ESCUELA SUPERIOR DE INGENIERÍA INFORMÁTICA

INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

PROYECTO FIN DE CARRERA

Plataforma de desarrollo de aplicaciones robóticas: **Jderobot 5.1**

Autor: Maikel González Baile

Tutor: Prof. Dr. José María Cañas Plaza

Curso académico 2012/2013

Lo que tenemos que aprender lo aprendemos haciendo.
Aristóteles

Agradecimientos

Quisiera aprovechar esta pequeña sección para dar las gracias a todas las personas que me han ayudado a lo largo de mi carrera puesto que sin ellas nada de esto hubiese sido posible.

En primer lugar agradecérselo a mis padres, con los que he compartido todas mis experiencias a lo largo de estos años motivándome en las malas y disfrutando en las buenas, como si de un equipo se hubiese tratado, gracias.

En segundo lugar a Beatriz, por su paciencia, comprensión y apoyo en todos esos momentos malos y porque marcó un antes y un después en mi vida, gracias.

En tercer lugar mis amigos, sin los cuales todos estos años no hubiesen sido tan especiales y con los que he compartido todo, desde la felicidad por superar los objetivos, hasta la desilusión de quedar alguno en el camino, gracias.

Y por último y no menos importante, mi tutor José María Cañas, por darme la oportunidad de formar parte de esta comunidad donde con esfuerzo y sacrificio he crecido como ingeniero, siempre de la mano de sus consejos, sabiduría, paciencia y apoyo, gracias.

Muchas gracias a todos.

Resumen

Este proyecto presenta la versión 5.1 de la plataforma Jderobot, entorno de software orientado al desarrollo de aplicaciones robóticas, domóticas y de visión artificial. Desde su nacimiento, la plataforma ha evolucionado junto a los aportes realizados por parte de la comunidad de robótica de la Universidad Rey Juan Carlos, dando lugar a un proyecto de software libre cuya implementación supera actualmente las 50.000 líneas de código repartidas entre librerías, herramientas y *drivers*.

La experiencia de esa comunidad utilizando Jderobot permite incorporar nuevas funcionalidades, o mejorar las ya existentes, liberando nuevas versiones con cierta periodicidad. La nueva versión, *5.1*, incorpora herramientas como *CMake*, que mejora la gestión, mantenimiento y usabilidad del proyecto. Otro aporte es la creación de *paquetes debian*, que facilitan a los usuarios su instalación, concretamente en distribuciones punteras como Ubuntu 12.04 y Debian Testing. Además, se han desarrollado nuevas aplicaciones y mejorado otras como *introrob*, utilizada como herramienta docente en las diferentes asignaturas de robótica.

Los aportes integrados en *Jderobot 5.1* dotan este proyecto de un fuerte carácter evolutivo y heterogéneo. Cabe destacar la existencia de usuarios reales, como los alumnos de los Máster de Visión y el Máster de Sistemas Telemáticos e Informáticos u otros desarrolladores del grupo de robótica. Todos han hecho uso de las nuevas mejoras y han proporcionado importante *realimentación*, utilizada para conseguir que éstas fueran cada vez más usables, robustas y funcionales.

Índice general

1. Introducción **5**

1.1. Robótica 5

1.1.1. Componentes de los robots 6

1.1.2. Aplicaciones de la robótica 8

1.1.3. Investigación en robótica 11

1.2. Software para robots 13

1.2.1. Player/Stage 14

1.2.2. ROS 15

1.2.3. Orca 16

1.3. Jderobot 16

1.3.1. JDE (Jerarquía Dinámica de Esquemas) 18

1.3.2. jde.c 19

1.3.3. jde+ 20

1.3.4. jdeneo.c 21

1.3.5. Jderobot 4.3 22

1.3.6. Jderobot 5.0 22

1.3.7. Jderobot 5.1 23

2. Objetivos **25**

2.1. Descripción del problema 25

2.2. Requisitos 27

2.3. Metodología de desarrollo 27

2.4. Plan de trabajo 29

3. Infraestructura **31**

3.1. Bibilioteca GTK+ y Glade 31

3.2. Biblioteca ICE 32

3.3. Biblioteca CWIID 33

3.4. Simulador Gazebo 34

3.5. Herramienta DPKG 35

3.6. Herramienta CMake 36

4. Descripción informática **37**

4.1. Componentes 37

4.1.1. Basic Component 38

4.1.2. Driver Gazeboserver 41

4.1.3. Teleoperator 47

4.1.4. Introrob 50

4.1.5. WiimoteServer y WiimoteClient 59

4.2. Compilación de Jderobot 5.1 con CMake 63

4.2.1. Cómo emplea CMake un usuario de Jderobot 68

4.2.2. Como emplea CMake un desarrollador de Jderobot 69

4.2.3. Pruebas de verificación 75

4.3. Paquetes debian de Jderobot 5.1 75

4.3.1. Pruebas de verificación 81

5. Conclusiones **83**

5.1. Conclusiones 83

5.2. Trabajos futuros 87

Bibliografía **89**

Índice de figuras

1.1. (a) Robot integrante de una cadena de montaje de automóviles (b) AdeptQuattro realizando tareas de envasado 6

1.2. (a) Sistema robótico adquirido por Mercadona. (b) Robots de Amazon gestionando un almacén. (c) Robot FlexPicker utilizado para el envasado de tortitas. 9

1.3. (a) Robot Da Vinci. (b) Sistema ROBODOC. 9

1.4. Sonda Spirit. 10

1.5. (a) Ladrillo Lego. (b) Robot Roomba. 10

1.6. (a) Robot empleado en el accidente de la centrla nuclear de Fukushima (b) Robot empleado en la labores de limpieza del *Prestige* 11

1.7. (a) Toyota Prius modificado por Google para conducir de manera autónoma. (b) Robot Asimo de Honda. 12

1.8. Robot Nao 12

1.9. Stage simulando múltiples robots. 14

1.10. Patrón típico de un esquema perceptivo (a) y de un esquema motor (b) en JDE 18

1.11. Arquitectura software con servidores y clientes. 19

1.12. (a) Interfaz gráfica con XForms en jde.c (b) Interfaz gráfica con GTK en jdeneo.c 22

2.1. Modelo en espiral. 28

3.1. Ejemplo de la interfaz de un componente con GTK+ 32

3.2. Ejemplo de Gazebo 1.0 35

4.1. Comunicación cameraserver-basic component 38
4.2. Diseño del nuevo esqueleto para componentes de Jderobot 5.1. 40
4.3. (a) Robot pioneer simulado en Gazebo (b) Robot pioneer real 42
4.4. Diseño del nuevo Gazeboserver (inferior) frente al antiguo (superior) conectándose a un componente ejemplo (introrob). 43
4.5. (a) Definición del chassis del pioneer (b) Definición del sensor láser (c) Definición del joint para el láser 45
4.6. *teleoperator* en ejecución. A la izquierda el simulador Gazebo y a la derecha las múltiples ventanas en las que se divide la GUI del componente. 48
4.7. Nuevo esqueleto aplicado en teleoperator. 49
4.8. Flujo de sincronización entre hilo de control y procesamiento. 50
4.9. Interfaz gráfica de Gazebo (esquina inferior izquierda) e introrob. 51
4.10. Esquema de comunicación Introrob-Gazebo. 52
4.11. Diagrama de clases Introrob. 54
4.12. Acceso a sensores y actuadores. 56
4.13. Métodos gráficos. 57
4.14. Métodos auxiliares. 58
4.15. Práctica de un alumno del Máster de Visión Artifical con Introrob. 59
4.16. (a) Representación del acelerómetro del wiimote (b) Wiimote 61
4.17. WiimoteClient & WiimoteServer. 63
4.18. Árbol de directorios de Jderobot y diseño de CMake. 66
4.19. Jerarquía de paquetes en Jderobot 5.1. 77

Capítulo 1

Introducción

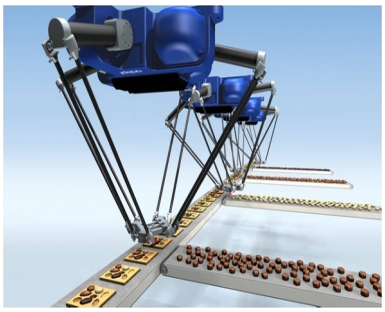
1.1. Robótica

Según la Real Academia Española, el término robótica se define como *técnica que aplica la informática al diseño y empleo de aparatos que, en sustitución de personas, realizan operaciones o trabajos, por lo general en instalaciones industriales*, donde los aparatos mencionados son los robots, que a su vez los definen como *(1) ingenio electrónico que puede ejecutar automáticamente operaciones o movimientos muy varios. (2) autómata.* El término robot debe su origen a la palabra checa *robota* (que significa trabajo forzado), usada por el escritor checoslovako Karel Capek, quién publicó en 1923 la novela Rossum's Universal Robots en la cual el autor hace uso de este término para referirse a seres mecánicos capaces de llevar a cabo las instrucciones dadas por su creador.

Las anteriores definiciones están fuertemente ligadas al carácter industrial que siempre ha acompañado a la robótica, puesto que es el sector que ha impulsado su crecimiento durante décadas y donde el ser humano siempre ha tratado de usar su ingenio para construir complejas máquinas que extendiesen o mejorasen sus capacidades. Por ejemplo, ya en el siglo XIX fueron creados los primeros telares automáticos capaces de realizar dibujos previamente programados en tarjetas perforadas. Sin embargo, no es hasta comienzos del siglo XX que la robótica comienza a abrirse un importante hueco en la industria, utilizándose para llevar a cabo aquellas tareas peligrosas para el ser humano, o bien automatizando otras cuyo proceso consta de diversas acciones que se repiten contínuamente en el tiempo. De hecho, la mayoría de robots que existen en la actualidad se encuentran emplazados en fábricas, formando parte de grandes cadenas de montaje tales como las utilizadas para la fabricación de automóviles. Un hito paradigmático en este marco fue la

aparición en los 70 de los brazos articulados PUMA (Prgrammable Universal Manipulator for Assembly), utilizados para soldar, pintar, transporte de materiales, etc.

(a)



(b)

Figura 1.1: *(a) Robot integrante de una cadena de montaje de automóviles (b) AdeptQuattro realizando tareas de envasado*

Poco a poco los robots están cobrando un papel más importante, unas veces teleoperados por el hombre extendiendo así sus capacidades y otras de manera autónoma, es decir, capaces de desenvolverse por sí mismos en entornos desconocidos y parcialmente cambiantes sin necesidad de supervisión. Es en este segundo aspecto donde profundizaremos más en el resto del documento. Esta autonomía no nace de la nada, sino que es el fruto de incorporar la informática en esta disciplina, dando lugar al software de control automático.

1.1.1. Componentes de los robots

Si tratamos de profundizar más en el término, un robot puede ser considerado como un sistema complejo dotado de sensores, actuadores y procesadores que le otorgan capacidades básicas como percepción, acción, procesamiento y memoria para interactuar con el entorno y conseguir así ciertos objetivos.

Los **sensores** le permiten obtener información de su entorno o de sí mismo, de este modo su comportamiento podrá establecerse de acuerdo al entorno en el cual el robot se haya inmerso. Existe una gran cantidad de sensores: de luz, temperatura, ultrasónicos, infrarrojos, láser, etc.

Los **actuadores** son dispositivos que le permiten interactuar con su entorno y/o realizar movimientos, de este modo, los robots móviles se caracterizan por no encontrarse en una posición fija. Los actuadores pueden ser de diversos tipos, destacando entre ellos el uso de motores en ruedas por su capacidad de adecuarse a cualquier tipo de suelo, aunque también se pueden encontrar robots articulados por patas, o incluso con capacidad de volar y nadar.

El tercer componente fundamental son los **procesadores**, encargados de procesar los datos obtenidos por los sensores y de materializar los algoritmos de decisión que posteriormente se transformarán en acciones comandadas hacia los actuadores.

Todos estos elementos ofrecen la base sobre la que se asienta el software, elemento que tiene gran importancia dentro de un robot puesto que es el encargado de dotar a éste del comportamiento inteligente que le diferencia del resto de las máquinas. Al igual que el desarrollo de software en otro tipo de aplicaciones, el software orientado a la robótica atiende a una serie de requisitos específicos que vienen determinados por el comportamiento que debe presentar un robot, entre los cuales podemos destacar:

- *Agilidad*: un robot se encuentra en constante interacción con el entorno, lo cual conlleva que el robot necesite respuestas rápidas de manera contínua.
- *Multitarea*: puesto que un robot no realiza una única acción de manera simultánea, sino que en un momento dado es posible que esté obteniendo información de su entorno a través de los sensores, procesando datos y enviando éstos hacia sus actuadores.
- *Distribuido*: una forma de diseñar su software es dividiendo el procesamiento de los datos en diferentes nodos de cómputo, consiguiendo así aumentar su capacidad. Este hecho conlleva a su vez la necesidad de disponer de un sistema de comunicación, que permita la transimisión de los datos entre cada uno de los nodos.
- *Hardware heteogéneo*: existe una gran variedad entre los diferentes elementos materiales que componen un robot: cámaras, láser, infrarrojos, etc.

El diseño de este software se lleva a cabo a través de una *arquitectura* para dotar al robot de comportamiento ingeligente o autonomía, entendiendo por **arquitectura** a *la organización de sus capacidades sensoriales, de procesamiento y de acción para conseguir un repertorio de comportamientos inteligentes interactuando con un cierto entorno.* Esta organicación cobra mayor importancia cuanto más complejo es el sistema que gobierna el robot, llegando al punto de que un mal diseño de ésta puede acarrear el incumplimiento de los objtevos previamente fijados. En un robot, podemos hablar de dos tipos de arquitecturas, por un lado la *arquitectura hardware* cuya función es establecer el mejor diseño en el que se deberán disponer los diferentes dispositivos que conforman el robot en sí para obtener los datos del exterior e interactuar con el entorno de la manera más precisa posible. Por otro lado la *arquitectura software*, la cual debe definir cómo debe

ser el programa que dote de la capacidad resolutiva al robot para decidir qué hacer en qué momento.

1.1.2. Aplicaciones de la robótica

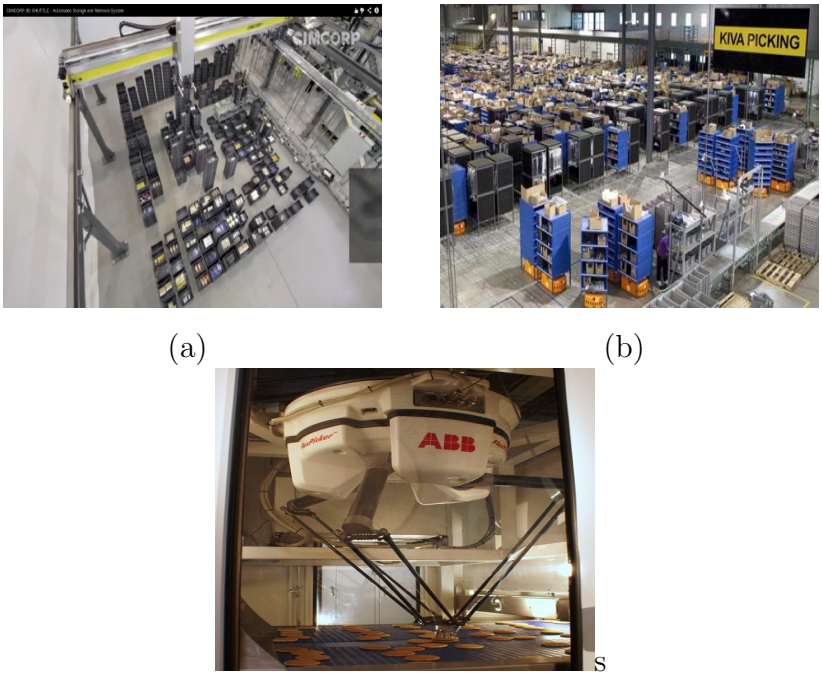
Desde su nacimiento, la robótica ha sido aplicada en numerosas situaciones donde su uso ha jugado un importante papel para la resolución de tareas inalcanzables por el hombre, o bien ayudándonos a resolverlas de forma más eficiente y ágil.

Si bien el impulso de la robótica en un principio está fuertemente ligado al avance industrial, en los últimos años su proliferación abarca sectores tan dispares como la medicina, la construcción, tareas militares, misiones espaciales, destinados al ocio y muchas aplicaciones que ayudan al hombre de a pie a realizar tareas tan comunes como conducir o ayudar a mantener limpio el hogar.

Todos estos sectores siguen acompañados por el contínuo avance en el sector industrial, donde podemos encontrar ejemplos tan próximos como la reciente adquisición por parte de Mercadona (figura 1.2(a)) de un complejo sistema robótico que permite la gestión de forma ágil de sus enormes almacenes. En esta misma línea, Amazon (figura 1.2(b)) adquirió la empresa Kiva Systems, la cual desarrolló una solución para mover volúmenes de material automáticamente con una flota de robots, capaces de desplazarse por todo el almacén trasladando dicho material. En este mismo sector tenemos otro ejemplo en la empresa ABB[1], dedicada a diseñar y construir brazos manipuladores, en particular de los robot *FlexPicker* (ver figura 1.2(c)), que son una familia de robots con múltiples usos, desde la soldadura o montaje hasta el envasado de alimentos.

En un campo tan importante como la medicina, la incorporación de robots se está consolidando gracias a que sus intervenciones eliminan los temblores humanos, permiten acceder a zonas inasequibles para los médicos y dañan menos tejido sano en las regiones afectadas, extendiendo así las capacidades de los propios cirujanos. Como ejemplo, en la cirugía laparoscópica, el uso del robot Da Vinci (figura 1.3(a)) ha demostrado sus capacidades siendo teleoperado por una persona, quién observa a través de las cámaras situadas en el robot la zona afectada del paciente y puede así manipular los brazos robóticos para llevar a cabo cirugías precisas. Otro ejemplo es el sistema ROBODOC (figura 1.3(b)), desarrollado por la empresa Surgical System Inc., utilizado con éxito en operaciones de sustitución de la cabeza del fémur por una prótesis de material biocompatible, siendo capaz

[1] http://www.abb.es/



(a) (b)

(c)

Figura 1.2: *(a) Sistema robótico adquirido por Mercadona. (b) Robots de Amazon gestionando un almacén. (c) Robot FlexPicker utilizado para el envasado de tortitas.*

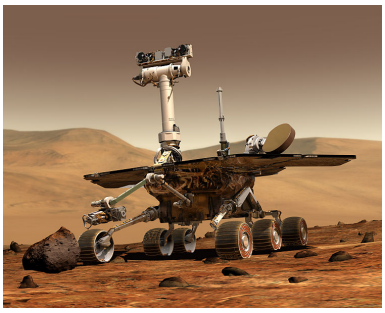
además de seleccionar la prótesis más adecuada y planificar la operación previamente a su ejecución.

(a) (b)

Figura 1.3: *(a) Robot Da Vinci. (b) Sistema ROBODOC.*

La industria espacial ha contribuído de manera significativa al uso de la robótica en tareas no convencionales. Los posibles beneficios que aporta su aplicación a este campo abarcan amplios ámbitos como la colaboración con la tripulación realizando trabajos de ensamblado o reparaciones extravehículares, a la realización de tareas hoy día imposibles para el ser humano, tales como exploraciones del sistema solar, por ejemplo. Los robots

pueden explorar lugares extraplanetarios, bien teleoperados desde la Tierra, o mediante comportamiento autónomo, interactuando con el entorno de manera independiente. El robot *Curiosity* o la sonda *Spirit* (figura 1.4(a)) son dos robots diseñados por la NASA para llevar a cabo misiones en Marte, obteniendo de él una gran cantidad de imágenes y análisis de muestras del terreno.



(a)

Figura 1.4: *Sonda Spirit.*

En los últimos años, la robótica está comenzando a entrar en el hogar por medio del ocio, como es el caso del ladrillo Lego (ver figura 1.5(a)), utilizado como juguete imaginativo por los más pequeños. Por otro lado también comienzan a entrar pequeños robots tales como el Roomba (ver figura 1.5(b)), capaces de aspirar el suelo de una casa guiado con sensores que le permiten realizar la tarea de manera autónoma sin chocar ni volcar. En el caso de este último, su éxito en ventas ha conseguido asentarlo en el mercado y ya son muchas las empresas que desarrollan su propio modelo de aspiradora robótica.

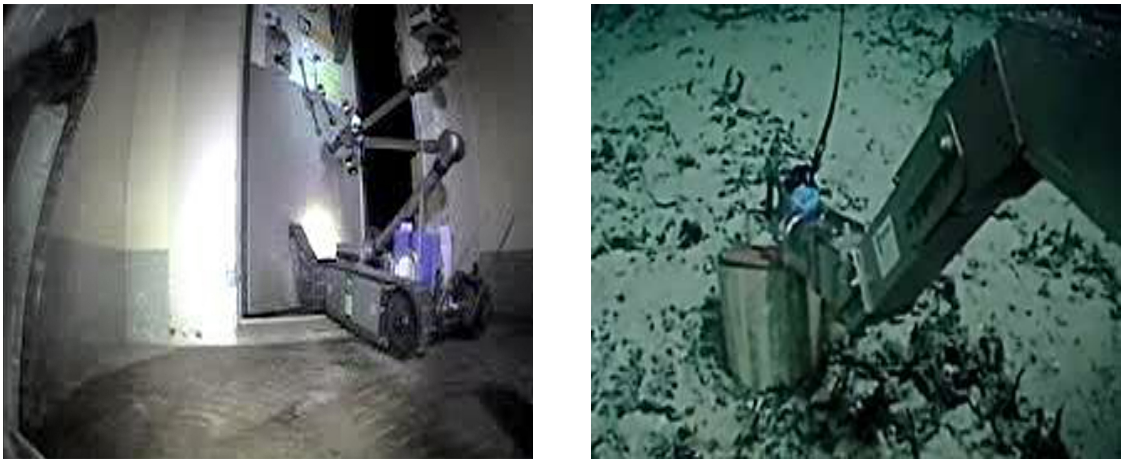
(a)

(b)

Figura 1.5: *(a) Ladrillo Lego. (b) Robot Roomba.*

Los robots también han tenido un papel importante ayudando al hombre tras catástrofes que han marcado la historia, como es el ejemplo de los *PackBot* (figura 1.6(a)), utilizados tras el accidente ocurrido en la central nuclear de Fukushima para medir la radiación o la

temperatura en los reactores de la planta nuclear. O el caso de *Frugo* (figura 1.6(b)) que ayudó en la inspección de las grietas existentes en el Prestige, barco hundido en la costas gallegas en el año 2002.



(a) (b)

Figura 1.6: *(a) Robot empleado en el accidente de la centrla nuclear de Fukushima (b) Robot empleado en la labores de limpieza del Prestige*

1.1.3. Investigación en robótica

Otro campo importante es el de la investigación, donde se desarrollan sistemas robóticos que si bien en la actualidad no son aplicados o utilizados por un sector genérico de la población, muestran las tendencias que podremos encontrar en un futuro. Entre este tipo de prototipos, se pueden destacar los vehículos desarrollados por Google, capaces de desplazarse sin necesidad de un conductor, es decir, de forma autónoma. Su director, Sebastian Thrun, fue el ganador del campeonato DARPA Grand Challenge en el año 2005, competición que consistía en el recorrido de un circuito por parte de estos coches no tripulados.

Otro proyecto importante es el del robot Asimo, llevado a cabo por la empresa automovilística Honda. En este caso se trata de un robot humanoide capaz de simular el comportamiento típico de una persona, como su forma de andar, correr, saltar, manipular objetos con las manos, transportar una bandeja, servir una bebida, entre otros tipos de acciones. Otorgar a un robot de todas estas capacidades le ofrece un carácter social que le permite servir de asistente personal, por ejemplo, a la población de avanzada edad. Gracias a su capacidad de desenvolverse con soltura en un recinto cerrado, el robot Asimo puede transportar material de un lugar a otro, servir como guía turístico u otras muchas tareas.

(a) (b)

Figura 1.7: *(a) Toyota Prius modificado por Google para conducir de manera autónoma. (b) Robot Asimo de Honda.*

El robot Nao (ver figura 1.8(a)) es utilizado actualmente en la competición de robots RoboCup[2], proporcionando una plataforma de desarrollo para la investigación y educación sobre inteligencia artificial.

(a)

Figura 1.8: *Robot Nao*

Todos estos hitos se deben al gran avance que ha experimentado la teconología en los últimos años, donde el hardware con el que contamos es cada vez más preciso y aumenta el rango de posibilidades que se puede conseguir combinándolo. Además, el aumento de procesamiento, y por ende, la capacidad de cómputo de los robots crece a una velocidad vertiginosa. Sin embargo no todo es potencia, y es cada vez más necesario el diseño de plataformas de programación que nutran a los desarrolladores de un amplio conjunto de

[2]RoboCup es un proyecto internacional fundado en 1997 para promover, a través de competencias integradas por robots autónomos, la investigación y educación sobre inteligencia artificial

herramientas para facilitar la implementación de sus ideas en los robots móviles actuales.

1.2. Software para robots

La autonomía de los robots no proviene de la nada, sino que es el resultado del desarrollo de complejos sistemas, infraestructuras y aplicaciones que dotan a éste de inteligencia autónoma. El desarrollo software de sistemas robóticos no difiere especialmente del desarrollo en otros ámbitos del software, donde el programador parte de una serie de requisitos y modela un diseño que finalmente será implementado.

Históricamente, el problema del desarrollo software de sistemas robóticos se abordaba con soluciones *ad-hoc*, es decir, su diseño e implementación estaba destinado a un modelo o robot específico. El propio fabricante era el que ofrecía los *drivers* para sensores y actuadores de un determinado robot y el programador hacía uso de ellos. De tal forma, que la posibilidad de portar un sistema de un robot a otro era mínima, siendo necesario realizar nuevamente todas las etapas que un sistema de estas características conlleva para su implementación.

En los últimos años, con el asentamiento de los fabricantes de robots y el aporte de numerosos grupos de investigación, comienzan a surgir plataformas más genéricas que permiten el desarrollo de aplicaciones robóticas de forma más eficiente y rápida. Esto facilita su aplicación en diferentes robots sin necesidad de reimplementar todo el sistema, o integrando nuevas características, algunas de las cuales son:

- *HAL (Hardware Abstraction Layer)*: o capa de abstracción al hardware, que permite a los desarrolladores abstraerse de los detalles a bajo nivel en la comunicación con los diferentes dispositivos de un robot.
- *Arquitectura software concreta*: la plataforma puede ofrecer un modelo que sirva como patrón a la hora de organizar los diferentes elementos que componen el software de un robot.
- *API (Application Programming Interface)*: las cuales proveen métodos cuya funcionalidad es común en el desarrollo de aplicaciones robóticas, y que además, ofrecen al desarrollador funciones de alto nivel que hacen más intuitivo el código a desarrollar.

Las últimas tendencias en el desarrollo de plataformas hacen uso de los componentes, los cuales se definen como pequeños módulos con una funcionalidad concreta que pueden

ser combinados para obtener un comportamiento más complejo. Además, la reutilización de estos módulos agiliza a los programadores la tarea de desarrollar software.

A continuación se muestran algunos de estos entornos presentes en la actualidad.

1.2.1. Player/Stage

Player/Stage [3] es un proyecto de software libre orientado a la investigación en el campo de la robótica. El nombre lo forman las dos partes en las que se divide el proyecto:

1. *Player*: Cuya principal funcionalidad es ofrecer una capa de abstracción hardware, es decir, es la parte que se encarga de comunicarse con el robot a bajo nivel y permitir al usuario la comunicación con éste a partir de código de una forma mucho más sencilla.

2. *Stage*: Entorno de simulación 2D de robots (ver figura 1.9). Su función es escuchar las instrucciones que el desarrollador le envía por medio de Player para posteriormente reflejar éstas sobre el robot simulado. Además, el robot simulado dispone de todos los sensores que incorpora un robot y ofrece los datos percibidos por ellos.

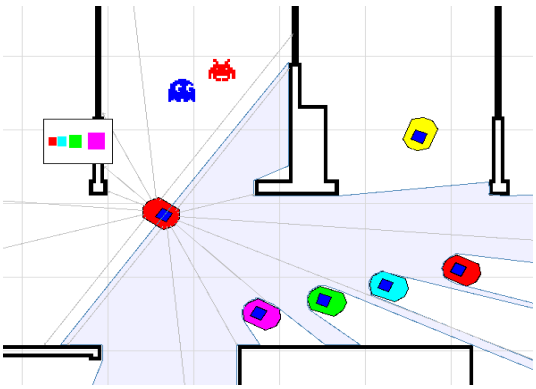


Figura 1.9: *Stage simulando múltiples robots.*

Otro componente añadido a posteriori pero soportado también por Player/Stage es el simulador 3D Gazebo, cuya funcionalidad es similar a la descrita en Stage pero ofreciendo un entorno de simulación mucho más rico en lo que a gráficos y texturas se refiere, creando un entorno casi real.

El entorno Player/Stage está diseñado para ser independiente del lenguaje de programación, pudiendo desarrollar sus aplicaciones (clientes Player) en lenguajes como

[3] http://playerstage.sourceforge.net/

C++, Java, Python o Tcl. Además, su carácter Cliente (Player) - Servidor (Stage) le ofrece la capacidad de crear sistemas distribuidos. Por otro lado, Player/Stage no ofrece una arquitectura definida para el desarrollo de sus aplicaciones, sino que únicamente provee una interfaz que facilita al desarrollador la tarea de definir él cómo será el diseño a implementar.

Player/Stage es considerado como uno de los entornos referentes en la robótica, gracias a su potente flexibilidad y su filosofía de código libre ha conseguido crear una gran comunidad que nutre día a día el proyecto.

1.2.2. ROS

ROS [4] es una plataforma también de software libre para el desarrollo de software orientado a robots que provee servicios típicos de un sistema operativo como la abstracción de acceso al hardware, control a bajo nivel de dispositivos, mecanismos de paso de mensajes entre procesos y una variada colección de herramientas comunmente utilizadas en un sistema robótico. Aunque fue originalmente desarrollado en 2007 bajo el nombre *switchyard* por el *Stanford Artificial Intelligence Laboratory*, actualmente es la empresa *Willow Garage* la encargada de su desarrollo y artífice del impulso que le ha llevado a ser uno de los entornos más completos a día de hoy.

Su diseño, en línea de las últimas tendencias, propone ser un sistema ligero cuya integración en cualquier equipo sea lo más sencilla e intuitiva posible. Todas sus librerías se han diseñado con la idea de ofrecer interfaces claros y limpios, ayudando al programador a centrarse en el desarrollo de algoritmos fácilmente integrables en la gran cantidad de robots que soporta este proyecto. Además, en el último año ROS ha decidido ayudar con financiación al simulador Gazebo y ha conseguido una gran integración con éste, obteniendo así un conjunto de herramientas muy completo que permite el desarrollo de complejas aplicaciones robóticas.

Otro punto clave de este proyecto es su gestión de todo el software que le rodea por medio de paquetes, facilitando así a los usuarios menos experimentados el acercamiento a su plataforma. De esta forma, el usuario tan sólo debe instalar los módulos que necesita para comenzar a usarlo en la mayor brevedad posible.

[4] http://www.ros.org/wiki/

1.2.3. Orca

Orca [5] es otra plataforma para el desarrollo de aplicaciones robóticas cimentada en el software libre. Está orientada a componentes, los cuales pueden ser ejecutados de manera independiente o combinándolos para formar aplicaciones más complejas. De esta forma permite una gran reutilización del código, no siendo necesario crear todos los elementos (componentes) que conforman un robot, sino utilizando los ya existentes. Además, ofrece una API que libera al desarrollador de la comunicación directa con el hardware (HAL).

Una característica importante de Orca es el uso de *ICE* para la comunicación remota de sus componentes. El uso de este *middleware*, además de facilitar la comunicación remota, permite crear sistemas distribuidos donde los componentes que conforman el sistema puedan estar desarrollados en lenguajes diferentes, ofreciéndole la capacidad de ser multiplataforma.

1.3. Jderobot

Una vez presentadas algunas de las plataformas pioneras en la actualidad, abrimos paso a una nueva que nace, en el año 2003, de una tesis doctoral ([Cañas Plaza, 2003]) y que recibe actualmente el nombre de Jderobot. Durante su evolución, la plataforma se ha visto nutrida gracias a los aportes realizados por la comunidad de robótica de la Universidad Rey Juan Carlos. Como plataforma, algunas de las características que aporta en la actualidad son:

- *Orientada a componentes*: entendemos como componente la unidad mínima de Jderobot. Si bien un componente es un proceso independiente con funcionalidad propia, lo más común es combinar varios de ellos para conseguir un comportamiento más complejo.
- *Distribuida*: estos componentes se encuentran distribuidos en distintas máquinas o como varios procesos en una misma. Su comuncación se lleva a cabo a través del middleware ICE.
- *Librerías*: Jderobot proporciona un conjunto de librerías propias que ofrecen métodos comunmente utilizados en el desarrollo de componentes para la plataforma.

[5] http://orca-robotics.sourceforge.net/

- *HAL*: existen componentes categorizados como *drivers* que se encargan de la comunicación con los deferentes dispositivos que posee un robot, abstrayendo al desarrollador de implementar dicha funcionalidad.
- *Software externo*: Jderobot no trabaja solo, sino que se apoya en una gran cantidad de software externo que le permite extender su funcionalidad. Entre este software podemos destacar:
  - *Simuladores*: Jderobot ofrece compatibilidad con algunos simuladores como Gazebo, a través de los cuales es posible poner a prueba los algoritmos desarrollados sin necesidad de disponer de un robot físico para ello.
  - *OpenCV*: esta librería ofrece a la plataforma potentes funciones para el procesamiento de imágenes.
  - *ICE*: a través de este middleware es posible realizar la comunicación de manera remota entre los diferentes componentes distribuidos.
  - *OpenNi y PCL*: algunos componentes existentes hacen uso de dispositivos de visión y profunidad como *Kinect*. Estas dos librerías ofrecen funciones para la manipulación de datos (nubes de puntos) y la captura de lecturas sensoriales.

Como proyecto, podemos destacar que:

- Se trata de un software con más de 50.000 líneas de código.
- Existe una comunidad de desarrolladores con la cual es posible interactuar a través de una lista de correo asociada a la plataforma, dónde es posible debatir temas, resolver dudas o compartir experiencias. Además, existe una wiki oficial desde la cual obtener información de cada uno de los elementos del proyecto.
- Jderobot es utilizado en la actualidad en un gran número de proyectos de investigación, marcado por la gran cantidad de alumnos que usan esta herramienta para el desarrollo de sus Trabajo Fin de Máster, diferentes Tesis Doctorales y/o Proyectos Final de Carrera.
- También es utlizado en aplicaciones concretas ya puestas en producción como son los proyectos de Eldercare [6] o Surveillance [7].

En las siguientes secciones se tratará cómo ha sido la evolución que ha experimentado esta plataforma desde su nacimiento hasta su última versión liberada, la *5.1*.

[6] http://jderobot.org/index.php/ElderCare
[7] http://jderobot.org/index.php/Surveillance

1.3.1. JDE (Jerarquía Dinámica de Esquemas)

Jderobot nace como una nueva arquitectura cognitiva (JDE), que ofrece una nueva forma de generar comportamiento autónomo para un robot, articulándose para ello en tres puntos claves. La fragmentación del comportamiento en pequeñas unidades denominadas esquemas, cuya cuantización facilita la reutilización de partes de la arquitectura. La clara diferenciación del problema común, generar comportamiento, en percepción y control, generando así esquemas destinados a modelar la percepción y otros para el control. Y por último, la combinación de estos esquemas en jerarquías.

Se define esquema como un flujo de ejecución independiente con un objetivo. Puesto que esta arquitectura trata la percepción y el control como dos objetivos imbricados pero diferentes, existirán esquemas de percepción y control respectivamente. Mientras que los perceptivos elaboran representación a partir de los diferentes sensores de un robot, los de control, también llamados esquemas motores, se encargan de comandar las decisiones procesadas a los actuadores.

Los esquemas son flujos de ejecución iterativos, es decir, realizan una determinada tarea de forma cíclica en el tiempo, siendo posible su interrupción en cualquier momento, así como su reactivación. Esta característica propia es destacable en un ámbito donde no todas las acciones y percepciones son útiles en todo momento, con lo que existir la posibilidad de dejar algunos de ellos en un estado dormido es útil para liberar de cierta carga computacional al sistema en esas situaciones donde estos datos son irrelevantes. Además, otra posibilidad es la modulación de estos esquemas a través de parámetros que sesgan ligeramente su funcionamiento.

Por otro lado, la fragmentación de la percepción y el control en unidades mínimas modulables facilita la reutilización de esquemas para diferentes comportamientos.

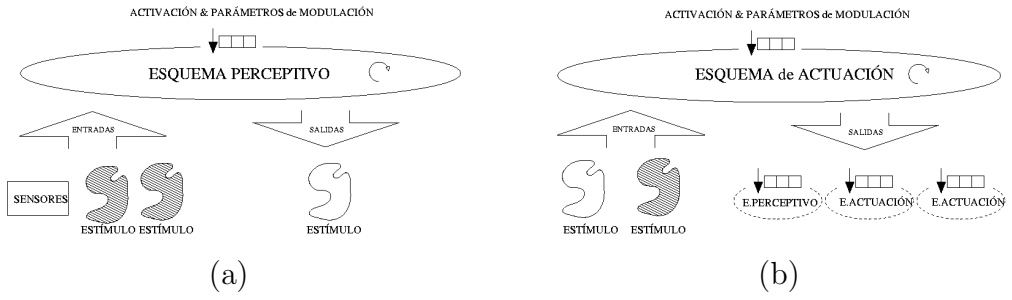


Figura 1.10: *Patrón típico de un esquema perceptivo (a) y de un esquema motor (b) en JDE*

La organización de los diferentes esquemas que componen la arquitectura se establece por medio de capas, en jerarquías que dependen del comportamiento deseado, de esta

forma, la variación de su organización dará como resultado diferentes comportamientos. La forma de construir esta jerarquía y dotarla de "vida"se realiza por medio de la activación y modulación de cada uno de los esquemas a través de parámetros, siendo a su vez éstos el resultado de la ejecución de otros situados un nivel superior. Así, cada uno de los esquemas perceptivos y de control pueden activar o desactivar esquemas situados en niveles inferiores ampliando la funcionalidad del sistema conforme se necesite en tiempo de ejecución.

1.3.2. jde.c

En sus inicios, la implementación de la plataforma fue de la mano del uso de diferentes robots reales programados utilizando el software (Saphira) del propio fabricante, estos robots fueron : un *Pioneer* y un *B21*. En ambos casos, el procesamiento de los datos sensados para su posterior envío a los actuadores debía realizarse en máquinas equipadas en el propio robot, por limitaciones del propio software. Para solventar esta situación JDE incorporó dos servidores, *otos* y *oculo*, encargados de ofrecer acceso a los sensores y los comandos motrices a cualquier programa cliente por medio de una *interfaz de mensajes*.

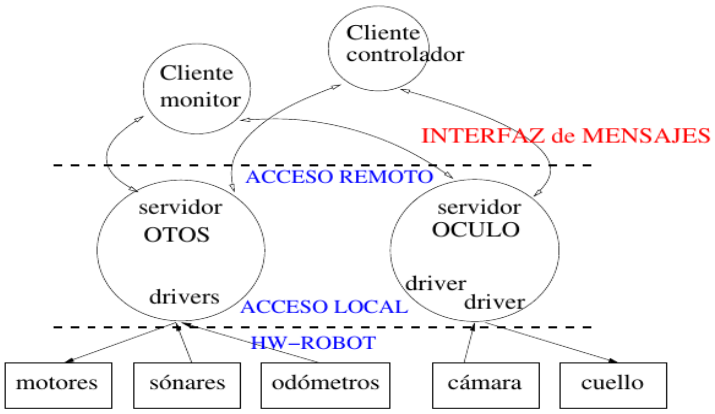


Figura 1.11: *Arquitectura software con servidores y clientes.*

El hecho de permitir a un programa usuario acceder remotamente a cuantos recursos necesite a través de la red, vía *sockets*, supone un valor añadido frente a entornos como Saphira que sólo ofrecen acceso local. Además, esta arquitectura también permitía aprovechar mejor la capacidad de cómputo disponible facilitando así el procesamiento distribuido y paralelo de la información que proviene de los sensores.

Por otro lado, se crea la posiblidad de exportar estos dos servidores a cualquier tipo de robot simplemente añadiendo los *drivers* para entablar la comunicación con éste,

manteniendo en todo momento la nueva interfaz de mensajes y creando así un sistema más independiente del robot, al cual se le pueden añadir incluso nuevos dispositivos que en principio podía existir la posibilidad de que el fabricante no los ofreciese.

Un esquema en *jde.c* es implantado como una hebra kernel, usando la extendida librería POSIX, dejando que el propio *scheduler* sea el encargado de repartir las porciones de tiempo destinadas a cada una de ellas. Estas hebras, tanto las de percepción como las de actuación, ejecutan de forma periódica y simultánea una determinada función cuando no se encuentran en estado DORMIDO, satisfaciendo así el requisito que fija la necesidad de que un esquema debe tratarse de un flujo iterativo e independiente del resto.

En cuanto a la comunicación entre esquemas, ésta se realiza por medio de variables globales (memoria compartida) cuya visibilidad está garantizada al correr todos bajo un mismo proceso. Por un lado, los estímulos perceptivos se materializan en variables que los esquemas perceptivos refrescan de forma periódica. Por otro lado, los esquemas de actuación (o motores) se encargan de leer esta información y ejecutar las acciones pertinentes.

Resumiendo, las principales ventajas de esta primera versión son:

- Implementa esquemas concurrentes y distribuidos, pero no intermáquina.
- Incorpora una interfaz de comunicación que permite liberar un poco más el software de la máquina (el robot) donde será aplicado.
- Facilita la reutilización de esquemas, los cuales sólo deben ceñirse a una *API* común que define la funcionalidad básica para cada esquema.

1.3.3. jde+

jde.c presentaba una serie de carencias que lo impedían ser más manejable, escalable y flexible. En este contexto nace una nueva versión de la plataforma, que aunque no llegó a ser aplicada de forma práctica, los conocimientos obtenidos en su desarrollo fueron esenciales en futuras versiones que poco a poco lograron incorporarlos. Este nuevo enfoque recibió el nombre de *jde+* ([Lobato Bravo, 2005]), y pretendía poner fin a varias limitaciones sufridas por *jde.c*.

En *jde.c* ya existía la posibilidad de crear tantas hebras como fuesen necesarias definiendo únicamente la función que debería cumplir cada una de ellas, sin embargo, al basarse la comunicación entre ellas en el uso de variables compartidas, la complejidad que alcanza la sincronización necesaria para evitar posibles condiciones de carrera entre

ellas aumenta a la par que son añadidas más hebras. Para solventar este problema, *jde+* implementa un mecanismo de comunicación mediante *paso de mensajes*, que por un lado permite desarrollar diferentes políticas de comunicación y por otro, restringe ésta para únicamente poder entablarse entre padres e hijos, tal y como se describe en la arquitectura cognitiva.

Otra limitación existente es la integración de nuevos esquemas al sistema, que aunque posible en *jde.c*, se trata de una ardua tarea que conlleva la programación del módulo correspondiente, modificar el programa principal para incluirlo y crearlo, y recompilar el sistema completo. La solución dada por *jde+* es la *carga dinámica de esquemas*, añadiendo la posibilidad de cargar esquemas en tiempo de ejecución y desacoplando así el sistema de los esquemas, siendo posible compilar cada parte por separado.

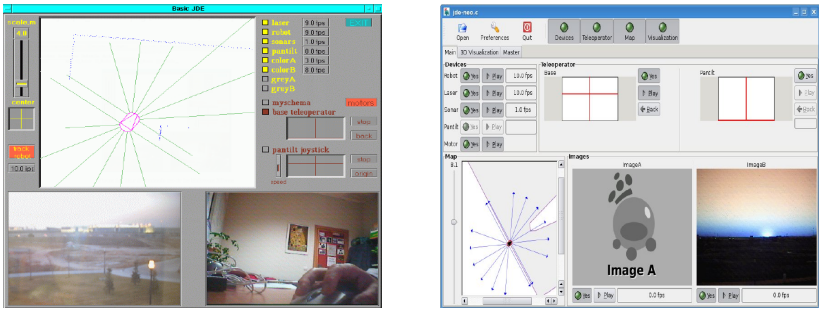
Otra novedad interesante es el cambio del lenguaje para el desarrollo de la plataforma. Si bien *jde.c* estaba desarrollada en C, con *jde+* se decide dar el paso al Paradigma Orientado a Objetos. Para reutilizar toda la funcionalidad ya existente el cambio se realiza hacia C++, siendo éste el lenguaje de programación sobre el que se tratará de implementar todos los nuevos aportes.

Además de las anteriores mejoras, *jde+* esboza las líneas maestras que simplifiquen la tarea de convertir JDE en un sistema distribuible que permita ubicar los esquemas en diferentes nodos de cómputo.

1.3.4. jdeneo.c

La siguiente versión, *jdeneo.c* ([Ruiz-Ayúcar Vázquez, 2007]), está basada en *jde.c*, aunque se nutre también de los nuevos enfoques procedentes de *jde+*. Los tres hitos que marcan a esta nueva versión son:

- La aplicación sobre JDE del concepto originado en *jde+*, la *carga dinámica de esquemas*.
- Portar la interfaz gráfica de la anticuada librería XForms por nuevas bibliotecas más potentes que comienzan a cobrar un importante papel en esas fechas, como GTK. Junto con este cambio, se apostará también por un rediseño de toda la interfaz gráfica buscando la usabilidad de cara al usuario.
- Mejorar la visualización de las aplicaciones, hasta entonces en 2D, hacia una nueva representación en 3D de la escena de forma nativa, utilizando para ellos las librerías de OpenGL.



(a) (b)

Figura 1.12: *(a) Interfaz gráfica con XForms en jde.c (b) Interfaz gráfica con GTK en jdeneo.c*

1.3.5. Jderobot 4.3

En Jderobot 4.3 empiezan a surgir los *esquema drivers* (o de bajo nivel) cuyo cometido es entablar la comunicación con el hardware y proporcionar variables para el acceso a éste, siendo posible su reutilización y con ella la abstracción a futuros desarrolladores de realizar la tarea manualmente.

Es en esta etapa donde la extensión de la plataforma crece, no sólo en cuanto a código, sino también en cuanto a la infraestructura que rodea a cualquier gran proyecto, surgiendo enriquecedoras herramientas como son las listas de distribución, una web con información del proyecto y documentación en línea, y una comunidad cada vez más amplia que nutre todo este ecosistema.

Es por tanto que Jderobot 4.3 ya no sólo es aquella arquitectura que ofrecía un núcleo (jde.c) desde el cual era posible activar y desactivar esquemas. Estamos ante un *framework* que extiende su funcionalidad gracias a una gran cantidad de herramientas, como el soporte para la gestión de interfaces gráficas, carga dinámica de esquemas, definición de interfaces para acceso al hardware a través de una amplia colección de *drivers*, integración con simuladores punteros donde es posible llevar a cabo desarrollos sin necesidad de hacer uso de robots reales, y un largo etcétera de características.

1.3.6. Jderobot 5.0

Jderobot 5.0 ([Lobato Bravo, 2010]) es la última versión estable concebida previa al presente proyecto. En este punto, la experiencia obtenida en el desarrollo de gran cantidad de proyectos marcan las trazas que debe conseguir esta versión. Esta experiencia se centra

en la idea principal de ayudar al desarrollador de aplicaciones robóticas por medio de sistemas más flexibles y metodologías que permitan una mayor reutilización del software, puesto que ambos aspectos permiten al desarrollador centrarse en la tarea de diseñar su sistema y avanzar más rápido en su desarrollo apoyándose en partes ya implementadas.

Para la obtención de un sistema más flexible y que haga uso de caraterísticas que comenzaban a ser tendencia en ese momento, como el uso de múltiples lenguajes para el desarrollo de las aplicaciones, fue precisa una completa revolución en toda la arquitectura. Se añade la capacidad de crear un sistema distribuido, donde el cómputo pueda ser realizado en diferentes nodos y que la información resultante esté disponible por cada uno de ellos de forma remota. Ambas funcionalidades son obtenidas por medio del *middleware* ICE (ver sección 3.2). Se integra Autotools en la plataforma como herramienta de construcción del código fuente. Además es creado un paquete debian que permite la instalación de toda la plataforma de forma sencilla por el usuario.

Resumiendo, los avances conseguidos en Jderobot5.0 fueron:

- *Sistema flexible*: El sistema debe aportar diferentes enfoques para organizar nuestras arquitecturas, o al menos no imponer ninguno concreto.
- *Sistema distribuido*: El sistema debe poder distribuirse entre múltiples nodos de cómputo utilizando mecanismos estándard, multi-plataforma y escalables.
- *Sistema multi-lenjuaje*: El sistema debe permitir implementar componentes en múltiples lenguajes.
- *Orientado a componentes*: El desarrollo debe orientarse a componentes (generalización de los esquemas), de modo que cada unidad mínima obtenida sea autocontenida, maximizando así las posibilidades de reutilización de los elementos desarrollados. Estos componentes son procesos independientes que contienen funcionalidad propia.

1.3.7. Jderobot 5.1

Este proyecto aborda la liberación de la última versión de la plataforma Jderobot, la 5.1. La nueva versión incorpora mejoras realizadas en diferentes componentes concretos de la plataforma, como es el caso de *introrob*, *gazeboserver* o *teleoperator*, entre otros. A parte de los componentes mejorados, también se han desarrollado nuevos, como *basic component*, que introducen nuevas funcionalidades y características no existentes hasta ahora.

También se han integrado nuevas herramientas en el proyecto que mejoran la usabilidad y mantenimiento de éste. En esta línea podemos destacar la incorporación de *CMake* como herramienta de construcción de código, sustituyendo a Autotools, cuyo manejo era mucho más complejo.

Por otro lado, la creación de nuevos *paquetes debian* llegan para facilitar la instalación de toda la plataforma y con ella el software del que hace uso.

Comentadas entre líneas algunas de las mejoras incorporadas en la nueva versión, los siguientes capítulos describirán de forma más detallada cada una de ellas. Comenzando con los *objetivos* propuestos y que debía alcanzar la versión 5.1, el siguiente capítulo, *infraestructura*, describirá algunas de las principales herramientas utilizadas para desarrollar ésta. En el capítulo *descripción informática* será donde se detallarán los nuevos aportes, los cuales han sido divididos en tres bloques: *componentes*, *CMake* y *paquetes debian*. En el último capítulo, *conclusiones*, se recopilarán todos los objetivos conseguidos y otros que han surgido en la consecución de éstos, así como las nuevas vías que se abren tras la liberación de Jderobot 5.1.

Capítulo 2

Objetivos

En el presente capítulo se describen de manera genérica los objetivos planteados en el Proyecto Final de Carrera, los requisitos impuestos para su resultado y la metodología utilizada durante su desarrollo.

2.1. Descripción del problema

El objetivo principal será el lanzamiento de una nueva versión estable para la plataforma Jderobot. Puesto que se trata de un avance evolutivo, el incremento en la versión se ha fijado en su iteración 5.1.

Con la experiencia adquirida durante el uso de Jderobot 5.0, se aprecia que algunas de las partes que forman la plataforma no aportan toda la funcionalidad deseada o ésta puede mejorarse. Por ejemplo, la herramienta Autotools, aunque ofrece a la plataforma la funcionalidad de construir su código, muy importante en proyectos de gran envergadura, se observa que su mantenimiento es muy complejo y por lo tanto costoso. Otro aspecto a mejorar es la facilidad en cuanto a instalación de Jderobot y todo el software externo que utiliza. En la versión 5.0 existe un único paquete que instala *todo* el software que forma Jderobot y que le rodea, este método no es del todo efectivo ya que el usuario no siempre quiere hacer uso de todos los elementos de la plataforma, no existiendo forma de seleccionar qué elementos se desean realmente instalar.

En cuanto a los elementos que forman la plataforma, se echa en falta un componente sencillo que integre las características que identifican a toda aplicación de Jderobot, pero cuyo comportamiento sea lo suficientemente simple como para que sirva de ejemplo a los nuevos desarrolladores que entran a formar parte de esta comunidad. En esta misma línea,

se observa cómo el software sobre el que se apoyan los aportes (simuladores, librerías, etc) liberan versiones nuevas, lo cual obliga a realizar un mantenimiento necesario en varios de los componentes si se desean utilizar todas las novedades que incorpora el software que les rodea.

La nueva versión de Jderobot agrupará todas las soluciones conseguidas para los problemas previamente citados. Además de las mejoras que serán descritas en el presente documento, también han sido añadidos diferentes desarrollos realizados por otros programadores dentro de la comunidad Jderobot, siendo necesaria la integración y agrupación de todos ellos en la versión 5.1 de la plataforma Jderobot.

El objetivo principal, la liberación de la nueva versión, ha sido dividido en tres subobjetivos concretos:

- *Desarrollo de nuevos componentes o mejora de existentes*: En este subobjetivo se agrupan todos los desarrollos orientados a la creación de nuevos componentes o a la mejora de ya existentes, añadiendo nuevas funcionalidades demandadas por los usuarios u optimizando las actuales. Entre estos componentes se encuentran *introrob*, *gazeboserver*, *teleoperator*, *wiimoteclient*, *wiimoteserver* y *basic component*.

- *Integración de CMake*: El siguiente subobjetivo aborda la integración de una nueva herramienta en Jderobot para la construcción de todo su software, CMake, sustituyendo a Autotools. El objetivo principal de este nuevo aporte será ofrecer una herramienta sencilla, tanto para que los usuarios puedan hacer uso de la gran cantidad de aplicaciones existentes dentro del proyecto, como para que los desarrolladores dispongan de facilidades a la hora de incorporar nuevos componentes.

- *Creación y liberación de la nueva versión por medio de paquetes debian*: Con este último subobjetivo se alcanzará la liberación, de manera oficial, de *Jderobot 5.1*. La creación de *paquetes debian* ofrecerá a los usuarios y desarrolladores de Jderobot una manera sencilla de instalar toda la plataforma (o las partes de ésta que se deseen) en su sistema, así como todo el software externo del que hace uso. Además, se tratará de una potente herramienta que facilitará la usabilidad, gestión y mantenibilidad de toda la plataforma ofreciendo su soporte en distribuciones punteras como son Ubuntu 12.04 y Debian Testing (Wheezy).

2.2. Requisitos

Los requisitos impuestos son similares en cada uno de los tres subobjetivos presentados:

- *Plataforma*: la plataforma sobre la que se cimienta Jderobot es GNU/Linux, siendo libre la decisión de la distribución a escoger. En el caso de los paquetes, éstos han sido concebidos para ser instalados en las versiones más recientes de Ubuntu 12.04 y Debian Testing (Wheezy).
- *Robustez*: en especial los componentes, deben presentar una ejecución libre de errores.
- *Flexibilidad*: para CMake y los paquetes, deben ser funcionales en cualquier sistema donde se van a utilizar, independientemente de la configuración que disponga la máquina del usuario que va a hacer uso de cualquiera de estas herramientas.
- *Escalabilidad*: los diseños elegidos para la integración de CMake y paquetizado de la plataforma deberán ser fácilmente extensibles para incorporar nuevos componentes y bibliotecas.
- *Soporte a comunidad y usuarios*: La incorporación de cambios en la plataforma llevará consigo un soporte constante tanto a la comunidad de desarrolladores, como a los usuarios que hacen uso de Jderobot para manejar las novedades. Para ello se llevará a cabo un contínuo mantenimiento del manual del proyecto alojado en su wiki[1], donde se explicará cómo utilizar todas las nuevas mejoras. Además, por medio de distintos foros[2] y listas de correo [3] se atenderá a las dudas que surjan por parte de los usuarios.
- *Documentación*: los nuevos aportes deben ser correctamente documentados en inglés dentro del manual contenido en la wiki oficial del proyecto Jderobot.
- *Licencia*: Todo el código de Jderobot está liberado bajo la licencia GPLv3.

2.3. Metodología de desarrollo

Como todo proyecto software, es necesario el uso de un modelo de su ciclo de vida que comprenda las diferentes fases que vive: desarrollo, verificación y mantenimiento. El

[1] http://jderobot.org/index.php/Manual-5

[2] http://docencia.gavab.es/course/view.php?id=11 — http://docencia.etsit.urjc.es/moodle/course/view.php?id=23

[3] http://gsyc.escet.urjc.es/pipermail/jde-developers/

modelo seguido en este caso ha sido el de *espiral*, puesto que se ajusta de forma óptima a las necesidades que requiere un Proyecto Final de Carrera, ya que está basado en un conjunto de iteraciones, cada una de las cuales da origen a una nueva versión del producto que puede ser evaluada, aumentando así su complejidad de manera progresiva. Cada una de estas iteraciones está dividida en las siguientes actividades:

- *Determinar objetivos*: de tal modo que el objetivo final será dividido en un conjunto de subojetivos alcanzables en cada iteración, aumentando así éstos de forma gradual.
- *Análisis del riesgo*: reduciendo la posibilidad de alcanzar el objetivo final con problemas que podrían haber sido detectados en alguna de sus iteraciones previas.
- *Planificar*: evaluando lo realizado hasta el momento y adaptando así el siguiente subobjetivo, siendo éste un nuevo avance o la mejora de los existentes.
- *Desarrollar, verificar y probar*: es necesario asegurar el correcto funcionamiento del trabajo realizado en cada una de las iteraciones con el fin de no arrastrar errores durante varios ciclos.

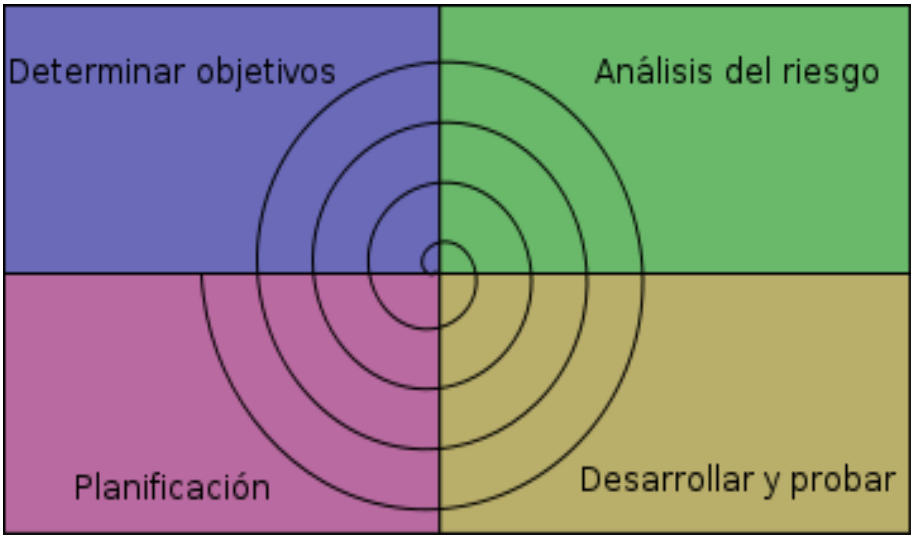


Figura 2.1: *Modelo en espiral.*

Adaptado al presente proyecto, cada una de las iteraciones fueron llevadas a cabo, en general, por medio de reuniones semanales. De esta forma, el tutor (José María Cañas) realimentaba de forma constante y era partícipe de la evolución de los nuevos aportes en todo momento.

2.4. Plan de trabajo

Dividiendo las diferentes etapas desde el inicio del proyecto de forma cronológica, se pueden resumir en:

1. Componentes:

   a) *Familiarización con la tecnología usada en componentes*: dado que el primer subobjetivo fue el desarrollo de componentes, fue necesario obtener previos conocimientos acerca de los detalles que caracterizan las aplicaciones desarrolladas para Jderobot, así como las tecnologías utilizadas para ello. Entre estas particularidades se encuentran tecnologías como ICE, para la comunicación de aplicaciones distribuidas, C++ como lenguaje base para todo el desarrollo o GTK+, utilizado en la implementación de las interfaces gráficas.

   b) *Desarrollo de componentes*: tras las iniciación previa, se dio paso al desarrollo de nuevos componentes y la refactorización de otros, ampliando y mejorando su funcionalidad.

2. CMake:

   a) *Familiarización con herramientas para la construcción de software*: el segundo subobjetivo fue la integración de una nueva herramienta para la construcción de código, *CMake*, en sustitución a la existente hasta entonces, *Autotools*. Para ello fue necesario el estudio previo de ambas utilidades, focalizando en mayor medida en CMake, puesto que sería la herramienta a dominar para su posterior integración en el proyecto.

   b) *Integración de CMake*: la siguiente tarea fue el diseño y desarrollo de la nueva solución usando CMake, y su integración en la plataforma Jderobot.

   c) *Pruebas de CMake*: Una vez la herramienta fue liberada, transcurrió un periodo de tiempo durante el cual todos los desarrolladores y usuarios hicieron uso de ella con el fin de ofrecer retroalimentación que permitiese la detección y depuración de los posibles errores encontrados en ella.

3. Paquetes:

   a) *Familiarización con las herramientas de paquetería*: nuevamente, nace una nueva curva de aprendizaje para conocer todas las herramientas necesarias destinadas a la creación de los paquetes que liberarían la nueva versión de Jderobot.

b) *Diseño de la solución para el empaquetado de Jderobot*: etapa en la que fueron generados los paquetes para los diferentes componentes, librerías y dependencias externas.

c) *Pruebas de los paquetes*: una vez los paquetes fueron creados, fue necesario un periodo de tiempo para que el resto de desarrolladores hicieran uso de ellos y verificar así su correcto funcionamiento en cualquier tipo de entorno. Estas verificaciones se llevaron a cabo utilizando un repositorio de prueba desplegado en el servidor de desarrollo. Una vez comprobado que los paquetes no presentaban problemas, fueron liberados en el servidor de producción. Tras la liberación de los paquetes para Debian, fueron realizadas las mismas etapas para crear los de Ubuntu.

Como se puede observar, la heterogeneidad del proyecto crea varias barreras, las cuales implican largos periodos de tiempo en los que es necesario el aprendizaje y la documentación en nuevas tecnologías, demorando así el proceso de desarrollo, pero ampliando el espectro de conocimiento adquirido durante éste.

Capítulo 3

Infraestructura

Este capítulo ofrece una breve descripción de las tecnologías utilizadas para el desarrollo de las diferentes partes del presente proyecto.

3.1. Bibilioteca GTK+ y Glade

GTK+[1] es un conjunto de bibliotecas que permiten el desarrollo de interfaces gráficas de usuario. Es multiplataforma y su uso predomina en los entornos gráficos de GNOME y XFCE. Su implementación puede ser llevada a cabo a través de lenguajes como C, C++, Java, Ruby, Perl, PHP o Python.

Su uso en proyectos de gran reconocimiento como son Firefox o GIMP, le dota de una amplia comunidad a sus espaldas otorgándole así la confianza necesaria como para seleccionar esta herramienta en cualquier proyecto.

Forma parte del software libre, distribuyéndose bajo licencia LPGL.

Cada elemento definido en la interfaz gráfica se denomina *widget*, estos elementos son enlazados a eventos gestionados por el *callback* principal *gtk_main*. Este *callback* principal se encarga de gestionar los eventos detectados, es decir, las interacciones realizadas por el usuario con los *widgets*, de tal forma que el programador puede abstraerse de esta gestión para centrarse en el desarrollo del algoritmo a ejecutar cuando uno de estos eventos es disparado.

Todas las interfaces gráficas de usuario presentes en los componentes desarrollados en este proyecto han sido implementadas utilizando este conjunto de bibliotecas ya que

[1] http://www.gtk.org/

ofrece los objetos necesarios para representar la información deseada, así como los *widgets* utilizados para la interacción por parte del usuario.

Para la maquetación del diseño de estos componentes se ha hecho uso de la herramienta Glade[2], la cual ofrece una sencilla interfaz desde la que poder dar forma al aspecto que tendrá el componente. Una vez se ha realizado la maquetación, la propia herramienta genera su especificación en XML para posteriormente ser importada y usada en la aplicación, ahorrando al desarrollador la ardúa tarea de realizar todo este proceso a mano.

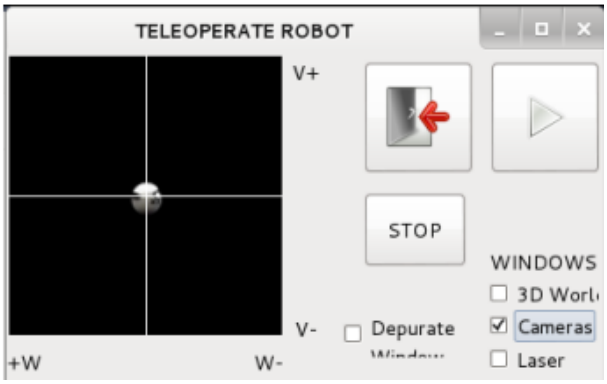


Figura 3.1: *Ejemplo de la interfaz de un componente con GTK+*

3.2. Biblioteca ICE

ICE[3] (Internet Communications Engine) es un *middleware* multiplataforma que permite el desarrollo de sistemas distribuidos heterogéneos orientado a objetos. Su desarrollo está fuertemente influenciado por CORBA, otro proyecto que permite la comunicación entre aplicaciones desarrolladas en distintos lenguajes y que pueden ser ejecutadas en múltiples máquinas de manera simultánea. ZeroC (empresa que desarrolla ICE) ofrece una herramienta que permite a los programadores llevar a cabo estas comunicaciones de forma más sencilla.

Al igual que CORBA, las aplicaciones pueden desarrollarse en diferentes lenguajes permitiendo así su heterogeneidad. Sin embargo, todos los componentes de este proyecto están desarrollados sobre C++.

Para el desarrollo de las interfaces que permiten la interoperabilidad entre aplicaciones se utiliza el lenguaje *slice*, el cual provee generadores de códigos para usar dichas interfaces

[2]http://glade.gnome.org/
[3]http://www.zeroc.com/

en C++, Python, Java o PHP.

ICE contiene otros servicios que amplían aún más esta herramienta, como:

- *IceStorm*: Servicio que permite gestionar la distribución de eventos de manera flexible y eficiente
- *IceGrid*: Conjunto de plataformas que permiten el balanceo de carga, de despliegue y administración de nodos.
- *IcePatch*: Facilita el despliegue de software basado en ICE de forma sencilla permitiendo la aplicación de parches en una aplicación de manera automática.
- *Glacier*: Servicio que permite la comunicación a través de *firewalls*.

Jderobot hace uso de este *middleware* para la comunicación entre los diferentes componentes distribuidos que se agrupan para generar una determinada funcionalidad.

3.3. Biblioteca CWIID

CWIID[4] es una librería que proporciona una sencilla API para realizar la comunicación, vía bluetooth, entre un dispositivo *wiimote* y un ordenador. Proporciona básicos métodos que permiten la lectura de cada uno de los sensores que contiene este dispositivo.

Similar al comportamiento descrito en GTK+, CWIID provee un *callback* principal capaz de detectar los eventos producidos por la acción de algún usuario sobre el dispositivo *wiimote*. Almacena cada uno de ellos en variables de modo que es posible comprobar cual fue el evento detectado y leer la información del sensor en cuestión para posteriormente ser utilizado por el programador.

Además, la conexión/desconexión entre máquina y wiimote es muy sencilla a través de su API, así como la detección e interacción con diferentes dispositivos conectados de forma simultánea, agrupando éstos en una estructura que contiene el identificador de cada uno de ellos.

Algunos componentes de Jderobot como wiimoteServer (ver sección 4.1.5), hacen uso de esta librería para la comunicación con un dispositivo *wiimote* y obtener a través de ella los datos de sus sensores.

[4]http://abstrakraft.org/cwiid/wiki/WikiStart

3.4. Simulador Gazebo

Gazebo es un potente simulador de robots. Desarrollado por Andrew Howard y Nate Koening surge de la idea de crear un entorno de simulación para la mayoría de robots populares, junto con sus dispositivos, y que éstos puedan interactuar con un entorno tridimensional donde es posible integrar diferentes tipos de objetos dinámicos.

Gracias a este simulador, liberado bajo licencia GNU Pública, existe la posibilidad de crear aplicaciones robóticas sin la necesidad de invertir altos presupuestos en la adquisición de sofisticado hardware prohibitivo en muchos casos. Además, permite el desarrollo de algoritmos que podrán ser probados en un entorno simulado pero muy cercano a la realidad para conseguir un producto estable que posteriormente podrá ser puesto en producción sobre un robot real. También permite que instituciones públicas, tales como la propia Universidad Rey Juan Carlos ofrezcan material docente utilizando un potente software como es este a coste cero, y que los alumnos desarrollen sus prácticas sin necesidad de adquirir un robot real.

Aunque Gazebo ofrece una gran variedad de sensores, actuadores, robots, objetos y mapas ya creados, también ofrece potentes herramientas para el diseño de nuevos elementos cuya integración se hace de manera muy sencilla. De esta forma, se abre un importante abanico de posibilidades para acercar aún más el entorno simulado al real, siendo posible la creación de escenarios muy similares a aquel donde finalmente será desplegado el robot físico.

El desarrollo de este proyecto ha coincidido con una gran evolución de este simulador desde su versión 0.9 a la nueva 1.5. Esta evolución viene de la mano de la fuerte financiación conseguida por parte de DARPA[5] (Defense Advanced Research Projects Agency), que ha elegido este simulador como base para su campeonato DRC (DARPA Robotics Challenge), creando así una nueva versión que conllevó una refactorización completa del código dando origen a una herramienta mucho más completa, estable, donde el detalle de las texturas ha evolucionado con creces.

Jderobot 5.0 ya hacía uso de Gazebo 0.9 en su componente *gazeboserver* (ver sección 4.1.2), pero la reestructuración completa del simulador hizo necesaria la reimplementación de un nuevo *driver* que permitiese la integración de la nueva versión de Gazebo con los componentes existentes en Jderobot.

[5]http://www.darpa.mil/

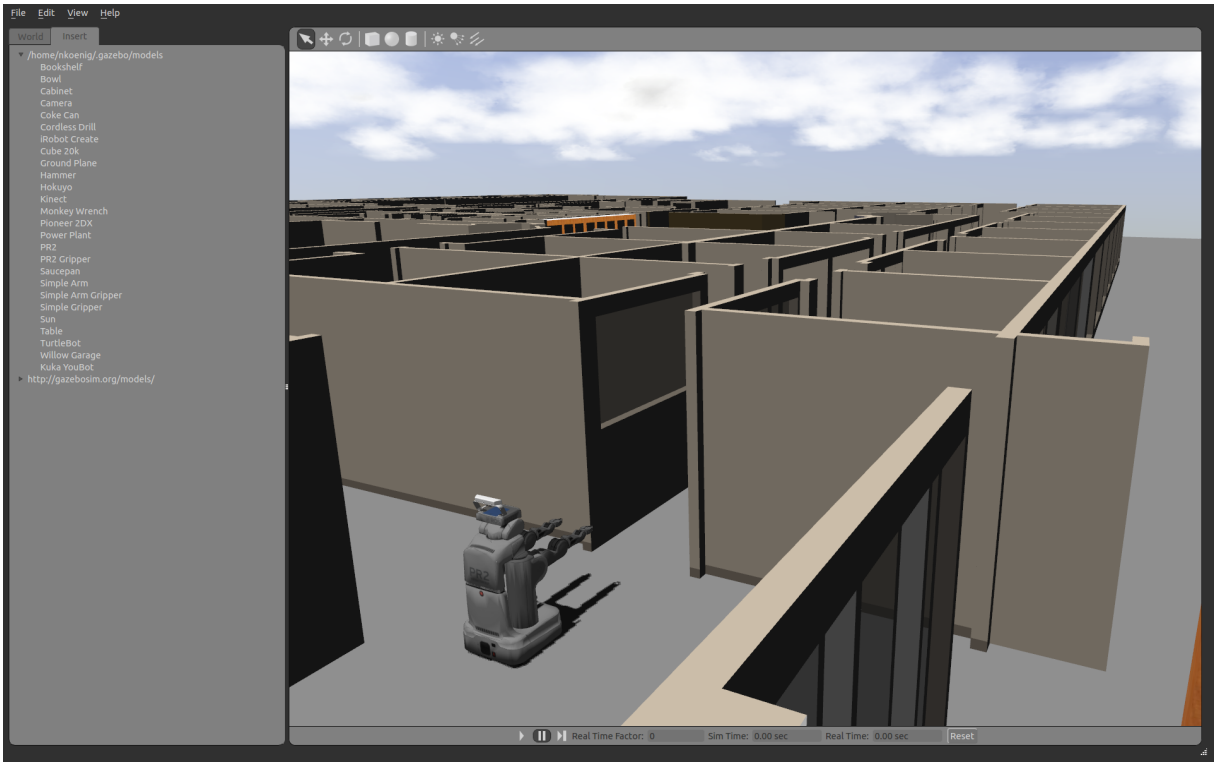


Figura 3.2: *Ejemplo de Gazebo 1.0*

3.5. Herramienta DPKG

Creado por Ian Jackson, dpkg es la base del sistema de gestión de paquetes de Debian GNU/Linux. Este programa ofrece un amplio conjunto de herramientas para la gestión de paquetes entre las que se encuentran:

- dpkg-source: Útil para empaquetar y desempaquetar las fuentes de un paquete Debian.
- dpkg-gencontrol: Genera un paquete binario a partir de uno previamente desempaquetado.
- dpkg-shlibdeps: Calcula las dependencias que posee ese paquete hacia otras librerías.
- dpkg-genchanges: Genera un fichero de cambios a partir de un paquete ya construído.
- dpkg-buildpackage: Herramienta utilizada en la creación de paquetes.

A partir de esta herramienta, junto con otras más, se han creado los nuevos paquetes para la versión 5.1 de JDErobot.

3.6. Herramienta CMake

CMake[6] es una herramienta de construcción de software multiplataforma que permite simplificar la tarea de crear una compleja cadena de compilación y enlazado gracias a las herramientas que ofrece para ello. Además de construir el software, generando los binarios y librerías, y resolviendo las dependencias necesarias para ello, esta herramienta es utilizada para la creación de paquetes combinándola con *dpkg*.

En este proyecto hemos usado esta herramienta para la construcción de todo el código de Jderobot, entre el cual se encuentran diversas aplicaciones, librerías y el uso de gran cantidad de software externo.

CMake será descrito con mayor detalle en la sección 4.2 dedicada a esta herramienta, puesto que supone uno de los puntos claves del presente proyecto.

[6]http://www.cmake.org/

Capítulo 4

Descripción informática

Una vez presentados los requisitos del proyecto y las herramientas utilizadas durante su desarrollo, en este capítulo se detallarán cada uno de los tres módulos correspondientes a los subobjetivos definidos en el captítulo 2, y que dan origen a la versión 5.1 de Jderobot: Componentes, CMake y Paquetes Debian.

4.1. Componentes

El primero de los bloques en los que se divide este proyecto se centra en la refactorización, actualización y creación de algunos componentes de Jderobot. Como ya se ha mencionado en secciones anteriores, un componente es una aplicación capaz de actuar por si misma o de forma distribuida junto con otros componentes. Algunos de ellos ofrecen al usuario (el programador) una serie de herramientas útiles en el desarrollo de aplicaciones robóticas. Por ejemplo, el acceso a los diferentes dispositivos que posee un robot (actuadores y sensores), una interfaz gráfica desde la cual es posible depurar los algoritmos desarrollados y la comunicación remota de una manera sencilla con un simulador, otro componente o el propio robot.

Según la funcionalidad del componente, éstos son catalogados en diversos tipos, entre los cuales nos centraremos:

- *Componentes-driver*: cuya funcionalidad es la de comunicarse con los diferentes dispositivos físicos, sensoriales y de actuación de un robot. Este tipo de componente es el encargado de servir la información obtenida a través de los sensores de un robot o enviar nueva hacia sus actuadores.

- *Componentes-herramienta*: este tipo de componentes hacen uso de los anteriores para comunicarse con el robot (real o simulado) y procesar la información que reciben a partir de ellos, para posteriormente enviar esa información procesada hacia el robot, nuevamente a través de los *componente-driver*.

Las novedades que se incorporan en esta nueva versión de Jderobot 5.1 en cuanto a sus componentes se pueden dividir en:

- Desarrollo de nuevos componentes que incorporan nuevas funcionalidades no existentes antes: *basic component*.
- Optimizaciones realizadas en componentes antiguos, incorporando a éstos mayor funcionalidad, mejorando la ya existente o eliminando aquella ya obsoleta: *introrob*.
- Renovación completa (desde cero) de aquellos componentes cuya tecnología ha quedado ya obsoleta y cuya actualización conlleva una nueva implementación ajustándose a las necesidades de las nuevas tendencias: *teleoperator*, *gazeboserver*, *wiimoteclient* y *wiimoteserver*.

4.1.1. Basic Component

Basic component es un componente de tipo *herramienta* cuya funcionalidad consiste simplemente en mostrar una serie de imágenes en una ventana. Tal y como se muestra en la figura 4.1, *basic component* hace uso del *componente-driver cameraserver*, el cual lee un fichero de vídeo y sirve las imágenes que éste contiene por medio de un interfaz ICE. De tal forma que *basic component* se conecta a dicho servidor ICE y obtiene así las imágenes, para posteriormente mostrarlas en una ventana usando GTK.

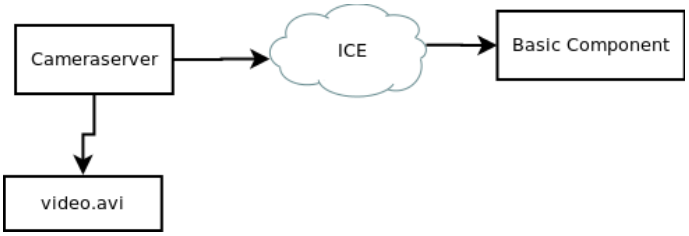


Figura 4.1: *Comunicación cameraserver-basic component*

Los objetivos que se abordan con un componente cuya funcionalidad es tan sencilla son:

- Diseñar un nuevo esqueleto basado en hebras que sirva de patrón para los futuros componentes desarrollados en Jderobot.

- Utilizar este componente como ejemplo para que los futuros programadores que comienzan a desarrollar para la plataforma tengan un modelo simple en el que fijarse, haciendo así más sencilla su curva de aprendizaje.

En cuanto al primer objetivo, se ha diseñado este esqueleto que satisface las características principales que identifican a cualquier componente de Jderobot, como son:

- *Comunicación remota con otros componentes*: los componentes de Jderobot son distribuidos, por lo que es necesario utilizar un sistema de comunicación entre ellos que les permita compartir la información. Para esta tarea Jderobot hace uso del *middleware ICE* (sección 3.2). Es por ello que el esqueleto de *basic component* mostrará cómo establecer esta conexión remota entre diferentes componentes.

- *División del flujo de ejecución en varios hilos*: un componente se caracteriza por realizar diferentes tareas simultáneamente. Esta capacidad se lleva a cabo dividiendo el flujo de ejecución principal en varios hilos, siendo dos el mínimo de ellos:

  - *Flujo de control o procesamiento*: se trata de una ejecución iterativa que se encarga de pedir los datos al *componente-driver* y de procesarlos posteriormente. Una vez procesados, los nuevos datos pueden ser enviados de vuelta hacia el *componente-driver*.

  - *Flujo de Interfaz Gráfica de Usuario (GUI)*: es un flujo iterativo que hace uso de los datos recibidos por el flujo de control o procesamiento, para mostrarlos a través de una interfaz gráfica. Además, a través de dicha interfaz gráfica es posible generar nuevos datos, por medio de la interacción del usuario con ella, que serán puestos a disposición para el flujo de control o procesamiento.

    Otro aspecto interesante es la posibilidad de activar o desactivar el hilo GUI a través de parámetros de configuración, ahorrando así la carga de cómputo que conlleva mostrar la interfaz gráfica si ésta no va a ser usada.

- *Comunicación asíncrona y concurrente entre los diferentes hilos*: el hecho de dividir el flujo de ejecución en varios hilos conlleva diseñar un método que facilite el compartir información entre ellos, eliminando condiciones de carrera y permitiendo la coordinación necesaria en caso de ser necesaria.

- Además de presentar las características anteriores, el nuevo esqueleto debe permitir aumentar su funcionalidad de una forma sencilla, ya que si bien en el caso de *basic component* se trata de algo tan simple como mostrar imágenes, los futuros componentes pueden ofrecer un comportamiento mucho más complejo.

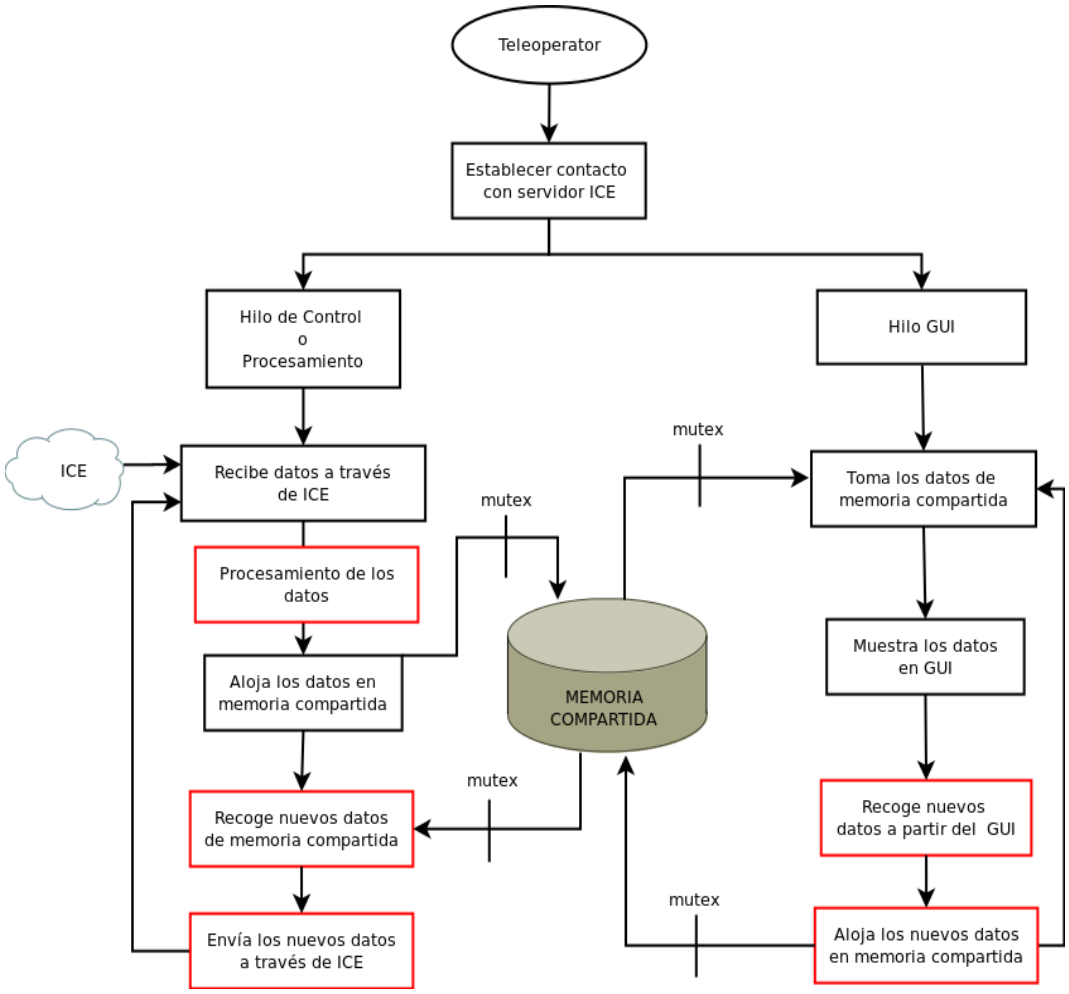


Figura 4.2: *Diseño del nuevo esqueleto para componentes de Jderobot 5.1.*

Partiendo de los requisitos anteriores, se diseña el nuevo esqueleto mostrado en la figura 4.2. En este esquema se pueden apreciar algunas partes marcadas en rojo, estas partes si bien pertenecen al nuevo esqueleto, no han sido implementados en el componente *basic component* dado que su funcionalidad no lo requería. A continuación se detalla de manera más concreta cual es el comportamiento que aborda cada una de las partes:

- Establecer el contacto, vía interfaces ICE, con el servidor (en este caso *cameraserver*) para la transmisión de los datos.

- Inicialización y creación del hilo que se encarga de mostrar la interfaz gráfica y cuyo flujo es:
  - Leer datos alojados por el hilo de control o procesamiento (en este caso imágenes).
  - Representar éstos en la interfaz gráfica.
  - OPCIONAL (marcado en rojo): En una aplicación completa, este hilo también se encargaría de alojar los datos obtenidos por medio de la interfaz gráfica en la memoria compartida.
- Bucle continuo del flujo de control o procesamiento en el que:
  - Se piden los datos al servidor ICE (cameraserver) y se alojan en la memoria compartida.
  - OPCIONAL (marcado en rojo): Procesamiento de los datos obtenidos antes de alojarlos en memoria compartida. Si el procesado de los datos es grande, el diseño podría ser ampliado añadiendo un nuevo flujo de procesamiento que realizase esta tarea.
  - OPCIONAL (marcado en rojo): Otra tarea común, sería enviar los datos alojados en memoria compartida por el hilo *GUI* hacia el servidor.

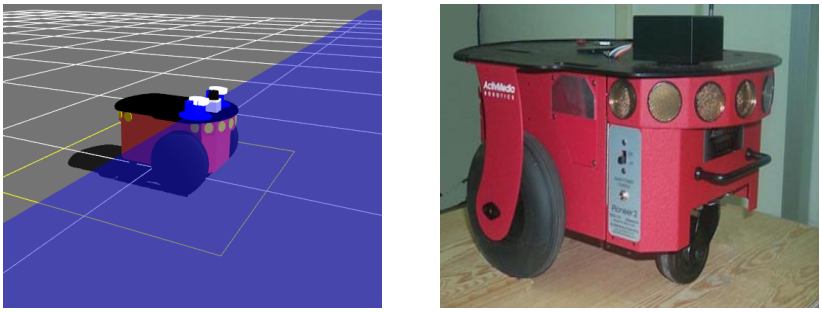
En los siguiente capítulos veremos como, partiendo del esqueleto aquí diseñado, se han desarrollado aplicaciones cuyo comportamiento es mucho más complejo, realizando la comunicación con otros *componentes-driver* que permiten teleoperar un robot, comunicarse con otros tipos de dispositivos, etc.

4.1.2. Driver Gazeboserver

El componente *gazeboserver* pertenece al grupo de *componentes-driver*. Su funcionalidad es la de hacer de intermediario entre el simulador Gazebo (sección 3.4) y otro/s *componentes-herramienta*, permitiendo así la comunicación entre ambos. Para llevar a cabo esta función, *gazeboserver* se conecta con un robot simulado en Gazebo a través de la propia *API* que ofrece el simulador, obtiene los datos que recibe el robot a través de sus diferentes sensores y sirve éstos por medio de un servidor ICE, poniéndolos a disposición de otro/s componentes. De igual modo, a través del servidor ICE, *gazeboserver* puede recibir nuevos datos que lleguen de estos componentes para enviarlos a los actuadores del robot simulado.

El robot simulado utilizado como referencia y con el que se conecta *gazeboserver* es un *pioneer2dx* (ver imagen 4.3(a)) equipado con:

- *Sensores*: dos cámaras, un láser y la odometría interna del robot.
- *Actuadores*: tres ruedas (dos de ellas motrices) y dos cuellos mecánicos, sobre los que montan sendas cámaras y a través de los cuales es posible variar su orientación.



(a) (b)

Figura 4.3: *(a) Robot pioneer simulado en Gazebo (b) Robot pioneer real*

El componente *gazeboserver* ya existía en Jderobot 5.0, siendo compatible con el simulador Gazebo en su versión 0.9. Sin embargo, en el último año el simulador ha dado el salto a su versión 1.5. Con esta actualización a una nueva versión, Gazebo ha visto como su arquitectura interna ha sido modificada por completo, así como la *API* que proporciona y la forma de comunicarse con aplicaciones externas. Este hecho conllevó que *gazeboserver*, existente en Jderobot 5.0, fuese incompatible con la nueva versión del simulador y requiriese la implementación de un nuevo componente que mantuviese la funcionalidad del anterior.

Los requisitos principales marcados para la nueva versión de *gazeboserver* son:

- Compatibilidad con la última versión de Gazebo.
- Mantener la compatibilidad con los componentes de Jderobot que ya hacían uso del antiguo *gazeboserver*, es decir, mantener la interfaz de comunicación que ya utilizaban éstos permitiendo así que no sea necesario modificar nada en ellos para utilizar la nueva versión de Gazebo.

En su anterior versión, *gazeboserver* estaba desarrollado para ser ejecutado como un proceso independiente que proveía todas las interfaces ICE necesarias para establecer la comunicación con cada uno de los sensores y actuadores integrados en el robot simulado

pioneer. El simulador hacía uso de ficheros con extensión *.world* donde se definían las características del mundo simulado, entre las cuales se encontraba la definición del robot *pioneer* y los dispositivos que éste portaba. Con el salto a la nueva versión, además de satisfacer los requisitos anteriores, se decide modificar algunos aspectos en *gazeboserver* en cuanto a su diseño interno, ilustrado en el diagrama 4.4, y los cuales se resumen en:

- *Gazeboserver* pasa de ser un proceso independiente a estar formado por un conjunto de librerías dinámicas o *plugins* que son cargados en tiempo de ejecución por el propio simulador Gazebo. Básicamente un *plugin* por cada sensor o sistema de actuadores.
- En los ficheros *.world* de Gazebo 1.x, además de definir los parámetros del mundo simulado al igual que en su anterior versión (robot, dispositivos del robot, terreno, objetos existentes en el mapa, etc), es necesario especificar qué *plugins* o librerías de *gazeboserver* serán cargados cuando se ejecute ese mundo.

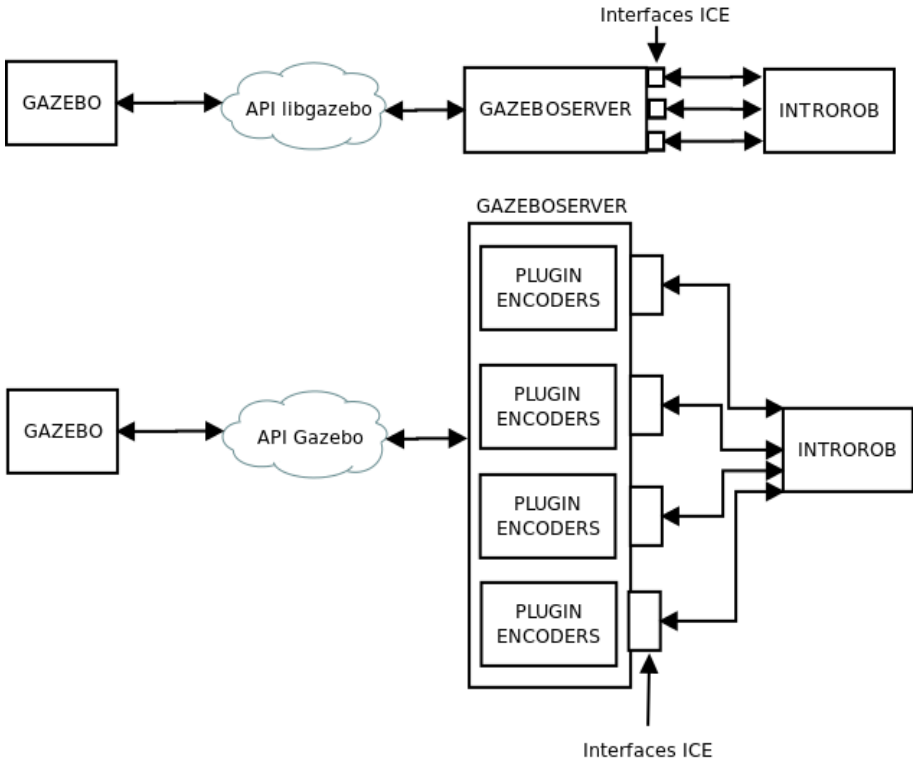


Figura 4.4: *Diseño del nuevo Gazeboserver (inferior) frente al antiguo (superior) conectándose a un componente ejemplo (introrob).*

La decisión de dividir *gazeboserver* en un conjunto de librerías dinámicas, una por cada dispositivo (sensor o actuador) del robot, en lugar de crear una única que contenga la funcionalidad completa para el robot *pioneer* proporciona las siguientes ventajas:

- *Flexibilidad del código*: en la anterior versión de *gazeboserver* todo el código que lo implementaba se encontraba en un único fichero fuente, de tal forma que cuando se quería añadir la funcionalidad para un nuevo dispositivo (por ejemplo una nueva cámara, un nuevo láser, etc) o modificar alguna ya existente, era necesario trabajar sobre una gran cantidad de líneas de código, algo complejo para los programadores menos experimentados. Con el nuevo diseño, cualquier modificación que se desee realizar se reducirá a una única librería que contendrá la implementación para ese dispositivo concreto, abstrayéndonos del resto y haciendo más sencillo su mantenimiento.

- *Más configurable*: en el antiguo *gazeboserver*, al tratarse de un proceso independiente, cuando queríamos añadir o quitar un determinado dispositivo del robot era necesario modificar su código fuente y recompilarlo para generar un nuevo ejecutable que reflejase ese cambio. Actualmente, poseemos un plugin para cada uno de los dispositivos, de tal forma que añadir o quitar uno de ellos se reduce a definir un mundo (*.world*) que especifique si dicho dispositivo se encontrará o no, evitando la necesidad de modificar *gazeboserver* con cada uno de estos cambios.

Para el desarrollo del nuevo componente y su comunicación con el simulador se ha hecho uso de una API que proporciona el simulador Gazebo [1], la cual ofrece los métodos básicos para inicializar (o cargar) cada uno de los plugins, así como el carácter iterativo necesario para el sensado del mundo simulado y el envío de órdenes hacia el robot.

También fue necesario crear el modelo del robot simulado haciendo uso del formato .sdf (ver figuras 4.5(a) y 4.5(b)) utilizado por el simulador, donde se especifica cada una de las partes que componen el robot, se les atribuye propiedades (masa, aceleración, gravedad, ...) y se enlaza a ellos lo que se conoce como articulaciones (*joints* en la sintaxis del simulador) (ver figura 4.5(c)) para otorgarles la movilidad a aquellos que lo necesiten. Estas articulaciones hacen de pegamento entre, por ejemplo, una cámara y el chasis del robot y ofrecen dos métodos básicos:

- *setVelocity*: a partir de la cual, usando un valor positivo o negativo, se le otorga movimiento rotacional al dispositivo al que está enlazado el *joint* en uno u otro sentido.

- *getPosition*: que obtiene la posición relativa del *joint* enlazado con respecto al robot.

[1] http://gazebosim.org/api.html

```
<link name="chassis">
  <pose>0 0 0.16 0 0 0</pose>
  <inertial>
    <mass>15.0</mass>
  </inertial>
  <collision name="collision">
    <geometry>
      <box>
        <size>0.445 0.277 0.17</size>
      </box>
    </geometry>
  </collision>
  <collision name="castor_collision">
    <pose>-0.200 0 -0.12 0 0 0</pose>
    <geometry>
      <sphere>
        <radius>0.04</radius>
      </sphere>
    </geometry>
    <surface>
      <friction>
        <ode>
          <mu>0</mu>
          <mu2>0</mu2>
          <slip1>1.0</slip1>
          <slip2>1.0</slip2>
        </ode>
      </friction>
    </surface>
  </collision>
  <visual name="visual">
    <pose>0 0 0.04 0 0 0</pose>
    <geometry>
      <mesh>
        <uri>model://pioneer2dx/meshes/chassis.dae</uri>
      </mesh>
    </geometry>
  </visual>
  <visual name="castor_visual">
    <pose>-0.200 0 -0.12 0 0 0</pose>
    <geometry>
      <sphere>
        <radius>0.04</radius>
      </sphere>
    </geometry>
    <material>
      <script>
        <uri>file://media/materials/scripts/gazebo.material</uri>
        <name>Gazebo/FlatBlack</name>
      </script>
    </material>
  </visual>
</link>
```

(a)

```
<link name="laser">
  <pose>0.22 .0 0.3 0 0 0</pose>
  <gravity>false</gravity>
  <inertial>
    <mass>0.1</mass>
  </inertial>
  <visual name="visual">
    <geometry>
      <mesh>
        <uri>model://hokuyo/meshes/hokuyo.dae</uri>
      </mesh>
    </geometry>
  </visual>
  <collision name="collision-base">
    <pose>0 0 -0.0145 0 0 0</pose>
    <geometry>
      <box>
        <size>0.05 0.05 0.041</size>
      </box>
    </geometry>
  </collision>
  <collision name="collision-top">
    <pose>0 0 0.0205 0 0 0</pose>
    <geometry>
      <cylinder>
        <radius>0.021</radius>
        <length>0.029</length>
      </cylinder>
    </geometry>
  </collision>
  <sensor name="laser" type="ray">
    <update_rate>20</update_rate>
    <ray>
      <scan>
        <horizontal>
          <samples>180</samples>
          <resolution>1</resolution>
          <min_angle>-1.57</min_angle>
          <max_angle>1.57</max_angle>
        </horizontal>
      </scan>
      <range>
        <min>0.08</min>
        <max>10</max>
        <resolution>0.01</resolution>
      </range>
    </ray>
    <plugin name="laser" filename="liblaser.so" />
    <always_on>1</always_on>
    <visualize>true</visualize>
  </sensor>
</link>
```

(b)

```
<joint type="revolute" name="joint_laser">
  <parent>chassis</parent>
  <child>laser</child>
  <axis>
    <xyz>0 0 0</xyz>
    <limit>
      <lower>0</lower>
      <upper>0</upper>
    </limit>
  </axis>
</joint>
```

(c)

Figura 4.5: *(a) Definición del chassis del pioneer (b) Definición del sensor láser (c) Definición del joint para el láser*

Además de definir el modelo del robot *pioneer* junto con todos sus sensores y actuadores, es necesario dotar de funcionalidad cada una de las articulaciones por los que está formado, permitiendo así interactuar con el robot por medio de otro componente. La funcionalidad de cada uno de los dispositivos que conforman el robot (cámaras, motores, láser y encoders) junto con sus articulaciones, tuvo que ser realizada a bajo nivel, ya que en este caso el simulador no proveía los métodos típicos utilizados en el comportamiento de cada uno de los dispositivos tales como: girar (o rotar) el robot, avanzar "x"metros, rotar las cámaras un número determinado de grados, etc, etc. Por ejemplo, haciendo uso de los métodos básicos que ofrece Gazebo para las articulaciones (*setVelocity* y *getPosition*), hemos desarrollado una interfaz de más alto nivel que ofrece la funcionalidad necesaria para poder mover una cámara a una velocidad determinada hasta conseguir una orientación concreta por medio del método *setPose3DEncoders*, que recibe como parámetros la velocidad de desplazamiento y la orientación que se desea que tenga la cámara.

Cada uno de los *plugins*, además de implementar la funcionalidad de los dispositivos del robot, dividen su flujo de ejecución en una segunda hebra en el momento que son cargados por el simulador Gazebo. Esta hebra será la encargada de crear el servidor ICE que ofrezca los datos obtenidos por el propio *plugin*, por ejemplo imágenes en el caso del *plugin* de las cámaras. A través de dicho servidor, también existirá la posibilidad de recibr datos desde un componente hacia el *plugin*, que será el encargado de materializar dicha información sobre el robot.

Para establecer la comunicación (por medio de ICE) entre los componentes y los diferentes *plugins* se hace uso de ficheros de configuración (*.cfg*). *Gazeboserver* utiliza un fichero de configuración por cada uno de los *plugins* que posee (ver listado 4.1), y en cada uno de ellos se define: identificador del servidor (utilizado posteriormente por el cliente, el componente externo), la IP en la que se encontrará el servidor y el puerto desde el que escucha.

```
Motors.Endpoints=default -h localhost -p 9999
```

Listado 4.1: *motors.cfg (plugin para los motores)*

Por otro lado, el componente externo dispondrá de su propio fichero de configuración utilizado para establecer la conexión con el servidor ICE que ofrece *gazeboserver* (ver listado 4.2).

```
introrob.Motors.Proxy=Motors:tcp -h localhost -p 9999
introrob.Camera1.Proxy=cam_sensor_left:tcp -h localhost -p 9991
introrob.Camera2.Proxy=cam_sensor_right:tcp -h localhost -p 9992
introrob.Encoders.Proxy=Encoders:tcp -h localhost -p 9997
introrob.Laser.Proxy=Laser:tcp -h localhost -p 9998
introrob.Pose3Dencoders2.Proxy=Pose3DEncodersRight:tcp -h localhost -p 9995
introrob.Pose3Dencoders1.Proxy=Pose3DEncodersLeft:tcp -h localhost -p 9996
introrob.Pose3Dmotors2.Proxy=Pose3DMotorsRight:tcp -h localhost -p 9993
introrob.Pose3Dmotors1.Proxy=Pose3DMotorsLeft:tcp -h localhost -p 9994
```

Listado 4.2: *Introrob.cfg*

En resumen, la implementación de esta nueva versión de *gazeboserver* ha requerido el diseño y desarrollo de un *componente-driver* completamente nuevo que permita la compatibilidad con las nuevas versiones de Gazebo, y mantiene las interfaces ICE de comunicación con los componentes utilizadas en su anterior versión. Siendo necesario además crear nuevos ficheros de configuración ICE (*.cfg*) o la definición de nuevos mundos (*.world*).

4.1.3. Teleoperator

Teleoperator es un *componente-herramienta* que permite teleoperar un robot *pioneer* a través de una Interfaz Gráfica de Usuario (ver imagen 4.6) con la cual es posible:

- Visualizar las imágenes captadas por el robot. (ventana CAMERAS)
- Teleoperar manualmente al robot estableciendo una velocidad lineal y rotacional a los motores por medio de un joystick virtual. (ventana MOTORS)
- Visualizar los datos tomados por el láser. (ventana LASER)
- Conocer en todo momento la posición del robot según su odometría en los ejes: x,y,theta. (ventana GPS)
- Teleoperar las cámaras que incorpora el robot por medio de un joystick virtual. (ventana Pose3D)

El robot a teleoperar podrá ser simulado en Gazebo (sección 3.4), haciendo uso del *componente-driver gazeboserver* (sección 4.1.2), o bien real. En ambos casos, la

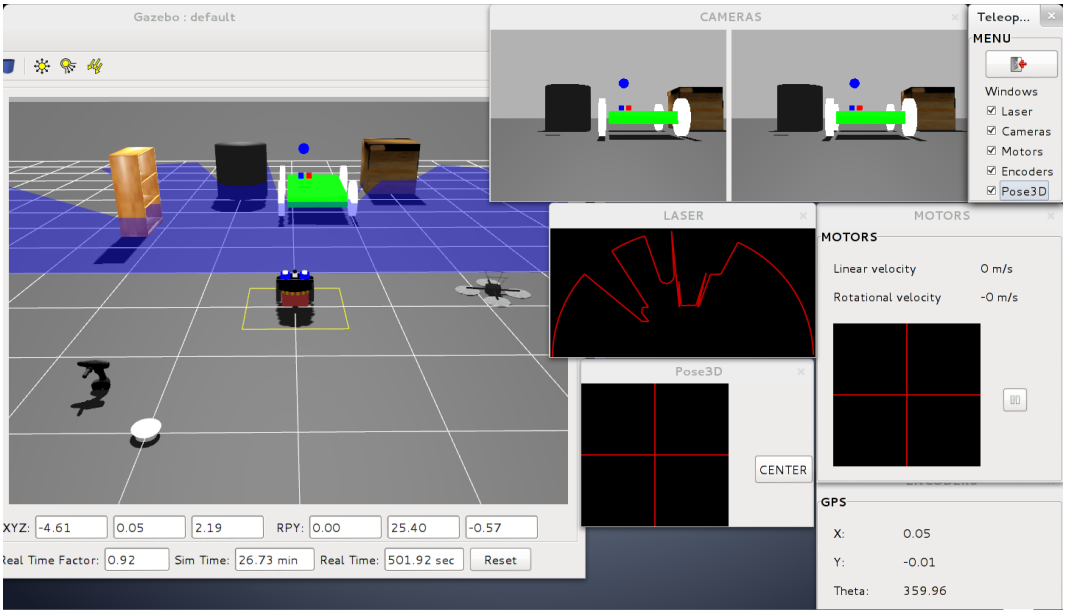


Figura 4.6: *teleoperator en ejecución. A la izquierda el simulador Gazebo y a la derecha las múltiples ventanas en las que se divide la GUI del componente.*

comunicación remota con cada uno de sus dispositivos se realizará por medio del *middleware* ICE, siendo sus interfaces utilizadas las mismas.

Para llevar a cabo esta funcionalidad, *teleoperator* aplica el nuevo esqueleto diseñado en *basic component*. Como se aprecia en la imagen 4.7, el flujo de ejecución es dividido en dos hebras:

- *Hilo de control o procesamiento*: se trata de un bucle continuo dividido en cuatro etapas, en las cuales el componente obtiene los datos del robot a través de *gazeboserver*, y aloja estos datos en memoria compartida, a disposición del hilo GUI, en sus primeras dos etapas. Las dos etapas siguientes tienen como función tomar de memoria compartida los nuevos datos que se hayan generado a través de la Interfaz Gráfica de Usuario (como modificar la velocidad del robot, la posición de las cámaras, etc) y enviar estos datos hacia el robot.

- *Hilo GUI*: es un bucle contínuo dividido en cuatro etapas. En sus dos primeras etapas son tomados los datos de memoria compartida alojados previamente por el hilo de control para ser mostrados en la Interfaz Gráfica de Usuario. Además, el usuario puede teleoperar el robot por medio de la Intefaz Gráfica (ver figura 4.6) que muestra *teleoperator*, de tal forma que los datos generados de su interacción son alojados en memoria compartida para el posterior envío por parte del hilo de control.

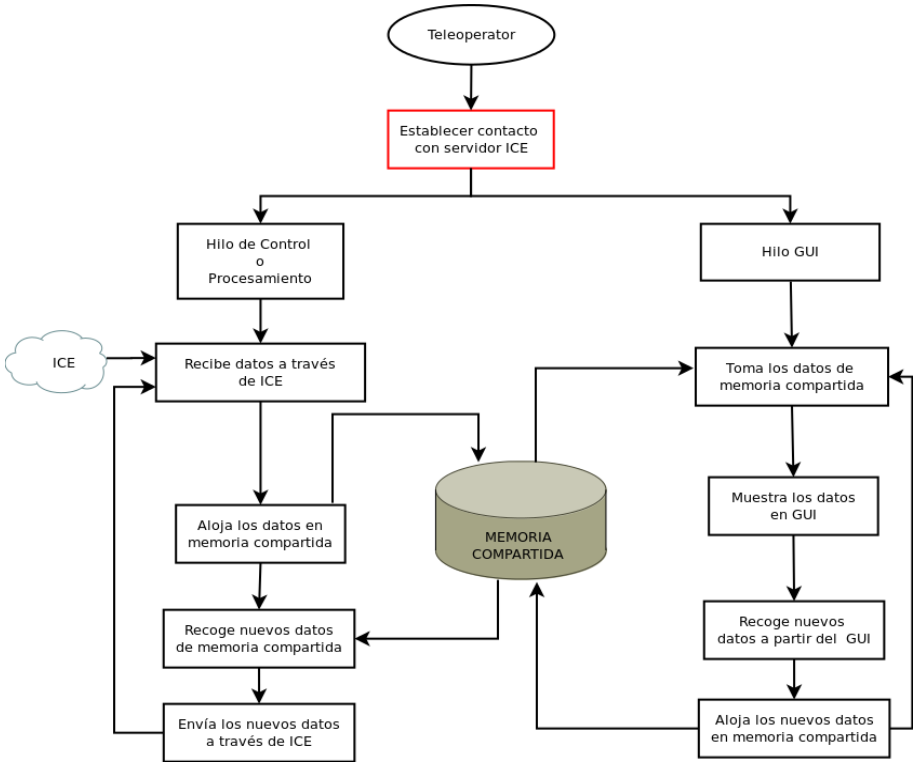


Figura 4.7: *Nuevo esqueleto aplicado en teleoperator.*

En el esquema 4.7 se puede observar como la primera etapa, establecer el contacto con el servidor ICE, está marcada en rojo. Hasta el desarrollo de *teleoperator*, la conexión por parte de un componente con un servidor ICE siempre era realizada al comienzo de su ejecución y ésta permanecía inalterable durante toda su ejecución. Con *teleoperator* se introduce el concepto de *interfaces ICE dinámicas*, cuyo significado es la posibilidad de activar o desactivar la conexión con alguno o algunos de los dispositivos que contiene el robot de forma dinámica (en tiempo de ejecución) por medio de *checkbox* contenidos en la interfaz gráfica. Con esta nueva funcionalidad podremos activar o desactivar la conexión, por ejemplo, con las cámaras del robot cuando se desee, ahorrando el ancho de banda que conlleva la transmisión de imágenes cuando no se estiman necesarias. Además, el componente pasa a ser más robusto en cuanto a fallos, puesto que si la conexión con alguno de los dispositivos es cerrada, dicho dispositivo podrá ser desacoplado del robot sin que esto repercuta en el funcionamiento de *teleoperator*.

El desarrollo de las *interfaces ICE dinámicas* conlleva un aumento de complejidad a la hora de sincronizar ambos hilos, puesto que el hilo de control debe saber si el usuario ha generado algún evento para activar una nueva interfaz, o bien desactivarla. Por otro lado, donde más influye que la sincronazición se lleve a cabo de manera satisfactoria es en el

lado del hilo GUI, puesto que éste deberá primero detectar la acción del usuario, informar al hilo de control de dicha acción y recibir la notificación por parte de éste de que dicho dispositivo ha sido activado y sus datos vuelven a estar accesibles para su renderizado (o desactivado, dejando de mostrar la información). Si la comunicación entre ambos falla, el hilo de procesamiento puede tratar de acceder a una zona de memoria donde en realidad no hay datos, generándose así lo que se conoce como *fallo de segmentación* e interrumpiendo la ejecución de toda la aplicación.

Para paliar esta situación, se ha hecho uso de cerrojos (*mutex*) protegiendo la memoria compartida utilizada para la sincronización de ambos hilos. En el diagrama de flujo 4.8 se puede ver un fragmento del algoritmo que controla esta sección del código para un dispositivo cualquiera.

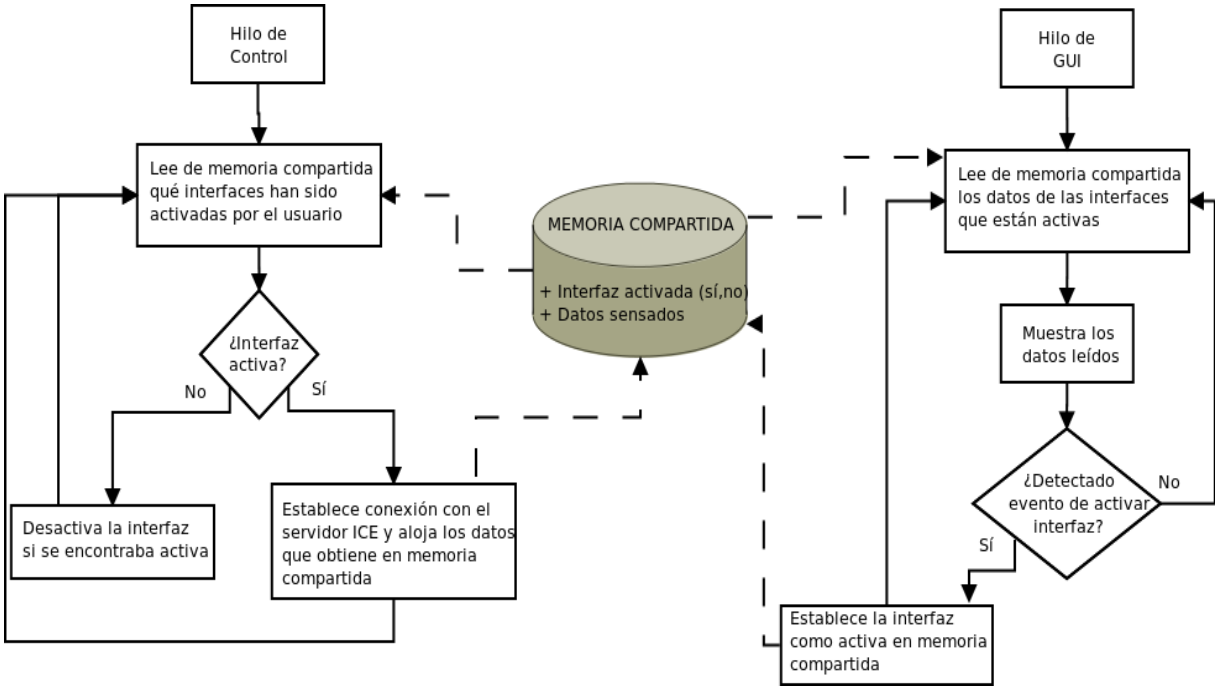


Figura 4.8: *Flujo de sincronización entre hilo de control y procesamiento.*

4.1.4. Introrob

Introrob es un *componente-herramienta* ya existente en la versión 5.0 de Jderobot, utilizado por parte de la Universidad Rey Juan Carlos en la docencia de las asignaturas de robótica impartidas en los máster de Sistemas Telemáticos e Informáticos y Visión Artificial. Su función es la de ofrecer una herramienta para el desarrollo de algoritmos

orientados al reconocimiento visual de objetos, algoritmos de navegación, comportamientos autónomos de un robot o proyecciones visuales a partir de imágenes captadas por cámaras.

En esta ocasión, lo que se pretende conseguir es añadir una serie de herramientas y funcionalidades demandadas por parte de los usuarios (los alumnos), así como mejorar las existentes e implantar el nuevo esqueleto concebido a partir de *basic component* (sección 4.1.1). Siendo necesario para ello la refactorización de la mayor parte del código, definición de nuevas clases, implementación de una completa API que permita la abstracción al alumno de funciones complejas como la comunicación con los diferentes dispositivos de un robot y la posibilidad de hacer uso de una interfaz gráfica (ver imagen 4.9) completamente renovada desde la cual puede:

- Teleoperar por medio de dos joystick virtuales tanto el robot como las cámaras que éste incorpora.
- Depurar sus resultados a través de una ventana que muestra las imágenes procesadas por sus algoritmos.
- Visualizar los datos internos del robot.
- Poner en marcha sus propios algoritmos mediante un simple click en el botón *PLAY*.

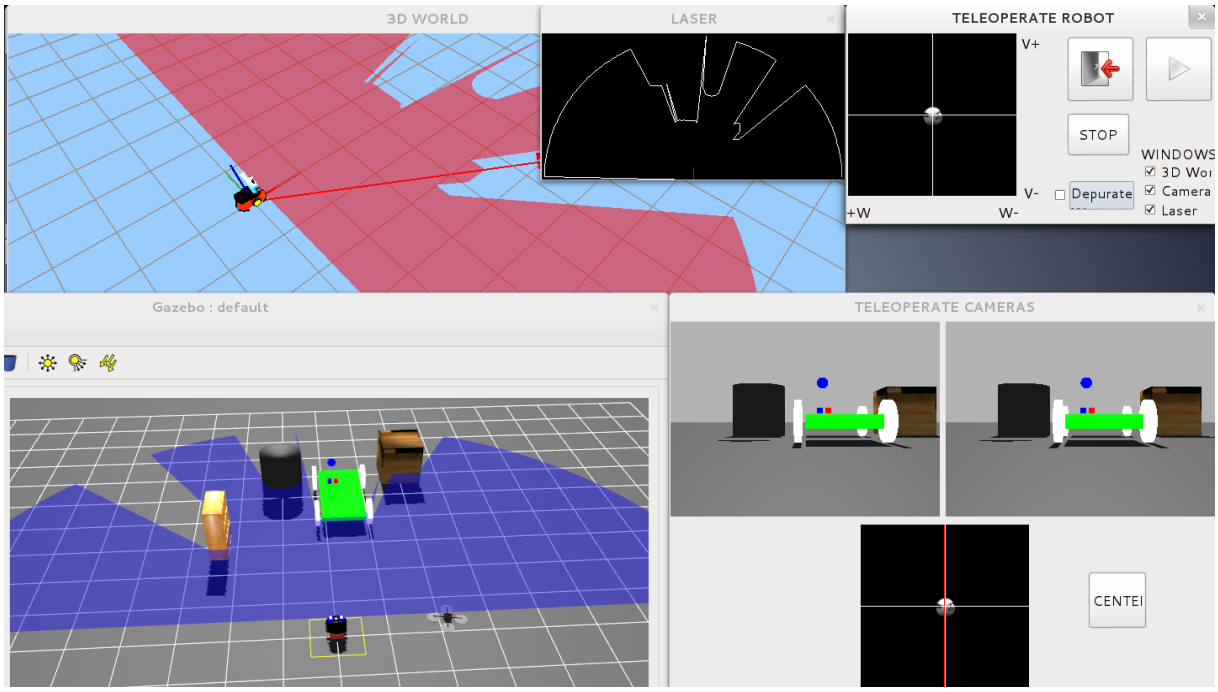


Figura 4.9: *Interfaz gráfica de Gazebo (esquina inferior izquierda) e introrob.*

Para el desarrollo de los diferentes algoritmos, *introrob* se conecta al simulador Gazebo (sección 3.4). El simulador despliega un entorno 3D en el cual son utilizados diferentes mundos previamente definidos que ofrecen las características necesarias para cada una de las prácticas que los alumnos deberán encarar, así como un robot que será utilizado para materializar estos algoritmos. Algunas de estas características son la inclusión en estos mundos de objetos para su posterior visualización, así como líneas trazadas en el suelo que posteriormente el algoritmo deberá reconocer, o incluso la posibilidad de coexistir dos robots en un mismo mundo. Además de los mundos, el simulador ofrece las herramientas necesarias para la definición de diferentes modelos de robots y todo el hardware que lo componen, concretamente con *introrob* es utilizado un robot *pioneer* equipado con una serie de sensores (cámaras, láser y GPS) y actuadores (encoders para las ruedas y el cuello mecánico que permiten la movilidad del robot y las cámaras respectivamente).

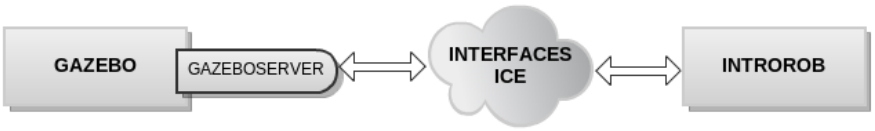


Figura 4.10: *Esquema de comunicación Introrob-Gazebo.*

Como se observa en el esquema 4.10, *introrob* no establece una comunicación directa con el simulador, sino que hace uso de un intermediario, *gazeboserver* (sección 4.1.2) el cual solicita a Gazebo los datos que el robot percibe de su entorno dentro del simulador y se los entrega por medio de ICE a *introrob*. Para configurar esta comunicación se hace uso de una serie de ficheros dónde se establece la información necesaria para las diferentes interfaces (cámaras, encoders, láser, ...) y que es leída en tiempo de ejecución por parte de los dos componentes. Un ejemplo de estos ficheros puede ser la siguiente:

```
introrob.Motors.Proxy=Motors:tcp -h localhost -p 9999
introrob.Camera1.Proxy=cam_sensor_left:tcp -h localhost -p 9991
introrob.Camera2.Proxy=cam_sensor_right:tcp -h localhost -p 9992
introrob.Encoders.Proxy=Encoders:tcp -h localhost -p 9997
introrob.Laser.Proxy=Laser:tcp -h localhost -p 9998
introrob.Pose3Dencoders2.Proxy=Pose3DEncodersRight:tcp -h localhost -p 9995
introrob.Pose3Dencoders1.Proxy=Pose3DEncodersLeft:tcp -h localhost -p 9996
introrob.Pose3Dmotors2.Proxy=Pose3DMotorsRight:tcp -h localhost -p 9993
introrob.Pose3Dmotors1.Proxy=Pose3DMotorsLeft:tcp -h localhost -p 9994
```

Listado 4.3: *Introrob.cfg*

```
Motors.Endpoints=default -h localhost -p 9999
```

Listado 4.4: *pioneer2dxMotors.cfg (gazeboserver)*

Para tener una visión global del nuevo diseño de *introrob*, el esquema 4.11 muestra las partes más importantes que lo conforman. Tal y como se aprecia en dicho esquema, *introrob* está dividido en dos partes. Por un lado, se agrupan todas las clases utilizadas para la correcta ejecución de la aplicación, siendo éstas:

- *Control*: Esta clase se encarga de realizar la función del flujo de control incorporado en el nuevo esqueleto, encargándose de realizar solicitudes de los datos sensados a *gazeboserver* y alojar éstos en la memoria compartida. Por otro lado, también se encarga de recoger los nuevos datos alojados por parte de la clase GUI para enviarlos a Gazebo en forma de órdenes a los diferentes actuadores.

- *GUI*: esta otra clase se encarga del procesamiento de los datos que se encuentran en la memoria compartida y mostrarlos a través de una interfaz gráfica (completamente renovada respecto a la versión anterior de *introrob*), a partir de la cual existe la posibilidad de teleoperar tanto el robot, como sus cámaras, haciendo más cómoda su navegación a través del mundo. Cuando el usuario teleopera el robot está generando nuevos datos para la velocdad lineal y/o rotacional de las ruedas que es alojada de nuevo en la memoria compartida y usada por parte de la clase Control. Además, dentro de esta clase se renderiza una simple representación realizada con OpenGL del robot en un espacio 3D a través de la cual se puede interactuar por medio del ratón para, por ejemplo, definir posiciones que serán usadas como objetivos para

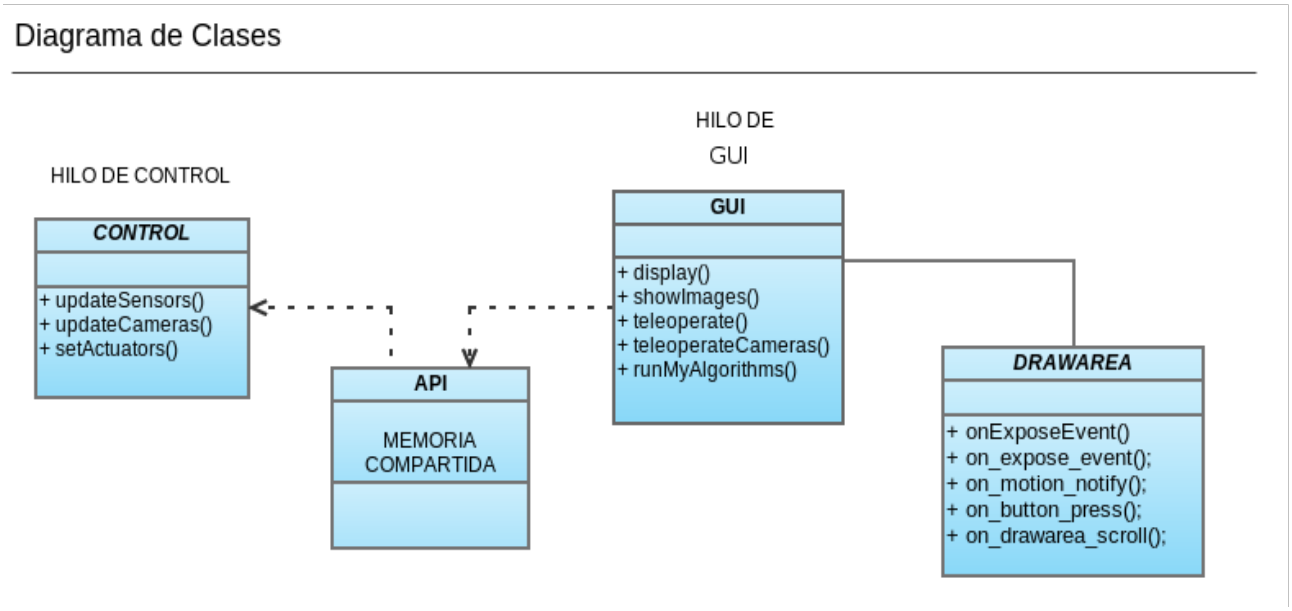


Figura 4.11: *Diagrama de clases Introrob.*

nuevos algoritmos de navegación, dibujar pequeñas esferas que, mediante algoritmos de visión pueden representar aquellos objetos recogidos por las cámaras del robot a través de diferentes métodos de transformación de puntos en el espacio.

Otra parte que se aprecia en el esquema es la que se centra en la sección que utilizará el usuario para el desarrollo de sus algoritmos. Para esta tarea se ha creado una completa API local que proporciona un conjunto de métodos lo más intuitivos posibles para abstraer al alumno de todo lo necesario para la comunicación con el simulador, el acceso al hardware del robot, etc. A continuación se muestra una lista de algunas de estas funciones con una breve explicación, tal y como lo ve el alumno en la documentación:

- Acceso a sensores y actuadores (ver cuadro 4.12).
- Métodos para procesar y depurar las imágenes obtenidas por las cámaras (ver cuadro 4.13).
- Métodos que ofrecen funcionalidad auxiliar para los algoritmos de los alumnos (ver cuadro 4.14).

Todo el desarrollo realizado por el alumno estará definido en un fichero destinado para este fin denominado *MyAlgorithms.cpp* donde se encuentra la lista mostrada de todos los métodos disponibles y la función principal que lanza la ejecución de todo este código que es llamada desde la clase GUI. Además, en este fichero se han añadido diversos ejemplos básicos utilizando la mayor parte de estos métodos para una mejor comprensión por parte del alumno.

Pruebas de verificación

El componente fue sometido a una gran cantidad de pruebas debido a que su posterior uso estaba destinado hacia los propios alumnos, tratando así de limitar al máximo posibles errores que conllevase la pérdida de tiempo por parte de ellos. Algunos de los test realizados fueron:

- *Medición de rendimiento*: Dado que la frecuencia del flujo de control y el de GUI son modulables mediante parámetros definidos en el código, fue necesario probar con diversos valores hasta conseguir un equilibrio entre consumo de recursos y fluidez de la aplicación lo más óptima posible en máquinas equipadas con cuatro y dos cores.

- *Pruebas de estabilidad*: Las pruebas realizadas para medir la estabilidad iban desde probar a estresar la aplicación interactuando con ella de una forma agresiva/rápida tratando de errores con posibles combinaciones realizadas al interactuar con su interfaz. Además, otra de las pruebas llevadas a cabo consistió en ejecutar el código del alumno y dejar el proceso corriendo durante horas para observar que no tuviese fugas relacionadas con la gestión de la memoria.

- *Pruebas de funcionalidad*: la funcionalidad, sobre todo de la API proporcionada, debía estar libre de errores para no alterar el comportamiento de los algoritmos implementados por los alumnos pudiendo confundir a ellos mismos al ver que el programa no realiza el funcionamiento para el que ellos lo diseñaron.

- *Puesta en producción*: los propios alumnos utilizaron el componente en el curso 2011-2012 y desarrollaron sus prácticas de forma satisfactoria reportando pequeños detalles que han sido pulidos para el presente curso.

/* INTROROB API:

Métodos para obtener los datos de los sensores:
getMotorV: Este método devuelve la velocidad lineal (mm./s.) del robot.
EJEMPLO: float v = getMotorV();

getMotorW: Este método devuelve la velocidad rotacional (deg./s.) del robot.
EJEMPLO: float w = getMotorW();

getLaserData: Este método devuelve una estructura con dos campos, por un lado el número de lasers del robot (180) y por otro un vector de 180 posiciones en cada una de las cuales se almacena la distancia obtenida por el láser, estando relacionada cada posición del robot con el ángulo del láser. Ejemplo:
Jderobot::LaserDataPtr laser = getLaserData()

getDistancesLaser: Este método devolvería únicamente el vector mencionado anteriormente.
EJEMPLO: Jderobot::IntSeq VectorDistances getDistancesLaser();

getNumLasers: Este método devuelve el otro campo mencionado, el número de grados del láser. EJEMPLO: int numLasers = getNumLasers();

getEncodersData: Este método devuelve una estructura con tres campos: robotx, roboty y robottheta, siendo la posición x , y y su orientación theta respectivamente.
EJEMPLO: Jderobot::EncodersDataPtr myPosition getEncodersData();

Métodos para manipular los actuadores:
setMotorV(float V): Con este método definimos la velocidad lineal (mm./s.) del robot.
EJEMPLO: setMotorV(40.)

setMotorW(float W): Con este método definimos la velocidad rotacional (deg./s.) del robot.
EJEMPLO: setMotorW(10.)

Figura 4.12: *Acceso a sensores y actuadores.*

/* Métodos gráficos (mundo 3D)
imageCameras2openCV(): Obtiene las imágenes captadas por las cámaras y las transforma al formato IplImage
para un mejor manejo de éstas por medio de OpenCV. Las imágenes son alojadas en:
* imageCameraLeft
* imageCameraRight
pintaSegmento: Traza una línea entre dos puntos dados a partir de un color dado.
USO:
CvPoint3D32f aa,bb;
CvPoint3D32f color;
bb.x= destino.x;
bb.y= destino.y;
bb.z=0.;
aa.x=encodersData robotx;
aa.y=encodersData roboty;
aa.z=0;
color.x = 1.; // Red
color.y = 0.; // Greenv color.z = 0.; // Bluev
pintaSegmento (aa, bb, color);
drawProjectionLines: Traza líneas desde el origen de coordenadas a un punto seleccionado (click izquierdo) en una de las cámaras del robot.
USO: drawProjectionLines();

drawSphere: Dibuja una esfera dado un punto y un color
USO: drawSphere(bb, color);

Figura 4.13: *Métodos gráficos.*

xclickcameraleft: Almacena la coordenada x del punto donde se ha hecho click en la camara izquierda
yclickcameraleft: Almacena la coordenada y del punto donde se ha hecho click en la camara izquierda
xclickcameraright: Almacena la coordenada x del punto donde se ha hecho click en la camara derecha
yclickcameraright: Almacena la coordenada y del punto donde se ha hecho click en la camara derecha

graficas2opticas: Transforma un punto (pointX,pointY) a su equivalente en el sistema de referencia usado por las camaras (progeo)
USO:
int pointX; Podemos usar el punto obtenido al hacer click en una camara de la GUI xclickcameraleft
int pointY; Podemos usar el punto obtenido al hacer click en una camara de la GUI yclickcameraleft
HPoint2D Point2DCam punto en el sistema de referencia de la camara
graficas2opticas(pointX,pointY,&Point2DCam);
De tal forma que:
+ Point2DCam.x contiene la coordenada x
+ Point2DCam.y contiene la coordenada y

opticas2graficas: Transforma un punto 2D (Point2DCam) en el sistema de refercia de las camaras a su equivalente en el sistema de referencia de las imagenes que se muestran en el interfaz grafico
USO:
int pointX;
int pointY;
HPoint2D Point2DCam punto en el sistema de referencia de la camara
this opticas2graficas(&pointX,&pointY,Point2DCam);

Figura 4.14: *Métodos auxiliares.*

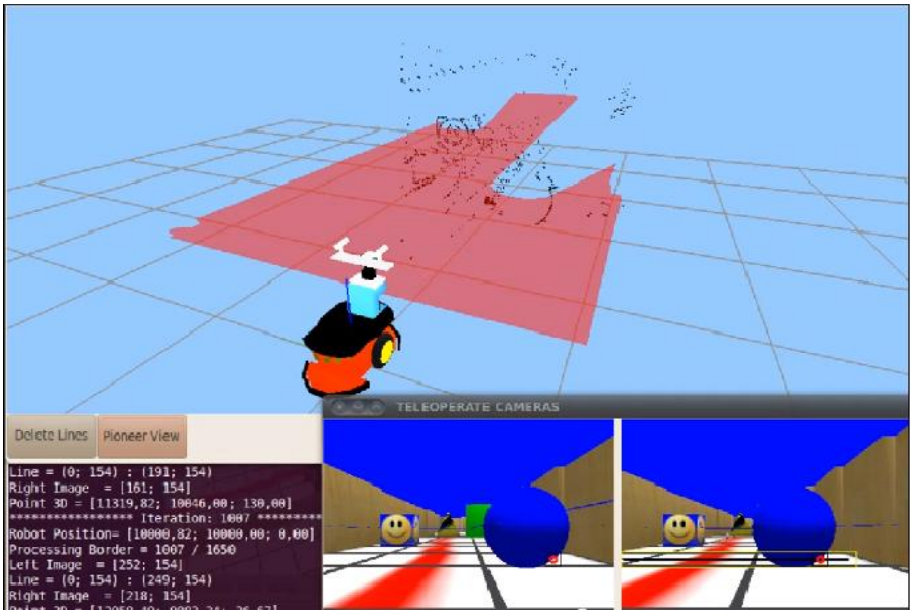


Figura 4.15: *Práctica de un alumno del Máster de Visión Artifical con Introrob.*

4.1.5. WiimoteServer y WiimoteClient

WiimoteServer y *WiimoteClient* son, respectivamente, un *componente-driver* y un *componente-herramienta* desarrollados para la integración del dispositivo *wiimote* dentro de Jderobot, permitiendo el desarrollo de nuevas aplicaciones que puedan interactuar y aprovechar la funcionalidad de este dispositivo. La funcionalidad de *wiimoteServer* es la de servir los datos obtenidos a partir del *wiimote*, mientras que *wiimoteClient* hace de cliente mostrando los datos obtenidos del servidor en una interfaz gráfica y permitiendo la interacción del usuario con el mando a través de ella (encender leds, activar/desactivar vibración, etc).

Ambos componentes existían en la versión de Jderobot 4.3, sin embargo, la evolución que vivió la plataforma en su paso a la 5.0 y la modificación completa de su arquitectura llevó a ambos componentes a la obsolescencia. Esta nueva versión incorpora todas las mejoras obtenidas en las últimas actualizaciones, como son:

- Uso del middleware Ice para la comunicación entre servidor y cliente/s en sustitución a la comunicación por memoria compartida utilizada en la Jderobot 4.3.
- Utilizado C++ y la metodología orientada a objetos que ofrece este lenguaje, en contraposición al lenguaje C con el que estaban implementados.
- La implementación de *wiimoteClient*, que sirve para que los demás desarrolladores

puedan ver cómo se realiza la interacción entre cliente y servidor, incorporando además el nuevo esqueleto explicado en el componente *basic component*.

WiimoteServer

Para la implementación de *WiimoteServer* y su comunicación con el *wiimote* (y su extensión *nunchuck*) ha sido necesario el uso de la librería *cwiid* (sección 3.3). Esta librería ofrece una API cuyos métodos aportan la funcionalidad necesaria para:

- Interactuar con el wiimote a través de la tecnología bluetooth.
- Acceso a cada uno de los botones y joystick que lo componen.
- Un *callback* para apilar los eventos realizados con el dispositivo y poder procesar éstos por medio de tipos de datos propios para cada uno de los controles.

Además, para permitir la posterior comunicación con los demás componentes, el driver añade un servidor ICE que hace uso de una nueva interfaz específica (no existente antes) para este dispositivo, cuya definición está implementada en el fichero *wiimote.ice* (ver listados 4.5 y 4.6) utilizando el lenguaje *slice* que proporciona el propio *middleware* Ice:

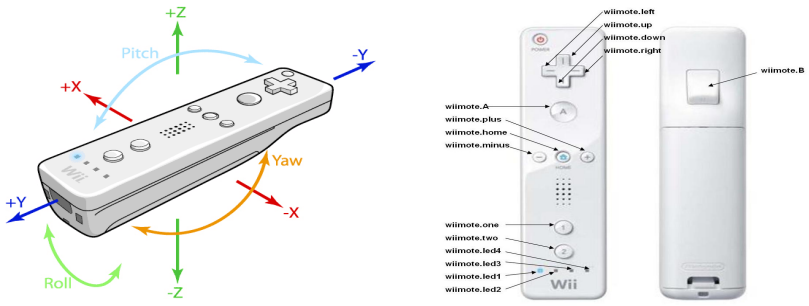
Como se observa en los listados 4.5 y 4.6, la interfaz ofrece los atributos y métodos necesarios para:

- Obtener los datos del acelerómetro que posee el *wiimote*.
- Obtener los datos del infrarrojos.
- Obtener el estado de la batería.
- Activar el modo vibración.
- Cambiar entre los distintos modos disponibles en el *wiimote*.
- Encender o apagar los distintos leds.

Para establecer la conexión, *WiimoteServer* hace uso del fichero de configuración *wiimoteServer.cfg* donde se define el nombre que identifica al servidor (para posteriormente

```
#ifndef WIIMOTE_ICE
#define WIIMOTE_ICE
#include <Jderobot/common.ice>
module Jderobot{
    /* Wiimote information */
  class AccelerometerData
  {
        IntSeq accelerometer; //Vector con los datos del acelerometro.
  };
  class InfraredData{
    IntSeq infrared1; //Datos para el sensor 1 infrarrojos
    IntSeq infrared2; //Datos para el sensor 2 infrarrojos
    IntSeq infrared3; //Datos para el sensor 3 infrarrojos
    IntSeq infrared4; //Datos para el sensor 4 infrarrojos
  };
  class NunchukData{
    int button; //Boton presionado en el nunchuck
    IntSeq stick; //Vector con la posicion del stick
    IntSeq acc; //Vector con los datos del acelerometro
  };
```

Listado 4.5: *Interfaz wiimote.ice (1)*



(a) (b)

Figura 4.16: *(a) Representación del acelerómetro del wiimote (b) Wiimote*

ser utilizado por *WiimoteClient*), la IP en la que se encuentra y el puerto desde el que escucha. El listado 4.7 muestra un ejemplo de como sería este fichero.

La estructura del driver consta de la parte que hace de servidor ICE cuyo objetivo es la transferencia de los datos hacia el cliente y la parte bluetooth que se encarga de

```
    /*
    * Interface to the Wiimote interaction.
    */
     interface wiiMote {
     //SET DIFFERENT MODES
          int changeRumbleMode(); //Poner el wiimote en modo vibracion
          int changeIrMode(); //Activar/desactivar el sensor infrarrojos
          int changeAccMode(); //Activar/desactivar el sensor acelerometro
          int changeButtonMode(); //Activar/desactivar los botones
          int changeNunchukMode(); //Activar/desactivar el mando nunchuck
          int activateLed(int led); //Activar el led determinado (int led)
     //GET DATA
          int getButtonData(); //Toma el identificador del boton presionado
     idempotent NunchukData getNunchukData(); //Toma un objeto NunchuckData
     idempotent AccelerometerData getAccData(); //Toma un objeto AccelerometerDat
     idempotent InfraredData getIrData(); //Toma un objeto InfrarredData
     int getBatteryStatus(); // Toma el estado de la baterÃ a
     };
}; //module
#endif //WIIMOTE_ICE
```

Listado 4.6: *Interfaz wiimote.ice (2)*

comunicarse con el *wiimote*, donde toda la lógica recae sobre un *callback* que proporciona la librería *cwiid* y en la que son procesados todos los eventos generados al interactuar con el dispositivo, como pueden ser:

- Hacer click en algún botón.
- Tomar datos del sensor de infrarrojos.
- Tomar datos del acelerómetro.
- Activar/desactivar los leds del *wiimote*.

```
WiimoteServer.Endpoints=default -h localhost -p 9999
```

Listado 4.7: *WiimoteServer.cfg*

- Mover alguno de los joysticks.

WiimoteClient

El diseño del componente *wiimoteClient* ofrece un ejemplo de cómo debe realizarse la conexión remota con el servidor a través del fichero de configuración *wiimoteClient.cfg*. El listado 4.8 muestra un ejemplo de este fichero.

```
wiimoteClient.Wiimote.Proxy=wiiMote1:tcp -h localhost -p 9999
```

Listado 4.8: *WiimoteClient.cfg*

Además, este *componente-herramienta* muestra cómo utilizar los métodos que contiene la interfaz explicada anteriormente y proporciona una interfaz gráfica donde se muestran los datos recibidos de *wiimoteServer*. Todo ello implementado, una vez más, sobre el nuevo esqueleto de *basic component*, donde el hilo de control realiza las peticiones necesarias hacia el servidor y el hilo GUI muestra la interfaz gráfica.

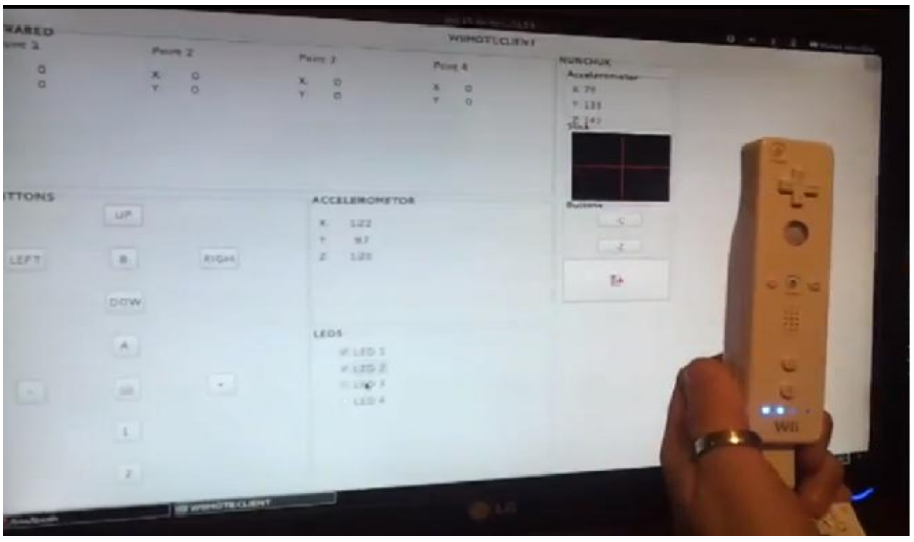


Figura 4.17: *WiimoteClient & WiimoteServer.*

4.2. Compilación de Jderobot 5.1 con CMake

Tras tratar algunos de los componentes que han sido incorporados o mejorados en la versión 5.1 de Jderobot, el segundo módulo en el que se dividen los nuevos aportes de este proyecto aborda la integración en la plataforma de una nueva herramienta de construcción

de software. Este tipo de herramientas es realmente útil cuando hablamos de un software muy amplio compuesto (como es el caso de Jderobot) por una gran cantidad de aplicaciones, librerías y el uso de mucho software de terceros. El objetivo de estas herramientas es generar todos los elementos (librerías y ejecutables) que conforman un software desde el código fuente, facilitando el proceso tanto a desarrolladores, quienes implementan su funcionalidad, como a los usuarios, que ven como con sencillos pasos logran construir todo el software en sus equipos. Existen diferentes herramientas que permiten compilar software de una manera más cómoda, entre ellas podemos destacar: Autotools, CMake, SCons, Boost bjam, etc.

En la versión 5.0 de Jderobot, la tarea de construcción de software era realizada con la herramienta *Autotools*. Sin embargo, la complejidad que conlleva desplegar la infraestructura (directorios y ficheros de configuración) que necesita esta herramienta para llevar a cabo el proceso de compilación había desembocado en su desuso.

En este contexto, se decidió integrar otra herramienta (sustituyendo la anterior) que facilitara el proceso tanto a desarrolladores como usuarios, siendo CMake la elegida. CMake (ver sección 4.2) es una herramienta de código abierto orientada a la construcción, realización de pruebas y creación de paquetes de software. Su configuración utiliza una sintaxis realmente intuitiva, y el hecho de que está basada únicamente en ficheros CMakeLists.txt elimina la necesidad de crear estructuras de directorios para su configuración que aumentarían en gran medida su complejidad. Además, cabe destacar que se trata de una herramienta multiplataforma, puesto que se adapta al compilador y arquitectura existentes en cada máquina para llevar a cabo el proceso de construcción.

Otro factor clave que ayudó a tomar la decisión de usar esta nueva herramienta es el auge que está teniendo en los últimos años en grandes proyectos como OpenCV, Gazebo, Player, Stage y Gearbox, entre otros. Esto ha creado una importante comunidad a su alrededor que nutre a la herramienta de documentación muy útil para conseguir la mejor configuración posible en nuestro proyecto.

Jderobot es una plataforma formada por una gran cantidad de elementos software, lo cual hace imprescindible contar con una herramienta que facilite su construcción a partir del código fuente. Entre estos elementos podemos encontrar:

- Librerías propias de Jderobot: *bgfgsegmentation, colorspaces, colorspacesice, fuzzylib, jderobotice, jderobotutil, pioneer, progeo, visionlib.*
- Componentes: divididos en:

    - Herramientas: *alarmgenerator, bgfglab, basic_component, calibrator, calibratorKinect, cameraview, kinectView, colortuner, giraffeclient, introrob, motiondetection, teleoperator-android, naooperator, opencvdemo, recorder, replayer, teleoperator, wiimoteClient, visualHFSM.*
    - Drivers: *cameraserver, cameraview_icestorm, gazeboserver, giraffeServer, naobody, naoserver, kinectServer, openniServer, playerServer, playserServer2.0, wiimoteServer.*
    - Otros: Existen componentes de aplicaciones que se mantienen fuera del repositorio de la plataforma pero forman parte del ecosistema de Jderobot.

- Interfaces ICE explícitas que permiten conectar componentes entre sí: *imageProvider, wiimote, camera, recordingManager, motors, laser, encoders, ptMotors, ptEncoders, sonars, pose3dEncoders, pose3dMotors, cloud points.*
- Software externo: *gazebo, gearbox, player, stage, pcl, openNi, fireware, opencv, ICE, cwiid, etc.*

La implantación de CMake en la plataforma Jderobot busca dotar a éste de una utilidad para generar todas las librerías y componentes que lo conforman. El diseño de la cadena de compilación con CMake debe satisfacer la siguiente lista de requisitos:

(a) Tratamiento de cada componente como una entidad individual. Es decir, si un usuario/desarrollador desea trabajar con sólo un componente, podrá abstraerse de todos los demás que forman Jderobot. Con eso ahorrará todo el tiempo y posibles problemas de dependencias que puedan ocasionar éstos y centrarse por tanto únicamente en aquél que desea tratar. A este método de construcción se le ha denominado *compilación por componentes*.

(b) También existe la posibilidad de generar *todas* las librerías y componentes que forman parte de Jderobot como si se tratase de una única entidad. De esta forma el usuario que lo desee podrá descargar todo el código completo que compone la plataforma y construir *todos* los elementos que la forman. A este otro método lo denominamos *compilación desde la raíz*.

(c) Adaptación a cualquier máquina. El diseño de la cadena de compilación deberá ser autosuficiente para conseguir alcanzar su objetivo (construir software), sea cual sea la máquina anfitrión dónde se esté ejecutando. A medida que aumentan las versiones del software del cual depende Jderobot, los directorios donde éste se instala puede variar,

por lo tanto la herramienta CMake deberá ser capaz de encontrar dónde se encuentra instalado dicho software, aislando así a los programadores y usuarios de la tarea de modificar los ficheros de configuración pertinentes.

(d) Facilidad de uso para usuarios y desarrolladores. Puesto que se trata de una herramienta necesaria para cualquier persona que quiera emplear Jderobot, su uso debe ser lo más sencillo e intuitivo posible ofreciendo así una primera experiencia agradable con Jderobot. Además, también se ha buscado en todo momento que los ficheros para configurar la herramienta sean intuitivos y fácilmente manipulables por parte de los desarrolladores con el fin de que incorporen a la construcción sus propios componentes.

Para cumplir todos estos requisitos fue necesario llevar a cabo un complejo diseño que integrase CMake dentro de Jderobot sin llevar a cabo grandes modificaciones en cuanto a la estructura de directorios que ya poseía el proyecto, y con la que estaban familiarizados desarrolladores y usuarios.

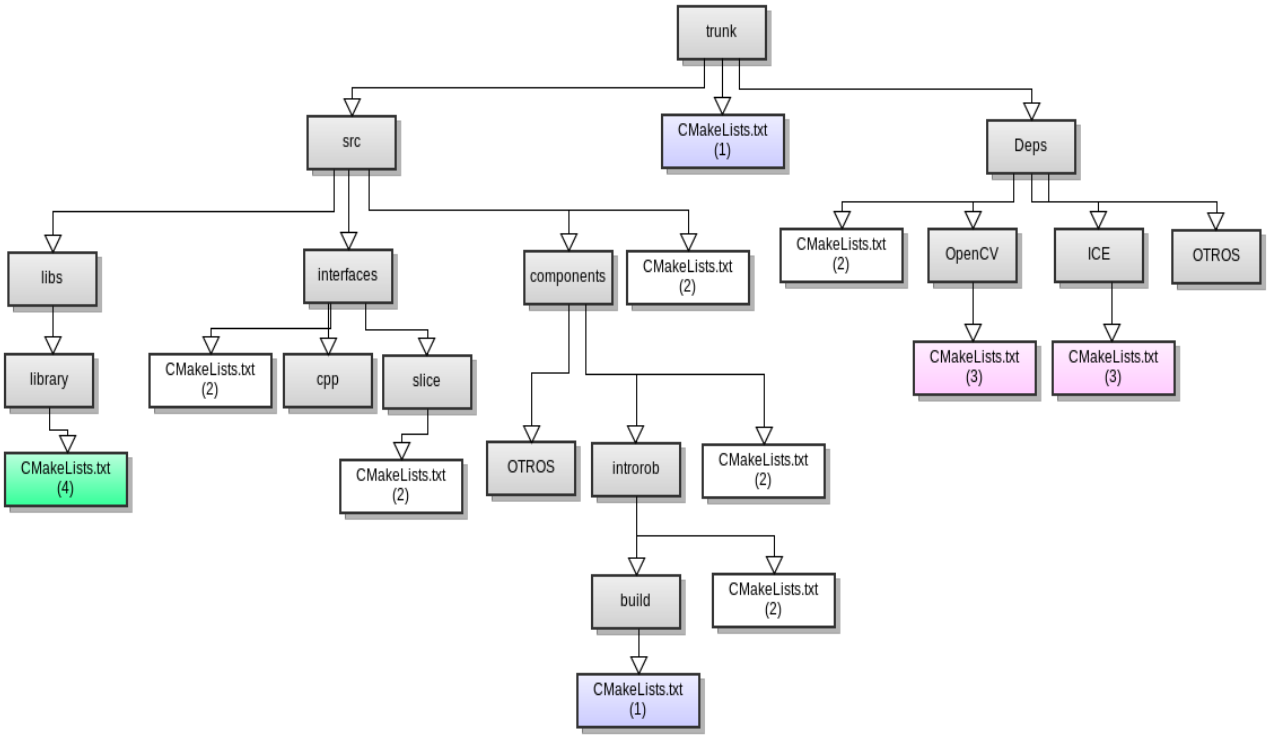


Figura 4.18: *Árbol de directorios de Jderobot y diseño de CMake.*

El esquema 4.18 ofrece una visión global del diseño elegido para incorporar CMake en Jderobot. Dicho esquema está dividido en diferentes "cajas", las cuales representan cada uno de los directorios en los que se divide todo el software que forma Jderobot y cuya raíz es el directorio "trunk/". Además, han sido representadas en diferentes colores aquellas

que simbolizan los ficheros de configuración de CMake (CMakeLists.txt) según su función en el proceso de construcción.

CMake identifica como un *proyecto* a la entidad que deseamos construir a partir de un determinado código fuente. Esta entidad puede ser un simple componente, una librería determinada o la plataforma completa. El nombre de ese proyecto debe ser definido en un fichero CMakeLists.txt que será el encargado de iniciar el proceso de construcción de dicha entidad. Partiendo de esta base, en el esquema 4.18 los diferentes ficheros CMakeLists.txt han sido etiquetados bajo un número identificativo según su funcionalidad, la cual puede ser:

- *Ficheros principales de configuración (1)*: Son los ficheros raíz, es decir, son aquellos donde es definido el nombre del proyecto y a partir de los cuales comienza la construcción de dicho proyecto o entidad. Además del nombre, en este fichero se definen las propiedades comunes que comparten todos los componentes que serán generados, de esta forma se consigue que la variación de los ficheros de configuración de los distintos componentes sea mínima, facilitando la inserción de nuevos componentes a Jderobot.

- *Ficheros de apoyo (2)*: Estos ficheros son llamados de forma recursiva desde el raíz y/o desde otros ficheros de apoyo. Son utilizados para definir propiedades más concretas de cada componente, librería o dependencia externa (gazebo, opencv, etc). Aunque su estructura en todos ellos sigue un mismo patrón, éste debe adaptarse a las necesidades de cada uno de los elementos.

- *Ficheros de dependencias (3)*: El directorio "Deps/", dónde se encuentran estos ficheros, contiene un directorio por cada una de las dependencias que no puede ser instalada por medio de paquetes (.deb) y que por tanto ha sido instalada directamente desde código fuente. El hecho de haber sido instalada a través de código fuente dificulta su adaptación a Jderobot, puesto que su instalación puede variar de una máquina a otra según la configuración determinada por el usuario en tiempo de compilación. Cada dependencia posee un fichero CMakeLists.txt que es llamado por cada componente que hace uso de ella, y es donde se especifican las reglas necesarias para buscar las librerías y cabeceras relacionadas con ella.

- *Ficheros de librerías*: El último tipo de ficheros de configuración es aquél destinado a la generación de las librerías propias de Jderobot, las cuales deben ser generadas en primer término para el posterior uso (enlazado) por parte del componente que

la demanda. En este sentido, una optimización realizada ha sido la reutilización de librerías previamente generadas, es decir, si el componente A ha generado las librerías X e Y durante su compilación (construcción) y posteriormente se desea compilar el componente B, el cual hace uso de la librería X, el sistema detectará que ésta ya ha sido generada anteriormente ahorrando así el tiempo que conlleva el proceso de generar dicha librería nuevamente.

4.2.1. Cómo emplea CMake un usuario de Jderobot

El uso de CMake en un proyecto de las dimensiones de Jderobot facilita en gran medida la experiencia de uso por parte de aquellos usuarios y desarrolladores que comienzan a utilizar este software por primera vez, puesto que en ambos casos podrán comenzar su andadura con dos simples comandos:

cmake arg1

siendo *arg1* la ruta donde se encuentra el fichero CMakeLists.txt principal, es decir, aquél a partir del cual es posible comenzar la cadena de compilación (tipo 1 en el esquema 4.18). Este comando genera todos los directorios auxiliares que utiliza CMake para cachear toda la información que necesitará durante el posterior proceso de compilación. El siguiente comando será:

make

Este comando inicia el proceso de compilación, resolución de dependencias y enlazado que generará los binarios para el componente (*compilación por componentes*) o componentes (*compilación desde la raíz*), así como las librerías de las que dependen.

De este modo, si un usuario desea construir toda la plataforma (*compilación desde la raíz*) debe acceder al directorio “trunk/build” y ejecutar:

cmake ..

haciendo referencia al directorio superior en el que se encuentra el CMakeLists.txt. Para iniciar el proceso de compilación deberá lanzar, en el mismo directorio que se encuentra:

make

Si el usuario desea constuir un componente concreto, debe acceder a su directorio correspondiente ”trunk/src/components/componente_x/build”(siendo componente_x el componente a construir) y ejecutar:

cmake .

haciendo referencia al directorio en el que se encuentra, dado que es donde está el CMakeLists.txt principal que inicia el proceso de cacheado. Y sin cambiar de directorio, ejecutar:

make

De este modo, el usuario puede comenzar a utilizar el componente deseado sin necesidad de conocer nada sobre su implementación interna, o modificarlo y generar el nuevo ejecutable fácilmente.

4.2.2. Como emplea CMake un desarrollador de Jderobot

Para añadir un nuevo componente que haga uso de alguna/s librería/s contenidas en Jderobot y no incorpore ninguna dependencia nueva no contemplada en el directorio "Deps/" (de lo contrario sería necesario añadir su fichero CMakeLists.txt correspondiente al directorio "Deps/") habría que llevar a cabo los siguientes pasos.

1. Añadir al fichero CMakeLists.txt contenido en "trunk/" el nombre del nuevo componente y asociarlo a la variable "COMPONENTS". De esta forma el componente será incluído en la *compilación desde la raíz.*

2. Crear un nuevo directorio en "trunk/src" con el nombre del nuevo componente, que debe coincidir con el nombre que se le dió en el paso anterior.

3. Añadir a dicho directorio el código fuente del componente y un nuevo directorio llamado "build/".

4. Añadir en build (/trunk/src/components/mi_componente/build) el CMakeLists.txt raíz, dónde se definirá:

En primer lugar, nombre del proyecto y variables globales utilizadas durante el proceso de compilación:

```
project (Jderobot_BASIC_COMPONENT)
cmake_minimum_required(VERSION 2.8)
# ENV VARS|
# Directorio donde se encuentran todas las interfaces ICE en C++|
SET( INTERFACES_CPP_DIR ${CMAKE_CURRENT_SOURCE_DIR}/../../../interfaces/cpp)
# Directorio donde se encuentran las librerías propias de Jderobot|
SET( LIBS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/../../../libs)
#Directorio donde se encuentran las cabeceras para las interfaces ICE|
SET( SLICE_DIR ${CMAKE_CURRENT_SOURCE_DIR}/../../../interfaces/slice)
#Conjunto de librerías propias de Jderobot que necesita el componente|
SET( LIBS_NEEDED   Jderobotice Jderobotutil colorspaces )
#Directorio donde se encuentran las definidas las dependencias
SET( DEPS_DIR ${CMAKE_CURRENT_SOURCE_DIR}/..
```

En segundo lugar, dependencias que necesesita el componente y son instaladas por medio de paquetes:

```
#  CHECK SYSTEM   #

###Conjunto de dependencias instaladas a partir de paquetes (.deb)

include(FindPkgConfig)

PKG_CHECK_MODULES(gtk20 REQUIRED gtk+-2.0)
include_directories(${gtk20_INCLUDE_DIRS})
link_directories(${gtk20_LIBRARY_DIRS})

PKG_CHECK_MODULES(gtkmm REQUIRED gtkmm-2.4)
include_directories(${gtkmm_INCLUDE_DIRS})
link_directories(${gtkmm_LIBRARY_DIRS})

PKG_CHECK_MODULES(libglademm REQUIRED libglademm-2.4)
include_directories(${libglademm_INCLUDE_DIRS})
link_directories(${libglademm_LIBRARY_DIRS})

###Conjunto de las dependencias contenidas en
Deps de las que hace uso el componente
include(${DEPS_DIR}/gearbox/CMakeLists.txt)
include(${DEPS_DIR}/ice/CMakeLists.txt)
include(${DEPS_DIR}/opencv/CMakeLists.txt)

include_directories(${LIBS_DIR}/progeo)

### Inicio del acceso recursivo hacia los ficheros de apoyo.
add_subdirectory (${CMAKE_CURRENT_SOURCE_DIR}/../../..
${CMAKE_CURRENT_SOURCE_DIR}/../../..)

add_subdirectory (${CMAKE_CURRENT_SOURCE_DIR}/..
${CMAKE_CURRENT_SOURCE_DIR}/..)
```

En

tercer lugar, estalecer las reglas para la instalación, es decir, dónde instalar qué:

```
#    INSTALL    #

#Librerías usadas por el componente.
INSTALL(FILES
${CMAKE_CURRENT_SOURCE_DIR}/../../
../../src/interfaces/cpp/Jderobot/libJderobotInterfaces.so
DESTINATION /usr/local/lib/Jderobot)

FOREACH(currentLibFile ${LIBS_NEEDED})
    SET (new_lib "lib${currentLibFile}.so")
    MESSAGE("${new_lib}")
    INSTALL (FILES
${CMAKE_CURRENT_SOURCE_DIR}/../../../../src/
libs/${currentLibFile}/${new_lib}
DESTINATION /usr/local/lib/Jderobot)
ENDFOREACH(currentLibFile)

# Cabeceras del componente.
FILE(GLOB_RECURSE HEADERS_FILES
${CMAKE_CURRENT_SOURCE_DIR}/../../../../src/libs/*.h)
FOREACH(currentSourceFile ${HEADERS_FILES})
    string(REGEX REPLACE ".*/(.*/).*.h" "\\1"
new_source1 ${currentSourceFile})
    INSTALL (FILES ${currentSourceFile} DESTINATION
/usr/local/include/Jderobot/${new_source1})
ENDFOREACH(currentSourceFile)

# Ejecutable del componente.
FILE(GLOB_RECURSE BIN_FILES ${CMAKE_CURRENT_SOURCE_DIR}/../*cfg)
    string(REGEX REPLACE ".*/(.*).cfg" "\\1" new_source1 ${BIN_FILES})
INSTALL (FILES ../${new_source1}  DESTINATION /usr/local/bin PERMISSIONS
OWNER_EXECUTE GROUP_EXECUTE WORLD_EXECUTE)
```

En quinto lugar, establecer las reglas para la desinstalación de los elementos instalados en el apartado anterior:

```
### Cabeceras de las interfaces.
FILE(GLOB HEADER_INTERFACE_FILES
${CMAKE_CURRENT_SOURCE_DIR}/../../../../src/interfaces/cpp/Jderobot/*.h)
INSTALL (FILES ${HEADER_INTERFACE_FILES} DESTINATION
/usr/local/include/Jderobot/Jderobot)

### Ficheros .ice
FILE(GLOB SLICE_FILES
${CMAKE_CURRENT_SOURCE_DIR}/../../../../src/interfaces/slice/Jderobot/*.ice)
INSTALL (FILES ${SLICE_FILES} DESTINATION
/usr/local/include/Jderobot/slice)

### Ficheros .conf
FILE(GLOB_RECURSE CONF_FILES  ${CMAKE_CURRENT_SOURCE_DIR}/../*cfg)
INSTALL (FILES ${CONF_FILES} DESTINATION /usr/local/share/Jderobot/conf)

### Ficheros .glade
FILE(GLOB_RECURSE GLADE_FILES ${CMAKE_CURRENT_SOURCE_DIR}/../*.glade)
INSTALL (FILES ${GLADE_FILES} DESTINATION /usr/local/share/Jderobot/glade)

#    UNINSTALL    #

### Descripcion para desinstalar todo aquello instalado previamente.

configure_file(
    "${CMAKE_CURRENT_SOURCE_DIR}/cmake_uninstall.cmake.in"
    "${CMAKE_CURRENT_BINARY_DIR}/cmake_uninstall.cmake"
    IMMEDIATE @ONLY)

add_custom_target(uninstall
    COMMAND ${CMAKE_COMMAND} -P
    ${CMAKE_CURRENT_BINARY_DIR}/cmake_uninstall.cmake)
```

Además, debemos añadir otro fichero CMakeLists.txt en el directorio del componente (/trunk/src/components/mi_componente) donde se definirán los ficheros fuentes que lo conforman, dónde buscar las cabeceras y librerías, nombre del ejecutable y librerías con las que enlazar:

```
### Definir que ficheros fuentes componen el componente.
SET( SOURCE_FILES  control.cpp basic_component.cpp API.cpp gui.cpp)

### Especificar rutas donde el sistema buscará las cabeceras (.h)
include_directories(
    ${INTERFACES_CPP_DIR}
    ${LIBS_DIR}
    ${CMAKE_CURRENT_SOURCE_DIR}
)

### Definir el nombre del ejecutable.
add_executable (basic_component  ${SOURCE_FILES})

###Especificar las librerías con las que enlazar el ejecutable.
TARGET_LINK_LIBRARIES(basic_component
    ${INTERFACES_CPP_DIR}/Jderobot/libJderobotInterfaces.so
    ${LIBS_DIR}/Jderobotice/libJderobotice.so
    ${LIBS_DIR}/Jderobotutil/libJderobotutil.so
    ${LIBS_DIR}/colorspaces/libcolorspacesmm.so
    ${libglademm_LIBRARIES}
    ${OpenCV_LIBRARIES}
        ${ZeroCIce_LIBRARIES}
        ${gtkmm_LIBRARIES}
        ${gtkmm3_LIBRARIES}
        ${gthread_LIBRARIES}
)
```

Además de la funcionalidad descrita en la *compilación por componentes*, ésta fue extendida permitiendo la compilación de un componente sin necesidad de descargar todo el árbol de directorios de Jderobot. Un usuario podrá entonces descargar un componente determinado y compilarlo, siempre y cuando haya instalado en su máquina previamente todas las librerías de Jderobot que pueda utilizar dicho componente. Los

ficheros CMakeLists.txt serán idénticos a los explicados previamente salvo que las rutas hacia los directorios definidos para la resolución de dependencias deberán ser absolutos, en lugar de relativos.

4.2.3. Pruebas de verificación

Durante la implantación de la nueva herramienta, ésta era sometida a pruebas en dos equipos con configuraciones diferentes para tratar de adaptarla a diferentes entornos. Para ello se instalaba/desinstalaba software utilizado por Jderobot y se estudiaba el resultado tras los cambios. Una vez la herramienta estuvo afinada se puso en producción y comenzó a extenderse su uso entre todos los desarrolladores de Jderobot y nuevos usuarios.

Con el aumento del uso de la herramienta y con ello la heterogeneidad de los equipos donde era usada, se generó una gran realimentación muy útil por parte de usuarios y desarrolladores, que llevó a un constante arreglo de fallos, mejoras y optimizaciones, consiguiendo así una cadena de compilación cada vez más estable y robusta que a día de hoy es usada por toda la comunidad de Jderobot con buenos resultados.

Para facilitar el mantenimiento de esta herramienta, decidimos añadir una *tabla de dependencias* en el manual oficial de la plataforma [2] desde la que es posible consultar los requisitos software (librerías o dispositivos necesarios) que debemos poseer en nuestro equipo para construir cada uno de los elementos que conforman Jderobot.

4.3. Paquetes debian de Jderobot 5.1

El tercer y último módulo en los que se divide el proyecto aborda el diseño y construcción de un conjunto de paquetes debian (.deb) que agrupen todas las novedades incorporadas en la nueva versión, tanto las explicadas en este proyecto como las realizadas por parte de la comunidad de desarrolladores. La finalidad de estos paquetes es ofrecer a los usuarios y desarrolladores un método sencillo de instalar los elementos que conforman la plataforma, reduciendo el proceso a un simple comando, *apt-get install*. Además de la facilidad, también se consigue una mejor mantenibilidad de Jderobot al poder actualizar los paquetes automáticamente cuando éstos reciban nuevas versiones.

El uso de estos paquetes no sólo sirve para que un usuario pueda descargar determinada aplicación de Jderobot y ejecutarla en su sistema, sino que aquellos interesados en un

[2] http://jderobot.org/index.php/Manual-5#Tables_of_components.2Flibraries.2Ftools_and_their_dependencies

posterior desarrollo de esa misma aplicación (componente) verán como todo el software externo, es decir, las dependencias, serán instaladas automáticamente obteniendo un entorno completamente listo para comenzar a desarrollar y usar la plataforma.

Las distribuciones elegidas para generar los paquetes han sido Ubuntu 12.04 (LTS) y Debian en su rama *testing*, ambas en su versión de 32 bits.

Como se pudo ver en la imagen 4.18, Jderobot está compuesto por una gran cantidad de componentes, que a su vez hacen uso de librerías e interfaces también incluídas en él. De entre todos ellos, aquellos eficiente con calidad y madurez fueron elegidos para ser añadidos al paquetizado. Dos opciones se barajaron a la hora de crear los paquetes necesarios:

- *Paquetización monolítica*: creación de un único paquete que instalase todo el software de Jderobot así como las dependencias externas que necesita, tal y como existía en Jderobot 5.0.
- *Paquetización atómica*: creación de un paquete por cada entidad existente, entendiendo por entidad cada uno de los componentes en el que se divide el proyecto, así como las librerías propias de éste.

La paquetización atómica fue la escogida, puesto que aportaba las siguientes ventajas con respecto a la otra:

- *Atomización de los paquetes*: cada componente se maneja como un único paquete, lo que permite la instalación, manipulación, actualización y mantenimiento de cada uno de ellos de manera independiente.
- *Instalación a medida*: El hecho de tener un único paquete conlleva la instalación de todas las librerías y componentes existentes, cuando lo más probable es que el usuario sólo necesite la instalación de alguno de ellos. Además, algunos componentes necesitan la instalación, para su uso, de software con un tamaño importante (por ejemplo componentes que utilizan *Kinect*) con lo que el usuario debería instalar una gran cantidad de librerías de terceros que posiblemente nunca llegase a usar.
- *Agrupación según categorías*: una característica importante añadida han sido *los paquetes virtuales* que permiten realizar agrupaciones de paquetes atómicos que tiene sentido se usen conjuntamente. Por ejemplo, si el funcionamiento de un componente hace uso de otro/s (por ejemplo *introrob* necesita de *gazeboserver*) se ha creado un paquete virtual conteniéndolos a todos ellos, simplificando aún más al usuario el proceso de disponer del software necesario a ejecutar.

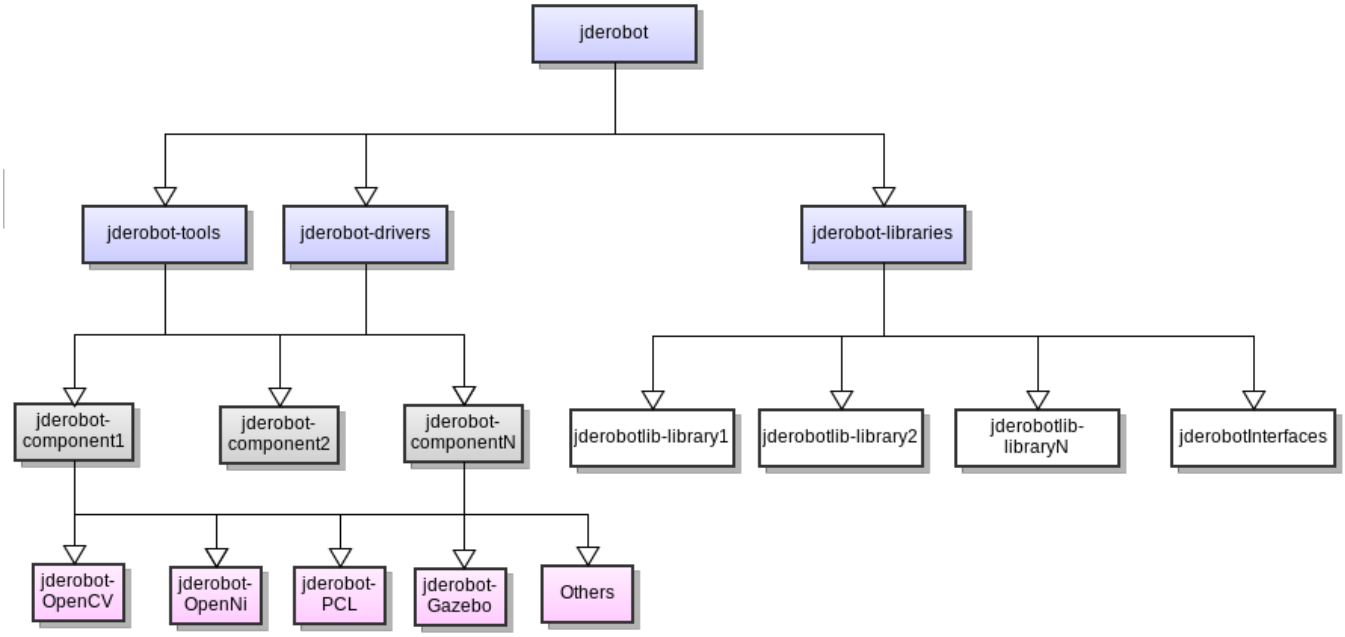


Figura 4.19: *Jerarquía de paquetes en Jderobot 5.1.*

La figura 4.19 muestra un esquema que ofrece una visión general sobre el diseño del paquetizado y cómo es la jerarquía de paquetes de Jderobot. En dicha figura se puede observar la existencia de una serie de niveles jerárquicos que, comenzando desde el nivel inferior, hacen referencia a:

- *Jderobot-componentXX*: Estos paquetes son el resultado de paquetizar cada uno de los componentes de manera independiente. En ellos están definidas todas las librerías (dependencias) que necesita, y por tanto los paquetes que deberán ser instalados para su correcta ejecución.

- *Jderobot-libraryXX*: Estas son las librerías propias de Jderobot, que son necesarias para la mayor parte de los componentes. Además, incluye el paquete *JderobotInterfaces* que agrupa todas las interfaces ICE utilizadas en el proyecto.

- *Jderobot-tools*: Paquete virtual que agrupa todos aquellos componentes categorizados como *herramientas* dentro del proyecto.

- *Jderobot-drivers*: Paquete virtual que agrupa todos aquellos componentes categorizados como *drivers* dentro del proyecto.

- *Jderobot-libraries*: Paquete virtual que agrupa todas las librerías existentes en el proyecto.

- *Jderobot*: Paquete virtual que engloba todo el software incluido en el proyecto Jderobot, de tal forma que instalando éste, se instalará todo el software relacionado con Jderobot.

- *Dependencias externas*: la mayoría de componentes hacen uso de librerías externas para su ejecución, las cuales en su mayor parte son distribuidas por sus desarrolladores por medio de paquetes, tanto para Ubuntu como para Debian. Sin embargo, existen algunos de ellos que deben ser instalados a través del código fuente, siendo necesaria su previa compilación con la complejidad que ésto supone (resolución de dependencias). Para facilitar al usuario la tarea de disponer de todo el software Jderobot de la manera más sencilla posible han sido creados paquetes para cada una de estas librerías externas que no lo incorporan de manera oficial, añadiendo el sufijo *jderobot-* para identificarlo como propio de la plataforma. De esta forma, el usuario puede abstraerse de descargar, compilar e instalar este software de manera independiente. Además, siempre existirá la posibilidad de reemplazar cualquiera de ellos por el original y que todo continúe funcionando.

La lista completa de paquetes generados para Jderobot 5.1 es:

- Librerías propias de Jderobot: *jderobotlib-bgfgsegmentation, jderobotlib-colorspaces, jderobotlib-colorspacesice, jderobotlib-fuzzylib, jderobotlib-jderobotice, jderobotlib-jderobotutil, jderobotlib-pioneer, jderobotlib-progeo, jderobotlib-visionlib, jderobotlib-jderobotinterfaces.*

- Componentes: *jderobot-bgfglab, jderobot-basic_component, jderobot-calibrator, jderobot-cameraview, jderobot-colortuner, jderobot-introrob, jderobot-opencvdemo, jderobot-recorder, jderobot-replayer, jderobot-teleoperator, jderobot-wiimoteclient, jderobot-cameraserver, jderobot-cameraview_icestorm, jderobot-gazeboserver, jderobot-playerServer, jderobot-wiimoteserver.*

- Software externo: *jderobot-gazebo, jderobot-gearbox, jderobot-player, jderobot-stage, jderobot-pcl, jderobot-opencv.*

Para el desarrollo de los paquetes fue necesaria la creación de diferentes jaulas con la herramienta *chroot*. Esta herramienta ofrece la posibilidad de instalar diferentes sistemas

operativos en una misma partición, pudiendo ejecutarse cada uno de éstos de manera simultánea haciendo uso de un terminal. La creación de estas jaulas es muy útil puesto que los sitemas operativos instalados en ellas traen las librerías básicas para su funcionamiento, emulando así un entorno donde no existe instalado nada relacionado con Jderobot, es decir, similares al entorno que tendrá el usuario en su máquina cuando desee instalar los paquetes. La simulación de estos entornos ofrece al desarrollador de paquetes la posibilidad de realizar las pruebas pertinentes sin necesidad de tener que instalar un sistema operativo desde cero, en una partición independiente, con todas las librerías adicionales (tales como la interfaz gráfica) que ocupan tiempo y espacio.

El proceso para crear cada una de las jaulas es muy sencillo utilizando las herramientas adecuadas:

1. Instalar las herramientas necesarias:

```
sudo apt-get install debootstrap
sudo apt-get install schroot
```

2. Crear un fichero donde se define la configuración del sistema a instalar:

```
sudo editor /etc/schroot/chroot.d/precise.conf
```

   Con el siguiente contenido:

```
[precise]
description=Ubuntu precise
directory=/var/chroot/precise
root-users=robotica
type=directory
users=robotica
```

   Donde *directory* es el directorio desde el que colgará el nuevo sistema.

3. Descargar y desempaquetar el sistema base por medio de la herramienta *debootstrap*.

```
sudo mkdir -p /var/chroot/precise
sudo debootstrap --variant=buildd --arch i386 precise /var/chroot/precise/
http://archive.ubuntu.com/ubuntu/
```

   Tras esto, se habrá instalado en el directorio /var/chroot/precise un sitema Ubuntu 12.04 listo para pruebas. Para arrancarlo sólo es necesario ejecutar:

```
sudo schroot -c precise -u root
```

Una vez listo el entorno desde el que crear los paquetes, el siguiente paso en el desarrollo de los paquetes es obtener las herramientas necesarias para ello:

- *dh-make*: Herramienta para convertir archivos de código fuente en paquetes de código fuente de Debian .
- *fakeroot*: ejecuta una orden en un entorno donde se simula que se tienen permisos de superusuario para la manipulación de ficheros.
- *devscripts*: Conjunto de scripts que facilitan la creación de los paquetes.
- *debhelper*: Un conjunto de programas que se pueden usar en un archivo de reglas de debian para automatizar las tareas comunes relacionadas con la generación de paquetes debian.
- *CMake*: Herramienta que generará los binarios y librerías a partir del código fuente para ser incluídos en los paqutes:

Con las herramientas descritas los comandos necesarios para crear el paquete se reducen a dos, primero:

```
dh_make --createorig
```

con el que se creará el directorio *debian/* que contendrá los ficheros de configuración necesarios para definir las reglas de los paquetes:

- *control*: En este fichero se definen datos básicos del paquete como son:
  - *Source*: Nombre del paquete.
  - *Maintainer*: Creador del paquete.
  - *Architecture*: Arquitecturas soportadas por el paquete.
  - *Depends*: Conjunto de dependencias del paquete.
- *rules*: En este fichero se definen las reglas para la creación del paquete. En este caso es recomendable dejar el fichero tal y como ha sido generado, aunque siempre es posible modificar alguna de sus funciones añadiéndole el sufijo *override* y describiendo el comportamiento que deseamos tenga esa función.

```
Source: Jderobot-gearbox
Section: unknown
Priority: extra
Maintainer: root <root@unknown>
Build-Depends: debhelper (>= 8.0.0), cmake
Standards-Version: 3.9.2
Homepage: <insert the upstream URL, if relevant>
#Vcs-Git: git://git.debian.org/collab-maint/gearbox.git
#Vcs-Browser: http://git.debian.org/?p=collab-maint/gearbox.git
;a=summary

Package: gearbox
Architecture: any
Depends: ${shlibs:Depends}, ${misc:Depends}
Description: <insert up to 60 chars description>
 <insert long description, indented with spaces>
```

Listado 4.9: *Ejemplo de un fichero control*

- *changelog*: En este fichero se almacena la información correspondiente a los cambios del paquete en cada una de las versiones.

Y segundo, una vez configurados los ficheros anteriores, tan sólo será necesario ejecutar el siguiente comando para crear el paquete:

```
dpkg-buildpackage -rfakeroot
```

4.3.1. Pruebas de verificación

Las pruebas realizadas para este módulo han sido muy intensivas, al igual que con CMake, puesto que se trata de un aporte para todos los desarrolladores y usuarios de Jderobot. Ello implica una gran heterogeneidad con respecto a los sistemas anfitriones donde podrá ser instalado. El proceso fue dividido en dos fases:

- *Servidor de pruebas*: Durante un período de tiempo los paquetes generados permanecieron en un servidor de desarrollo (o de pruebas), en el que fue necesario crear un repositorio de paquetes, con el fin de permitir a usuarios y desarrolladores la instalación de éstos y el reporte de errores vía lista de correo. Este hecho llevó a

```
#!/usr/bin/make -f
# -*- makefile -*-

%:
	dh $@

override_dh_shlibdeps:

override_dh_usrlocal:

override_dh_autobuild:
	make -j4
```

Listado 4.10: *Ejemplo de un fichero control*

una constante revisión de los paquetes y liberaciones continuas de nuevas versiones arreglando los problemas reportados.

- *Servidor de producción*: Cuando la comunidad de desarrolladores aprobó el correcto funcionamiento de los paquetes éstos fueron liberados al servidor de producción, ofreciendo así otra rama desde la que proveer paquetes estables a los usuarios.

Cabe destacar que actualmente este servidor de producción está siendo usado de forma oficial en Jderobot y es usado tanto por usuarios y desarrolladores, como alumnos del Máster de Visión Artificial y el Máster de Sistemas Telemáticos e Informáticos de la Universidad Rey Juan Carlos.

Capítulo 5

Conclusiones

En los anteriores capítulos se han definido de manera específica los objetivos, la infraestructura utilizada para el desarrollo del Proyecto Fin de Carrera, así como los componentes y herramientas implementados a partir de ésta. A continuación se explicará cuales son las conclusiones alcanzadas tras toda su evolución y los posibles trabajos futuros a los que da pie éste.

5.1. Conclusiones

El objetivo principal conseguido ha sido liberar la nueva versión 5.1 de la plataforma Jderobot, agrupando toda la evolución que ésta ha vivido desde su anterior versión.

Esta nueva versión debía incorporar un conjunto de aportes propios que hemos dividido en tres módulos:

- En primer lugar *componentes*: hemos creado el componente *basic component*, con el que se ha diseñado un nuevo esqueleto que sirve como patrón básico para el desarrollo de aplicaciones de Jderobot.

  También:

  - Se ha refactorizado *gazeboserver*: el *driver* utilizado para conectar el simulador Gazebo con otros componentes por medio de interfaces ICE. Este componente ha sido creado de cero para ofrecer compatibilidad con la nueva versión 1.5 de Gazebo manteniendo todas las interfaces ICE que ya ofrecía *gazeboserver* en Jderobot 5.0, eliminando así la necesidad de tener que modificar el resto de componentes que hacían uso del driver. El nuevo diseño consta de un conjunto

de *plugins* que implementan la funcionalidad de cada uno de los dispositivos (sensores y actuadores) que monta el robot en el simulador.

- Se ha mejorado la herramienta *introrob*: utilizada para el desarrollo de algoritmos robóticos y empleada en docencia. Ha sido incorporado el nuevo esqueleto, mejorada su GUI (interfaz gráfica de usuario) y mejorada y ampliada la API local que se ofrece a los programadores para facilitarles la tarea de implementar sus algoritmos.
- Se ha refactorizado *teleoperator*: utilizado para la teleoperación manual, por medio de una interfaz gráfica, de un robot simulado o real. La nueva versión de este componente se ha llevado a cabo desde cero, aplicando el nuevo esqueleto y añadiendo en él una nueva funcionalidad conocida como *interfaces ICE dinámicas*. Esta nueva característica permite crear o destruir en tiempo de ejecución una conexión a la interfaz ICE de cada uno de los sensores y actuadores que contiene el robot teleoperado.
- Se han refactorizado *wiimoteClient* y *wiimoteServer*: estos dos componentes hacen uso del dispositivo *wiimote* para capturar los datos obtenidos por sus sensores (vía bluetooth) y servirlos a través de interfaces ICE (*wiimoteServer*), así como mostrarlos en una interfaz gráfica desde la que es posible interactuar con el propio dispositivo (*wiimoteClient*).

- En segundo lugar, se ha diseñado una cadena de compilación con *CMake*: diseño e integración de una nueva herramienta que sustituyera Autotools y permitiese la construcción de todos los componentes que conforman Jderobot, junto a sus librerías y software externo que utilizan. Su funcionalidad permite construir toda la plataforma o aquellos componentes que se deseen a través de simples comandos. Facilita a los desarrolladores la tarea de incluir nuevos componentes a la plataforma o modificar ya existentes, y a usuarios poder construir éstos.

- En tercer lugar, se han diseñado y creado *paquetes debian*: diseño y creación de un conjunto de paquetes atómicos que permiten instalar aquellos elementos de la plataforma (componentes, librerías, interfaces, etc) que se deseen de forma individual. Diseño y creación de paquetes virtuales que agrupan paquetes atómicos que tiene sentido sean instalados conjuntamente. Gracias a estos paquetes se mejora y facilita el mantenimiento y la instalación de la plataforma a todos sus usuarios y desarrolladores en plataformas punteras como Ubuntu y Debian.

Para conseguir estos objetivos, en el camino fueron necesarios numerosos aportes *extras*

que si bien no han sido mencionados de manera directa en la presente memoria, sí merecen una especial mención. Algunos de estos aportes han sido:

- Desarrollo de diferentes recursos asociados al uso de Jderobot en docencia como entorno de prácticas robóticas:
  - *mundos para Gazebo*: además de refactorizar *gazeboserver*, ha sido necesaria la creación de diferentes mundos simulados útiles para las prácticas realizadas por los alumnos que cursan la asignatura de robótica. Estos mundos incorporaban una serie de características necesarias para crear diferentes situaciones que los alumnos deberían abordar por medio de algoritmos, como por ejemplo:
    - Introducir dos robots en un mismo mundo, ofreciendo la posibilidad de interactuar entre ellos.
    - Introducir en los mundos diferentes objetos que posteriormente los alumnos deberían percibir y posicionar en 3D por medio de las imágenes captadas por las cámaras.
    - Crear escenarios similares a laberintos donde el robot debe ser capaz de localizarse.
  - *modelos de robots para Gazebo*: Además de los anteriores mundos, con la llegada de la nueva versión de Gazebo hubo que definir nuevos modelos (*Pioneer2dx.model*) que integraron los *plugins* desarrollados en *gazeboserver* y ofreciesen las características necesarias para simular su comportamiento y equipamiento sensorial. Por ejemplo, fue necesario añadir dos cámaras en el frontal del pioneer, así como un láser fijado también en su parte superior, dado que el conjunto pioneer-cámaras-láser no lo aportaba Gazebo de manera predeterminada.
- Participación y documentación de los nuevos aportes:
  - *Soporte a dudas en la lista de Jderobot*: el uso, por parte de esta comunidad, de las herramientas incorporadas en la nueva versión ha generado un gran flujo de información alrededor de ellas, siendo necesaria una participación constante que facilitase a todos, tanto desarrolladores como usuarios, su adaptación a ellas. Este hecho me ha permitido conocer de primera mano cómo un proyecto no sólo es su desarrollo, sino también su mantenimiento y comunidad.

- *Documentación en su wiki oficial*: Todos los nuevos aportes debían ser documentados en la *wiki* oficial del proyecto [1], siempre usando para ello el inglés. Además de documentar los componentes, se realizaron otros aportes como tutoriales [2] y manuales [3] para aprender a usar las nuevas herramientas introducidas en la nueva versión.
- *Charla sobre el lanzamiento oficial de Jderobot 5.1*: Una vez liberada la versión 5.1 de la plataforma, realizamos una exposición en la que fueron presentadas sus principales características y las nuevas herramientas incorporadas, tales como CMake y paquetes *debian*.

El desarrollo de todos mis aportes en la plataforma Jderobot me ha permitido adquirir y mejorar mis aptitudes como ingeniero. Gracias, por ejemplo, a la cooperación con otros desarrolladores de la comunidad, ya que Jderobot no es desarrollado por una única persona, evidentemente, ha sido necesaria la cooperación (en mayor o menor medida según la situación) con los demás integrantes de la comunidad.

También ha sido necesaria la puesta en producción de los nuevos aportes, donde todos los desarrollos, tanto de herramientas como de aplicaciones, introducidos en la nueva versión han sido puestos en producción tras su finalización. Este hecho ha desembocado en su uso real por parte de dos tipos de clientes:

- Usuarios: los alumnos de los Másteres de Visión Artificial y de Sistemas Telemáticos e Informáticos de la URJC, quiénes utilizan la plataforma para el desarrollo de sus prácticas en la asignatura de robótica y por lo tanto demandan su eficiencia y usabilidad.
- Comunidad de Jderobot: desde desarrolladores que comienzan a utilizar la plataforma, y que han visto facilitada su adaptación gracias a nuevas aplicaciones destinadas a facilitar la curva de aprendizaje, como otros más veteranos que han sido partícipes del potencial existente en las nuevas herramientas introducidas.

Y en tercer lugar, la manipulación de amplio software externo y tecnologías. Debido al carácter heterogéneo del Proyecto, donde han sido implementadas diferentes aplicaciones, drivers y herramientas, ha sido necesaria la manipulación de un gran número de tecnologías diferentes. Por otro lado, Jderobot se apoya en una amplia cantidad de software externo:

[1] http://jderobot.org/index.php/Main_Page
[2] http://jderobot.org/index.php/CMAKE_FOR_DEVELOPERS
[3] http://jderobot.org/index.php/Manual-5#Building_JDErobot_5.0_with_CMake

OpenCV, Gazebo, Gearbox, ICE, etc. que hace necesario no sólo un gran conocimiento de la propia plataforma, sino de todo sobre el que se apoya.

Heterogéneo es el adjetivo que mejor describe este Proyecto Fin de Carrera, que si bien genera numerosas curvas de aprendizaje en cada una de sus etapas, también nutre a su vez de una gran cantidad conocimientos.

El desarrollo del proyecto, aunque de una plataforma robótica se trata, no se ha centrado en la implementación de algoritmos como localización, visión, navegación u otros comportamientos propios de un robot. Su enfoque ha tenido un carácter más genérico, de Ingeniería de Software, aplicable a cualquier proyecto, más aún teniendo en cuenta que Jderobot tiene un tamaño considerable y se asienta sobre un escenario idéntico (wiki, repositorio, listas, blog, etc) sobre el que se puede asentar actualmente cualquier proyecto de software libre.

En este trabajo, el número de líneas de código, entre todos los componentes, aportadas al proyecto suman unas 9000, de las cuales un 50 % de éstas son comunes en todos ellos dado su carácter reutilizable, y un 30 % reutilizado de componentes ya existentes, además de los paquetes y la herramienta CMake.

5.2. Trabajos futuros

Como todo proyecto de ingeniería, su funcionalidad puede ser afinada dando origen a otros tantos que permitan la mejora de todos los aportes que en éste se han realizado, tales como:

- Creación de nuevos componentes siguiendo la nueva estructura incorporada en esta versión.
- Continuar optimizando el componente *introrob* y seguir adaptándolo a las prácticas que se llevan a cabo en la asignatura de robótica, para ofrecer año tras año un mejor recurso a los alumnos que hacen uso de él.
- Continuar mejorando y manteniendo la estructura CMake diseñada, puesto que esta herramienta seguirá creciendo junto con Jderobot y será necesario un mantenimiento para que contínue sirviendo a todos los desarrolladores que hacen uso de ella, así como mejorarla para optimizar al máximo su proceso de compilado y enlazado.

- Ampliar *gazeboserver*, ya que sólo ofrece soporte para un robot *Pioneer*, pero se ha abierto camino para ser más sencillo el desarrollo de nuevos drivers que permitan la comunicación con otros tipos de robots como el Nao.

- Actualizar paquetería, ya que al igual que CMake, en pocos meses los paquetes actuales estarán obsoletos y habrán sido añadidas diferentes mejoras en cada uno de los desarrollos que será necesario ofrecer soporte a través de esta herramienta.

Bibliografía

[Cañas Plaza *et al.*, 2006] José María Cañas Plaza, Antonio Pineda, Jesús Ruíz-Ayúcar, José A. Santos, and Javier Martín. Programación de robots con la plataforma jde.c. *Informe Técnico. Universidad Rey Juan Carlos*, 2006.

[Cañas Plaza, 2003] José María Cañas Plaza. *Jerarquía Dinámica de Esquemas para la generación de comportamiento autónomo.* PhD thesis, Universidad Politécnica de Madrid, 2003.

[Cañas Plaza, 2009] José María Cañas Plaza. Programación de robots con la plataforma jderobot. *Informe técnico. Universidad Rey Juan Carlos*, 2009.

[CMa, 2012] Cmake reference manual. `http://www.cmake.org/cmake/help/v2.8.10/cmake.html`, 2012.

[Davies, 2011] Tracy Davies. Dawn of kate's gazebo simulation. *California Polytechnic State University*, 2011.

[Gerkey *et al.*, 2003] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. *International Conference on Advanced Robotics*, 2003.

[GTK, 2011] Gtk reference manual. `http://developer.gnome.org/gtk/`, 2011.

[Gutiérrez *et al.*, 2013] Marco A. Gutiérrez, A. Romero-Garcés, B. Bustos, and J. Martínez. Progress in robocomp. *Journal of physical agents*, 7(1), 2013.

[Henning and Spruiell, 2010] Michi Henning and Mark Spruiell. *Distributed Programming with Ice.* ZeroC, `http://www.zeroc.com/doc/Ice-3.4.1/manual/`, 2010.

[Lobato Bravo, 2005] David Lobato Bravo. jde+: Una plataforma de desarrollo para aplicaciones robóticas. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2005.

[Lobato Bravo, 2010] David Lobato Bravo. Jderobot 5: Entorno de desarrollo basado en componentes para aplicaciones robóticas. Master's thesis, Universidad Rey Juan Carlos, 2010.

[Pac, 2012] Ubuntu packaging guide. `http://developer.ubuntu.com/packaging/html`, 2012.

[Plaza, 2006] José María Cañas Plaza. Programación de robots móviles. *Revista Iberoamericana de Automática e Informática Industrial*, (2):99–110, 2006.

[Quigley *et al.*, 2009] Morgan Quigley, Gerkey Brian, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. *Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA)*, 2009.

[Ruiz-Ayúcar Vázquez, 2007] Jesús Ruiz-Ayúcar Vázquez. jdeneo.c: Una plataforma para desarrollo de aplicaciones robóticas. *Proyecto Fin de Carrera. Ing. Informática - Universidad Rey Juan Carlos*, 2007.

[Vázquez Pereda, 2010] Javier Vázquez Pereda. Docencia de robótica con jderobot5. Master's thesis, Universidad Rey Juan Carlos, 2010.