquier aplicación gráfica Qt son QtCore y QtGui.

El módulo QtCore forma parte de todas las ediciones de las librerías Qt, dependiendo el resto de módulos de la librería de este módulo. Para incluir la definición de las clases del método debemos incluir en nuestro código la sentencia:

```
1 #include <QtCore>
```

En primer lugar incluye la definición del namespace <Qt>, donde se incluyen numerosos identificadores usados en toda la librería. Incluye asimismo la definición de numerosas clases y objetos que son básicas para usar en otros elementos o widgets del resto de la librería.

Algunos ejemplos importantes se recogen en la tabla 3.4.

MÓDULO	DESCRIPCIÓN
QCoreApplication	Bucle de eventos para aplicaciones Qt en consola
QPointF	Define un punto en el plano con precisión float precision
QString	Cadena de caracteres Unicode
QTime	Funciones de Tiempo y Reloj
QTimer	Temporizadores repetitivos y disparadores
QTimerEvent	Parámetros que describen un evento de tiempo
QIODevice	Clase interfaz base para todos los dispositivos de I/O en Qt
QFile	Interfaz para leer y escribir en ficheros
QEvent	Clase base de todas las clases de eventos

Tabla 3.4: Clases Módulo QtCore.

El módulo QtGui extiende las funcionalidades del módulo QtCore incorporando todas las definiciones de clases utilizadas para el desarrollo de interfaces gráficas de usuario. Incorpora las definiciones de clases recogidas en la tabla 3.5.

Para incluir las definiciones de ambos módulos bastaría con la sentencia:

MÓDULO	DESCRIPCIÓN
QAbstractButton	Clase base abstracta para elementos tipo botón
QAction	Acción abstracta de la interfaz de usuario insertada entre dos widgets
QActionEvent	Evento generado cuando una QAction es añadida, eliminada o cambiada
QApplication	Gestiona el control del flujo de la aplicación gráfica y sus parámetros principales
QBitmap	Mapas de píxeles monocromos (1 bit de profundidad)
QButtonGroup	Contenedor para organizar grupos de widget buttons
QCheckBox	Checkbox con etiqueta de texto
QComboBox	Botón combinado y lista desplegable
QDialog	Clase base para las ventanas de diálogo
QDialogButtonBox	Widget que presenta botones en un layout apropiado
QDrag	Soporte para acciones de arrastrar y soltar MIME-based
QGraphicsItem	Clase base para todos los elementos gráficos en una QGraphicsScene
QGraphicsScene	Superficie para la gestión de un gran número de elementos gráficos bidimensionales
QGraphicsView	Widget que muestra el contenido de una QGraphicsScene
QIcon	Iconos escalables y de diferentes modos y estados
QImage	Tratamiento de imágenes
QPushButton	Botón de Acción
QLCDNumber	Dmuestra un número con formato de display LCD

Tabla 3.5: Clases Módulo QtGui.

```
1 #include <QtGui>
```

El módulo QtGui es parte de la Edición Ligera de Escritorio de Qt, la Edición de Escritorio Qt y la Edición Open Source de Qt.

Estos dos módulos recogen la mayoría de elementos que se utilizarán en cualquier aplicación basada en Qt. Si queremos añadir funcionalidades de red, de gráficos 3D, etc., debemos recurrir a otros módulos de Qt o a librerías externas, pero *¿cómo podemos comunicar entre sí todos estos elementos?*

Mediante los llamados Signals y Slots. El mecanismo de comunicación de objetos mediante señales y slots es un elemento central de Qt y probablemente la parte que más se diferencia de las características proporcionadas por otros frameworks de desarrollo.

En la programación de interfaces de usuario gráficas, cuando cambiamos un widget, a menudo queremos notificárselo a otro widget. De manera más general, queremos que los objetos de cualquier tipo sean capaces de comunicarse entre sí.

Por ejemplo, si un usuario hace clic en un botón de cierre, es probable que

quieren llamar a la función para cerrar la ventana close().

En sistemas más antiguos, este tipo de comunicación se lograba con las devoluciones de llamada (callbacks).

Un callback es un puntero a una función, así que si quieres una función de procesamiento para informarle sobre algún evento, debías pasar un puntero a otra función (la devolución de llamada) a la función de procesamiento.

La función de procesamiento a continuación, llama a la devolución de llamada cuando corresponda. Los sistemas basados en callbacks tienen dos defectos fundamentales: en primer lugar, no son de tipo seguro.

Nunca podemos estar seguros de que la función de procesamiento llamará a la devolución de llamada con los argumentos correctos. En segundo lugar, la devolución de llamada está fuertemente ligada a la función de procesamiento ya que la función de procesamiento debe saber a que callback llamar.

En Qt, tenemos una alternativa a la técnica de devolución de llamada: utilizamos señales y slots. Una señal se emite cuando se produce un evento en un widgets en particular. Los widgets de Qt tienen muchas señales predefinidas, pero siempre podemos hacer una subclase del widget para agregar nuestras señales propias para ellos.

Un slot es una función que se llama en respuesta a una señal particular. Igualmente los widgets de Qt tienen muchos slots predefinidos, pero es una práctica común crear subclases de widgets y añadir sus propias slots para que pueda manejar las señales que nos interesen.

El mecanismo de señales y slots sí es de tipo seguro: la firma de una señal debe coincidir con la firma del slot de recepción.

De hecho, un slot puede tener una firma más corta que la señal que recibe, ya que puede pasar por alto parámetros extra.

Dado que las firmas son compatibles, el compilador puede ayudarnos a detectar los desajustes de tipo.

Señales y slots están débilmente acoplados: una clase que emite una señal ni sabe ni le importa qué slot va a recibir la señal.

Las señales y slots de Qt aseguran que si se conecta una señal a un slot, el slot se llama con los parámetros de la señal en el momento adecuado. Señales y slots pueden tener cualquier número de argumentos y de cualquier tipo. Es por tanto un

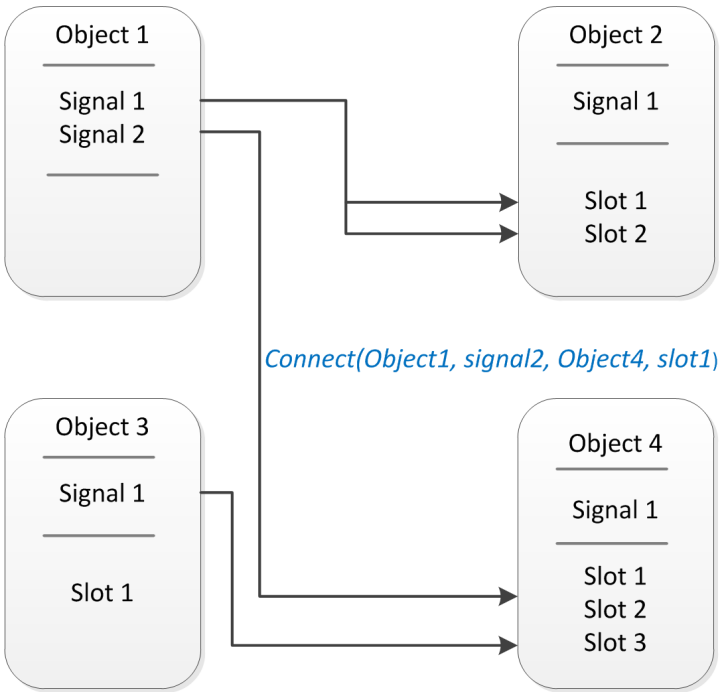


Figura 3.4: Diagrama de conexión de señales y slots diversos objetos Qt.

mecanismo totalmente seguro.

Todas las clases que heredan de QObject o una de sus subclases (por ejemplo, QWidget) pueden contener señales y slots. Las señales son emitidas por objetos cuando cambian su estado de forma que puede ser interesante para los otros objetos. Todo esto es lo que el objeto hace para comunicarse. No sabe ni le importa si algo está recibiendo las señales que emite. Esta es la encapsulación de información veraz, y asegura que el objeto puede ser utilizado como un componente de software.

Los slots se pueden utilizar para la recepción de señales, pero también son miembros de las funciones normales. Así como un objeto no sabe si algo recibe sus señales, un slot no sabe si tiene señales conectadas a ella. Esto asegura que con Qt podemos crear componentes verdaderamente independientes.

Se puede conectar tantas señales como se requiera a un único slot, así como también es posible conectar una misma señal a los slots que sea necesario. Incluso es posible conectar una señal directamente a otra señal. Esta emitirá la segunda señal de inmediato cada vez que la primero se emite.

Juntos, señales y slots constituyen un potente mecanismo de programación de componentes.

Qt no es sólamente las librerías con sus clases, elementos y mecanismos de comunicación. Actualmente para facilitar la escritura de aplicaciones Qt disponemos de completos entornos de desarrollo, que separados en distintos módulos facilitan enormemente el proceso de desarrollo de una aplicación.

En concreto, para el desarrollo de la interfaz de Romeo se ha aprovechado la integración con Qt que implementa el conocido entorno de desarrollo KDevelop así como QtDesigner para la creación de los formularios y ventanas de la aplicación.

QtDesigner en un potente generador de layouts y formularios para el diseño de interfaces de gráficas de usuario. Al igual que el resto de Qt es totalmente multiplataforma. QtDesigner permite diseñar y contruir rápidamente widgets y diálogos con formularios que formarán parte de la aplicación final. Los formularios y ventanas creadas con QtDesigner son totalmente funcionales y pueden generarse una preview de tal modo que podemos comprobar y asegurar el correcto funcionamiento y apariencia que teníamos en mente.

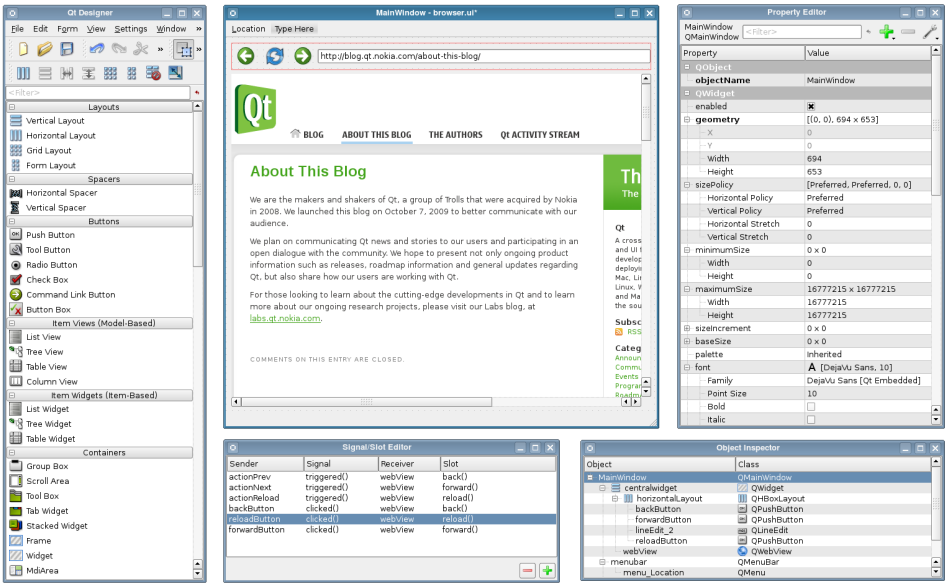


Figura 3.5: Qt Designer.

Características y beneficios:

- Diseño de interfaces rápido con la funcionalidad Arrastrar y Soltar.
- Genera widgets a medida o utiliza los estándares de la librería.

- Previews instantáneas.
- Generación de código C++ o Java de los prototipos de la interfaz.
- Integración de QtDesigner con múltiples IDEs (Visual Studio, Eclipse, Kdevelop, ...).
- Construcción de interfaces de usuario totalmente funcionales mediante el uso de las señales y slots propias de Qt.

Comentar en último lugar que la versión más actual del desarrollo ha sido lanzada en su versión Qt 5.0 en el momento de escritura de este texto, mientras que la aplicación Romeo HMI fue desarrollada con la versión 4.3 de la librería.

3.4. OPENCV

OpenCV (Open Source Computer Vision) [6] es una librería de código abierto para la programación de sistemas de visión por ordenador en tiempo real.

Figura 3.6: OpenCV Logo.

OpenCV está escrito en C y C++ y es multiplataforma, ejecutándose bajo Linux, Windows y Mac OS X.

Actualmente existen desarrollos en activo trabajando en interfaces para Python, Ruby, Matlab y otros lenguajes de programación.

OpenCV fue diseñado para obtener eficiencia computacional en ejecución y con un fuerte enfoque para las aplicaciones en tiempo real. Está escrito en lenguaje optimizado en C y además puede beneficiarse del uso de procesadores multinúcleo.

Uno de los objetivos del proyecto OpenCV es proveer de una infraestructura de visión por computador fácil de usar, que facilite a los desarrolladores construir sofisticadas aplicaciones de visión de manera rápida y eficiente. La librería OpenCV contiene más de 500 funciones que incluyen muchas áreas de la visión por computador, tales como inspección en la fabricación de productos, imágenes médicas, seguridad, interfaces de usuario, calibración de cámaras, visión estéreo, así como aplicaciones específicas para robótica.

OpenCV está estructurado en cinco componentes principales, cuatro de las cuales se muestran en la figura 3.7.

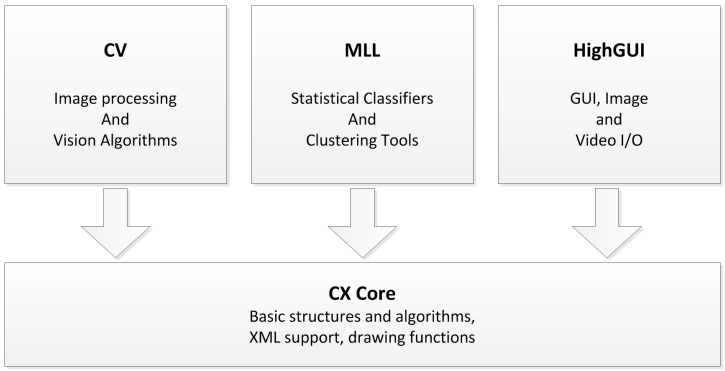


Figura 3.7: Estructura librería OpenCV.

El componente CV contiene el procesamiento base de imágenes y algoritmos de nivel superior de visión por computador. MLL (Machine Learning Library) es la librería de aprendizaje automático, que incluye a muchos clasificadores estadísticos y herramientas de clustering. HighGUI contiene rutinas de salida y funciones para el almacenamiento y carga de vídeo e imágenes, y CXCore contiene las estructuras de datos básicos y el contenido.

La figura 3.7 no incluye el último módulo CvAux, que contiene áreas abandonadas del proyecto (integrado HMM y reconocimiento facial) y algoritmos experimentales (segmentación de fondo/primer plano).

CvAux no está especialmente bien documentado aunque cubre las siguientes áreas:

- Eigen objects, una técnica de reconocimiento computacionalmente eficiente, es decir, en esencia, un patrón de reconocimiento de formas.
- 1D y 2D modelos ocultos de Markov, una técnica de reconocimiento de estadística resuelto por programación dinámica.
- Los HMM incrustado (las observaciones de una HMM padres mismos son HMM).
- Reconocimiento gestual apoyado en visión estéreo.
- Extensiones a la triangulación de Delaunay, secuencias, etc.
- Visión estéreo.
- Descriptores de textura.

- Ojos y seguimiento de la boca.
- Seguimiento 3D.
- Búsqueda de esqueletos (vías centrales) de los objetos en una escena.
- Distorsión intermedios puntos de vista entre dos puntos de vista de la cámara.
- Antecedentes en primer plano de segmentación.
- Vigilancia por video.
- Clases para calibración de cámaras de C++ (las funciones de C y el motor están en CV).

Figura 3.8: Proyecto URUS. Seguimiento visual de personas para experimento de guiado.

Capítulo 4

Interfaz Romeo HMI

4.1. Introducción

En primer lugar hemos de determinar el uso específico de la aplicación. La interfaz gráfica Romeo HMI se utiliza para controlar y monitorizar el vehículo autónomo Romeo-4R. Está orientada a facilitar el uso del robot durante la realización de experimentos de navegación y guiado, por lo cual lo primero que debemos destacar es que no se trata de una interfaz hombre máquina orientada para un usuario final, sino que viene a satisfacer algunos problemas que la experiencia ha demostrado que pueden resultar muy útiles en cuanto tiempo y facilidad de uso para los desarrolladores de Romeo-4R.

Es por ello, que desde la aplicación se tiene acceso a los datos provenientes de los múltiples sensores que dispone el robot, lo cual pueden servir de orientación para saber que está viendo en cada momento el robot, o por ejemplo para comprobar si se está produciendo algún error excesivo en la localización del mismo.

También se creó un mecanismo de introducción de las trayectorias que debe seguir el robot, algo que facilitó mucho el anterior mecanismo basado en la generación de ficheros de texto plano, así como un mecanismo para lanzar y detener los distintos módulos de funcionamiento que han sido explicados en el capítulo 2 de este texto.

Comentar por último que los aspectos gráficos de la interfaz, como el tamaño de los botones y widgets por ejemplo, han sido diseñados teniendo en cuenta el modelo de pantalla táctil que utiliza Romeo-4R, que se corresponde con una pantalla táctil de 15 pulgadas de la serie ET1515L del fabricante ELO, así como también la resolución utilizada por defecto de *1024 x 768 píxeles*.

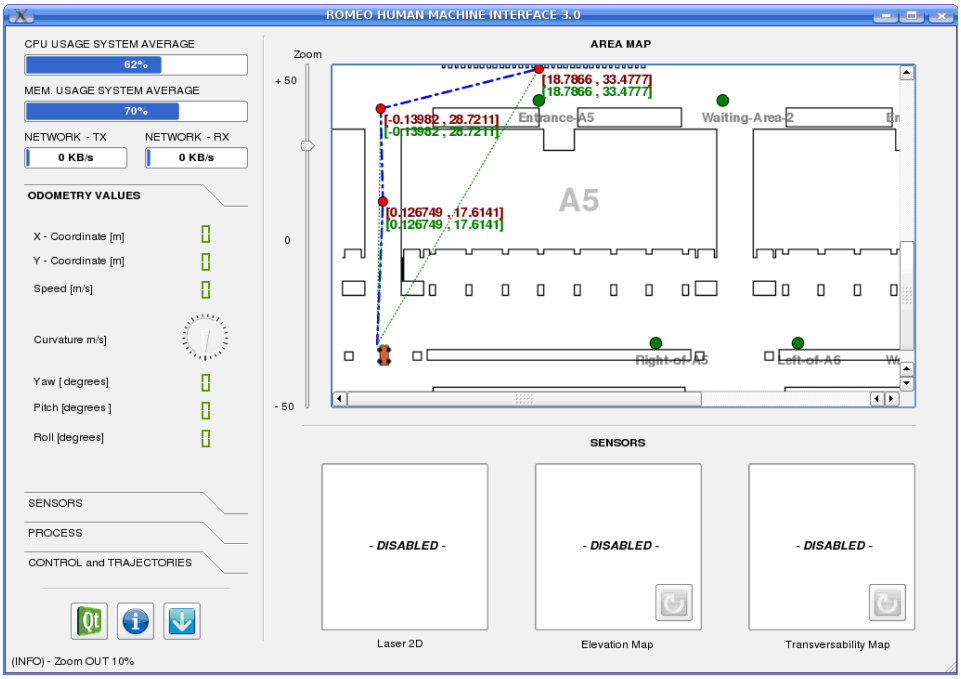


Figura 4.1: Interfaz gráfica Romeo HMI.

4.2. Definición de clases principales

Veremos a continuación una descripción de las clases principales implementadas en la aplicación así como una descripción funcional de las mimas. Los diagramas que aparecen en esta sección han sido creados con la herramienta de generación automática de documentación Doxygen [3].

4.2.1. RomeoMainWindow

Sin duda RomeoMainWindow es la clase principal de la aplicación. Partimos de la clase base QMainWindow, que nos proporciona el marco para la creación de la interfaz de usuario de la aplicación.

En Qt disponemos de QMainWindow y sus clases relacionadas para la gestión de la ventana principal. QMainWindow tiene su propio diseño a la que puede agregar QToolBars, QDockWidgets, un QMenuBar e incluso una QStatusBar.

De hecho, el main.cpp de la aplicación se reduce a algo tan sencillo como lo recogido en la tabla 4.1.

Donde como vemos simplemente se genera un objeto de la clase QApplication,

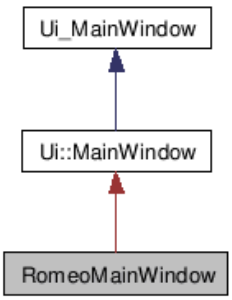


Figura 4.2: Diagrama de Herencia de clase RomeoMainWindow.

que gestiona el flujo de la aplicación con interfaz gráfica, así como los ajustes principales y se genera la pantalla principal de la aplicación.

La clase QApplication contiene el bucle de eventos principal, donde todos los eventos del sistema de ventanas y otras fuentes son procesados y enviados. También maneja la inicialización de la aplicación y finalización de la misma, proporciona la administración de sesiones y gestiona todos los ajustes de la aplicación.

Una vez generada la pantalla principal, mediante la función exec() entramos en el bucle de eventos principal una vez ha sido generada como decimos la pantalla principal. Es necesario llamar a esta función para iniciar la gestión de eventos.

```
1  #include <QApplication>
2  #include "romeomainwindow.h"
3
4  int main(int argc, char *argv[])
5  {
6          Q_INIT_RESOURCE(application);
7          QApplication app(argc, argv);
8          new RomeoMainWindow();
9          return app.exec();
10 }
```

Tabla 4.1: ***Proyecto URUS – main.cpp.***

El bucle de eventos principal recibe los eventos del sistema de ventanas y los redirige a los distintos widgets de la aplicación.

Parte de la definición de la clase RomeoMainWindow se recoge en la tabla 4.2, simplemente para mostrar los elementos más significativos. Recogemos aquí las variables y funciones principales de la clase, incluyendo la definición de los puer-

tos YARP a utilizar, la definición de un objeto de la clase MapScene (ver apartado 4.2.2) así como algunas de las variables de control utilizadas (ver apartado 4.3).

La definición de los widgets que componen la interfaz, así como su disposición en los distintos frameworks de la misma han sido generados mediante el uso de la aplicación visual QtDesigner, basada en el método arrastrar y soltar, que guarda el resultado un fichero con extensión .ui (User Interface), el cual es simplemente un fichero de texto plano en formato XML, del que recogemos un ejemplo (extracto) en la tabla 4.3.

Utilizando el llamado enfoque de herencia múltiple, todos los componentes de interfaz de usuario definido en la ventana son directamente accesibles en el ámbito de la subclase, y permite conexiones de forma habitual mediante señales y slots. Únicamente tenemos necesidad de incluir el encabezado del archivo que genera el UIC (user interface compiler) desde el archivo con extensión .ui, que en nuestro caso se recoge en la tabla 4.4.

Finalmente el constructor de la subclase es el que realiza las tareas de generación de la interfaz y colocación de los widgets y la configuración de los atributos de los mismos.

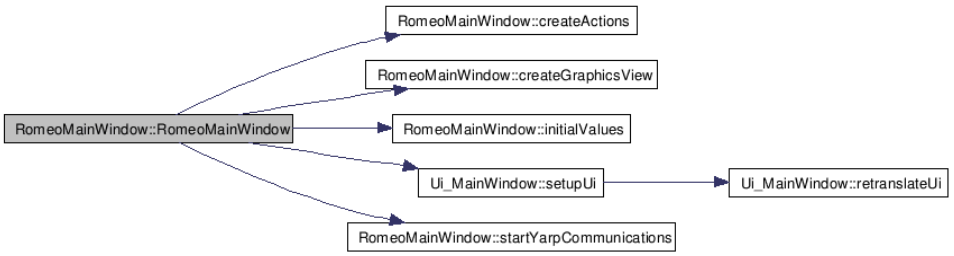


Figura 4.3: Llamadas desde el constructor RomeoMainWindow.

Este procedimiento es el que se utiliza también para la generación del resto de formularios que se utilizan en la aplicación, tal y como se recoge en la figura 4.4.

4.2.2. MapScene

La clase MapScene es una clase derivada de QGraphicsScene, implementada para mejorar la gestión de mensajes al realizar ciertos eventos sobre el área gráfica interactiva que muestra el mapa y la localización del robot (ver sección 4.4). La definición de la clase se recoge en la tabla 4.6.

La figura 4.5 nos muestra el gráfico de dependencias en el fichero mapscene.cpp.

```
class RomeoMainWindow:public QMainWindow,private Ui::
   MainWindow
{
        Q_OBJECT

        public:
                RomeoMainWindow(QWidget *parent=0,Qt::
                   WFlags fl = 0);
                ~RomeoMainWindow();

        signals:
                void  message(QString); // Info Text
                   Message Signal

        protected:
                void closeEvent(QCloseEvent *event);

        private slots:
                void exitApplication();
                void openMap();
                void loadMap(QString mapName);
            ...

        private:
                void createActions();
                void initialValues();
                void createGraphicsView();
                void startYARPCommunications();
                QProcess *LAUNCHER_Process;
                QString LAUNCHER_Program;
                QString LAUNCHER_Directory;
                MapScene scene;
                QPixmap map;
                ScaleDialog MapScale;
                trajectoryDialog SelectTrajectory;
                bool trajectory_edit_mode;
                bool trajectory_executing_mode;
                bool setting_initial_point;
                bool theta_edit_mode;
                bool trajectoriesBYinterface;
                bool yarpRunning;
                CDcxData *pyarpDcx;     // Fron DCX
                CLocData *pyarpEkf;     // From EkFloc
                yarp::os::BufferedPort<CLocData> locInput;
                yarp::os::BufferedPort<CDcxData> odomInput;
            ...
}
```

Tabla 4.2: ***Proyecto URUS – romeomainwindow.h.***

```
class Ui_MainWindow
{
public:
    QAction *actionAboutQT;
    QAction *actionAboutRomeoHMI;
    QAction *actionOpenMap;
    QAction *actionOpenTrajectory;
    QWidget *centralwidget;
    QToolBox *toolBox;
    QWidget *page;
    QLabel *XcoordinateLabel;
    QLCDNumber *YcoordinatelcdNumber;
    QLabel *PITCHlabel;
    QLCDNumber *XcoordinatelcdNumber;
    ...

    void setupUi(QMainWindow *MainWindow)
    {
            if (MainWindow->objectName().isEmpty())
            MainWindow->setObjectName(QString::fromUtf8("
               MainWindow"));
            MainWindow->setWindowModality(Qt::WindowModal);
        MainWindow->resize(1024, 695);
            actionAboutQT = new QAction(MainWindow);
            ...
            retranslateUi(MainWindow);
            toolBox->setCurrentIndex(3);

        QMetaObject::connectSlotsByName(MainWindow);
        } // setupUi
        ...
};
namespace Ui {
    class MainWindow: public Ui_MainWindow {};
} // namespace Ui
```

Tabla 4.3: ***Proyecto URUS – ui-MainWindow-HMI2.h.***

```
#include "ui_MainWindow_HMI2.h"
```

Tabla 4.4: ***Proyecto URUS – ui-MainWindow-HMI2.h.***

```
RomeoMainWindow::RomeoMainWindow(QWidget* parent, Qt::
    WFlags fl):QMainWindow(parent, fl), Ui::MainWindow()
{
        setupUi(this);
        initialValues();
        createActions();
        createGraphicsView();
        check_YARP_network();
        startYARPCommunications();
        statusbar->showMessage(tr("(INFO) - Ready"));
}
```

Tabla 4.5: ***Proyecto URUS – romeomainwindow.cpp.***

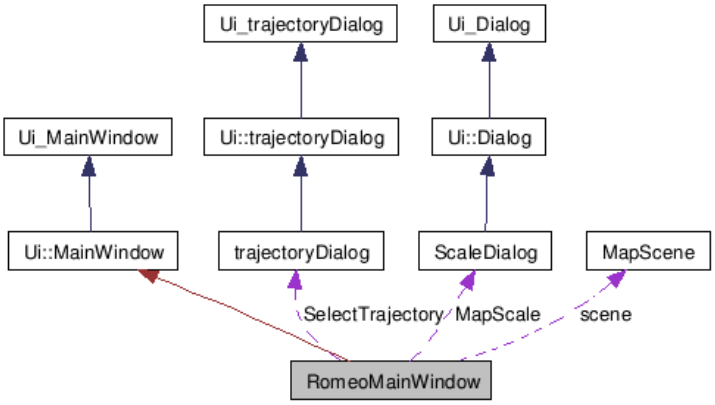


Figura 4.4: Diagrama de colaboración de clase RomeoMainWindow.

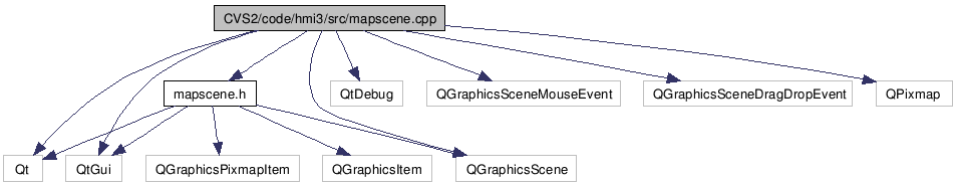


Figura 4.5: Gráfico de dependencias fichero mapscene.cpp.

```
1  class MapScene : public QGraphicsScene
2  {
3          Q_OBJECT
4
5  public:
6          MapScene();
7          ~MapScene();
8
9  signals:
10         void messagemap(QString);
11         void mapScenePressEvent(QPointF clickedPoint);
12         void changeRobot(QPoint);
13         void ungrabRobot();
14
15 protected:
16
17         void mousePressEvent (QGraphicsSceneMouseEvent *
               event);
18         void dragMoveEvent (QGraphicsSceneDragDropEvent *
               event);
19         void dropEvent (QGraphicsSceneDragDropEvent * event
               );
20 };
```

Tabla 4.6: ***Proyecto URUS – mapscene.h.***

No habría sido estrictamente necesario la realización de esta clase, pues podría haberse implementado su funcionalidad con la clase genérica QGraphicsScene, sin embargo se mantuvo en el desarollo para implementar el sistema de paso de mensajes a la barra de estado de la aplicación.

4.3. Variables

Se han utilizado mútiples variables a lo largo de la aplicación. Vamos a realizar una clasificación de las mismas según su utilidad. Distinguimos varios grupos, entre ellos:

- **Variables para sensores robot Romeo-4R**

  Las variables de la tabla 4.7 recogen la información de localización del robot en en el mundo real, relativas siempre al mapa de la zona, al denominado en el capítulo 2 como World Coordinate System (WCS).

```
double theta;
double previous_theta;
double theta_offset;
double locale_romeo_x_ref;
double locale_romeo_y_ref;
double romeo_speed_reference;
```

Tabla 4.7: ***Romeo HMI – Variables locales de localización del robot.***

Mientras que las variables de la tabla 4.8 recogen la información de localización que el módulo EKFLOC exporta y se corresponden con la información del sistema de referencia local del robot Romeo Coordinate System (RCS).

```
double x_ref;
double y_ref;
double orient_ref;
double curv_ref;
double previous_orient_ref;
```

Tabla 4.8: ***Romeo HMI – Variables de localización del robot - módulo EKFLOC.***

Otros valores de odometría procedentes de las lecturas de los encoders en las ruedas que nos dan la velocidad (exportado por el módulo DCX) así como las estimaciones de los ángulos de Tait-Bryan, Yaw, Pitch y Roll (exportados

```
double speed_ref;        // From DCX
double yaw_ref;          // From IMU
double pitch_ref;        // From IMU
double roll_ref;         // From IMU
```

Tabla 4.9: ***Romeo HMI – Variables odométricas del robot - módulos IMU y DCX.***

por el módulo IMU).

- **Variables de la vista gráfica correspondiente al mapa**

  Estas variables contienen la información necesaria para localizar al robot en el mapa así como para poder escalar el mismo con sus dimensiones reales, pudiendo mostrar en cada instante la estimación de la posición del robot en el mapa.

```
double map_width;        // in Pixels
double map_height;       // in Pixels
double map_x_axis_length;        // in meters
double map_y_axis_length;        // in meters
double map_X_scale;      // Pixels/meter
double map_Y_scale;      // Pixels/meter
double scene_x_axis_length;
double scene_y_axis_length;
double x_ini_global;     // Romeo Initial Position
double y_ini_global;
```

Tabla 4.10: ***Romeo HMI – Variables vista gráfica del mapa.***

- **Variables Estado de Sistema**

  Con los valores de estas variables se va actualizando los datos de la información del sistema, carga de CPU, Memoria y estadísticas de red, que aparecen en la parte superior del frame izquierdo de la aplicación (ver sección 4.4).

- **Variables Estado Aplicación**

  La aplicación sigue un diagrama de estados de funcionamiento en función de los valores de estas variable binarias, que inicialmente tienen todos sus valores a FALSE.

```
double m_stats[4];      // Values from /proc/stat
double m_stats_new[4];  // For CPU Usage evaluation
double tx_net_stats;    // Values from /proc/dev/net
double tx_net_stats_new;
double rx_net_stats;    // Values from /proc/dev/net
double rx_net_stats_new;
```

Tabla 4.11: ***Romeo HMI – Variables Monitor Estado del Sistema.***

```
bool trajectory_edit_mode;
bool trajectory_executing_mode;
```

Tabla 4.12: ***Romeo HMI – Variables estado aplicación.***

El diagrama de estados de funcionamiento de la aplicación, recogido en la figura 4.6 es en realidad muy sencillo. Tras un cuasi estado de inicialización de todo el sistema (estado 0), se entra en el modo general de espera (estado 1).

Utilizando los controles de trayectoria entramos en el estado de introducción de trayectoria (estado 2). Si se cancela la trayectoria se vuelve al estado de espera (estado 1).

Una vez preparada la trayectoria se envía al módulo PURE PURSUIT entrando en modo de ejecución de trayectoria (estado 3).

Una vez la trayectoria ha sido ejecutada o si bien se interrumpe la ejecución de la misma se vuelve al modo general de espera (estado 1).

ESTADO	NOMBRE	VARIABLES ACTIVADAS
0	Inicialización del Sistema	Ninguna
1	Modo de Espera	Ninguna
2	Modo Introducción de Trayectoria	trajectory_edit_mode
3	Modo Ejecución de Trayectoria	trajectory_executing_mode

Tabla 4.13: Romeo HMI – Tabla de estados aplicación.

Por tanto vemos que el modo de uso de la aplicación tiene un comportamiento sencillo, totalmente cíclico y secuencial.

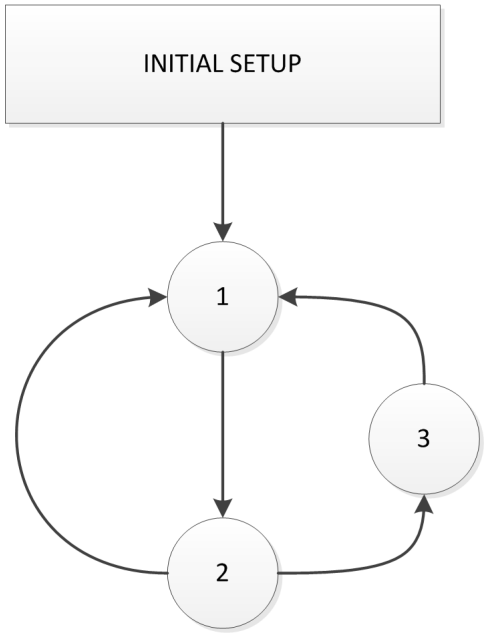


Figura 4.6: Diagrama estados funcionamiento aplicación Romeo HMI.

4.4. Descripción de la interfaz gráfica

La interfaz gráfica se estructura en tres frames principales, tal y como vemos en la 4.7.

1. **Frame Izquierdo.** Dividido a su vez en dos segmentos. El primero de ellos se dedica a mostrar información del sistema que está ejecutando la propia aplicación, que por lo general ejecutará también la mayor parte de los módulos que componen el sistema. Es por tanto un monitor del estado del sistema, mostrando promedios en tiempo real de datos de consumo de CPU, uso de memoria del sistema y utilización de red (transmisión y recepción).

   El segundo segmento está formado por un widget QToolBox, que nos permite ordenar diferentes widgets en cada una de las pestañas que lo componen, mostrando siempre los widgets bajo la pestaña seleccionada en cada instante. Se han asignado cuatro ventanas, con los siguientes identificadores:

   - *ODOMETRY VALUES*
     Bajo esta pestaña se muestran los valores más significativos de la odometría y de algunos sensores del vehículo.

     Se muestra la posición en el plano estimada del robot, la velocidad instantánea y el ángulo de giro del volante, así como los ángulos Yaw,

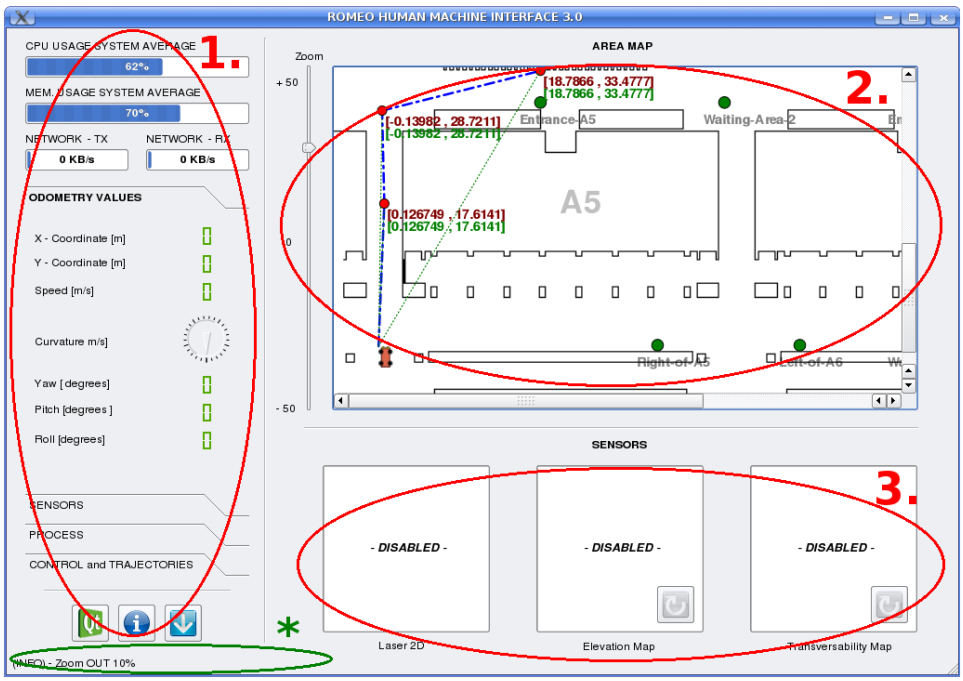


Figura 4.7: Frames aplicación Romeo HMI.

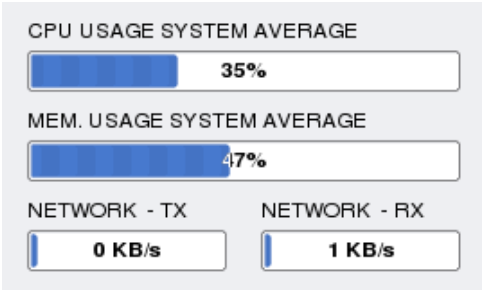


Figura 4.8: Monitor del sistema.

Pitch y Roll.

Figura 4.9: QToolBox Frame izquierdo.

Toda esta información procede de los puertos YARP correspondientes a los módulos DCX, EKFLOC e IMU. La información se actualiza cada segundo, aunque el intervalo de actualización es modificable.

- *SENSORS*
  Pestaña que contiene los controles que permiten la visualización de los sensores indicados en el área de visualización correspondiente (Frame Inferior Derecho).

  La visualización de los sensores en tiempo real es comprometida para el rendimiento general del sistema, al aumentar considerablemente el flujo de datos a transmitir por la red YARP, ya que se deben generar y transmitir las imágenes que capta el módulo Laser 2D o bien las que generan los mapas de elevación y de transversabilidad.

- *PROCESS*
  En esta pestaña aparece el controlador de arranque y parada de un script generado para inicializar secuencialmente todos los módulos de control, sensorización y navegación que componen el cerebro de Romeo-4R que fueron explicados en el capítulo 2.

Figura 4.10: Controles visualización de los sensores.

Figura 4.11: Controles del Launcher.

Estos controles sirven pues para iniciar y detener el funcionamiento del robot como tal.

- *CONTROL and TRAJECTORIES*
  Esta pestaña contiene los controles para la generación y el paso de trayectorias al robot.

  Se ha creado un mecanismo de introducción de waypoints secuenciales donde una vez escrita la trayectoria se pasa al módulo correspondiente por un puerto YARP.

  Se han creado mecanismos para indicarle al robot su posición y orientación inicial en el mapa, así como métodos para eliminación y carga tanto de trayectorias como de mapas.

  Al cargar un nuevo mapa (imagen bmp) aparece una ventana emergente que nos permite calibrar el mismo, es decir, hacer la correspondencia entre píxeles – metros.

  Esta pestaña incluye un botón de parada de emergencia por software que detiene todos los procesos de control del robot provocando su parada cuasi inmediata.

Figura 4.12: Controles de inserción de trayectoria.

El frame izquierdo finaliza con unos iconos donde se muestran accesos directos a los créditos de Qt, créditos de la aplicación así como una opción de salida de la aplicación, respectivamente.

Figura 4.13: Accesos directos frame izquierdo.

2. **Frame Derecho Superior.** En esta zona de la aplicación se ha incluido el mapa de la zona y los controles de zoom de la misma. El mapa cargado se muestra en una clase MapScene que es totalmente interactiva.

   En tiempo de ejecución, se va mostrando la posición actual del robot y puede superponer los mapas de elevación y de transversabilidad que se van generando (con el coste computacional que ello supone).

   En el modo de introducción de trayectoria la zona del mapa responde de forma interactiva según la opción que le hayamos indicado, mostrando los próximos waypoints o mostrando la localización del robot.

3. **Frame Derecho Inferior.** En la zona inferior derecha de la pantalla se muestra la información proveniente del laser frontal (Laser 2D) lo cual sirve de

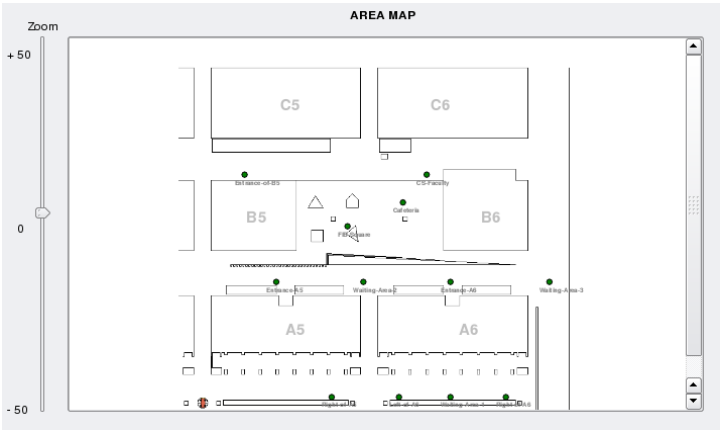


Figura 4.14: Zona de visualización mapa (I).

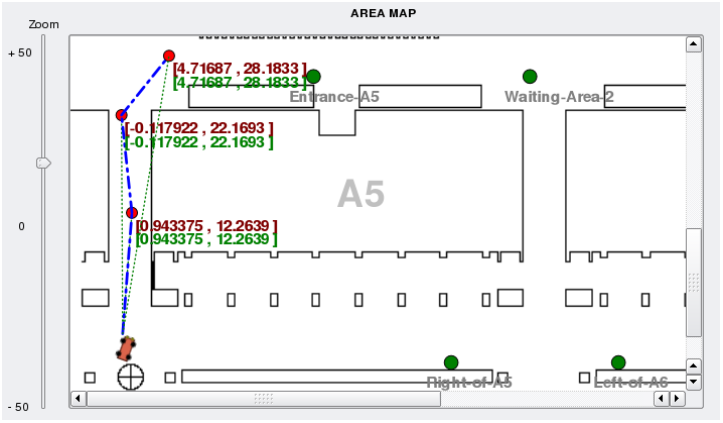


Figura 4.15: Zona de visualización mapa (II).

utilidad para *comprobar qué está viendo el robot* así como para mostrar los mapas de elevación y de transversabilidad del entorno que Romeo-4R va generando en tiempo real.

Figura 4.16: Sensores Romeo HMI.

La información a mostrar es seleccionable y puede ser integrada con la zona de visualización del mapa.

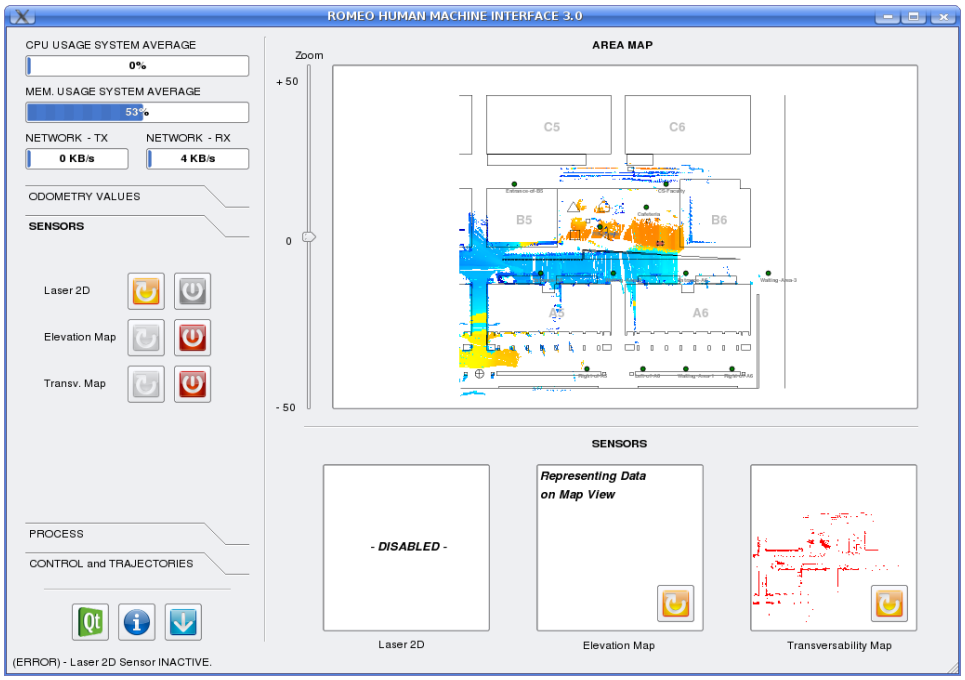


Figura 4.17: Mapa de elevación integrado en la visualización del mapa.

Destacar también que la aplicación cuenta con una barra de estado donde van apareciendo los acontecimientos más significativos y distintos mensajes de información, warning o error, que se van produciendo en tiempo de ejecución.

Capítulo 5

Sistema de Análisis Post-Misión (Logplayer)

5.1. Introducción

En el diseño de cualquier sistema robótico móvil, la simulación de los algoritmos diseñados para navegación o filtrado de datos de sensores es un paso previo fundamental a la realización de los experimentos reales.

En efecto, la realización de un experimento real en un entorno de exteriores es un proceso muy complejo y en ocasiones altamente costoso e incluso, puede llegar a resultar peligroso para la integridad física de las personas.

La simulación de experimentos en un entorno de computación es una herramienta altamente poderosa en el proceso de diseño y refinado de los algoritmos y de la inteligencia del robot.

En el proyecto URUS se desarrolló el Sistema de Análisis Post-Misión, también denominado como herramienta LogPlayer que permite reproducir, estableciendo una base de tiempos común todos los logs captados en anteriores experimentos, de tal modo, que no es necesario reproducir realmente un experimento, sino que dichos datos interpretados pueden ser utilizados como fuente u origen de datos para los módulos que componen la inteligencia del robot, reemplazando de manera transparente a los módulos que proporcionan los datos reales. Por tanto, un entorno en simulación podría corresponderse con el siguiente esquema, donde se han sustituido algunos módulos de la figura 5.1 por el módulo LogPlayer.

Es decir, en este esquema el módulo EKFLOC recibe la información que ha sido logueada en un experimento anterior en lugar de información real proveniente de los sensores reales, del mismo modo que le ocurre a los módulos que generan los mapas de elevación y transversabilidad. Evidentemente, y como veremos más

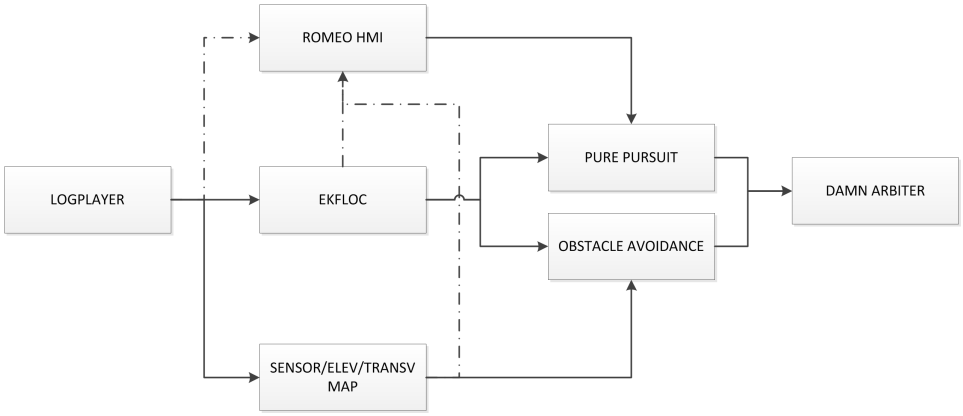


Figura 5.1: Módulo LOGPLAYER reemplazando sensores.

adelante, podemos reproducir no sólo datos de los sensores sino también cualquier resultado o dato que sea susceptible de ser enviado por un puerto YARP.

En primer lugar vamos a ver el formato que tienen los ficheros de log generados por el módulo PORTLOGGER, implementado por D. Francisco J. Real Pérez. La información que exportan los distintos módulos del sistema por sus puertos YARP es logueada por este módulo en un fichero de texto plano, con una cabecera que indica por columnas las magnitudes o datos que se van guardando así como la unidad de los mismos.

Posteriormente, cada fila se corresponde con una trama de datos, identificada por temporalmente por dos valores de distinta precisión, una en segundos y otra en nanosegundos.

Veamos por ejemplo los logs de los datos generados por los módulos DCX e IMU.

time [s]	**time [ns]**	**v_ref [m/s]**	**V [m/s]**	**v_dev [m/s]**	**k_ref [1/m]**	**K [1/m]**	**k_dev [1/m]**
1247847886.000	440439748.000	0.0	0.0	0.0	0.0	-0.0	0.0
1247847886.000	520507941.000	0.0	0.0	0.0	0.0	-0.0	0.0
1247847886.000	592376273.000	0.0	0.0	0.0	0.0	-0.0	0.0
1247847886.000	664198115.000	0.0	0.0	0.0	0.0	-0.0	0.0
1247847886.000	783997555.000	0.0	0.0	0.0	0.0	-0.0	0.0
1247847886.000	855816185.000	0.0	0.0	0.0	0.0	-0.0	0.0
1247847886.000	928600571.000	0.0	0.0	0.0	0.0	-0.0	0.0

Tabla 5.1: LogPlayer - Fichero de log dcx.log.

time [s]	time [ns]	roll [deg]	yaw [deg]	pitch [deg]
1247847887.00000	158343417.00000	1.62000	-17.80000	-2.41000
1247847887.00000	261949390.00000	1.71000	-17.60000	-2.33000
1247847887.00000	357754939.00000	1.69000	-17.60000	-2.34000
1247847887.00000	457529967.00000	1.70000	-17.60000	-2.34000
1247847887.00000	561504884.00000	1.70000	-17.60000	-2.40000
1247847887.00000	661595484.00000	1.78000	-17.50000	-2.36000
1247847887.00000	761932839.00000	1.70000	-17.60000	-2.33000

Tabla 5.2: LogPlayer - Fichero de log imu.log.

Es importante destacar que la cantidad de datos generada durante un experimento de un minuto de duración es realmente alta, en torno a 1 Gbyte de información.

La aplicación LogPlayer parte de estos distintos ficheros de log, para crear unos puertos YARP equivalentes a los de cada módulo y exportar por ellos los datos guardados, recorriendo cada fichero de log línea a línea y todo ello de manera síncrona, reproduciendo el experimento tal y como fue logueado. La reproducción de cada módulo se realiza con un hilo independiente, implementando como hemos dicho un método de sincronismo entre los distintos hilos que veremos a continuación.

Nota: en lo que sigue de capítulo hablaremos indistintamente de puerto, puerto YARP, log o módulo, refiriéndonos siempre al módulo que se va reproducir.

5.2. Definición de Clases

Veremos a continuación una descripción de las clases principales implementadas en la aplicación así como una descripción funcional de las mismas. Los diagramas que aparecen en esta sección han sido creados con la herramienta de generación automática de documentación de código fuente Doxygen [3].

5.2.1. LogPlayer

La clase LogPlayer es la clase base para la generación de los distintos reproductores. Dicha clase hereda de una clase utilizada en el proyecto URUS, que sirve de abstracción para la generación de hilos para no tener que usar directamente la librería POSIX Threads <pthread.h>.

Contiene como vemos en su declaración funciones para acceder a la especificación inicial de tiempo del log, para leer cada una de las líneas del fichero de log

```
class LogPlayer: public Thread {
public:
        virtual ~LogPlayer() {};
        void getInitialTime(timespec* time);
        void setOffset(const timespec& offset);

protected:

        virtual bool readNext(timespec* nextTime) = 0;
        virtual void send() = 0;
        void run();
        timespec offset_ts, dataFirstShot_ts;
        TimeReference tref;
        double discardLapse;
};
```

Tabla 5.3: ***LogPlayer – Declaración clase LogPlayer (LogPlayer.h.)***

y enviarlas por el puerto YARP correspondiente, así como una serie de estructuras temporales que se usarán para el sincronismo de los distintos logs a reproducir.

Sin embargo, tal y como se desprende del capítulo 2, cada log lleva un tipo de datos característico, por tanto, para evitar la repetición de código, se han utilizado una serie de templates, que nos permiten generar las estructuras de datos necesarias para cada log, con la clase TypedPlayer<T>.

5.2.2. PlayerFactory

En la declaración de esta clase encontramos el código que nos permite crear los distintos tipos de puertos YARP que vamos a reproducir.

El utilizar templates nos permite agilizar el proceso en el momento que se haga necesario reproducir algún tipo de log más.

Si se añadiera un nuevo módulo software al robot, por ejemplo correspondiente a algún sensor nuevo, una vez creada su clase específica de comunicaciones (CNewClassData), bastaría con añadirla aquí y en el fichero LogPlayer.cpp para que la reproducción del nuevo módulo quedase totalmente integrada en el sistema.

5.2.3. TimeReference

Estructura temporal utilizada para el sincronismo de los distintos logs.

```
class PlayerFactory {
public:

        static LogPlayer *createPlayer(  std::string
           classname, std::string port, std::string
           filename ) {

            if ( !classname.compare("CImuData") )
            { return new TypedPlayer<CImuData>( port,
               filename ); }
                if ( !classname.compare("CLaser2dData") )
            { return new TypedPlayer<CLaser2dData>( port,
               filename ); }
                if ( !classname.compare("CLocData") )
            { return new TypedPlayer<CLocData>( port,
               filename ); }
                if ( !classname.compare("CDcxData") )
            { return new TypedPlayer<CDcxData>( port,
               filename ); }
                if ( !classname.compare("CGyroData") )
            { return new TypedPlayer<CGyroData>( port,
               filename ); }
                if ( !classname.compare("CGpsData") )
            { return new TypedPlayer<CGpsData>( port,
               filename ); }
                if ( !classname.compare("CLoc6DData") )
            { return new TypedPlayer<CLoc6DData>( port,
               filename ); }
                if ( !classname.compare("CNewClassData") )
            { return new TypedPlayer<CNewClassData>( port,
               filename );
                                                    }
                throw std::runtime_error("PlayerFactory::
                   createPlayer: Unsupported classname.");
        }
};
```

Tabla 5.4: ***LogPlayer – Declaración clase PlayerFactory.***

```
//! Template Specializations for each data type

template class TypedPlayer< CImuData >;
template class TypedPlayer< CLaser2dData >;
template class TypedPlayer< CLocData >;
template class TypedPlayer< CDcxData >;
template class TypedPlayer< CGyroData>;
template class TypedPlayer< CLoc6DData >;
template class TypedPlayer< CGpsData >;
template class TypedPlayer< CNewClassData >;
```

Tabla 5.5: ***LogPlayer – Añadir nueva clase de comunicaciones en LogPlayer.cpp (PlayerFactory.h).***

```
class TimeReference{

public:
        TimeReference();
        double toTimeReference( timespec a );
        double getClockTime();
        double nanoSleep( double );

protected:
        timespec bigbang;
};
```

Tabla 5.6: ***LogPlayer – Declaración clase TimeReference (timeutil.h).***

5.2.4. TypedPlayer<T>

La clase TypedPlayer<T> hereda de la clase LogPlayer, creando un hilo para la generación y transmisión de un tipo de datos por su puerto YARP correspondiente.

Contiene las funciones para la lectura del fichero de log y para la transmisión de la estructura de datos correspondiente.

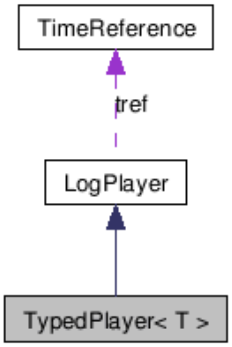


Figura 5.2: LogPlayer - Herencia TypedPlayer<T>.

El mecanismo de sincronización queda reflejado en el apéndice C.

```
template < typename T >
class TypedPlayer: public LogPlayer {
public:
        TypedPlayer(std::string portName, std::string
           fileName, double discardLapse = 0.05);
        ~TypedPlayer();

protected:
        bool readNext(timespec* nextTime);
        void send();
        std::string portOutputName;
        std::ifstream fileStream;
        yarp::os::BufferedPort<T> portOutput;
};
```

Tabla 5.7: ***LogPlayer – Declaración clase TypedPlayer (LogPlayer.h).***

5.3. Variables

En este caso recogemos una lista con las variables principales utilizadas en el bucle principal de la aplicación para posteriormente realizar una breve descripción de las mismas.

```
1 double timeOffset = atoi(argv[2]);
2 timespec startTime, playerStartTime;
3 timespec now, totalOffset;
4
5 typedef list<LogPlayer *> LogPlayerCollection;
6 LogPlayerCollection players;
```

Tabla 5.8: ***LogPLayer – Lista de variables (main.cpp).***

- *double timeOffset.* Offset inicial de tiempo. Parámetro que se utiliza para saltar los segundos iniciales que existen en todos los logs, correspondientes al tiempo necesario para inicializar todos los sistemas y que el robot reciba una trayectoria correcta y empiece a ejecutarla.

- *timespec startTime.* Estructura timespec utilizada para almacenar la hora real del sistema, accedida mediante el identificador del reloj en tiempo real de todo el sistema CLOCK_REALTIME.

- *timespec playerStartTime.* Variable del tipo timespec que se utiliza para guardar el menor valor de timeSpec de los diferentes logs a reproducir.

- *timespec now.* Estructura timespec utilizada para almacenar la hora real del sistema, accedida mediante el identificador del reloj en tiempo real de todo el sistema CLOCK_REALTIME.

- *timespec totalOffset.* Estrcutura timespec que contiene el offset temporal que realmente se aplica a los diferentes reproductores.

- *LogPlayerCollection players.* Lista que contiene todos los logs (reproductores o players) que se van a reproducir.

Todas estas variables temporales se usan para asegurar el sincronismo en la reproducción de los diferentes logs. El mecanismo seguido para lograr este objetivo está recogido, como ya hemos comentado, en el Apéndice C de este mismo texto.

5.4. Descripción de la aplicación

5.4.1. Opciones línea de comandos

Las opciones que admite el programa por línea de comandos son totalmente estáticas. Debe introducirse como primer argumento el directorio donde se encuentran almacenados los ficheros de logs del experimento que se pretende reproducir y posteriormente un offset inicial de tiempo.

Dicho offset suele ser un valor de unos 12 segundos, el tiempo típico que se ha comprobado el sistema loguea sin que haya movimiento del robot.

Si la introducción de parámetros no es correcta la aplicación nos recuerda el formato a utilizar, tal y como vemos en la figura 5.3.

Figura 5.3: Salida error consola LogPlayer.

El programa recibe la información de los módulos que va a simular a través de un fichero de configuración que es descrito en la sección siguiente.

5.4.2. Fichero de Configuración

En el siguiente ejemplo se muestra una configuración en la que se reproducirán los datos de los módulos GPS, DCX y del láser trasero izquierdo HOKUYO LEFT.

Al ejecutar la aplicación con el fichero de configuración arriba indicado, se muestra por consola la creación de cada uno de los players con su información respectiva de clase de comunicaciones, nombre del puerto YARP, fichero de log y el correspondiente puerto YARP activo, tal y como vemos en la figura 5.4.

Para la correcta lectura del fichero de configuración se ha utilizado la librería *sparser* creada por D. Pablo Soriano Tapia, miembro del GRVC, para lectura de datos con formato estructurado en ficheros de texto plano.

```
logPlayer {

        log {
                class = CGpsData;
                port = /romeo/gps/data;
                file = gps.log;
        }
        log {
                class = CDcxData;
                port = /romeo/dcx/data;
                file = dcx.log;
        }
        log {
                class = CLaser2dData;
                port = /romeo/hokuyoLeft/data;
                file = hokuyoLeft.log;
        }
}
```

Tabla 5.9: ***Fichero de Configuración logPlayer.conf.***

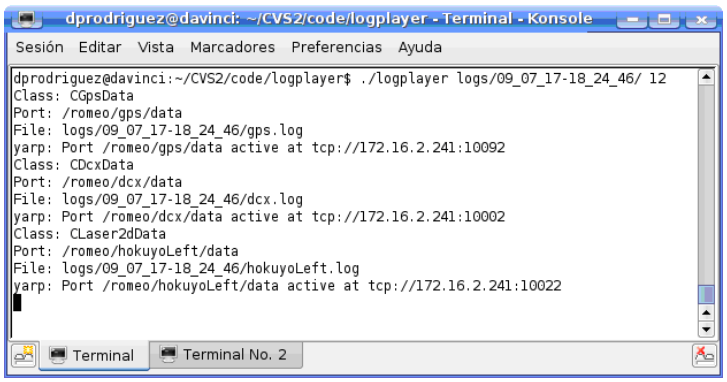


Figura 5.4: Salida correcta consola LogPlayer.

Capítulo 6

Conclusiones y Resultados

6.1. Conclusiones

Tras la realización de este proyecto se ha llegado a las siguientes conclusiones:

- La robótica es un elemento cada vez más presente en la vida cotidiana de las personas. Proyectos de investigación como URUS vienen demostrando que la robótica puede ser integrada en la vida diaria de las personas como una potente herramienta que flexibilice y facilite la vida de las mismas. Incluso en el mercado actual tenemos ejemplos de integración de robots en nuestro entorno, como el caso del famoso robot limpiador Roomba, de la empresa iRobot, que actualmente comercializa la tercera generación de su producto con un gran éxito comercial. La integración de la robótica en los entornos urbanos más allá de los límites de nuestro hogar será una realidad en los años venideros.

- Se ha comprobado la viabilidad del uso de las librerías gráficas Qt como una herramienta de desarrollo potente y versátil para la realización de interfaces de usuario. Aprovechando la cantidad de herramientas de desarrollo explicadas en el capítulo 3 la realización de interfaces de usuario, en cualquier grado de complejidad, es una tarea que se ve ampliamente facilitada al usar Qt.

- La utilización de la librería YARP como middleware para comunicaciones ha supuesto un gran acierto, ya que se ha consolidado como una poderosa herramienta que libera de gran cantidad de trabajo para el desarrollo de otros aspectos del robot, ofreciendo una abstracción de las comunicaciones en el concepto de puerto, muy fácil de implementar y de integrar en el resto del sistema. Se puede entender YARP como el *sistema circulatorio* del robot.

- La utilización de la herramienta LogPlayer ha resultado de gran utilidad para la simulación y el refinamiento de los algoritmos diseñados en la fase final del proyecto URUS.

- Personalmente, haber formado parte del equipo de desarrollo de URUS ha sido una gran experiencia formativa. Me ha permitido adquirir y desarrollar conocimientos en desarrollo en C++ bajo entornos linux, integración de librerías de desarrollo diversas, conocimientos hardware sobre sensores, desarrollo de drivers, etc., todo ello de la mano de unos excelentes profesionales y personas.

6.2. Futuras líneas de desarrollo

La interfaz gráfica Romeo HMI ha sido diseñada y realizada pensándose como una herramienta para el manejo del robot, pero orientada siempre para los desarrolladores del sistema. Por tanto, podemos pensar en dos vías de desarrollo bastante claras a partir de este proyecto. En primer lugar mejoras que no se llegaron a implementar en la interfaz actual o solucionar o mejorar problemas que aparecieron en el desarrollo de la misma y por otro lado la realización de una interfaz gráfica orientada para usuarios finales.

6.2.1. Mejoras en la interfaz gráfica actual

- *Integración de Qt y openCV.* Uno de los mayores problemas encontrados fue la integración entre Qt y openCV. La librería openCV, utilizada en módulos como sensormap o elevationmap, utiliza una serie de formatos gráficos de imagen propios, que deben ser convertidos a bitmaps para poder ser utilizados posteriormente por Qt. Esto deriva en un consumo excesivo de tiempo de procesado. Desarrollar una clase que implemente esta integración entre Qt y openCV facilitaría muchos trabajos amén de abrir nuevas posibilidades de visualización y de desarrollo.

- *Integración Google Maps.* En espacios donde el módulo GPS estuviese disponible sería muy interesante programar un widget que haciendo uso de la API de google maps nos permitiese aprovechar las funcionalidades de esta aplicación web, como por ejemplo mostrar la ubicación de Romeo-4R en una imagen satelital, cálculo de rutas, búsqueda de información en los alrededores del robot, etc.

6.2.2. Interfaz gráfica para usuarios finales

El siguiente paso lógico en la evolución del trabajo realizado en URUS por parte del GRVC consistiría en realizar desde cero una interfaz gráfica orientada a los

usuarios finales, donde se facilite la selección de las labores que pueda realizar el robot, taxi o seguimiento, ocultando al usuario información que realmente no necesita conocer, como la generación de los mapas de elevación, posiciones estimadas del robot, etc., mostrando un funcionamiento totalmente transparente al mismo.

6.3. Resultados obtenidos en Proyecto URUS

Como se comentó en la introducción, el trabajo realizado en este proyecto fin de carrera, se enmarca dentro de un proyecto de investigación Europeo, el proyecto URUS, por lo que para finalizar el contenido de esta memoria, recogemos algunos de los resultados más importantes que fueron obtenidos, por el equipo de trabajo del GRVC, dirigido por D. Aníbal Ollero Baturone y D. Luis Merino Cabañas, director de este proyecto fin de carrera y recogidos en [19] y [13].

Las áreas peatonales y urbanas, como por ejemplo un campus universitario, presentan una serie de retos para robots móviles tipo coche, tales como escaleras, rampas, pequeños bordillos, árboles, etc., a lo que debemos añadir personas en movimiento, por lo que la evitación de obstáculos dinámicos es un requerimiento esencial en este tipo de sistemas.

Por tanto, para una correcta recreación del entorno se hace necesario construir un modelo tridimensional del mismo, a partir del cual, el robot Romeo-4R es capaz de construir un mapa de transversabilidad (tal y como vimos en el apartado 2 de la memoria), en el que se indica para cada celda del mapa si ésta es transversable o no.

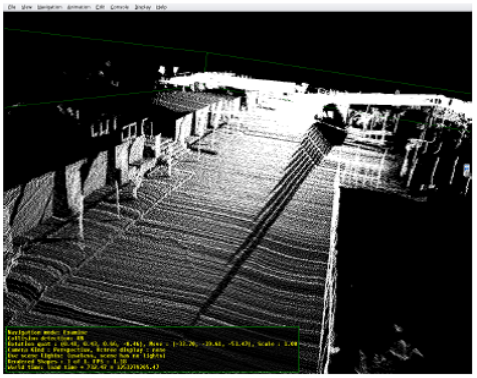


Figura 6.1: Proyecto URUS. Recreación entorno 3D.

Utilizando la videocámaras a bordo del robot Romeo-4R podemos obtener la posición de personas (lo que se puede emplear en tareas de guiado o de interacción con personas). Los algoritmos para determinar esta posición se basan en una combinación de detección y seguimiento.

El algoritmo de seguimiento se basa en la técnica mean-shift. Un algoritmo de detección de caras [26] se ejecuta en paralelo; los resultados de ambas técnicas se combinan, de modo que cuando el seguimiento se pierde, el detector permite recuperar a la persona.

Como resultado, el robot puede estimar la posición de la cara de una persona en el plano de la imagen. Aplicando razonamientos geométricos, sería posible estimar la posición de la persona con relación al robot.

Figura 6.2: Proyecto URUS. Reconocimiento facial.

Otro elemento de apoyo existente en el escenario de los experimentos, comentado en la introducción de este texto, es una red de cámaras IP fijas, las cuales pueden cubrir parte del entorno, y permiten el seguimiento de objetos de interés. Esta información puede combinarse con la propia información que obtienen los robots para mejorar la percepción del entorno.

El sistema empleado es capaz de seguir objetos de interés tanto dentro de una cámara como entre cámaras diferentes sin necesidad de una calibración explícita e incluso en ausencia de solape en sus campos de visión. La técnica ha sido desarrollada por Gilbert y Bowden [16].

La figura 6.3 muestra un ejemplo en el que la misma persona es seguida por 3 cámaras que tienen muy poca o ninguna superposición.

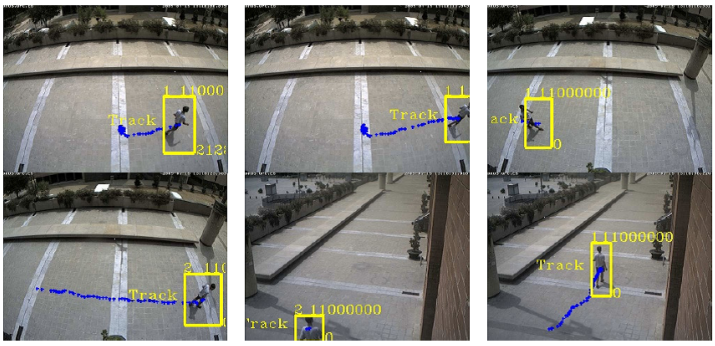


Figura 6.3: Proyecto URUS. Red cámaras IP - Identificación y seguimiento.

Para el guiado de una persona, el robot debe tener capacidad de navegación. La

figura 6.4 muestra algunos resultados de navegación de Romeo en el escenario del proyecto URUS. Uno de estos experimentos corresponde a una hipotética misión de guiado, en la que el robot debe guiar a una persona de un punto a otro del mapa.

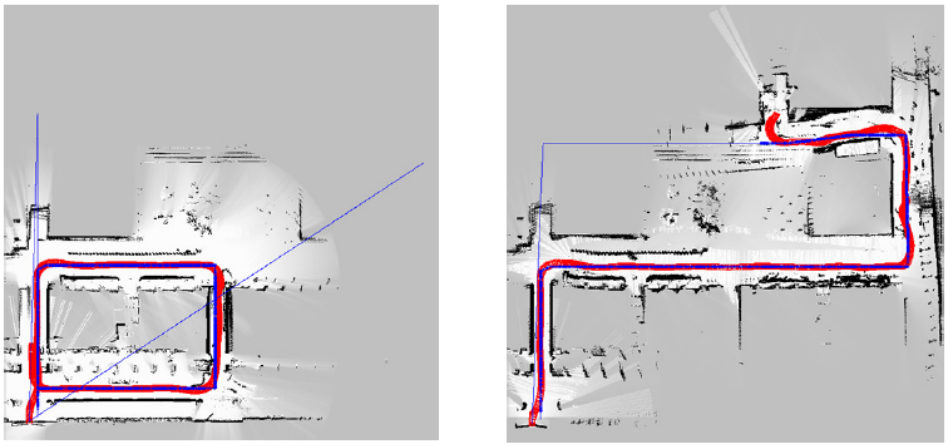


Figura 6.4: Proyecto URUS. Experimentos Navegación Romeo-4R.

Para poder guiar a alguien, es necesario estimar la posición de dicha persona para ajustar el movimiento del robot al de la persona. Tal y como se comentó igualmente en la introducción, para ayudar en las tareas de reconocimiento del entorno disponemos de una red de sensores desplegada por toda la zona de los experimentos, compuesta por decenas de nodos estáticos capaces de monitorizar diversas variables del entorno.

Al mismo tiempo, esta red se emplea para estimar la posición de un nodo móvil de la misma red a partir de la potencia de señal radio recibida por los nodos estáticos. Así podemos conseguir medidas adicionales para mejorar el seguimiento de personas, tal y como vemos en la figura 6.5.

El algoritmo para la estimación de la posición del nodo móvil está basado en un filtro de partículas. En dicho filtro, la estimación actual de la posición del nodo se describe por un conjunto de partículas, cada una de las cuales representa una hipótesis sobre la posición real de la persona que lleva el nodo. En cada iteración del filtro, modelos de movimiento de la persona e información del mapa se emplea para predecir la posición futura de las partículas. El uso de mapas permite descartar movimientos poco probables.

Cuando los nodos estáticos reciben nuevos mensajes del nodo móvil, el peso de las distintas partículas se ajusta teniendo en cuenta la potencia recibida.

Utilizando modelos de propagación de la señal radio, es posible calcular la

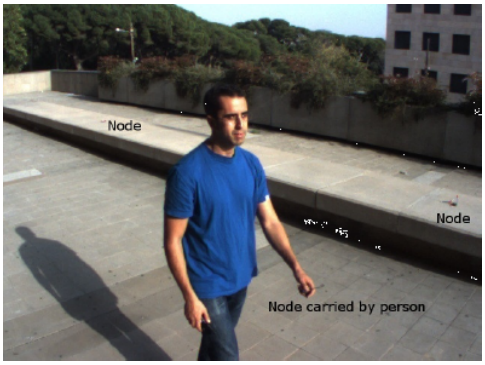


Figura 6.5: Proyecto URUS. Redes de Sensores (I).

verosimilitud de una hipótesis (partícula) a partir de la distancia al nodo receptor según dicha hipótesis [13].

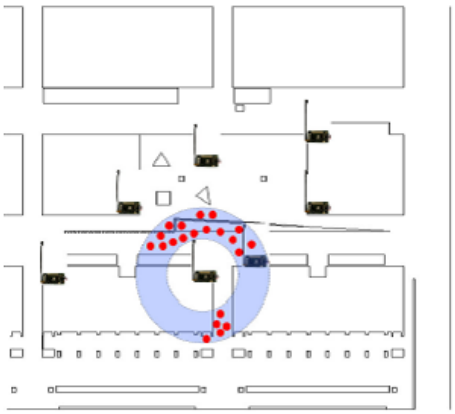


Figura 6.6: Proyecto URUS. Redes de Sensores (II).

Cada mensaje restringe la posición de las partículas a un anillo circular alrededor del nodo receptor en función de la potencia recibida, situación representada en la figura 6.6

Como resultado, el filtro proporciona estimaciones de la posición 3D del nodo móvil con una precisión que puede llegar a ser de 1 metro en función de la densidad de nodos de la red. La figura 6.7 muestra la evolución de las partículas para un experimento de guiado.

La figura 6.8 muestra la posición estimada, comparada con la posición del robot que la guía (que se encuentra unos metros más adelante).

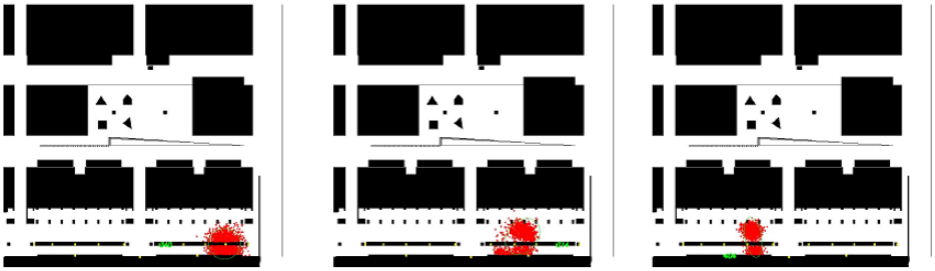


Figura 6.7: Proyecto URUS. Experimento de guiado (I).

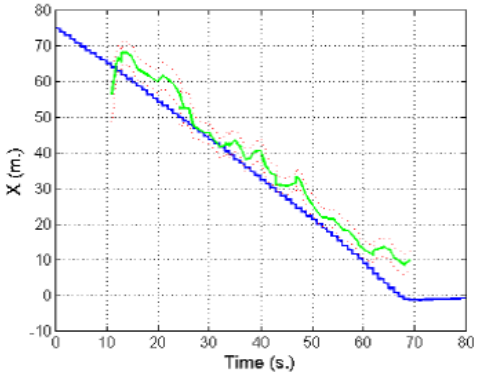


Figura 6.8: Proyecto URUS. Experimento de guiado (II).

6.4. Uso de las lecciones aprendidas en posteriores proyectos

La positiva experiencia en el desarrollo de interfaces gráficas basadas en Qt se ha visto plasmada en posteriores proyectos realizados por el *Grupo de Robótica, Visión y Control*, donde con los conocimientos adquiridos en el desarrollo de este proyecto fin de carrera, se han extendido las posibilidades de las librerías Qt, integrándolas en entornos multirobot y con otras librerías gráficas tales como Open Scene Graph (librería para gráficos 3D) y Open Street Map (librería para servicios de información geográfica), tal y como la Ground Control Station desarrollada para trabajar con múltiples UAVs en el entorno del Proyecto INTEGRA.

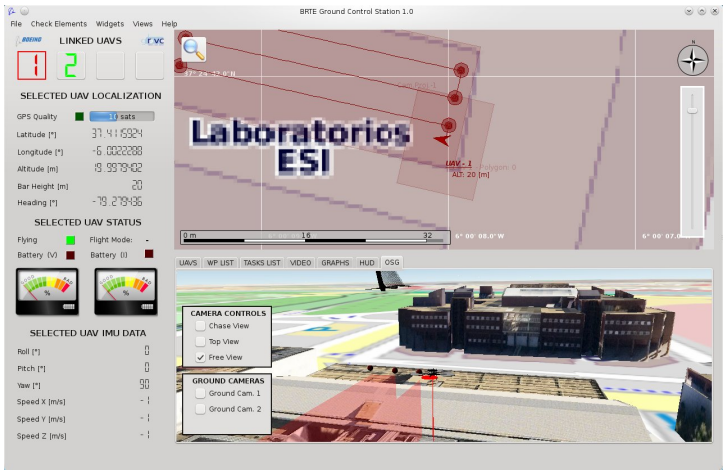


Figura 6.9: Qt Multi UAV Ground Control Station.

Apéndice A

Romeo HMI - Include dependency graph for main.cpp

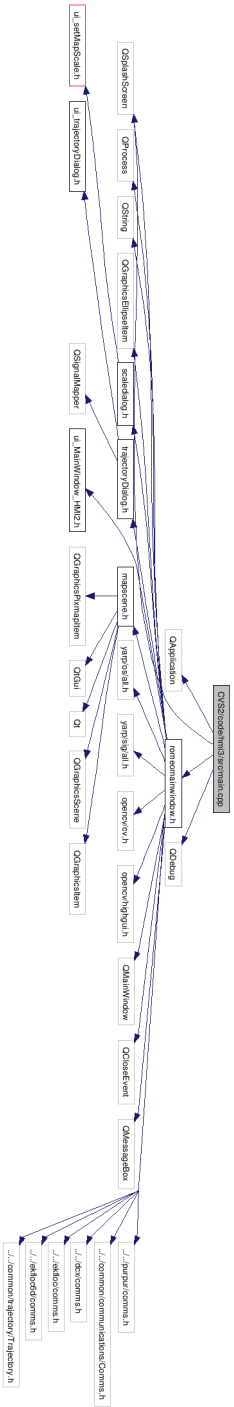


Figura A.1: Romeo HMI - Include dependency graph for main.cpp.

Apéndice B

LogPlayer - Include dependency graph for main.cpp

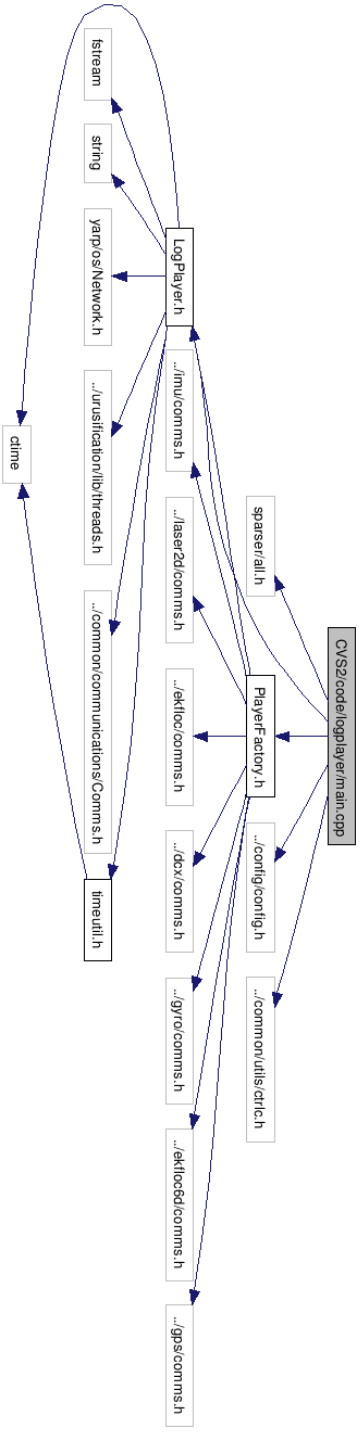


Figura B.1: LogPlayer - Include dependency graph for main.cpp.

Apéndice C

LogPlayer - Mecanismo de Sincronización de Logs

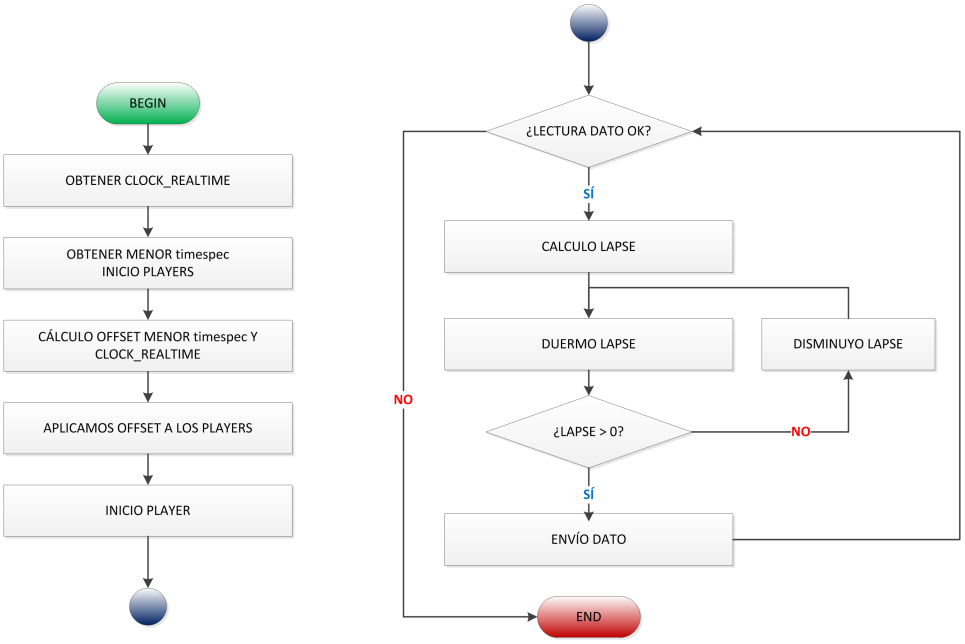


Figura C.1: LogPlayer - Mecanismo de Sincronización de Logs.

ACRÓNIMOS

ROMEO-4R	Robot Movil para Exteriores Cuatro Ruedas
ROMEO-3R	Robot Movil para Exteriores Tres Ruedas
EKF	Extended Kalman Filter: Filtro de Kalman Extendido
IMU	Inertial Movement Unit: Unidad de Movimiento Inercial
GPS	Global Positioning System: sistema de posicionamiento mundial
HMI	Human Machine Interaction: Interfaz/Interacción Hombre Máquina
HCI	Human Computer Interaction: Interfaz/Interacción Hombre Computador
IPO	Interacción Persona Ordenador
IDE	Integrated Development Environment: Entorno de Desarrollo Integrado
GUI	Graphic User Interface: Interfaz Gráfica de Usuario
NRS	Network Robot System: sistema de robots en red
YARP	Yet Another Robot Platform
QT	Librería Gráfica de Nokia
GTK	The Gimp Toolkit. Librerías gráficas
OPENCV	Librería de Visión por Computador
VXL	Vision – X – Library
CORBA	Common Object Request Broker Architecture
MIRO	Middleware utilizado en robótica
PLAYER/STAGE	Conjunto de Aplicaciones y librerías libres para robótica
CAMELLIA	Librería de visión por computador
GGA	Trama de datos GPS

Bibliografía

[1] Camellia. librería de código abierto para procesamiento de imágenes y visión por computador. [Online]. Available: http://camellia.sourceforge.net/

[2] Darpa grand challenge. [Online]. Available: http://www.darpa.mil/grandchallenge

[3] Doxygen. herramienta de generación automática de documentación para código fuente. [Online]. Available: http://www.stack.nl/~dimitri/doxygen/

[4] Gtk. the gimp toolkit. [Online]. Available: http://www.gtk.org/

[5] Miro. robotic middleware. [Online]. Available: http://miro-middleware.berlios.de/

[6] Opencv. librería para visión por computador. [Online]. Available: http://opencv.willowgarage.com/wiki/

[7] The player project. aplicaciones libres para robótica. [Online]. Available: http://playerstage.sourceforge.net/

[8] Proyecto urus: Ubiquitous networking robotics in urban settings. [Online]. Available: http://www.urus.upc.es

[9] Vxl. librerías c++ para visión por computador. [Online]. Available: http://vxl.sourceforge.net/

[10] Yarp. yet another robot plattform. [Online]. Available: http://eris.liralab.it/yarp/

[11] (2008) Qt. documentación de referencia. [Online]. Available: http://doc.trolltech.com/4.4/

[12] R. Baecker and W. Buxton, *Readings in human-computer interaction: A multidisciplinary approach*. San Mateo, CA: Morgan Kaufmann Publishers, 1987.

[13] F. Caballero, L. Merino, P. Gil, I. Maza, and A. Ollero, *A probabilistic framework for entire WSN localization using a mobile robot*. Robotics and Autonomous Systems, 2008.

[14] R. Conlter, *Implementation of the pure pursuit path tracking algorithm*. The Robotics Institute Carnegie Mellon University, 1992.

[15] A. Fod, A. Howard, and M. Mataric, *A laser-based people tracker*. Proceedings. ICRA '02. IEEE International Conference, 2002.

[16] A. Gilbert and R. Bowden, *Incremental Modelling of the Posterior Distribution of Objects for Inter and Intra Camera Tracking*. Proc. BMVC'05, Oxford UK, 2005.

[17] T. Hewett, R. Baecker, and et al., *Curricula for human–computer interaction*. Association for computing machinery (ACM), 1992.

[18] F. MacIntyre, K. Estep, and J. Sieburth, *The cost of user-friendly programming: MacImage as example*. Journal of FORTH Application and Research 6(2), 1990.

[19] L. Merino, J. Capitán, and A. Ollero, *Robótica cooperativa e integración con sensores en el ambiente. Aplicaciones en entornos urbanos*. Workshop Robot'09, Barcelona (Spain), 2009.

[20] G. Metta, P. Fitzpatrick, and L. Natale, *YARP: Yet Another Robot Platform*. International Journal on Advanced Robotics Systems, 2006.

[21] J. Minguez and L. Montano, *Nearness diagram (nd) navigation: Collision avoidance in troublesome scenarios*. IEEE Transactions on Robotics and Automation, vol. 20-nº1, 2004.

[22] B. Myers and M. Rosson, *Survey on User Interface Programming*. Proceedings of ACM CHI'92 Conference on Human Factors in Computing Systems, 1992.

[23] A. Ollero, *Robótica : manipuladores y robots móviles*. Marcombo, 2006.

[24] F. Real, *Navegación reactiva y seguimiento en entornos urbanos mediante percepción láser y mapas probabilísticos*. Universidad de Sevilla, Escuela Superior de Ingenieros. Proyecto fin de Máster, 2008.

[25] J. Rosenblatt, *DAMN: A Distributed Architecture for Mobile Navigation*. PhD thesis, Robotics Institute, Carnegie Mellon University, Pittsburgh, PA, 1997.

[26] P. Viola and M. Jones, *Robust Real-Time Face Detection*. International Journal of Computer Vision, 2004.