UNIVERSIDAD REY JUAN CARLOS

Máster Universitario en Sistemas Telemáticos e Informáticos

Curso académico 2009/2010

Trabajo fin de máster

jderobot 5
Entorno de desarrollo basado en componentes para aplicaciones robóticas

Autor: David Lobato Bravo

Tutor: José María Cañas Plaza

Ad utrumque paratus

Resumen

La aplicación de la robótica en los últimos años ha ido mas allá del ámbito industrial donde nació. En la actualidad han aparecido robots en campos tan insospechados como el ocio y el entretenimiento, y en general en entornos habitados como por ejemplo el robot aspirador Roomba. Pero el hardware de un robot en sí mismo no hace nada *motu proprio*, es inerte. Lo que da vida a estos componentes materiales es el *software* que los gobierna, los programas. Ellos recogen la información proporcionada por los sensores y calculan los comandos que se envían a los actuadores, determinando de este modo el comportamiento del robot. Esos programas siguen cierta arquitectura software y habitualmente se engloban en un entorno de desarrollo que aporta herramientas y utilidades para facilitar la tarea del desarrollador.

El objetivo global de este trabajo fin de máster es crear una nueva implementación para la plataforma de desarrollo *jderobot* que aplique las técnicas encontradas en las últimas tendencias en la materia, que apuestan por diseños que facilitan la reutilización (orientación a componentes), no fijan ningún tipo de organización de sus partes, permiten ser programadas en múltiples lenguajes y son capaces de ejecutarse en diversas plataformas. El trabajo se apoya en todo el conocimiento adquirido con versiones anteriores, introduciendo las nuevas tendencias de la mano del entorno de desarrollo *Orca*, que se ha utilizado como base de este proyecto. La tecnología subyacente se apoya en el *middleware* de comunicaciones Ice de ZeroC que nos permitirá abordar la programación multi-lenguaje y multi-plataforma con un esfuerzo mínimo. El uso de proyectos de terceros como parte de nuestro software refuerza la idea principal tras este trabajo: *la reutilización del software*. Ya sea desde el punto de vista de crear software que pueda ser fácilmente reutilizado por terceros, como desde el punto de vista de no desarrollar aquello para lo que ya exista una solución aceptable, es decir, *no reinventar la rueda*.

jderobot 5 pretende ser la plataforma estándar para el desarrollo de aplicaciones dentro del grupo de robótica de la URJC y es por ello que se ha validado desarrollando diferentes aplicaciones sobre ella. Una se describe en este trabajo y consiste en la re-escritura de *Carspeed* sobre *jderobot* 5. Además se han desarrollado algunos componentes básicos que han servido de base para otras aplicaciones.

Agradecimientos

Me gustaría agradecer a todos los que de alguna manera me han impulsado y dado ánimos para llevar a cabo este trabajo fin de máster. El camino has sido largo y duro, pero gracias a ellos he seguido adelante y he llegado a la meta.

Principalmente a mi madre y a mi padre que siempre han estado ahí, lloviese o hiciese sol.

A mis amigos, principalmente a los miembros del *consejo*, que siempre han tenido una palabra sabia y amable en momentos de debilidad. Y también a Laurita que siempre me saca una sonrisa.

Thanks also to my friends Brendon and Cathy that help me to continue my journey.

Y por último, a José María Cañas por empujarme y prestarme la ayuda y soporte necesario para llevar a cabo este trabajo.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1 Un poco de Historia

La palabra *robot* proviene del término checo *robota*, cuyo significado es "servidumbre, trabajo forzado o esclavitud". Principalmente usado para referirse a los siervos Checoslovacos de la época feudal, que eran obligados a trabajar dos o tres días a la semana en las tierras de los nobles sin ningún tipo de retribución. Pasado el sistema feudal, siguió usándose el término, para referirse al trabajo que no era realizado de manera voluntaria. En la actualidad, el término es usado por la juventud Checa y Eslovaca para referirse al trabajo aburrido o poco interesante. Según la Real Academia Española, el término robot se refiere a:

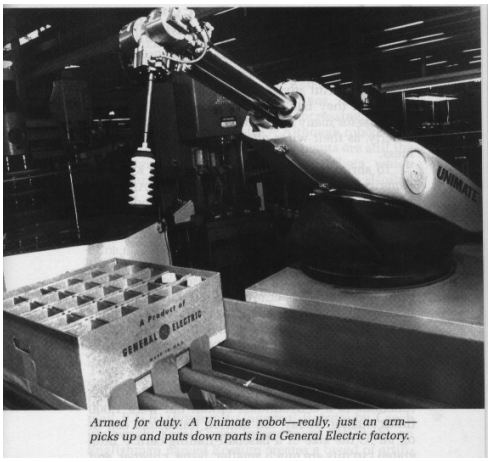
> *Máquina o ingenio electrónico programable, capaz de manipular objetos y realizar operaciones antes reservadas sólo a las personas*

Fue Karel Capek, quien utilizó por primera vez el término *robot* en su obra teatral *Rossum's Universal Robots*[Capek, 1923]. La obra, estrenada en 1921, narra la historia de Rossum, un científico, que consigue crear *robots* capaces de servir a la clase humana. Tras su perfeccionamiento, los *robots* se rebelan contra sus amos destruyendo toda vida humana. El término continúa usándose en varios escritos posteriores pertenecientes al género literario de la ciencia-ficción, pero no fue hasta años más tarde, en 1942, cuando Isaac Asimov utilizó por primera vez el termino robótica para referirse a la tecnología de los robots, en la novela *Runaround*.

Como precursores históricos de los robots actuales, tenemos diversos autómatas, diseñados y construidos por grandes mentes muy avanzadas a su época. Como ejemplos citar el Gallo de la catedral de Estrasburgo de autor desconocido en 1352, el guerrero mecánico de Leonardo Da Vinci en 1495, varios muñecos animados de Jacques Vaucanson en 1738, autor también del primer telar mecánico, o la muñeca dibujante de Henry Maillardet en 1805.

La robótica como hoy la conocemos, surge hacia la década de los 50, con la construcción de las primeras computadoras. Aunque no será hasta la década de los 70 con los primeros microprocesadores cuando comience su vertiginoso avance. Las primeras patentes aparecen en 1946, describiendo primitivos robots para el traslado de maquinaria. Será en 1954 cuando George Devol diseñe el primer mecanismo programable aplicado a la industria, el *Universal Automation*, más tarde reducido a Unimation y que sería el corazón del primer brazo robótico industrial (figura 1.1). Junto con Joseph F. Engelberger, G. Devol formará la primera compañía comercial dedicada a la robótica, Unimation Inc., dedicada a robots industriales para cadenas de montaje.

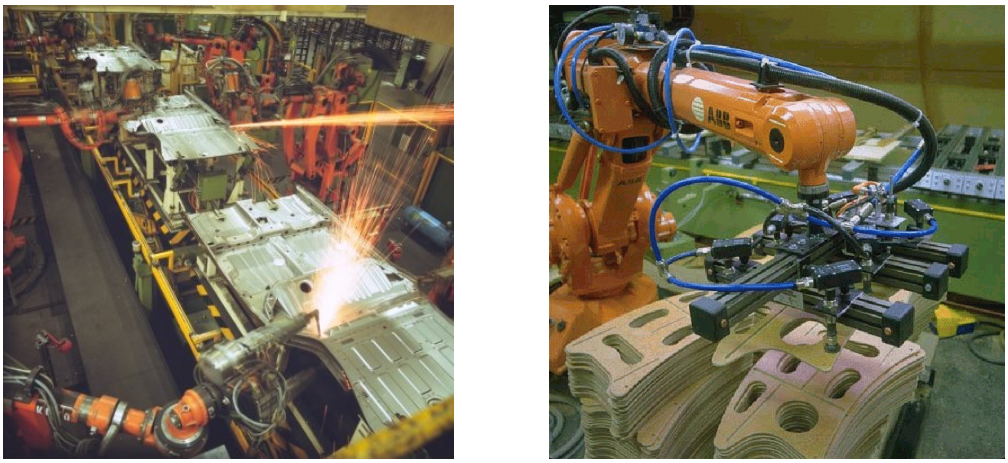
Un hito paradigmático en este escenario son los brazos articulados tipo PUMA (Programmable Universal Manipulator for Assembly). Se utilizan con frecuencia para soldar, pintar, transporte de materiales, etc. tal como ilustra la figura 1.2. Por ejemplo



Armed for duty. A Unimate robot—really, just an arm—picks up and puts down parts in a General Electric factory.

Figura 1.1: *Unimate: Primer brazo robótico industrial.*

los puntos de soldadura se le dan predefinidos en una lista con la que se programa el recorrido del brazo. Las principales ventajas que ofrece un brazo robotizado frente a un operario humano para estas labores repetitivas son la precisión y la rapidez. Desde el punto de vista del empresario también hay que considerar que los robots no hacen huelga, no se aburren y pueden trabajar 24 horas al día, todos los días del año. Además salvando cierta inversión inicial, su coste de mantenimiento puede ser menor que el equivalente para que la misma labor la realicen operadores humanos.



(a) (b)

Figura 1.2: *Robots soldadores en una factoría de coches (a) y brazo transportando materiales (b)*

Los robots también aparecen en otros escenarios fuera de la fábrica. En entornos altamente peligrosos para las personas se utilizan robots para reemplazarlas. Por ejemplo, se utilizan robots teleoperados para explorar zonas de alta radiación en el interior de reactores nucleares, para rastrear en las profundidades oceánicas, para sondear volcanes o pirámides, para inspeccionar y desactivar bombas o minas. En estos casos el robot suele estar equipado con cámaras de vídeo y es guiado remotamente por un operador humano.

En los últimos años está creciendo el uso de robots para tareas agrícolas [Stentz et al., 2002]. Las tareas de fumigación o cosecha son potencialmente automatizables, porque son repetitivas y tediosas, e incluso conllevan cierto riesgo químico para el operario humano. Por ejemplo, hay robots recolectores de tomates, melones, pepinos o champiñones, que utilizan visión artificial para identificar el fruto adecuado [García Pérez, 2004]. Otra aplicación relevante en esta línea son los cosechadores automáticos de cereales, que re-

corren de modo semiautónomo los inmensos campos de cereales segando y recogiendo el grano.

Las exploraciones espaciales suponen otro campo de aplicación de la robótica. Se han utilizado robots en varias misiones. Por ejemplo, la misión Pathfinder puso al robot Sojourner en la superficie de Marte en 1997. El robot tenía 6 ruedas y básicamente se movía teleoperado desde la Tierra. Tomó varias imágenes, analizó el suelo y algunas rocas del planeta rojo, enviando los resultados a la Tierra. En esta línea la NASA ha invertido y sigue invirtiendo mucho en proyectos que desarrollan robots capaces de desenvolverse en entorno hostil tan lejano.

Uno de los campos insospechados por donde está creciendo la oferta robótica es el ocio y el entretenimiento. Por ejemplo, el perrito Aibo de Sony se vendió como mascota hasta marzo del 2006 y los robots humanoides Nao desarrollados por Aldebaran Robotics[1], aunque más ligados al mundo de la investigación, también permiten el acceso a tecnologías robóticas a un público entusiasta. También el ladrillo de Lego[2] se vende como juguete imaginativo. En este sentido hay que destacar también la RoboCup[3], que es una competición internacional anual en la que equipos de robots juegan al fútbol en distintas categorías.

Otro de los entornos donde la robótica toma fuerza es el hogar o en general entornos habitados (oficinas, hospitales,...), donde se intenta introducir tecnología robótica que nos permita automatizar determinadas tareas, o bien dotar de cierta inteligencia al entorno que nos rodea (domótica). Una muestra es el aspirador robótico *Roomba*[4] (figura 1.3) que deambula por una habitación de forma autónoma sin chocar con las paredes, tragando pelusas y deslizándose debajo de las camas y los sofás. Sus sensores evitan que choque contra las paredes o los muebles, o se caiga por el hueco de las escaleras. Otro ejemplo lo tenemos en el sistema *ElderCare* [5] desarrollado por el grupo de robótica de la Universidad Rey Juan Carlos, capaz de monitorizar una habitación y alertar si se detecta que una persona se ha caído y permanece inmóvil en el suelo.

En general, el uso de la robótica va creciendo paulatinamente, aumentando el rango de sus aplicaciones a medida que ésta aumenta en autonomía y funcionalidad. Precisamente la falta de autonomía es el principal cuello de botella que ha impedido el uso generalizado de robots. En la mayoría de las aplicaciones robóticas de hoy día los robots utilizados no tienen autonomía, son teleoperados o autómatas que ejecutan una sola tarea. Aunque algunos escenarios admiten teleoperación, en ciertas aplicaciones la autonomía es un requisito ineludible y en la mayoría, una característica muy ventajosa. Por ejemplo, en robots para exploraciones espaciales la teleoperación se hace demasiado lenta cuando se alejan de la Tierra y las ondas de radio tardan demasiado en llegar. En los robots de entretenimiento la gracia radica en que sean mascotas independientes, con su propias capacidades, y no meras marionetas pasivas. En las tareas agrícolas la ventaja principal se consigue al automatizar de modo completo el laboreo. Con ello se reduce al mínimo la intervención humana, quizás a una sencilla supervisión. Del mismo modo sucede con las tareas domésticas.

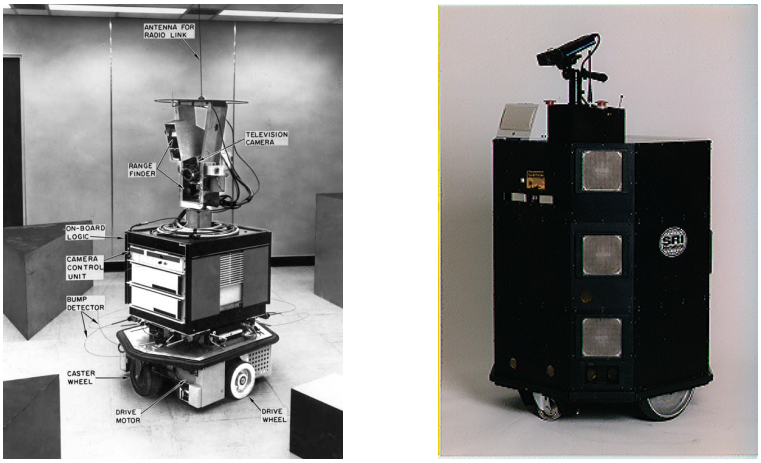
El camino hacia los robots móviles actuales ha sido largo. Los primeros proyectos en este área datan de principios de los años 70, cuando por primera vez se reúnen en una misma plataforma sensores, motores y procesadores [Serradilla, 1997]. Desde entonces se ha ido consolidando como un campo específico dentro de la robótica, en el cual el

[1] http://www.aldebaran-robotics.com
[2] http://www.legomindstorms.com
[3] http://www.robocup.org
[4] http://www.irobot.com
[5] http://jderobot.org/index.php/ElderCare

Figura 1.3: *Roomba de iRobot*

comportamiento principal es el movimiento y donde hay una especial preocupación por la autonomía. Además, la evolución de este campo ha ido definiendo sus propios problemas como la navegación, la construcción de mapas, la localización, etc.

El robot Shakey [Nilsson, 1969] (figura 1.4(a)) es un pionero dentro la robótica móvil, y fue construido en 1970 en el Stanford Research Institute (SRI). Tenía una cámara, motores y sensores odométricos. Era capaz de localizar un bloque en su entorno y empujarlo lentamente. El procesamiento de las imágenes se realizaba en un ordenador fuera del robot. Dentro del mismo instituto, un digno sucesor fue el robot Flakey (figura 1.4(b)), construido en 1984 con numerosos avances tecnológicos sobre Shakey, y que ya incluía en su interior ordenadores para realizar a bordo cualquier cómputo necesario para su comportamiento.



(a) (b)

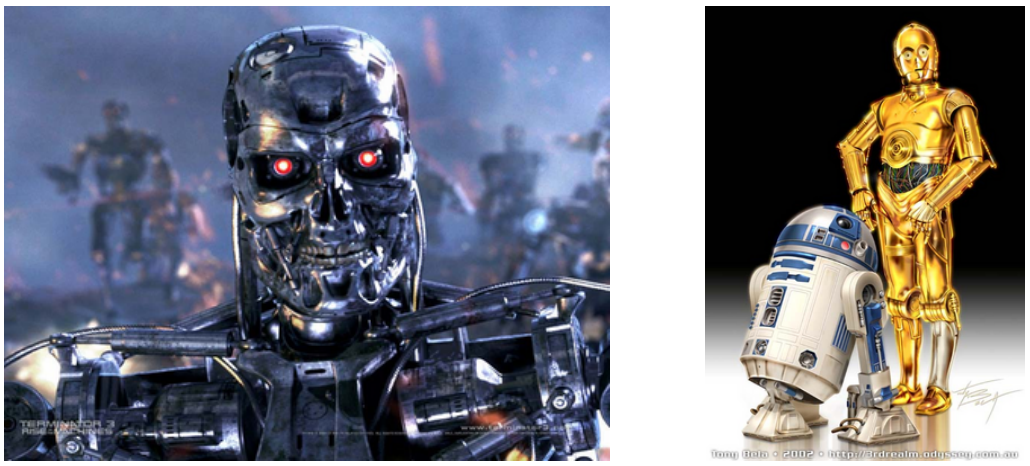
Figura 1.4: *Robots Shakey (a) y Flakey (b)*

Actualmente no es ciencia ficción ver a un robot móvil guiando a los visitantes por un museo[6], navegando por entornos de oficinas[7], o conduciendo un coche de un

[6]Minerva: http://www.cs.cmu.edu/~minerva
[7]Xavier: http://www.cs.cmu.edu/~xavier

extremo a otro de Estados Unidos (NavLab-CMU[8]). Estos prototipos son el fruto de la investigación de varias universidades y centros situados a la vanguardia de la robótica, como Carnegie Mellon University, el Massachusetts Institute of Technology o el Stanford Research Institute.

Aunque se ha progresado mucho aún estamos lejos de las expectativas levantadas por el cine o la literatura. Por ejemplo, robots móviles tan versátiles como C3PO y R2D2 de la Guerra de las galaxias, o los que aparecen en Blade runner o Terminator siguen siendo a día de hoy ciencia ficción. De hecho estamos muy lejos que conseguir una máquina que pase con buena nota el test de Turing [Turing, 1950]. En este terreno la imaginación ha ido muy por delante de la realidad y ha levantado unas expectativas demasiado prometedoras. Expectativas que no se han satisfecho, lo que provoca cierta frustración y descrédito. En este sentido las comunidades robótica y de inteligencia artificial se han encontrado con unas barreras muy complejas y difíciles de superar: el lenguaje natural, el aprendizaje, autonomía, adaptación, etc. La construcción de máquinas que se comporten de modo autónomo sigue siendo un reto y un problema abierto.



(a) (b)

Figura 1.5: *Robótica en la ciencia-ficción: Terminator (a), androides C3PO y R2D2 (b)*

En el terreno material, el crecimiento de la robótica móvil se ha beneficiado enormemente de la revolución electrónica que hemos experimentado en las últimas décadas. Las mejoras tecnológicas y en microelectrónica han disparado la capacidad de cómputo disponible y el desarrollo de los más variados sensores, motores y dispositivos. Por ejemplo, sensores precisos como el láser, el GPS, o las cámaras en color, y motores de continua, servos, etc. aparecen frecuentemente como parte del equipamiento de los robots móviles. La velocidad de los computadores se ha multiplicado en los últimos años considerablemente y esto ha hecho factible la materialización de algunas aproximaciones a la robótica móvil que demandan gran capacidad de cálculo. Pero todo este hardware, por potente que sea resulta inútil sin una buena programación que sepa como procesar sus datos y dar las respuestas adecuadas. Por lo que a pesar de estas mejoras tecnológicas, de disponer de los procesadores más rápidos, los sensores y actuadores más avanzados, el cómo combinarlos para generar comportamiento autónomo sigue siendo un problema abierto. Los progresos en el hardware han puesto en evidencia la importancia de una buena organización interna, que se ha convertido en un factor crítico para conseguir el comportamiento autónomo robusto.

[8]http://www.ri.cmu.edu/labs/lab 28.html

En este aspecto se ha trabajado largo y tendido desde un punto de vista teórico, describiendo como se organizan el sensado del entorno y la actuación de un robot, en busca de un comportamiento observable. Esta organización teórica se denomina arquitectura de control robótico o *arquitectura cognitiva*. Su aplicación práctica conduce a un problema de diseño software, que desemboca en un sistema software real. Dicho sistema permite el desarrollo de aplicaciones robóticas fundamentadas en una teoría de organización. A dichos sistemas los denominaremos *arquitecturas software*, aunque habitualmente estaremos hablando de entornos de desarrollo o *frameworks*, ya que estos sistema suelen reunir a su lado una serie de herramientas que facilitan la tarea del desarrollador de aplicaciones robóticas, tales como herramientas para depuración o para la configuración.

A día de hoy no se ha llegado a una conclusión sobre el mejor enfoque cognitivo para organizar comportamientos complejos, teniendo que en la mayoría de aplicaciones prácticas, se tiende a organizaciones híbridas, que permiten mayor flexibilidad a la hora de describir un comportamiento. De ahí que la tendencia en arquitecturas software se enfoque más en resolver problemas de organización y diseño software (sistemas basados en componentes, objetos, API's para acceso a hardware,...), dando total flexibilidad al usuario para organizar sus comportamientos como mejor le parezca o como mejor se aplique al escenario del problema en concreto. A este tipo de sistemas los clasificaremos dentro de las arquitecturas software neutras, que no fijan requisitos sobre organización, sino que permiten aplicar diferentes enfoques cognitivos.

En el grupo de robótica de la Universidad Rey Juan Carlos somos conscientes de ello, y esta es la principal de las motivaciones que nos mueve a dar un salto con este trabajo fin de máster hacia otra manera de organizar nuestros desarrollos, buscando una mayor flexibilidad y liberándonos en parte de la carga cognitiva (*JDE* [Cañas and Matellán, 2002]) que imponíamos hasta ahora a nuestras arquitecturas software, introduciendo las ventajas que aportan los enfoques de la ingeniería software basada en componentes al desarrollo de sistemas complejos como los robóticos. Por ello, el objetivo de este trabajo es la introducción de nuevas técnicas y herramientas para seguir desarrollando software robótico de calidad mucho más flexible y simple.

1.2 Arquitecturas cognitivas de control

Como se vio en el apartado anterior, una de las claves para la generación de comportamientos robustos es la correcta organización interna de los sensores y actuadores, así como del software que los gobierna. En esta sección mostraremos el estado del arte desde el punto de vista cognitivo de la arquitectura, mostrando las principales lineas, y desde el punto de vista práctico, veremos ejemplos de arquitecturas software relevantes.

Desde un enfoque sistémico un robot móvil se puede contemplar como un sistema que tiene por entradas los datos sensoriales y por salidas los posibles comandos sobre los actuadores. En este ámbito, de modo informal, entendemos que arquitectura es todo lo que hay entre los sensores y los actuadores. De manera más precisa llamamos arquitectura cognitiva, teórica, o simplemente *arquitectura* de un robot autónomo a *la organización de sus capacidades sensoriales, de procesamiento y de acción para conseguir un repertorio de comportamientos inteligentes interaccionando con un cierto entorno*. La arquitectura determina los comportamientos que exhibe un robot autónomo.

La importancia de la arquitectura radica en que los robots móviles son sistemas con un alto grado de complejidad, y cuanto más complejo es un sistema más relevancia

cobra el papel de la organización de sus componentes. Normalmente para tareas sencillas hay muchas maneras de abordarlas y el modo en que se resuelven, *el cómo*, casi no tiene importancia. Sin embargo al aumentar la complejidad la organización influye más en el resultado, puede incluso llegar a ser determinante: con una buena organización se consigue el objetivo y con una mala no. Por ejemplo, el papel de la arquitectura es más visible cuando la complejidad aumenta, bien porque el robot deba exhibir un repertorio completo de diferentes comportamientos, bien porque los sensores proporcionan una gran cantidad de datos del entorno, no todos relevantes. Además *el cómo* puede influir notablemente en la calidad del sistema, ya sea en su funcionamiento, en el tiempo de desarrollo necesario para construirlo o en otros aspectos. Para lograr sistemas repetibles y adaptables su organización interna ha de ser clara, accesible y fácilmente modificable. El estudio de cómo organizar estos procesos internos en robots es lo que denominamos arquitectura. Avances en la arquitectura conducen a comportamientos aun más complejos y a aumentar la autonomía de modo fiable.

El hardware de un robot en sí mismo no hace nada *motu proprio*, es inerte. Lo que da vida a estos componentes materiales es el *software* que los gobierna, los programas. Ellos recogen la información proporcionada por los sensores y calculan los comandos que se envían a los actuadores, determinando de este modo el comportamiento del robot. Si queremos que el robot se comporte de determinada manera tenemos que programarlo para que lo haga. Esos programas siguen cierta arquitectura software. Puede ser algo tan sencillo como un programa único en bucle infinito que ejecuta cíclicamente la secuencia: leer datos de los sensores, pensar, actuar. También puede ser varios procesos concurrentes que se comunican entre sí enviándose datos, estableciendo flujos de información entre ellos. O quizá habrá procesos que se dediquen a elaborar cierta información de salida desde sus datos de entrada, procesos que tomen decisiones, etc. Las distintas arquitecturas conceptuales se concretan en cierta arquitectura software, es decir en los programas que se ejecutan en los procesadores del robot.

En una primera aproximación generar un comportamiento autónomo se puede ver como un simple problema de control, en el que se trata que un sistema físico actúe de determinada forma en cierto entorno. Sin embargo, el problema de la arquitectura es más complejo que un problema de control. Hablamos de un controlador cuando tiene por entrada señales con la información relevante que están definidas con nitidez, cuando los objetivos no cambian de naturaleza (si acaso cambian las referencias) y cuando lo "único" que hay que hacer es elegir entre determinadas posibilidades de actuación para conseguir el objetivo. Por el contrario, hablamos de arquitectura cuando incluimos la tarea de construir o seleccionar esas señales de entrada y los sensores ofrecen una cantidad desbordante de información no específica. También cuando la naturaleza de los objetivos puede variar enormemente o cuando el robot tiene varios objetivos, con prioridades dinámicas que dependen de las necesidades actuales y de la situación del entorno. Tenemos un problema de arquitectura cuando el sistema ofrece un *repertorio* muy variado de comportamientos y hablamos de la interacción entre unos comportamientos y otros, cambio de un modo a otro, etc.

No existe *la arquitectura* válida para todos los entornos y para todos los comportamientos. Tradicionalmente el estado del arte en este campo se ha dividido en tres grandes corrientes: los sistemas deliberativos, los reactivos y los híbridos. Clasificaciones similares existen en [Murphy, 2000, Arkin, 1998, Arkin, 1995, Coste-Maniére and Simmons, 2000] y en muchas tesis de los años 90 [Schneider Fontán, 1996, Serradilla, 1997]

1.2.1 Sistemas deliberativos

La disciplina de la *Inteligencia Artificial* ha influido notablemente en el desarrollo de la robótica móvil. Las arquitecturas tradicionalmente consideradas como *deliberativas*,

o de la escuela simbólica, son las herederas de las investigaciones en IA clásica y fueron las primeras en surgir en el mundo de la robótica móvil para generar comportamiento, siendo las dominantes hasta finales de los años 80.

En general, estas arquitecturas asumen que generar comportamiento en un robot consiste en ejecutar una secuencia de acciones calculada de antemano (*plan*), que se elabora razonando sobre cierto modelo simbólico del mundo. El robot necesita una descripción simbólica del estado de su entorno (por ejemplo, un mapa de ocupación, con sus paredes, pasillos y puertas, o una relación de hechos lógicos sobre ese estado), y de su funcionamiento (por ejemplo, el efecto de sus propias acciones). La inteligencia del robot se halla principalmente en el planificador, el cual delibera sobre los símbolos del modelo del mundo y genera un plan de actuación explícito. El plan elaborado contiene el curso deseado de la acción en el futuro para conseguir el objetivo. Por ejemplo, la secuencia de operadores para conseguir el estado objetivo a partir del estado actual o la ruta óptima para llegar al punto destino.

Su inserción en robots reales consiste en ejecutar infinitamente el bucle *Sensar-Modelar-Planificar-Actuar* [Murphy, 2000]. En el primer paso (*Sensar*) se obtienen los estímulos del mundo exterior. En el segundo (*Modelar*) se construye una representación simbólica del mundo. En el tercero (*Planificar*) se elabora un plan, razonando sobre la representación simbólica del mundo que se obtuvo antes. Y por último (*Actuar*), se ejecutan las acciones del plan elaborado.

Las principales limitaciones de este tipo de arquitecturas son, que la planificación presenta explosión combinatoria en cuanto el problema a manejar crece en complejidad [Agre and Chapman, 1987] y, que la idea de mundo cerrado invariable no se ajusta a la realidad y los mecanismos lógicos no prevén cambios arbitrarios en el mundo, de manera que este tipo de sistemas no maneja bien la incertidumbre del mundo real.

Como ejemplos de este tipo de arquitecturas, tenemos el robot Shakey[9] (figura 1.4(a)), el robot Hilare[10], o los sistemas SOAR y RCS.

1.2.2 Sistemas Reactivos

A finales de los años 80, era notable la falta de flexibilidad de los robots reales conseguidos hasta la fecha, guiados por el enfoque deliberativo. Quizá motivados por esto, varios investigadores comenzaron a replantearse el modo de generar comportamiento, y el uso de los planes, que era la pieza central del paradigma deliberativo [Payton, 1990] [Agre and Chapman, 1990] [Firby, 1987].

Fruto de este replanteamiento nace el *enfoque reactivo*. Esta escuela reactiva, quizá liderada por los trabajos de Rodney Brooks [Brooks, 1986] [Brooks, 1991], provocó un giro drástico en cuanto al modo de generar comportamiento. El nuevo enfoque hace hincapié en una ligazón más directa entre los sensores y los actuadores, sin pasar por las etapas intermedias que utilizan los robots deliberativos. De esta manera la reacción a los eventos era mucho más rápida. Uno de los pasos intermedios que se consideran innecesarios es el modelo del mundo, y otro el manejo de símbolos. En este sentido el enfoque reactivo es subsimbólico, y argumenta que no es necesaria la representación simbólica, ni el razonamiento sobre símbolos para generar comportamiento. En términos de Murphy [Murphy, 1998], este planteamiento reduce las primitivas esenciales a sensar y actuar, que están directamente ligadas, y deja a un lado la de planificar.

Dentro de este enfoque podemos distinguir entre sistemas reactivos puros y sistemas basados en comportamientos. Teniendo ambos puntos en común, los segundos representan una evolución sobre los primeros.

[9] http://www.sri.com/technology/shakey.html
[10] http://www.laas.fr/∼matthieu/robots/

Los *sistemas reactivos puros* se basan en la conexión de situaciones y acciones para generar comportamiento. El funcionamiento principal es un rápido bucle que chequea los valores sensoriales, busca en la situación que le corresponde y ejecuta la acción recomendada por la asociación. Lo que implica que no hay que esperar a tener el entorno completamente modelado para emitir una actuación ventajosa. Esta asociación situación-acción se almacena de algún modo dentro del sistema, ya sea con una tabla rasa, con unas reglas borrosas, manualmente, con un programador de autómatas, etc.

La principales limitaciones de estos sistemas son las siguientes. Requiere la anticipación en tiempo de diseño de *todas* las situaciones en las que se va a encontrar el robot y el cálculo de la respuesta adecuada para todas ellas, siendo incapaces de improvisar ante situaciones desconocidas. La ausencia de anticipación temporal no les permite reaccionar hasta haber detectado (sensado) una situación determinada (ej. un obstáculo).

La característica principal de los *sistemas basados en comportamientos* (SBC) es que abogan por la distribución del control y la percepción en varias unidades que funcionan en paralelo, llamadas de diversas maneras: niveles de competencia, comportamientos, esquemas, agentes, controladores, etc. Esta descomposición por actividades contrasta con la descomposición funcional y el control monolítico típicos de las arquitecturas deliberativas. La idea subyacente es que el comportamiento complejo de un sistema es una propiedad emergente que surge de las interacciones de los componentes básicos entre sí y con el entorno.

Cada componente es un bucle rápido, reactivo, desde los sensores a los actuadores, que tiene un objetivo propio. En general cada componente unitario implementa las reacciones adecuadas ante ciertos estímulos accediendo directamente a los datos sensoriales que necesita y emitiendo control a los actuadores. Por ello, además de suponer una distribución del control, este paradigma también propone la distribución de la percepción. En este sentido no necesita representación centralizada del entorno para generar comportamiento, ni modelos simbólicos. La información necesaria está directamente en las lecturas de los sensores.

Las arquitecturas basadas en comportamientos se diferencian unas de otras en el modo en que interaccionan los diferentes comportamientos y en el método elegido para coordinarlos. Desde la jerarquía y la activación por prioridades, hasta la anarquía y la activación por un autómata finito de estados que hace las veces de árbitro.

Las principales limitaciones son la escalabilidad de estos sistemas, muy complejos por encima de 4 niveles de competencia, la falta de anticipación al no mantener una representación simbólica y la dificultad para fijar objetivos explicitos.

1.2.3 Sistemas Híbridos

A la luz de las limitaciones observadas en las escuelas reactiva [Mataric, 2002] y deliberativa [Brooks, 1991] han surgido muchas aproximaciones híbridas, que tratan de combinar las ventajas de ambos en una única arquitectura. Por ejemplo, aunar las habilidades de anticipación y orientación a objetivos que proporciona la planificación con la flexibilidad frente a imprevistos que proporciona la reactividad.

El paradigma híbrido, dentro de su heterogeneidad, combina los principios de la escuela deliberativa y de la reactiva, tratando de complementarlos y conseguir de esta manera comportamientos más robustos y complejos. La parte reactiva se ha mantenido muy próxima a los sensores y actuadores reales, dotando al sistema de flexibilidad. Sin embargo, esta parte no se escala hasta completar todo el sistema, como en los sistemas basados en comportamientos, sino que tiene por encima una capa deliberativa que

modula su funcionamiento. Esta parte deliberativa incluye la representación explícita de objetivos y la planificación como motor final del comportamiento.

La separación en parte deliberativa y reactiva atiende también a un principio de ingeniería software, que tiende a agrupar las cosas similares juntas. Así, a la hora de incorporar una nueva tarea o componente en una arquitectura híbrida hay que considerar si maneja información numérica que varía continuamente, o si la información que utiliza tiene una dinámica más lenta y es más abstracta.

Dentro de este enfoque se encuentran la arquitectura software TCA de Simmons y la arquitectura software Saphira de Konolige, arquitecturas software pioneras en su tiempo.

1.2.4 JDE

JDE[Plaza, 2003] es el nombre de la arquitectura de control ideada por el grupo de robótica de la URJC. Para JDE, una aplicación robótica tiene partes de percepción y partes de actuación, ejecutadas de manera concurrente. Cada una de estas partes se denomina esquema, y se agrupa con otras en jerarquía para producir un comportamiento observable. Las características principales de un esquema son que puede ser activado o desactivado a voluntad y que su funcionamiento puede ser modulado a través de parámetros. Así, unos esquemas *usan* la funcionalidad de otros activándolos y modulándolos según sus necesidades en un instante determinado. Por ello, la jerarquía (relación entre esquemas) varía dinámicamente según el contexto. Podemos enmarcar JDE dentro de los sistemas basados en comportamientos, aunque incluye ideas de otras tendencias como la etología [11].

La unidad básica de JDE es el esquema, que nos permite encapsular una determinada funcionalidad con la idea de formar comportamientos mediante la agrupación de varios esquemas. Podemos definir un esquema como *un flujo de ejecución independiente con un objetivo; un flujo que es modulable, iterativo y que puede ser activado o desactivado a voluntad*[Plaza, 2003]. Hay dos tipos de esquemas, *esquemas perceptivos* y *esquemas motores o de actuación*. Los esquemas perceptivos se encargan de elaborar los *estímulos o percepciones sensoriales*, que pueden ser leídos por otros esquemas. Las entradas de estos esquemas pueden ser los sensores del sistema, o las percepciones de otro esquema. Los esquemas de actuación utilizan los datos obtenidos de otros esquemas perceptivos para generar sus salidas, que pueden ser comandos para los actuadores del sistema, o señales de activación y/o modulación para otros esquemas de nivel inferior (perceptivos o motores). La figura 1.6 lo describe gráficamente.

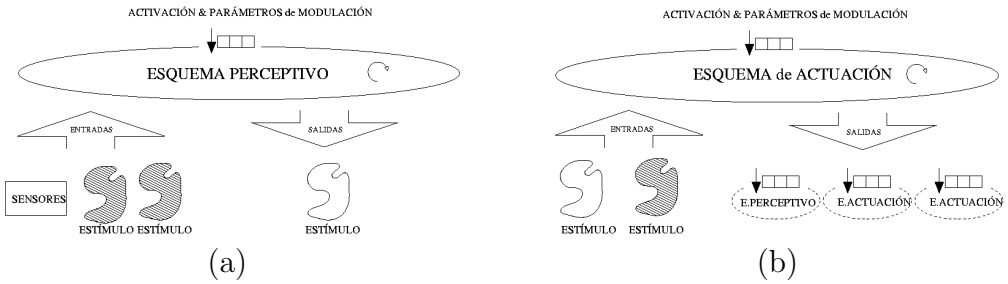


Figura 1.6: *Patrón típico de un esquema perceptivo (a) y de un esquema motor (b) en JDE*

Los esquemas son por definición modulables, permitiendo el ajuste de su funcionamiento interno con dependencia de una serie de parámetros, variando el comportamiento de dicho esquema según lo module uno u otro esquema de nivel superior.

[11] Ciencia que estudia el comportamiento animal

Tiempo real	Un robot esta conectado a la realidad física mediante sensores y actuadores, por lo que su software debe ser ágil y tomar decisiones con vivacidad. Por esto, el software de un robot actúe en tiempo real, sino estricto, al menos blando.
Concurrencia	Un robot debe realiza numerosas tareas de manera simultánea, leer sensores, procesarlos, generar actuaciones, ... Para conseguir esto se requieren sistemas multitarea, para repartir las tareas en múltiples flujos de ejecución independientes.
Interfaz	El software para robots suele incorporar interfaces (habitualmente gráficas) para facilitar la depuración del sistema, o para la interacción con él. En entornos de desarrollo resulta imprescindible para poder representar de alguna manera el estado interno del sistema.
Distribución de cómputo	Las aplicaciones para robots son cada vez más distribuidas, permitiendo crear sistemas multi-robot, o bien ubicar la carga computacional en varios nodos de cómputo.
Heterogeneidad	La gran variedad de sensores, actuadores y demás dispositivos usados para la robótica requieren que los sistemas robóticos sean capaces de interoperar con múltiples interfaces.

Cuadro 1.1: *Requisitos específicos de los sistemas robóticos*

Además, los esquemas son iterativos, realizando su misión en iteraciones que se ejecutan periódicamente. Dicho periodo es un parámetro de modulación principal, que ajusta la cadencia de ejecución del esquema, que variará según interese su función (ej. un esquema perceptivo de visión, puede generar 10,15,20... fps, según se ajuste este parámetro). Por último, un esquema puede suspenderse dejando de producir salidas y de consumir recursos.

La arquitectura cognitiva JDE ha sido inspiradora de la arquitectura software *jderobot* (actualmente en su versión 4.3) que constituye el punto de partida de los desarrollos de este trabajo fin de máster.

1.3 Arquitecturas software y entornos de desarrollo

El desarrollo de sistemas robóticos software no difiere especialmente del desarrollo aplicado a otros ámbitos del software. El desarrollador parte con una serie de requisitos, modela un diseño y finalmente realiza una implementación. La peculiaridad es que comparten una serie de requisitos específicos, que condicionan las características de este tipo sistemas software. Varios de estos requisitos se recogen en la tabla 1.1.

Históricamente el problema del desarrollo de sistemas robóticos generalmente se abordaba con soluciones *ad-hoc*, diseñadas e implementadas para un sistema específico y difícilmente portable. Con el paso de los años, el asentamiento de los fabricantes de robots y el trabajo de numerosos grupos de investigación, han ido apareciendo las arquitecturas software o diseños software, que se plasman en plataformas de desarrollo que permiten abordar el problema de construir aplicaciones robóticas de una manera más fiable, sencilla y eficiente.

Las arquitecturas software basadas en las doctrinas deliberativas consisten generalmente en una serie de módulos, agrupados en una librería, que se ejecutan de manera

secuenciada siguiendo el bucle *Sensar-Modelar-Planificar-Actuar* [Murphy, 2000]. El uso de la abstracción funcional hace que sea generalmente sistemas monohilo.

Por el contrario, las arquitecturas software basadas en las doctrinas reactivas suelen acoplarse en sistemas concurrentes donde múltiples módulos se ejecutan de manera concurrente. Esto se debe a que este tipo de sistemas debe atender gran cantidad de tareas, como ya vimos.

Las arquitecturas software híbridas aúnan los dos enfoques, aplicando lo que mejor se adapta a cada situación. Este tipo de arquitecturas tuvo desarrollos muy utilizados en el mundo de la robótica, algunos de ellos muy destacados en su momento como TCA de Simmons [Simmons, 1994] y Saphira de Konolige [Konolige and Myers, 1998].

1.3.1 TCA

El *control estructurado* [Simmons, 1994] es la propuesta de Reid Simmons para combinar en una misma arquitectura aspectos deliberativos y aspectos reactivos. La implementación de esta idea es la arquitectura software TCA (*Task Control Architecture*), que propone una descomposición del sistema en módulos concurrentes que intercambian mensajes entre sí. Unos módulos utilizan la funcionalidad de otros a través de paso de mensajes que necesitan ser tratados. Cada uno de ellos emite ciertos mensajes y tiene registrados manejadores para los mensajes que es capaz de procesar. TCA imbrica la planificación y la ejecución. Además introduce explícitamente la monitorización, y con ella la planificación de percepción (una suerte de atención).

Dentro de TCA los mensajes que puede emitir un módulo son: (1) informativo (*Information Message*), para comunicar cierta información a quien le pueda interesar, tipo multidifusión; (2) petición (*Query Message*), para preguntar cierta información, el receptor elabora la respuesta y se la devuelve; (3) objetivo (*Goal Message*), para introducir un nuevo objetivo al sistema; (4) comando (*Command Message*), para solicitar cierta actuación, enviar comando a los actuadores; (5) monitor (*Monitor Message*), para programar cierta comprobación de condiciones y (6) excepción (*Exception Message*), cuando ha detectado cierta emergencia. Tal y como muestra la figura 1.7(b), hay un módulo central que se encarga de coordinar los distintos módulos del sistema encaminando los mensajes hacia aquellos que son capaces de manejarlos.

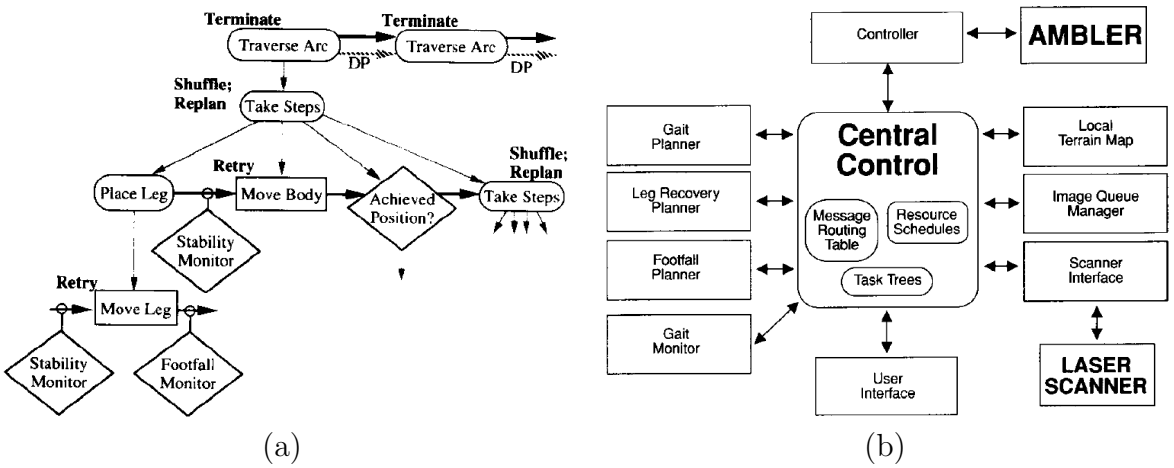


Figura 1.7: *Árbol de tareas en cierto instante para el robot Ambler (a) y la arquitectura de control como colección de módulos (b).*

Esta arquitectura software ha sido utilizada en multitud de robots, entre los que destacan Ambler y Xavier [Simmons et al., 1997]. Ambler es un robot de exteriores con

patas, diseñado para caminar por terrenos escabrosos. Xavier es un robot de interiores con ruedas capaz de entregar correo en entornos de oficina.

1.3.2 Saphira

Una arquitectura muy exitosa en robots de interiores es Saphira [Konolige and Myers, 1998], creada por Kurt Konolige en SRI International. La arquitectura software asociada ha sido integrada y difundida en la plataforma de los robots comercializados por Activmedia. Se ha reescrito en C++ con el nombre de ARIA bajo licencia GPL. La orientación cliente-servidor de esta arquitectura software ha facilitado su uso en robots tan distintos como Flakey del SRI, los modelos Kephera de K-Team, los B21 de iRobot o los Pioneer de ActivMedia.

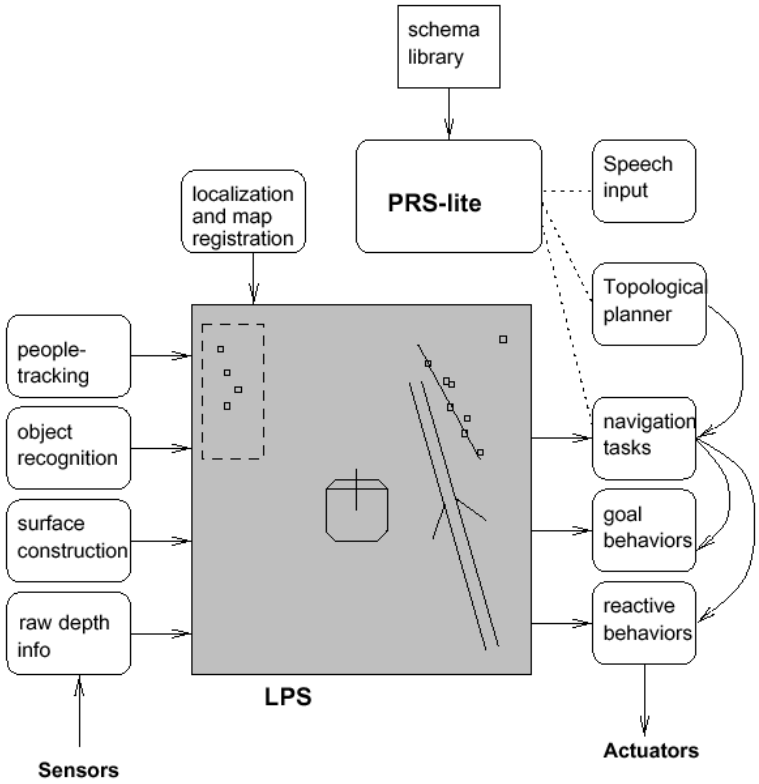


Figura 1.8: *Arquitectura Saphira*

En cuanto a la percepción, Saphira construye y actualiza constantemente una representación interna llamada *espacio perceptivo local*, en la cual fusiona información sobre el entorno próximo del robot. Este modelo está compuesto por *artefactos*, que representan internamente algún objeto real y que están en permanente correspondencia con los objetos homólogos de la realidad. Estos *artefactos* también pueden representar alguna configuración útil para guiar un comportamiento o algún objetivo instanciado. Varios procesos concurrentes mantienen ese modelo coherente, con sus símbolos *anclados* en el mundo real (*perceptual anchoring*), tal y como muestra la figura 1.8.

En cuanto al control, Saphira presenta dos niveles. En el nivel inferior dispone de unos comportamientos básicos escritos como controladores borrosos, es decir, como una colección de reglas borrosas con antecedentes y consecuentes. Estos comportamientos materializan las reacciones básicas, bien a los estímulos del entorno, bien a los objetivos instanciados (por ejemplo dirección de avance). Además, se engarzan con el espacio perceptivo local, de manera que los antecedentes de sus reglas hacen referencia a información almacenada en ese espacio, que se traduce a variables borrosas. La actuación

efectiva se calcula promediando de modo borroso las salidas individuales de todos los comportamientos. Ese promedio es una combinación lineal donde los pesos de cada comportamiento dependen de su prioridad, que es fija, y su adecuación a la situación actual, que se calcula con una reglas de contexto (*context-dependent-blending*).

En el nivel superior Saphira se encarga de relacionar los comportamientos básicos con la consecución de metas intermedias y objetivos del robot. En este nivel se determina la activación y coordinación de los diferentes comportamientos básicos. El planificador PRS-Lite se encarga de ello, permitiendo la descomposición jerárquica de tareas en subtareas y ofreciendo un rico repertorio de mecanismos para especificar secuencias, despliegues condicionales de planes, ordenación de subobjetivos y de actuaciones.

Sin embargo, la tendencia en los entornos de desarrollo robótico actuales va hacia la flexibilidad en cuanto a organización interna de los diferentes módulos, componentes o partes que componen un sistema robótico, en contraposición con la idea subyacente en las arquitecturas software para robots, en que de alguna manera se fija dicha organización. De esta manera, podríamos hablar de arquitecturas software neutras en las que no se fija ningún tipo de organización, sino que se aportan las herramientas para que el programador de aplicaciones pueda implementar la organización que mejor se adapte al contexto del problema que pretende resolver. Por ello, los entornos de desarrollo robótico más usados en la actualidad se basan en mecanismos de acceso al variado hardware para robots y una serie de herramientas que facilitan el desarrollo de tareas comunes en aplicaciones robóticas, pero intentan dar máxima flexibilidad al usuario a la hora de organizar su sistema robótico.

Otra idea que cada vez toma más fuerza en la comunidad robótica es la reutilización del software para poder afrontar el reto de crear sistemas robóticos más complejos, con mayor funcionalidad y más robustos. Unido a las nuevas ideas procedentes de la ingeniería del software basada en componentes, los enfoques en los nuevos sistemas robóticos van hacia la creación de componentes software con funcionalidades concretas que pueden usarse para construir sistemas más complejos, como si de bloques de construcción se tratase [Schmidt, 1999].

Entre los entornos de desarrollo más destacados tenemos Player/Stage [Collett et al., 2005, Gerkey et al., 2003, Vaughan et al., 2003], Orca [Brooks et al., 2007] y ROS [Quigley et al., 2009]. Todos ellos proyectos de software libre (en contraposición con los sistema propietarios de los primeros entornos de desarrollo) y con una comunidad de usuarios bastante amplia, sobre todo en Player/Stage, considerado el actual estándar de facto en la comunidad robótica. A continuación repasamos las características más destacables de estos tres sistemas, aunque los repasaremos con más detalle en el capítulo 3.

1.3.3 Player/Stage

Player/Stage[12] [Collett et al., 2005, Gerkey et al., 2003, Vaughan et al., 2003] es un entorno de desarrollo para aplicaciones robóticas de código abierto. Esta formado por dos piezas independientes, Player, un servidor de red para control de hardware robótico y Stage, un entorno de simulación multi-robot (ver figura 1.9. Ambas piezas son desarrolladas dentro del mismo proyecto. En la actualidad se ha unido también el proyecto Gazebo, que es un entorno de simulación tridimensional. De esta manera, es habitual referirse al entorno con los tres nombres o simplificando P/S/G.

P/S/G está diseñado para ser independiente del lenguaje de programación, y desde sus inicios todo el API ha contado con un recubrimiento SWIG, que permite generar *bindings* en multitud de lenguajes. De este modo, los clientes de Player pueden ser

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

desarrollados en lenguajes como C++, Java, Python o Tcl. Además P/S/G puede ser distribuido a través de la red, dada su arquitectura cliente-servidor. EN P/S/G no se hace ninguna asunción acerca de cómo estructurar las diferentes partes de un sistema robótico, únicamente se aporta la abstracción `interfaz` usada para comunicar los diferentes módulos de dicho sistema. Esto permite expresar las relaciones entre la diferente funcionalidad encapsulada en los módulos en términos de *provee/requiere* (o implementa/usa). Esta característica acerca P/S/G al paradigma de software basado en componentes.

P/S/G es en la actualidad el estándar de facto en la comunidad robótica. Su gran flexibilidad ha hecho que gente de todas partes del mundo soporten e implemente software basado en esta plataforma.

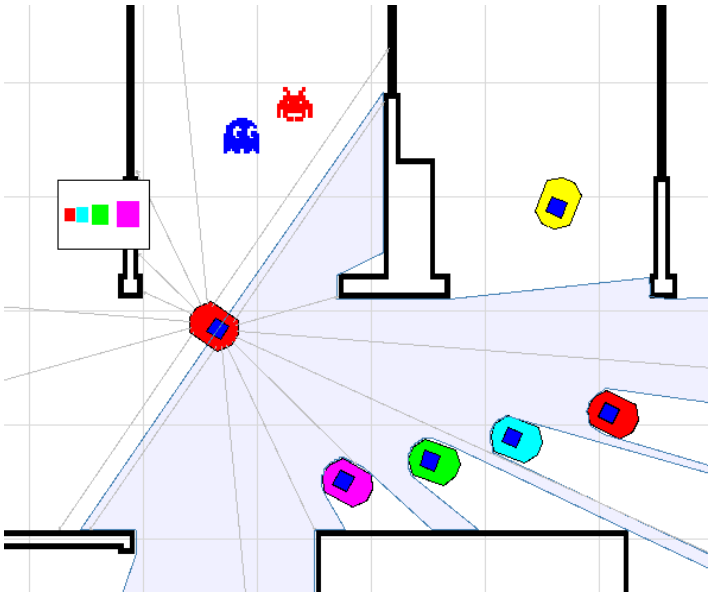


Figura 1.9: *Stage simulando múltiples robots*

1.3.4 Orca

Orca[13] [Brooks et al., 2007] es otro entorno de desarrollo de código abierto basado en componentes. Su principal enfoque se centra en la utilización del *middleware* Ice para la comunicación entre componentes, que es un proyecto externo, enfocando su esfuerzo al desarrollo de elementos robóticos y no al relacionado con *middleware*, que por otra parte resultan complicados de resolver para desarrolladores que no tienen un perfil concreto. De esta manera, Orca pretende reducir al mínimo lo que se considera arquitectura, volcándose más en desarrollar nuevos *drivers* o algoritmos. En la figura 1.10, extraída de su página web, muestran este hecho, comparando su esfuerzo, en términos de líneas de código, con Player. Ahí podemos ver como el esfuerzo en infraestructura es prácticamente mínimo en Orca desde que adoptaron Ice, mientras que en Player mantiene un ritmo creciente.

Como resultado del uso de Ice, Orca es multi-lenguaje y multi-máquina. La plataforma proporciona multitud de interfaces bien definidos para acceder tanto a hardware robótico, como a componentes software que implementan diferentes algoritmos. Además, han desarrollado bastantes herramientas orientadas a depuración.

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

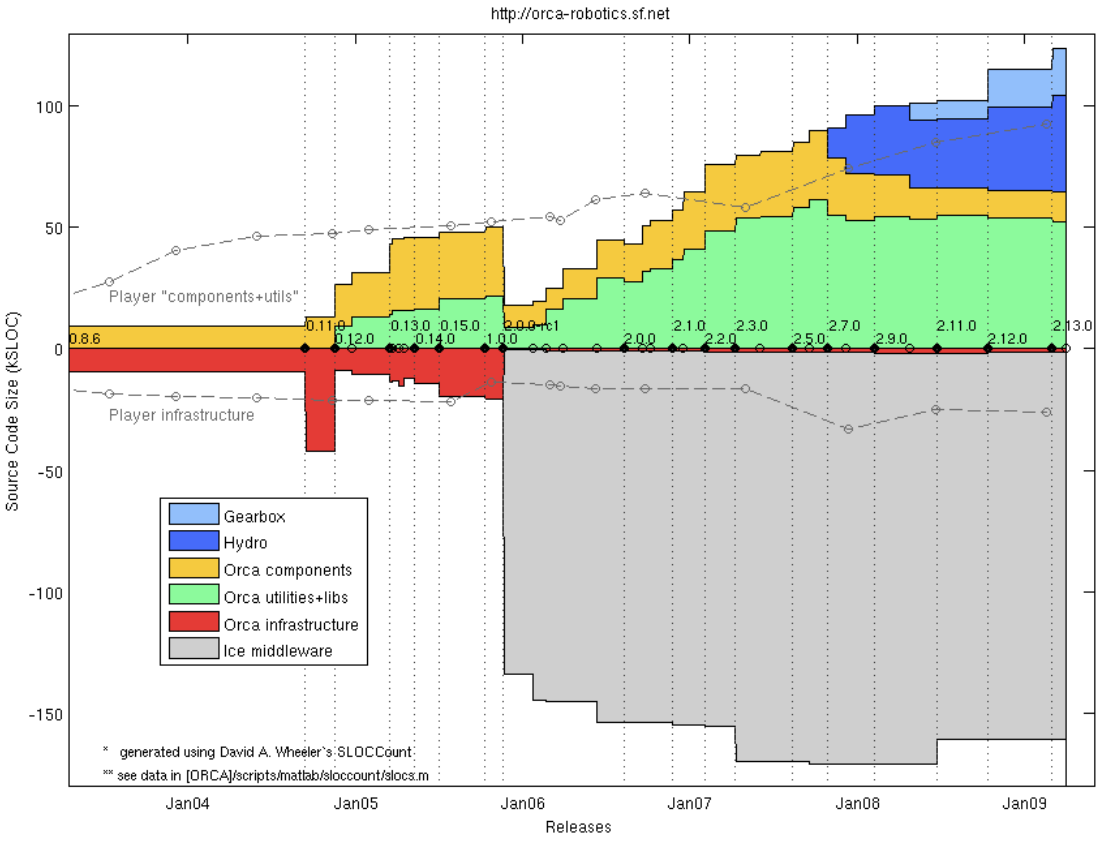


Figura 1.10: *Comparación de esfuerzos Orca vs Player*

1.3.5 ROS

ROS[14] [Quigley et al., 2009] es un meta sistema operativo para robots, también bajo licencia de código abierto. Provee servicios típicos de un sistema operativo, como son abstracción de acceso al hardware, control de bajo nivel para dispositivos, mecanismos de paso de mensajes entre procesos y una variada colección de la funcionalidad más comúnmente usada en un sistema robótico. Se propone como una capa mediadora entre el robot y el sistema operativo sobre el que se montan los procesos que lo gobiernan, con la intención de que sea un software multi-plataforma (a día de hoy sólo la versión para Linux es considerada estable y completamente funcional). Por el momento, los lenguajes de desarrollo soportados son C++ y Python, aunque hay intención de soportar algunos más proporcionando librerías cliente que puedan acceder al API de ROS.

Su diseño va en la línea de las últimas tendencias, de ser lo más ligero posible y capaz de ser integrado o utilizado fácilmente por otros sistemas existentes, para fomentar la reutilización de software robótico. Sus librerías se han diseñado con la idea de ofrecer interfaces claros y limpios. Además, dado que el equipo de desarrollo comparte *casa* con el proyecto *OpenCv*, su integración es muy grande, aprovechando ambos equipos la realimentación que surge del intercambio.

Otro punto a destacar es su sistema de gestión del software, que provee mecanismos para la distribución e integración de paquetes de software con funcionalidad determinada. Incluye la estructura de directorios que un paquete debe tener y cómo deben usarse, descripciones de la funcionalidad contenida, y detalles de alto nivel de cómo se comunica dicho software.

[14] http://www.ros.org/

El objetivo global de este proyecto es diseñar y programar la plataforma de desarrollo que será usada por el grupo de robótica de la URJC en los siguientes años. La hemos denominado versión 5, ya que rompe completamente con la línea de desarrollo seguida hasta la fecha con las versiones 4.

El énfasis en esta nueva versión no va hacia la construcción de un sistema completo que incluya todas las características que se esperan de una arquitectura software, tal y como las hemos descrito. El énfasis va en la dirección de las últimas tendencias en software para robots: *la flexibilidad*. Y es por ello que nos vamos a apoyar en las herramientas existentes a fecha de hoy y describir una manera de usarlas, aportando soluciones en aquellos campos en los que el grupo de robótica tiene más experiencia. De este modo, podemos decir, *jderobot* 5 más que una arquitectura software robótica, será un framework o entorno de desarrollo, más una colección de componentes y una colección de librerías capaces de ser reutilizadas por otros proyectos robóticos.

La memoria de este trabajo sigue la siguiente estructura. En el capítulo 2 expondremos los objetivos y requisitos de este proyecto, muy brevemente introducidos en el párrafo anterior. En el capítulo 3 describiremos el estado del arte en materia de entornos para desarrollo de aplicaciones robóticas, entrando en detalle con alguna de las implementaciones más exitosas. A continuación, en el capítulo 4 daremos todos los detalles referentes al desarrollo de este trabajo. Y por último en el capítulo 5 veremos las conclusiones a las que hemos llegado y expondremos las lineas de trabajo futuro que se abren tras este trabajo.

Capítulo 2

Objetivos y metodología

Como hemos visto en la introducción, la creciente complejidad de las aplicaciones robóticas requiere planteamientos y herramientas cada vez más potentes para conseguir el éxito. En este contexto surgen las arquitecturas cognitivas que plantean diferentes visiones acerca de la organización de los sensores y actuadores de un robot. Dichas teorías se plasman en sistemas software, sobre los que un desarrollador crea sus aplicaciones robóticas. Dichos sistemas suelen incorporar mecanismos y herramientas que facilitan la tarea de desarrollo. También hemos visto la tendencia hacia los *frameworks* para desarrollo de aplicaciones robóticas donde el énfasis se apoya en la flexibilidad y la gran libertad que tiene desarrollador para organizar su sistema.

El presente trabajo, *jderobot 5: Entorno de desarrollo basado en componentes para aplicaciones robóticas*, pretende argumentar las decisiones que hay tras la nueva línea de desarrollo de *jderobot*, que rompe con la arquitectura seguida en las versiones 4. Esta nueva versión aplica nuevas herramientas a los paradigmas que se ha pretendido alcanzar en las últimas versiones 4 pero que no siempre han tenido resultados positivos, como la orientación a componentes o la programación multi-lenguaje. *jderobot* 5 además rompe con los requisitos de anteriores versiones, que se diseñaban partiendo de las tesis expuestas en la arquitectura teórica JDE[Plaza, 2003]. En esta ocasión, la organización de las diferentes piezas se dejará de la mano del programador, ofreciéndole la máxima flexibilidad para que aplique aquella organización que considere más oportuna en cada contexto.

Comenzamos este capítulo introduciendo algunos conceptos principales de las versiones antecesoras, de donde pasaremos a la última versión estable de nuestro software para poder centrar la motivación que nos mueve al desarrollo de este trabajo. A continuación describiremos los objetivos concretos del trabajo de manera detallada, y enumeraremos los requisitos del sistema *jderobot* 5.0. Y por último describiremos la metodología empleada en el desarrollo del trabajo.

2.1 *jderobot*

Desde los inicios como grupo de investigación, el grupo de robótica de la URJC siempre ha tenido claro que para llevar a cabo aplicaciones complejas es necesaria una buena organización del software. Es por ello que desde la primera aplicación, siempre se ha trabajado con un diseño que se caracterizaba por la manera de integrar cada una de las partes. Diseño que siempre ha venido influenciado por la teoría contenida en la arquitectura cognitiva JDE[Plaza, 2003].

2.1.1 Antecedentes

Los primeros pasos vinieron con los servidores `otos` y `oculo`, que eran capaces de conectarse a diferente tipo de hardware y servir mediante un protocolo de mensajes sus valores (sensores y actuadores). El protocolo utilizaba tanto peticiones directas, como suscripción, permitiendo a un cliente preguntar por los datos o recibirlos cuando estuviesen listos. El abanico de hardware soportado era bastante pequeño en dicha fecha, limitándose al interfaz con robots *pioneer* y un puñado de cámaras. Su uso, resolvía el acceso al hardware, de modo que las aplicaciones que se desarrollaban se centraban en crear algoritmos que procesaran los datos sensoriales y generasen las órdenes adecuadas para los actuadores.

El siguiente hito lo marcó la versión 3.4, que ya recogía el conocimiento acumulado por diferentes trabajos fin de carrera. Su principal característica era el aportar un conjunto de variables que todos los esquemas eran capaces de acceder, unos para escribir su valor y otros para leerlo. Las principales variables de que se disponía eran los valores de la odometría, los valores obtenidos desde los sensores de ultrasonido y los valores de velocidad del robot. Todas ellas eran obtenidas desde los servidores comentados en el párrafo anterior. De esta manera la plataforma era básicamente un cliente de `otos` y `oculo` que compartía los datos entre varias hebras internas, cada una de las cuales representaba un esquema.

En estos momentos no había una gestión clara del proyecto, y la distribución del software se llevaba a cabo con paquetes comprimidos *tgz* y una colección de *Makefiles* para su compilación. El numerado de versiones tampoco atendía a un *roadmap* predefinido, por lo que en función de la ruptura que producían los cambios que se iban aplicando con versiones anteriores, así se iba haciendo avanzar el número de versión. De hecho, cada aplicación que se creaba aplicaba sus cambios, haciendo en la mayoría de ocasiones que arquitectura y aplicación estuviesen tan acopladas que no había una frontera clara entre ambas. Esto hacía que cambios con consecuencias favorables fuesen complicados de integrar a nivel de arquitectura para que otras aplicaciones se beneficiasen de ellos, quedando muchas veces en el olvido del ámbito muy reducido de una aplicación concreta.

En este punto surgió *jde+* [Lobato Bravo, 2005], que partiendo desde un diseño cuidadosamente extraído de la arquitectura teórica JDE, pretendía aportar una nueva versión que incorporase todo aquello para lo que *jderobot* no era capaz de dar una solución. *jde+* pretendía ofrecer herramientas que se pensaba serían interesantes dentro de una arquitectura robótica, siempre en el marco teórico de JDE. Los principales conceptos que dicho proyecto implementó fueron la carga dinámica de esquemas y el paso de mensajes entre esquemas. *jde+* no llegó a utilizarse en el grupo como base para sus aplicaciones, pero aportó nuevas ideas que se implementarían más tarde en *jderobot*.

Otro proyecto que surgió como propuesta para mejorar *jderobot* fue *jdeneo.c* [Ruiz-Ayúcar Vázq esta vez partiendo de la base de *jderobot* 3.4. En este proyecto se exploraron las herramientas que aportaba la librería `Glib` para aplicar sus tipos de datos C, se investigaron los mecanismos para creación de interfaces gráficas con `GTK+` y se implementó una versión de la carga dinámica de esquemas. Conceptos que de nuevo se aplicarían en las futuras versiones de *jderobot*.

La toma de conciencia de que *jderobot* debía moverse en una única dirección se plasma a partir de la rama de desarrollo 4. En esta rama se empieza a crear la idea de que la arquitectura software debe aportar determinados servicios a las aplicaciones y de que las aplicaciones deben estar perfectamente delimitadas dentro de los esquemas que las realizan. Empiezan a aparecer los conceptos de *esquemas drivers*, o esquemas de bajo nivel que se encargan de hablar con el hardware y proporcionar variables para

su acceso, *esquemas de servicio* que realizan alguna tarea que aporta algún servicio a las aplicaciones, como la el acceso a una interfaz gráfica. Todos estos conceptos principalmente fueron extraídos de la experiencia ganada con los proyectos *jde+* y *jdeneo.c*. Además, es en esta etapa cuando se empieza a gestionar el proyecto usando las herramientas de *Autotools* y creando una comunidad de usuarios y desarrolladores a su alrededor gracias a herramientas de comunicación como listas de distribución, una web con información del proyecto y documentación *online*, adquiriendo mayor peso como proyecto de software.

Tras un par de versiones más, llegamos a *jderobot* 4.3, última versión estable de la arquitectura software. Dicha versión incluye una gran colección de *drivers* para acceso a muy variado hardware (robots, simuladores, cámaras,dispositivos X10,etc.), un robusto soporte para la gestión de interfaces gráficas, carga dinámica de esquemas, definición de interfaces para acceso al hardware y un largo etcétera de características. Sobre las últimas versiones de la plataforma *jderobot* se han desarrollado con éxito múltiples proyectos como un robot guía [Vega Pérez, 2008], CarSpeed (aplicación para medir la velocidad de coches desde una cámara) [Hidalgo Blázquez, 2008] o diferentes aplicaciones con entornos tridimensionales.

Y es en este punto donde arranca este proyecto, guiado por las últimas tendencias en arquitecturas robóticas.

2.1.2 *jderobot* 4.3

jderobot 4.3 es la última versión estable de la arquitectura software desarrollada por el grupo de robótica de la URJC. Dicho software ha sido utilizado con éxito como base de muchos trabajos realizados en el grupo en entornos robóticos, domóticos y de visión por computador.

jderobot 4.3 se cimenta sobre la base cognitiva de JDE[Plaza, 2003] organizando las aplicaciones como una jerarquía dinámica de esquemas. En esta última versión la carga cognitiva es prácticamente inexistente y no se impone ningún tipo de restricción a la organización interna de los esquemas, ni se pone demasiado empeño en el uso de los mecanismos de arbitraje. Su diseño procede de los conceptos extraídos de la ingeniería del software orientada a componentes, donde las unidades mínimas, los componentes, se comunican a través de una serie de interfaces bien definidos. En *jderobot*, dichas unidades mínimas son los esquemas, y según JDE existen dos tipos: perceptivos y actuadores. Los primeros tratan la información obtenida desde sensores u otros esquemas de nivel inferior, y los segundos generan órdenes de actuación. El sistema está programado íntegramente en el lenguaje C.

Los principales mecanismos que aporta *jderobot* 4.3 son:

Abstracción de hardware : Mediante el uso de interfaces bien definidos, se abstrae el uso de los diferentes dispositivos que puede contener un robot, incluso soportando aquellos simulados desde Player/Stage.

Comunicación entre esquemas : El sistema implementa mecanismos para que los esquemas puedan exportar/importar variables con las que comunicarse.

Carga dinámica de esquemas : El sistema implementa un mecanismo de carga dinámica de esquemas a modo de *plugins*.

Comunicación remota entre esquemas : El sistema implementa mecanismos muy básicos para comunicación remota entre esquemas ejecutando en nodos de cómputo diferentes.

Ejecución cooperativa : El sistema implementa mecanismos básicos de comunicación entre procesos *jderobot*, de manera que es posible crear aplicaciones con procesos *jderobot* cooperantes, incluso ejecutándose en máquinas diferentes.

Una aplicación típica desarrollada usando *jderobot* 4.3 tiene la forma que se muestra en la figura 2.1. En dicha figura vemos que un tipo especial de esquemas, denominados *drivers* (D1,D2,etc.), acceden al hardware, ya sea real o simulado. De esta manera, exportan los diferentes valores obtenidos en forma de variables que el resto de esquemas (S1,S2,etc.) utilizan para conseguir sus objetivos. El conjunto de esquemas forma lo que denominamos jerarquía, sobre la que se crean una serie de dependencias. Además vemos otros elementos que se denominan servicios (Srv GTK y Srv Xforms), que gestionan el acceso concurrente al entorno gráfico, sobre el cual el esquema que lo requiera puede pintar su interfaz de usuario.

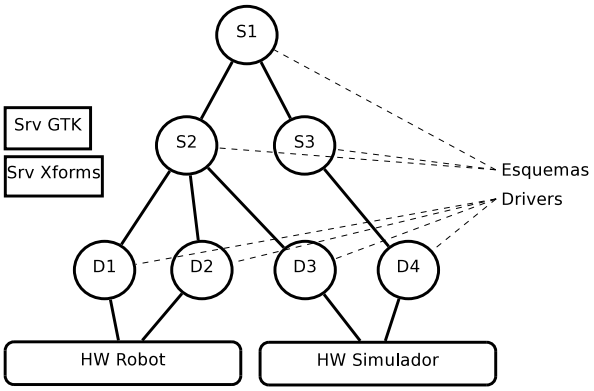


Figura 2.1: *Aplicación típica en jderobot 4.3*

El sistema *jderobot* 4.3 se ha usado con éxito desde 2006 para el desarrollo de prácticas de la asignatura de Robótica que imparte el grupo, además de en las siguientes aplicaciones:

Eldercare : Aplicación que es capaz de detectar que una persona ha sufrido una caída a partir de la información obtenida desde varias cámaras. Se apoya en algoritmos de detección 3D usando cámaras calibradas [Marugán Alonso, 2009].

Carspeed : Aplicación capaz de estimar las velocidades a las que circulan los coches sobre un tramo de carretera usando únicamente una cámara [Hidalgo Blázquez, 2008].

GuideRobot : Aplicación que permite a un robot funcionar como guía de un entorno como un museo o una oficina. Implementa diferentes algoritmos para navegación local, global y localización [Vega Pérez, 2008].

2.1.3 Motivaciones para *jderobot* 5

El camino recorrido hasta la fecha nos ha dado perspectiva sobre qué debe solucionar una arquitectura software y los diferentes enfoques que se pueden utilizar para conseguirlo. Quizá la idea principal es que debe ayudar al desarrollador de aplicaciones robóticas con sistemas más flexibles y metodologías que permitan una mayor reutilización de software. La flexibilidad en cuanto a cómo desarrollar una aplicación (organización, leguaje de programación,etc.) permite al desarrollador sentirse más libre a la hora de diseñar su sistema. La reutilización permite avanzar más rápido dado que podemos apoyarnos en partes ya implementadas y probadas.

El inicio de este trabajo lo marca el intento de añadir determinadas características que en el campo de la robótica ya empezaba a ser tendencia: El uso de múltiples lenguajes para programar las aplicaciones. Hasta el momento, todo ha sido programado en C, mezclando en ocasiones algo de código C++. Pero *jderobot* era incapaz de utilizar lenguajes que han ganado fama en los últimos años, como por ejemplo Python. Es por ello, que de nuevo nos aventuramos hacia la versión 4.4, que debía entre otras cosas, aportar capacidades multi-lenguaje a *jderobot*.

La versión 4.4 nunca llegó a un estado estable dentro de la rama de desarrollo 4. En parte solucionaba los puntos por los que se había iniciado su desarrollo, pero de una forma muy *ad-hoc* y poco escalable, dadas las limitaciones impuestas por la versión 4 de *jderobot* y todo el código C heredado. Resultaba muy complejo intentar introducir características para las que C no tiene solución.

La arquitectura *jderobot* 4 ha permitido llegar a aplicaciones muy complejas, pero que carecen de flexibilidad. Por una parte se apoyan en un modelo cognitivo que no siempre es el más adecuado, no es posible o resulta complicado la introducción de nuevos lenguajes de programación y la reutilización de sus partes mínimas es difícil dada la alta cohesión con los mecanismos de la arquitectura. A todo esto sumamos un carencia que no ha sido hasta hoy objetivo de *jderobot*, la distribución entre nodos de cómputo conectados mediante una red de los diferentes componentes o esquemas de una aplicación.

La imposición de un modelo cognitivo que fuerza a sus unidades mínimas a seguir una determinada estructura interna no se ha demostrado el mecanismo más flexible, por el contrario las nuevas tendencias dejan al desarrollador la tarea de decidir cúal es la mejor manera de organizar los elementos de una aplicación. Dicha tendencia ha ido entrando en las últimas versiones de *jderobot* donde la carga cognitiva ha pasado a ser mínima. Más allá de un sistema híbrido en el que se mezclan conceptos de diferentes arquitecturas teóricas, la tendencia es simplemente a no hacer ninguna asunción acerca de cómo se debe organizar el software de un robot. Para ello se tiende a crear componentes cerrados que interaccionan usando mecanismos básicos capaces de ser integrados de múltiples maneras. Ni siquiera la estructura interna de estos componentes ha de seguir una estructura fija. Este será el enfoque utilizado en *jderobot* 5, donde cada unidad mínima se encerrará en un componente o esquema (por mantener la nomenclatura) y desarrollaremos mecanismos genéricos con los cuales puedan interaccionar y desarrollar sistemas robóticos más complejos.

La proliferación de lenguajes de alto nivel, cada uno con sus características favorables para determinadas aplicaciones, hace requisito indispensable que un sistema que pretende ser flexible pueda ser programado en múltiples lenguajes. De esta manera, cada programador o cada parte especifica podrán beneficiarse de ser programadas en el lenguaje que mejor se adapte a sus gustos o necesidades. Como primera aproximación y como parte de este trabajo se ha abordará el problema con una nueva versión experimental de *jderobot* (4.4), con la que exploraremos posibles vías y enfoques para cumplir este objetivo. En *jderobot* 5 se pretende abordar este problema con una solución más elegante.

Quizá uno de los conceptos más ampliamente discutidos en la ingeniería del software, la reutilización del software. Su beneficio es indudable en cualquier sistema software, disponer de partes probadas capaces de ser añadidas sin coste adicional permite desarrollos más estables y económicos (ya hablemos de dinero o de tiempo). Los últimos pasos de la rama 4 permitieron la reutilización en gran medida de muchos de los esquemas desarrollados, haciendo que muchas aplicaciones se beneficiasen de código desarrollado y probado para otras aplicaciones. El propio proyecto *jderobot* se ha aprovechado de la reutilización de otros proyectos como Player/Stage, del que se han utilizado sus librerías para simuladores y para acceso a hardware. *jderobot* 5 pretende elevar el listón, orga-

nizando de una manera más efectiva todo el desarrollo de que disponemos y añadiendo dependencias externas que nos permitan avanzar más rápido y hacia aplicaciones de más calidad, que a su vez reincidan en un beneficio de la comunidad robótica.

Y por último el concepto de distribución de componentes por una red de nodos resulta a día de hoy una característica más que deseable en un sistema robótico. Por un lado permite la distribución de la carga de cómputo y por otro permite abordar otro tipo de aplicaciones, que por su naturaleza requieren estar distribuidas, por ejemplo sistema de seguridad con múltiples cámaras y sensores distribuidos por un edificio. Por ello *jderobot* 5 asume esta característica como objetivo, debiendo aportar una solución que permita ejecutar sus componentes de manera distribuida.

2.2 Objetivos

Habiendo mostrado las motivaciones de este proyecto, en esta sección se enumeran los objetivos perseguidos, que deben desembocar en la implementación del sistema *jderobot* 5. Estos serán el hilo conductor de todo el proceso de desarrollo junto con los requisitos marcados en la sección siguiente. Los organizamos según dos enfoques, diseño e implementación. Desde el enfoque de diseño, marcamos las ideas o características generales que se pretenden incluir en el sistema, visto como sistema abstracto (o sistema de diseño). Desde el enfoque de implementación, marcamos la metodología y características concretas del sistema, visto ahora como un programa.

El objetivo principal que persigue este proyecto es construir el sistema *jderobot* 5, un entorno de desarrollo software basado en componentes para aplicaciones robóticas. En esta ocasión el esfuerzo se centra en diseñar e implementar un sistema flexible y fácil de usar, de acuerdo con las tendencias en el campo. Para su diseño no vamos a imponer ningún requisito en cuanto a carga cognitiva, como veníamos haciendo en versiones anteriores, apoyándonos en que la flexibilidad del sistema debería permitir múltiples enfoques.

Para la consecución del objetivo principal de este proyecto, se deben alcanzar múltiples metas o subobjetivos. En lo que resta de sección describimos dichos subobjetivos, desde los enfoques comentados al inicio, numerándolos para poder referirnos a ellos en lo sucesivo.

Como subobjetivos enfocados al diseño de *jderobot* 5 tenemos:

Sistema flexible : El sistema debe aportar diferentes enfoques para organizar nuestras arquitecturas, o al menos no imponer ninguno concreto.

Sistema capaz de ser distribuido : El sistema debe poder distribuirse entre múltiples nodos de cómputo utilizando mecanismos estándar, multi-plataforma y escalables.

Sistema multi-lenguaje : El sistema debe permitir implementar componentes en múltiples lenguajes, como mínimo en los más utilizados actualmente (C++, Python, Java, PHP).

Orientado a componentes : El desarrollo debe orientarse a componentes, de modo que cada unidad mínima obtenida sea autocontenida, para maximizar las posibilidades de reutilización de los elementos desarrollados. Cada componente ofrecerá una serie de interfaces bien definidos con los que acceder a su funcionalidad.

Enfocando ahora los subobjetivos desde el punto de vista de la implementación del sistema tenemos:

Eficiente : El uso de recursos por parte de *jderobot* debe ajustarse al mínimo, de modo que no interfiera con las capacidades requeridas por los algoritmos que se implementen sobre él. Dado que las aplicaciones de visión por computador son uno de los campos en los que el grupo enfoca sus esfuerzos, se debe garantizar la eficiencia en este contexto.

Implementado para plataformas GNU/Linux y Android como mínimo : La solución óptima debería permitir que un componente *jderobot* pueda ejecutarse en cualquier plataforma. Dicha solución es muy ambiciosa, de modo, que la rebajaremos el subobjetivo a que como mínimo funcione en las plataformas sobre las que trabajamos actualmente GNU/Linux y Android.

Componentes básicos : Se deben implementar algunos de los componentes básicos junto con sus interfaces para permitir desarrollar alguna aplicación de prueba que verifique el sistema.

Aplicación prueba : Se deberá implementar alguna aplicación prueba para verificar el sistema construido.

2.3 Requisitos

A continuación se enumeran los requisitos que se plantean inicialmente para el desarrollo del sistema *jderobot* 5. En esta ocasión no consideraremos como requisitos del sistema las restricciones impuestas por la arquitectura teórica JDE[Plaza, 2003], dado que uno de los objetivos es hacer un sistema flexible, abierto a diferentes organizaciones. Por mantener la nomenclatura usada hasta el momento, nos referiremos a las unidades mínimas del sistema como componentes o esquemas de manera indiferente, ya que ambos representan lo mismo. Diferenciamos entre requisitos funcionales y no funcionales.

Requisitos funcionales

Estos requisitos describen las características requeridas del sistema que describen capacidades del mismo.

Requisito(1) ***"Unidad mínima Componente":*** La unidad mínima del sistema es el componente o esquema. El sistema permitirá con diferentes mecanismos la comunicación entre ellos.

Requisito(2) ***"Multilenguaje":*** El sistema debe poder programarse utilizando los lenguajes más utilizados (C++,Java,Python,PHP).

Requisito(3) ***"Distribución":*** El sistema debe cumplir el requisito 1 aun cuando dichas unidades mínimas se encuentren ejecutando en diferentes nodos de cómputo.

Requisito(4) ***"Portabilidad":*** Al requisito 3 se suma la independencia de la plataforma sobre la que se ejecutan las unidades mínimas. Este requisito se considera cumplido si la portabilidad se consigue al menos entre plataformas GNU/Linux y Android.

Requisitos no funcionales

Estos requisitos describen las características requeridas del sistema o del proceso de desarrollo que señalan restricciones.

Requisitos del sistema

Requisito(5) ***"Rendimiento":*** Los mecanismos considerados parte del sistema deben consumir una cantidad despreciable de recursos para no restar capacidad de cómputo a las aplicaciones que los usen.

Requisito(6) ***"Robustez":*** El sistema implementado debe ser robusto frente a fallos, disponiendo todos los mecanismos al alcance para el control de errores y su tratamiento más adecuado.

Requisitos del proceso de desarrollo

Requisito(8) ***"Lenguaje":*** Dada la intención de crear un sistema que permita utilizar múltiples lenguajes (R2), no se fija ningún requisito en este aspecto.

Requisito(9) ***"Plataforma":*** De nuevo, dado R4, no se especifican restricciones de plataforma, aunque en este caso se recomienda trabajar en entornos GNU/Linux que son los que se han utilizado principalmente en los últimos años y es donde mayor experiencia se tiene en el grupo.

Requisito(10) ***"Reutilización":*** Se aplica como restricción, por una parte, para forzar a que antes de implementar ningún mecanismo se haga un estudio de que soluciones existen y si se podrían aplicar a nuestro sistema. De esta manera evitaremos el problema de *reinvención de la rueda* y podremos avanzar más rápidamente hacia un sistema más completo. Y por otra parte, se requiere que todo el software desarrollado en este proyecto sea *software libre*, de modo que la comunidad robótica puede utilizarlo y beneficiarse.

Requisito(11) ***"Documentación":*** Todo el código desarrollado debe documentarse usando comentarios que puedan ser procesados por `doxygen`.

Requisito(12) ***"Integración al proyecto* jderobot*":*** Todos los productos desarrollados con este trabajo, código, documentación y herramientas de empaquetado (debian) se deben integrar al desarrollo oficial de *jderobot*.

2.4 Metodología de desarrollo

Todo proyecto software debe estar guiado por una metodología de trabajo o proceso de desarrollo software. Dependiendo de las características de dicho proyecto software deberemos aplicar una u otra metodología. Este proyecto es en definitiva un desarrollo software y por tanto debemos utilizar una metodología que guíe el proceso.

La metodología de desarrollo debe definir los productos que se deben obtener en el proceso de desarrollo. Los productos más comunes son la documentación y el código fuente que trasformaremos en los programas. También se debe fijar un plan de trabajo, marcando hitos y fechas de entrega.

Este proyecto tiene las siguientes características:

Dos integrantes . En el desarrollo de este proyecto tan solo están involucradas dos personas, el tutor y el autor.

Gran bagaje . La base del proyecto es el gran bagaje experimental que se acumula a lo largo de las diferentes versiones de *jderobot*. De alguna manera muchos de los requisitos se imponen por haberse demostrado un éxito, pero otros tantos persiguen continuar añadiendo innovación a la plataforma.

Complejidad media . La complejidad del sistema a desarrollar es media. Se pretende introducir mecanismos que habitualmente resultan complicados como: concurrencia, comunicaciones, y programación multiplataforma. Pero por otra parte se pretende reutilizar toda herramienta que encaje con el proyecto, de modo que dediquemos el menos esfuerzo en un futuro al desarrollo de arquitectura software.

Plazo de entrega cerrado . El plazo de entrega esta fijado para Junio de 2010.

El número de integrantes reducido de este proyecto implica que la documentación a realizar no será tan estricta, ya que la comunicación será constante y directa. Sin embargo, la pretensión de establecer la versión 5 como la nueva versión estable para el grupo de robótica de la URJC requiere que todo el código quede debidamente documentado. La complejidad media del proyecto requerirá un desarrollo incremental, que vaya incorporando poco a poco las diferentes funcionalidades. Y por último, dado que el plazo de entrega esta cerrado se debe preparar un plan de trabajo que permita finalizar el proceso en la fecha indicada.

Por estas razones, la metodología que se eligió es el desarrollo en espiral, y los productos esperados son esta memoria, el sistema *jderobot* 5 que incorpore algún componente básico y una aplicación para probar dicho sistema. El desarrollo en espiral nos permite realizar un desarrollo incremental, obteniendo versiones funcionales en cada iteración. El plan de trabajo consiste en obtener versiones preliminares de los productos en las fechas marcadas por el tutor del proyecto. Para la coordinación de tareas se realizan reuniones semanales, bien presenciales, bien por mensajería instantánea y se mantiene una bitacora *online* (mediawiki) con los progresos realizados.

En cada iteración del proceso se analizan los requisitos del sistema, se diseña e implementa una versión que incorpore dichos requisitos, se realizan pruebas y se planifica la siguiente versión. En total se han completado cuatro iteraciones, al final de cada una se obtuvo lo siguiente:

1. Aproximación con *jderobot* 4.4. Se inicia la investigación de mecanismos multi-lenguaje.
2. Versión 5 preliminar. Mecanismos mínimos para crear componentes con los que experimentar.
3. Afianzamiento de 5. Se establecen mecanismos de compilación y se crean primeros interfaces estables. Se inicia a portar código de la aplicación Carspeed.
4. Versión 5 final. Apoyada en Orca, y con implementación de algunos componentes completamente funcionales. Carspeed funcional.

El desarrollo de esta memoria se inicia en la segunda iteración y se ha ido intercalando con el resto de iteraciones.

Capítulo 3

Estado del arte

En este capítulo introducimos la actualidad en el campo del software para robots, presentando algunos de los proyectos software más exitosos del momento, de los cuales expondremos sus características más destacables, ya introducidas brevemente en la sección 1.3. Cada uno de ellos ha influido de alguna manera en este proyecto, ya que ante todo se persigue seguir las últimas tendencias. En una segunda sección hablaremos de las herramientas software que se han utilizado, previa introducción a la tecnología que utilizan, con las que se pretenderá abordar algunos de los objetivos marcados en este proyecto.

3.1 Entornos de desarrollo para aplicaciones robóticas

En los últimos años han surgido diferentes proyectos software libre en el campo de la robótica apoyados por una creciente comunidad de usuarios, que no sólo los utiliza, sino que los hace crecer tanto en funcionalidad como en robustez. Estos proyectos acercan la robótica a más y más usuarios.

En el grupo de robótica de la URJC llevamos trabajando en esta línea desde hace ya casi una década. Producto de este esfuerzo es la arquitectura software *jderobot*, de la que ya introdujimos sus antecedentes, estado actual y crítica en la sección 2.1. Ésta última fija las motivaciones del presente trabajo fin de máster, que además ha tenido como referencia los proyectos más destacados del momento. A continuación pasamos a describir algunos de ellos.

3.1.1 Player/Stage

Player/Stage [Collett et al., 2005, Gerkey et al., 2003, Vaughan et al., 2003] es uno de los proyectos software en el campo de la robótica que más entusiasmo ha levantado en los últimos años. Se considera el actual estándar de facto entre la comunidad robótica y es usado en multitud de proyectos robóticos. Player/Stage es software libre y dada su amplia comunidad, goza de soporte para un amplio abanico de hardware e incorpora una gran colección de algoritmos típicos en el terreno robótico.

El proyecto consta de tres partes. Primera, Player un servidor de dispositivos que permite el acceso remoto a uno o varios clientes a través de un API establecido. Segunda, Stage es un simulador en 2D que permite conectar los dispositivos servidos por Player a dispositivos simulados de una amplia gama. Y por último la tercera, añadida más recientemente Gazebo, que es un simulador en 3D que implementa física de movimiento y diferentes tipos de cámaras, además del resto de dispositivos típicos. Por

ello es común referirse al proyecto completo como P/S/G (primera letra de cada una de las partes).

Como hemos introducido, Player es un servidor que permite el acceso a los dispositivos de un robot con el objetivo de controlarlos de manera remota, desde un cliente que envía sus comandos a través de una red TCP/IP. Ejecutando Player en nuestro robot éste provee un canal de comunicación con interfaces definidos para acceder a sus sensores y actuadores. Originalmente Player soportaba la familia de robots de ActiveMedia Pioneer 2, aunque actualmente soporta una amplia gama de robots y hardware encontrado típicamente en aplicaciones robóticas. Esto en parte gracias a que Player es un proyecto de software libre y recibe muchas colaboraciones de su activa comunidad de usuarios. Algunos de los interfaces que encontramos en Player son `laser`, `camara` o `motores`.

Entrando en más detalle con Player, tenemos que se ha diseñado con la idea de que sea independiente tanto de la plataforma como del lenguaje de programación. Un cliente puede ejecutarse desde cualquier tipo de sistema con una conexión de red y puede programarse en cualquier lenguaje que soporte sockets TCP. Actualmente hay varias implementaciones en C++, Tcl, Java y Python. La transmisión de datos se apoya en una librería para el empaquetado/desempaquetado requerida para enviar datos por la red, que también forma parte del proyecto.

Player presenta un dispositivo a través de un interfaz de mensajes con un API bien definido. Dicho interfaz es independiente del dispositivo que lo implementa, de modo que diferentes dispositivos pueden presentar el mismo interfaz. Esto nos permite que código desarrollado sobre un determinado interfaz pueda manejar diferentes dispositivos (diferentes tipos de robots, o robots simulados,etc.). Aunque inicialmente un interfaz representaba un dispositivo físico (sensor/actuador) en la actualidad también existen interfaces que representan algoritmos (por ejemplo el interfaz `localize`). Player soporta el acceso concurrente desde múltiples clientes, permitiendo distribuir el control en diferentes nodos de cómputo. Player no hace ninguna asunción acerca de cómo se debe estructurar el software de un robot, sino que deja al desarrollador elegir lo que más convenga en cada contexto.

En la figura 3.1 vemos la estructura típica de un sistema basado en P/S/G. El servidor Player se conecta a los dispositivos indicados en su fichero de configuración, que bien pueden ser dispositivos reales, o bien pueden estar simulados con Stage o Gazebo. Dichos dispositivos son accesibles por red, utilizando unos APIs conocidos por ambas partes (cliente/servidor). Múltiples clientes pueden conectarse a un servidor Player. Los clientes utilizan una librería cliente que les da acceso al servidor Player desde el lenguaje de programación usado por el cliente. Esta configuración es muy flexible y permite desarrollar múltiples organizaciones.

Stage es un simulador 2D capaz de simular una buena colección de dispositivos usados habitualmente en robots, como lasers, sensores de ultrasonidos, motores, sensores odométricos, etc. . También puede simular cámaras, aunque en este caso son simples proyecciones 2D del mundo simulado. Su diseño permite la simulación de múltiples robots. El enfoque de su simulación se basa en modelos computacionalmente poco costosos, a costa de una menor fidelidad. Habitualmente se utiliza como *plugin* de Player, aunque puede usarse en otros entornos, incluso como librería para introducir modelos simulados en nuestros programas.

Gazebo por su parte es un simulador 3D que persigue simular entornos reales con la mayor fidelidad posible, incluyendo los efectos físicos de la inercia y la gravedad. Además de poder simular los dispositivos de Stage, Gazebo puede simular cámaras tanto monoculares, como pares estéreo, facilitando en gran medida el trabajo de investigación en el campo de la visión por computador orientado a la robótica. De nuevo, es posible usarlo como una librería sin tener que pasar por Player.

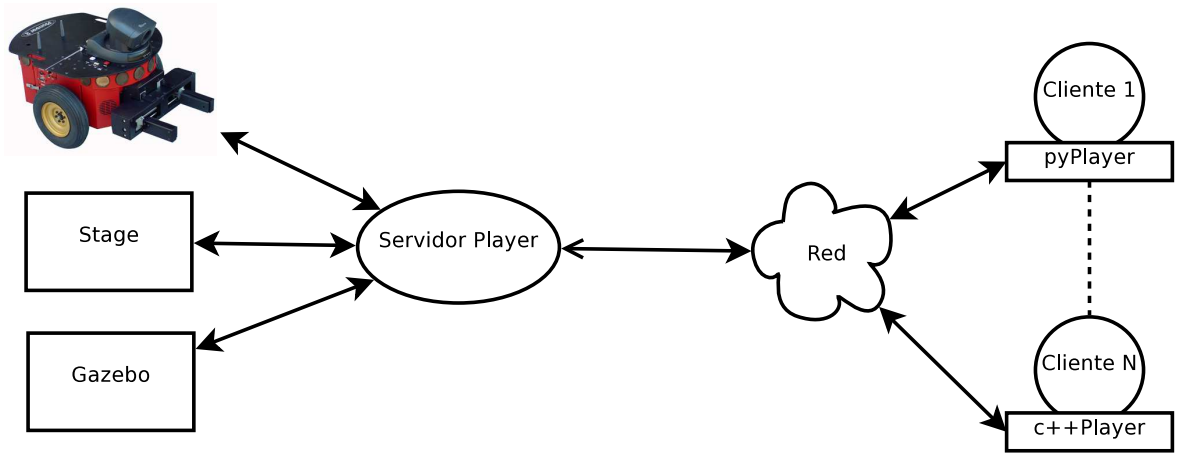
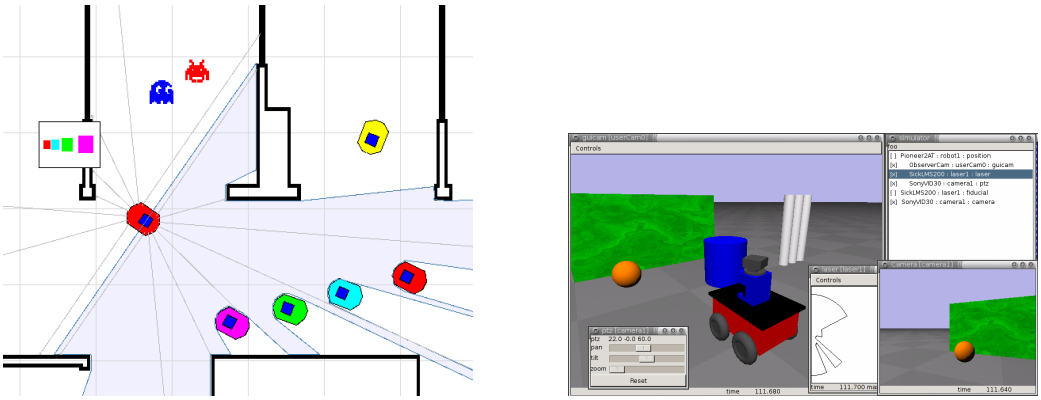


Figura 3.1: *Sistema basado en P/S/G*

En la figura 3.2 podemos ver ambos simuladores en funcionamiento.



(a) (b)

Figura 3.2: *Stage (a) y Gazebo (b)*

3.1.2 Orca

Orca [Makarenko et al., 2006, Brooks et al., 2007] es otro entorno para desarrollo de aplicaciones robóticas, cuyo enfoque principal es el desarrollo de un entorno lo suficientemente robusto como para permitir el desarrollo de aplicaciones robóticas industriales, pero al mismo tiempo lo suficientemente flexible y fácil de usar para poder ser utilizado en entornos de investigación.

Orca es un entorno de desarrollo basado en componentes, entendiendo por componentes a bloques mínimos con los que construir sistemas de cualquier complejidad, desde simples vehículos capaces de evitar obstáculos a complejas redes de sensores. Sus creadores persiguen la idea de la reutilización software a nivel de componente para sistemas robóticos, donde componentes desarrollados por equipos diferentes son capaces de interoperar, llegando así a soluciones más complejas y robustas a los problemas actuales en el campo de la robótica.

A su vez, Orca no indica ninguna restricción a la hora de organizar sus componentes, ni a la estructura interna de un componente, dejando dicha decisión al desarrollador, que como ya hemos comentado, es el enfoque de moda en las últimas tendencias. La única restricción se aplica en el interfaz del componente, que constará de una colección de llamadas que representa el contrato entre usuario-componente.

La primera versión de Orca implementaba su propio *middleware* para comunicación entre componentes, que gestionaba toda la maquinaria de llamadas remotas implicada. Sin embargo, sus desarrolladores se dieron cuenta de que la creación y mantenimiento de un *middleware* es una tarea que requiere mucho esfuerzo, de modo que en la versión 2 de Orca pasaron a utilizar un *middleware* externo, Ice de ZeroC. Así, el esfuerzo anteriormente dedicado a desarrollar el *middleware* se enfoca al desarrollo de aplicaciones robóticas. Orca 2 tan sólo implementa algunos mecanismos habitualmente usados que recubren alguna funcionalidad de Ice y simplifican su uso. El resto del proyecto Orca se compone de *drivers* para acceso a diferente hardware, algoritmos utilizados habitualmente en la robótica, una serie de interfaces y una colección de componentes que los implementan.

Cada componente se ejecuta de manera independiente en un proceso propio como norma general, aunque es posible ejecutarlos dentro del servidor de aplicaciones que incluye Ice, *IceBox*. Cada componente implementa uno o varios interfaces y utiliza otros tantos exportados por otros componentes, creándose así un grafo de dependencias entre componentes. Orca implementa mecanismos que evitan que esto sea un problema, de modo que un componente que depende de otro se queda a la espera hasta que su dependencia aparezca, todo ello apoyado en los ricos mecanismos que aporta Ice.

En la figura 3.3, obtenida de su página web, vemos la aplicación *Reactive Walker* compuesta de varios componentes. Esta aplicación hace moverse a un robot evitando obstáculos a medida que se detectan. Para ello, el componente `ReactiveWalker` utiliza los datos obtenidos por el interfaz `LaserScanner2d` para determinar si hay algún obstáculo y los datos del interfaz `Odometry2d` para saber el punto 2d en que nos encontramos. Con estos datos, aplica sus comandos actuadores a través del interfaz `VelocityControl2d`. Estos interfaces se implementan con varios componentes, `Laser2d` y `Robot2d`. Como se puede apreciar, los componentes que acceden al hardware del robot se apoyan en *Player/Stage*, lo que refuerza la idea de la reutilización que sus desarrolladores persiguen. ¿Para qué implementar lo que *P/S* ya hace? Basta recubrirlo y tener accesible todo el software de *P/S*, principalmente su simulador. Es importante destacar que los componentes pueden estar distribuidos en una red de computadores.

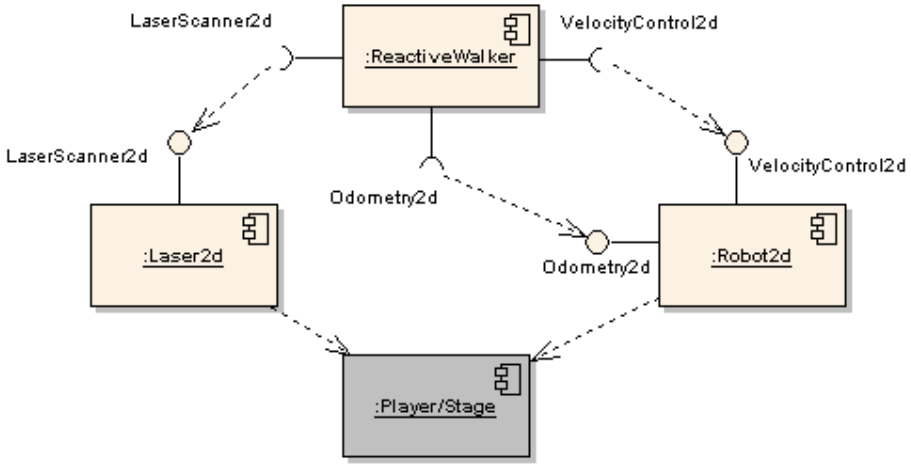


Figura 3.3: *Sistema basado en Orca*

Además de una variada colección de componentes, más o menos simples, Orca se distribuye con una serie de librerías que implementan variadas utilidades. Unas permiten registrar los datos producidos por un interfaz y para posteriormente repartirlos, otras permiten mantener la configuración de todos los componentes de manera centralizada, y otras permiten inspeccionar el estado de un componente.

Por último, se debe comentar que gracias al uso de Ice como *middleware* de comunicaciones, un componente puede ser programado en cualquiera de los lenguajes soportados por Ice, que a día de hoy incluye una gran lista (C++,Python,Java, Tcl, PHP,Ruby,etc.). También puede ejecutar en un gran número de plataformas, desde equipos PC (GNU/Linux,Windows, Mac OSX) , hasta dispositivos móviles (Android, IPhone). Todo esto permite desarrollar aplicaciones robóticas en entornos muy heterogéneos.

3.1.3 ROS

ROS [Quigley et al., 2009] se define como *un meta sistema operativo para robots* que ofrece los servicios que se esperarían de un sistema operativo, tal como abstracción de hardware, control a bajo nivel de dispositivos, implementación de la funcionalidad más habitualmente usada (en el campo de la robótica), mecanismos de paso de mensajes entre procesos (IPC) y un sistema de gestión de paquetes software. Aun así, podemos definir ROS dentro de los entornos para desarrollo robótico, dado que de nuevo tenemos un conjunto de herramientas que no fuerzan una estructura fija para combinar sus componentes, ni cómo deben ser internamente. El enfoque de ROS es crear un entorno que cubra todos los aspectos del desarrollo de una aplicación robótica, tanto los mecanismos para programarlas, para desplegarlas en un sistema como para distribuirlas.

ROS se apoya en tres niveles conceptuales:

1. Sistema de ficheros ROS
2. Grafo de cómputo ROS
3. Comunidad ROS

El nivel de sistema de ficheros ROS (*ROS Filesystem level*) define una serie de conceptos en torno a cómo se organizan los ficheros que componen una aplicación. Se definen los siguientes conceptos:

Paquetes : Son las unidades principales para organizar software dentro de ROS. Un paquete puede contener ejecutables, librerías independientes, conjuntos de datos, ficheros de configuración y otro tipo de utilidades.

Manifiestos : Incluyen la meta-información acerca de un paquete, como información sobre licencias y dependencias, o incluso información especifica para la compilación del paquete.

Pilas (Stacks) : Agrupan paquetes que ofrecen funcionalidad relacionada, como por ejemplo, pila de navegación que agrupa todos aquellos paquetes que tienen alguna funcionalidad relacionada con la navegación. Las pilas además se asocian unas con otras (dependencias) y utilizan un número de versión.

Manifiestos de pilas : Incluyen la meta-información relacionada con una pila.

Tipos de mensajes (msg) : Descripción de las estructuras de datos intercambiadas en los mensajes.

Tipos de servicios (srv) : Descripción del API de un servicio.

ROS organiza sus procesos como una red extremo-a-extremo (*peer-to-peer*) que se denomina grafo de cómputo ROS. En dicho grafo aparecen los siguientes conceptos:

Nodos : Son procesos que realizan algún tipo de cálculo. Ésta es la unidad mínima, e idealmente una aplicación robótica estará compuesta por múltiples nodos. Un nodo ROS se apoya en la librería cliente de ROS, de la que actualmente hay versiones estables en C++ y Python.

Maestro : El *Maestro* ROS provee el servicio de resolución de nombres, con el que unos nodos pueden encontrar a otros usando un nombre lógico para identificarlos.

Servidor de configuración : Es parte del *Maestro* y permite centralizar la configuración.

Mensajes : Los nodos se comunican entre ellos usando mensajes. Cada mensaje consiste en una estructura de datos.

Temas (Topics) : Los mensajes se transmiten a través de un sistema con semántica de publicación/suscripción. El *topic* es un nombre que identifica un canal de comunicación. Un nodo puede emitir información relativa a dicho tema y otro puede suscribirse si le interesa. Puede haber múltiples emisores concurrentes, al igual que múltiples receptores. Este mecanismo desacopla completamente la comunicación entre nodos.

Servicios : Permite un mecanismo de comunicación petición/respuesta o RPC entre nodos. Se define como un par de tipos de datos, que representan los datos de la petición y los datos de la respuesta.

Bolsas (Bags) : Son mecanismos para almacenar y retransmitir los datos que se han transmitido por un *topic* determinado en un intervalo de tiempo. Su uso simplifica la depuración, dado que resulta muy simple repetir una y otra vez los datos.

La figura 3.4 muestra todos estos conceptos y la relación entre ellos en una supuesta aplicación con dos nodos.

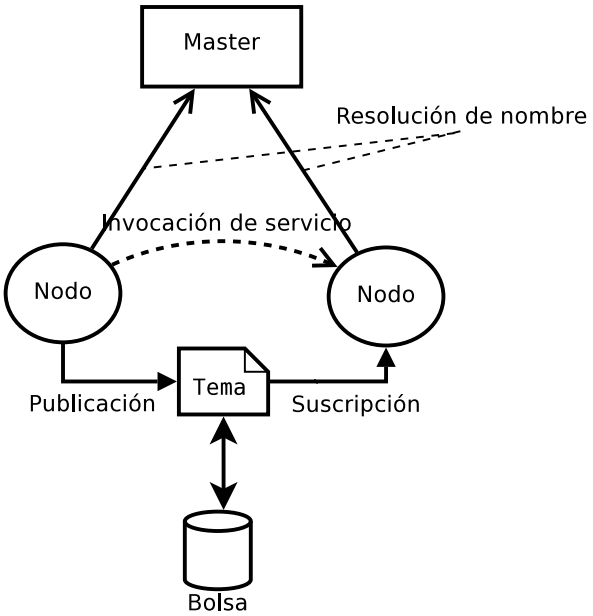


Figura 3.4: *Conceptos de ROS*

En el nivel comunidad ROS describe otros tantos elementos, que atienden a la estructura común de un proyecto de software libre:

Distribuciones : Una colección de pilas en determinada versión componen una distribución.

Repositorios : ROS se apoya en una red federada de repositorios que permite que diferentes grupos de desarrollo mantengan sus aportaciones al sistema ROS.

Wiki : Es el mecanismo principal de documentación donde toda la comunidad de ROS puede añadir su documentación describiendo software, tutoriales, etc.

Sistema de trazado de errores : Permite seguir la evolución de un error que se ha detectado, viendo quien toma parte y que solución se aporta.

Lista de distribución : Permite mantener un canal de comunicación para toda la comunidad ROS.

Blog : Es el sitio web donde se publican noticias relacionadas con ROS.

Actualmente la comunidad de usuarios de ROS está creciendo, y empieza a ser un proyecto bastante activo en la comunidad robótica. A esto se suma el esfuerzo aportado por la empresa Willow Garage[1], interesada en acelerar el uso de robots en entornos no militares. En el mes anterior a la fecha de este proyecto, Willow Garage ha lanzado el programa *PR2 Beta Program* con el que ha donado durante 2 años robots PR2, fabricados por ellos, que integran ROS. En la figura 3.5 podemos ver el prototipo PR2.

Figura 3.5: *Prototipo PR2 de Willow Garage*

[1] www.willowgarage.com

3.2 Herramientas software

Entre los requisitos marcados para este proyecto, se encuentra la *reutilización* del software, no sólo aplicado a que el diseño del sistema debe facilitar su reutilización por terceras personas, sino también enfocado a utilizar todas aquellas herramientas que se consideren oportunas para evitar desarrollar cosas que ya existan. De esta manera, no sólo reduciremos el esfuerzo de implementación, sino que reduciremos el esfuerzo requerido para mantener el sistema en las subsiguientes revisiones.

Dos de los requisitos comentados en la sección 2.3, **R2-Multilenguaje** y **R3-Distribuido**, implican mecanismos para los que hay una basta literatura y cantidad de implementaciones de código abierto. Dado que su implementación requiere un esfuerzo bastante grande, vamos a apoyarnos en alguna de las implementaciones existentes que mejor se adapte a nuestro contexto. Pero antes revisaremos los conceptos.

3.2.1 Interoperabilidad multi-lenguaje

La interoperabilidad multi-lenguaje se refiere a la habilidad de que diferentes lenguajes de programación (para simplificar nos referiremos a los lenguajes A y B) puedan comunicarse entre ellos para llevar a cabo algún tipo de acción, por ejemplo A utiliza una función escrita en B. Esto se puede conseguir de dos maneras. La primera consiste en envolver B con algún tipo de capa escrita en A (llamado *wrapper*) de modo que A puede hablar con B a través de esta capa. La segunda forma de conseguir esto es poner algún mecanismo intermedio que comunique las entidades programadas en diferentes lenguajes. Dicho mecanismo se denomina *middleware* y utiliza lo que se denomina IDL (*Interface Definition Language*) para describir el interfaz que hay entre ambas entidades. A continuación desarrollamos con más detalle cada una de las técnicas.

La técnica de envoltorio nos permite que dos lenguajes puedan interoperar. Para ello se crea un envoltorio o *wrapper* que implementa mecanismos para que las funciones disponibles del lenguaje B sean accesibles desde A, o viceversa, siendo habitual que los recubrimientos sólo funcionen en un sentido. Para ello, cuando desde A hacemos una llamada a B se hace una traducción de la llamada, que incluye la traducción de los parámetros, se realiza la llamada y se devuelve el resultado que previamente se ha traducido. En la figura 3.6 vemos un ejemplo de llamada con todos los pasos que hemos descrito.

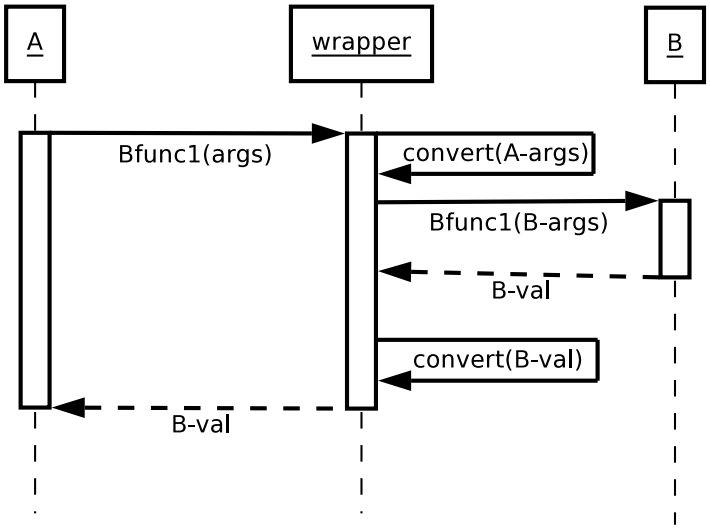


Figura 3.6: *A y B interoperando con un envoltorio*

Envolver código es una técnica especifica de cada lenguaje, dado que cada uno maneja las llamadas a función y representa los datos de una manera diferente. Habitualmente este mecanismo se apoya en alguna librería existente para interconectar dicho lenguaje con otros. Como ejemplo tenemos:

- Python → C con Python C API
- Java → C con JNI
- Perl → C con C::DynaLib

, habiendo mecanismos para hacer el camino contrario, aunque implican algo más de complejidad debido a que se requiere embeber el intérprete dentro del código C. De este modo, podemos ver que la técnica de recubrimiento requiere el esfuerzo de escribir código muy específico para cada par de lenguajes que queramos hacer interoperar. Esto es una tarea compleja, con tendencia a tener errores y difícil de mantener en un sistema grande cada vez que alguna función de su API requiera alguna modificación. Para solventar en parte este problema se creó SWIG [Beazley, 1996][Beazley and Fulton, 2008], que permite mediante un simple lenguaje de definición recubrir código C/C++ para ser usado en multitud de lenguajes modernos como Python,Java, Perl, PHP, etc. Hablaremos con más detalle de esta herramienta un poco más adelante en este capítulo (3.2.3).

La técnica de emplear un *middleware* es algo más elaborada y requiere previamente definir una interfaz de llamadas para la entidad (habitualmente denominado componente) que queremos hacer interoperar. Dicha definición se realiza con lo que se denomina un lenguaje de descripción de interfaz o *IDL*, que generalmente es diferente para cada *middleware*. Con esta descripción, a través de herramientas que acompañan al *middleware*, se genera código que nos permite enlazar la implementación de nuestro componente. A partir de este momento, el componente puede registrar las interfaces que implementa, y el sistema puede llegar a ellas usando los mecanismos de que el *middleware* disponga, independientemente del sentido de la comunicación y del lenguaje en que se programó cada componente. Un ejemplo lo podemos ver en la figura 3.7, donde dos componentes se comunican usando los mecanismos que aporta el *middleware*. Vemos como A exporta un interfaz que B usa.

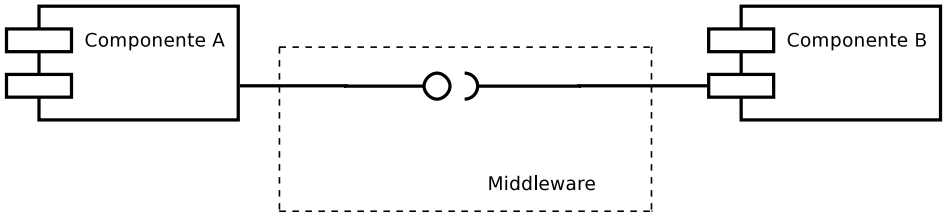


Figura 3.7: *Componentes A y B usando un middleware*

Habitualmente un *middleware* aporta más mecanismos que nos permiten distribuir en una red nuestros componentes, o que nos permiten localizar componentes mediante servicios de nombres, o crear sistemas redundantes. Algunos de los *middleware* más conocidos son:

CORBA : (*Common Object Request Broker Architecture* – arquitectura común de intermediarios en peticiones a objetos) es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. Es un estándar definido y controlado por OMG y existen múltiples implementaciones.

Java RMI : (*Java Remote Method Invocation* – Invocación de métodos remotos para Java) es un mecanismo ofrecido por Java para invocar un método de manera remota. Forma parte del entorno estándar de ejecución de Java y provee de un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java.

COM : (*Component Object Model* – Modelo de componentes) es una plataforma de Microsoft para componentes de software introducida por dicha empresa en 1993. Esta plataforma es utilizada para permitir la comunicación entre procesos y la creación dinámica de objetos, en cualquier lenguaje de programación que soporte dicha tecnología. El término abarca las tecnologías OLE, OLE Automation, ActiveX, COM+ y DCOM.

Ice : (*Internet Communications Engine* – Motor de comunicaciones para internet) [Henning, 2004], es una plataforma *middleware* orientada a objetos que ofrece llamadas remotas, computación GRID y mecanismos de suscripción/publicación desarrollada por ZeroC y licenciada tanto GPL como de manera propietaria para aplicaciones comerciales. Soporta múltiples plataformas y una gran variedad de lenguajes, compitiendo en muchos aspectos con implementaciones de CORBA.

De la plataforma Ice de ZeroC hablaremos más adelante en este capítulo (3.2.4), dado que se ha elegido como base para el sistema *jderobot* 5.0.

3.2.2 Sistemas distribuidos

Un sistema distribuido es aquel que permite resolver un problema distribuyendo diferentes tareas entre más de un nodo de cómputo, habitualmente conectados mediante algún tipo de red de comunicaciones. De esta manera, un problema complejo que requiera de un tiempo x para ser resuelto podría idealmente resolverse en x/N, siendo N el número de nodos. Otras veces la distribución atiende a la distribución de la carga de cómputo, de manera que una serie de tareas que resultarían pesadas para un solo nodo, o que requeriría de un nodo muy potente y caro, se reparten en varios nodos más baratos. Otro escenario para la distribución es la redundancia, donde una misma tarea es repartida por varios nodos y algún mecanismo de detección de errores es capaz de hacer que un nodo esté siempre realizando la tarea indicada.

Existen múltiples tipos de arquitecturas que abordan el problema de los sistemas distribuidos, algunas de las más conocidas son:

Cliente-Servidor : La capacidad de proceso se reparte entre clientes y servidores, aunque su principal ventaja es la gestión centralizada de la información y la separación de responsabilidades, que facilita y clarifica el diseño del sistema.

3-tier : Es un arquitectura diseñada sobre 3 capas, en la que cada una tiene una responsabilidad. La capa de presentación presenta el sistema al usuario, la capa de negocio gestiona la lógica de la aplicación y capa de datos se encarga del almacenamiento y recuperación de los datos. Esta arquitectura puede distribuirse en más de un nivel (nodo físico) dependiendo de la necesidad.

N-tier : Es la generalización de un sistema en 3 capas con N capas. De nuevo puede distribuirse en más de un nivel.

Sistemas cluster altamente acoplados : El término cluster se aplica a los conjuntos o conglomerados de computadoras construidos mediante la utilización de componentes de hardware comunes y que se comportan como si fuesen una única computadora.

P2P : Es una red de computadoras en la que todos o algunos aspectos de ésta funcionan sin clientes ni servidores fijos, sino una serie de nodos que se comportan como iguales entre sí. Es decir, actúan simultáneamente como clientes y servidores respecto a los demás nodos de la red.

Algunas de las tecnologías de sistemas distribuidos más conocidas son NUMA (cluster), Folding@home, CORBA, DCOM, Java RMI, ICE, etc.

El campo particular de la robótica permite de manera natural usar sistemas distribuidos para sus objetivos, dado que un robot puede ser visto como un sistema compuesto de diferentes nodos que procesan la información obtenida desde los sensores, o podemos pensar en un conjunto de robots que de alguna manera trabajan juntos en la resolución de un problema.

Como indicamos anteriormente, hemos elegido el middleware Ice de ZeroC para el desarrollo de *jderobot* 5.0, que además de sus capacidades multi-lenguaje, tiene mecanismos muy potentes para la distribución del cómputo entre múltiples nodos.

3.2.3 SWIG

SWIG es una herramienta de desarrollo que permite conectar programas escritos en C y C++ con una gran variedad de lenguajes de alto nivel como Perl, PHP, Python, Tcl o Ruby, entre otros. SWIG es un proyecto de software libre que surge como ayuda a la comunidad científica para permitir el acceso a la basta colección de librerías de alto rendimiento escritas en C/C++ desde los lenguajes de alto nivel que han surgido en los últimos tiempos, uniendo así el rendimiento de C/C++ con la facilidad de uso de estos lenguajes.

La manera en que SWIG consigue esto consiste en generar envoltorios en el lenguaje destino (alguno de los mencionados) de las llamadas de los programas o librerías programados en C o C++ a los que queremos acceder. Dichos envoltorios son capaces de realizar la traducción de las llamadas en el lenguaje destino a llamadas C/C++ y los datos que se transmitan durante la misma. SWIG utiliza las declaraciones de dichos programas C/C++ y genera los envoltorios necesarios mediante una serie de herramientas. Dichas declaraciones pueden ser leídas directamente de las cabeceras C/C++ (habitualmente ficheros .h) o de una versión *anotada* de éstas llamada definición de interfaz SWIG, que permite añadir algunas directivas que facilitan y dan mayor flexibilidad a la interpretación.

El flujo de trabajo habitual de SWIG se muestra en la figura 3.8. El ejemplo de la figura muestra cómo crear un envoltorio Python para las declaraciones contenidas en `example.i` que definen los elementos contenidos en `example.c`.

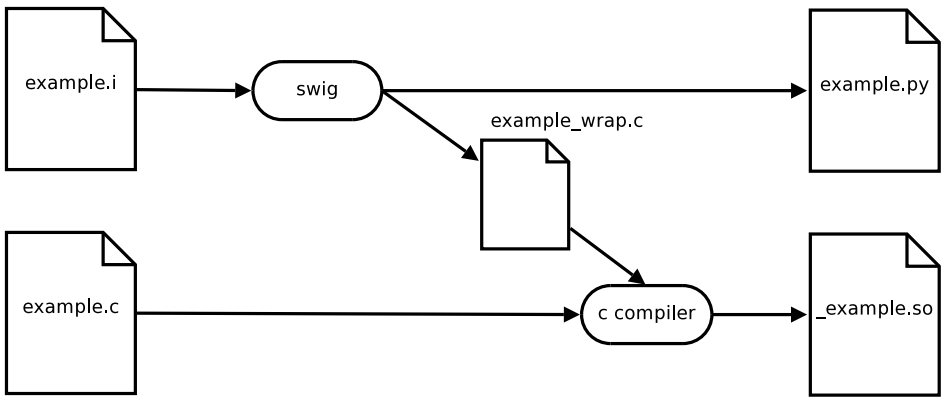


Figura 3.8: *Flujo de trabajo de Swig*

Tras el proceso obtendremos `example.py`, un módulo Python que contiene todo el código necesario para acceder a las funciones y datos contenidos en el código C. Este módulo puede usarse con un simple `import` dentro de Python para acceder a las funciones programadas en `example.c`. La librería dinámica `_example.so` contiene el código objeto del envoltorio, que el módulo Python se encargará de cargar y utilizar. Para el ejemplo mostrado en la figura 3.8 los fichero `example.c` y `example.i` podrían ser como se muestra en los listados 3.1 y 3.2 respectivamente. Suponemos que existe un fichero cabecera `example.h` que incluiría la declaración de las funciones.

```
1  int my_func1(int a, int b) {
2      ....
3      //código de nuestra función
4      ....
5  }
6
7  int my_func2(int a, char * s) {
8      ....
9      //mas código
10     ....
11 }
```

Listado 3.1: *example.c*

```
1  %module example
2  %{
3  #include "example.h"
4  %}
5
6  //Declaraciones ANSI C
7  int my_func1(int a, int b);
8  int my_func2(int a, char * s);
```

Listado 3.2: *example.i*

Para usar este código en Python bastará cargar el módulo y utilizar las funciones, como se muestra en el listado 3.3, donde se puede ver la simplicidad con que se puede acceder al código escrito en C desde Python.

```
1  import example
2
3  a=example.my_func1(2,3)
4  b=example.my_func(2,'hello')
5
6  ...
7  ...
```

Listado 3.3: *Usando example.py*

SWIG permite envolver prácticamente todos los elementos de C como variables, constantes, punteros y tipos de datos. Haciendo uso de la directiva *extend* de SWIG podemos dar un aspecto orientado a objetos de los tipos de datos C, a los que se puede asociar funciones para su construcción, destrucción y acceso. De esta manera, en aquellos lenguajes destino que estén orientados a objetos, los tipos de datos aparecerán

con estas funciones a modo de métodos. Existen otras tantas directivas que a modo de meta información permiten generar envoltorios más sofisticados y más cercanos a los lenguajes destino que facilitan el uso del código C recubierto. SWIG tiene una documentación muy extensa y bien estructurada [Beazley and Fulton, 2008].

3.2.4 Ice

Ice (*Internet Comminications Engine*) es un moderno middleware desarrollado por ZeroC con soporte para una gran variedad de lenguajes y plataformas hardware. Permite desarrollar sistemas distribuidos heterogéneos orientados a objetos, para los que aporta una gran variedad de mecanismos y herramientas. Su idea original persigue crear un *middleware* potente y flexible, capaz de competir con CORBA, pero mucho más fácil de usar y que solucione algunos de los problemas de diseño de éste[Henning, 2004]. Está licenciado GPL para proyectos de código abierto, pero también dispone de licencia comercial para proyectos cerrados.

El elemento central de Ice es *slice*, un IDL con el que se describen los interfaces de los objetos que formarán parte del sistema. Con dicha definición se genera todo el código necesario para enlazar con las implementaciones de dichos objetos. *slice* cuenta con generadores de código para los lenguajes orientados a objetos más populares, como C++, Python, Java o PHP. Gracias a los mecanismos del middleware, sea cual sea el lenguaje en que estén implementados nuestros objetos, estos podrán interoperar a través de los APIs definidos con *slice*.

La plataforma Ice además incluye una serie de servicios que permiten el desarrollo de prácticamente cualquier sistema distribuido, ya sean simples o con requisitos tan exigentes como redundancia o alta disponibilidad. Los servicios más destacados son:

IceGrid : Servicio para *grid computing* que incluye servicios de nombrado, de balanceo de carga, de alta disponibilidad, de despliegue y de administración de nodos.

IceStorm : Servicio de publicación/suscripción que permite distribuir eventos de una manera muy flexible y eficiente entre los objetos de un sistema.

Freeze y FreezeScript : Servicio que permite almacenar objetos en una base de datos.

Glacier2 : Servicio para comunicaciones a través de *firewalls*.

IcePatch2 : Servicio de distribución y aplicación de parches para actualización de los componentes que forman una aplicación de manera automática.

Todos estos servicios ofrecen APIs muy flexibles con los que acceder a toda la funcionalidad que ofrecen en tiempo de ejecución.

Para demostrar las capacidades de Ice, vamos a desarrollar un ejemplo típico *Hola Mundo* en C++ funcionando bajo una arquitectura cliente-servidor. Lo primero que tenemos que hacer es definir el interfaz del objeto que implementaremos para este ejemplo, que tendrá por nombre `Printer`. Simplemente definiremos una operación que imprimirá por la salida estándar la cadena de caracteres que le pasemos como parámetro. El listado 3.4 muestra la definición usando *slice*. Slice requiere que las definiciones sucedan dentro de módulos, que se utilizan para organizar las diferentes declaraciones (es el equivalente a un *namespace* de C++), en este caso hemos llamado al módulo `Demo`. Dentro declaramos el interfaz `Printer` con la operación `printString` que acepta un parámetro de tipo `string`. Suponiendo que esta definición *slice* está en el fichero `printer.ice`, generaríamos el código necesario para usarlo en C++ con el comando:

```
$> slice2cpp printer.ice
```

, que genera los ficheros `printer.h` y `printer.cpp`, declaración y definición respectivamente del interfaz `Printer` en C++.

```
1 module Demo {
2     interface Printer {
3         void printString(string s);
4     };
5 };
```

Listado 3.4: *Definición de Printer*

Una vez definido el interfaz necesitamos implementarlo con un objeto que se ejecute en la parte servidora y que atiende las llamadas a `printString`. El listado 3.5 muestra esta implementación y el programa principal que conformará el servidor, donde se muestra como se registra un objeto en el sistema. Lo primero que encontramos en el código del servidor es la clase `PrinterI` que implementa el método `printString` del interfaz `Printer` (en C++ una clase con métodos abstractos). A continuación, el programa principal (`main`) inicializa los mecanismos Ice y registra una instancia de la clase `PrinterI` que atenderá las llamadas. Dicha instancia se conocerá en el sistema como `SimplePrinter` y que en nomenclatura de Ice se denomina identidad del objeto, y que el cliente utilizará para referirse a ella. Al final del programa principal se controlan las posibles excepciones que pudiesen darse durante la ejecución del mismo.

La parte cliente se muestra en el listado 3.6. En este caso únicamente tenemos un programa principal que conecta con el servidor, obtiene la referencia a la instancia del objeto que implementa el interfaz `Printer` y llama a su método `printString` pasando la cadena de caracteres *Hola Mundo*.

Ambos programas se comunican a través de una red usando el protocolo TCP. En este caso, tanto servidor como cliente han fijado los parámetros de dicha comunicación en el código. El servidor indica que sus servicios serán accesibles por tcp en el puerto 10000 con:

```
Ice::ObjectAdapterPtr adapter
            = ic->createObjectAdapterWithEndpoints(
                "SimplePrinterAdapter", "tcp -p 10000");
```

, y el cliente intenta localizar la instancia con identidad `SimplePrinter` servida por servidor en la misma dirección con:

```
Ice::ObjectPrx base = ic->stringToProxy(
                                "SimplePrinter:tcp -p 10000");
```

Para ejecutar este ejemplo, iniciaríamos el servidor en primera instancia, y a continuación iniciaríamos el cliente que llamaría al método `printString` y que haría que el servidor imprimiese por su salida estándar el mensaje enviado.

Una posible mejora de esta ejemplo sería el uso del servicio de nombrado de *IceGrid*, que no requeriría indicar la localización del servidor, de modo que el cliente podría hacer una petición de resolución de nombres y obtener la dirección concreta en la que se encuentra el servidor, haciendo completamente transparente para ambos su localización dentro de la red.

Indicar por último que cliente y servidor podrían haberse programado en cualquiera de los lenguajes soportados en Ice, e incluso ejecutarlos en plataformas completamente diferentes. Todos estos detalles y los del resto de la plataforma se encuentran muy extensamente descritos en la documentación de Ice[Henning and Spruiell, 2010].

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

class PrinterI : public Printer {
public:
    virtual void printString(const string& s,
                             const Ice::Current&);
};

void
PrinterI::
printString(const string& s, const Ice::Current&)
{
    cout << s << endl;
}

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectAdapterPtr adapter
            = ic->createObjectAdapterWithEndpoints(
                "SimplePrinterAdapter", "tcp -p 10000");
        Ice::ObjectPtr object = new PrinterI;//instanciación
        adapter->add(object,
                     ic->stringToIdentity("SimplePrinter"));//registro de la
        adapter->activate();
        ic->waitForShutdown();
    } catch (const Ice::Exception& e) {
        cerr << e << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic) {
        try {
            ic->destroy();
        } catch (const Ice::Exception& e) {
            cerr << e << endl;
            status = 1;
        }
    }
    return status;
}
```

Listado 3.5: *Servidor Printer*

```
#include <Ice/Ice.h>
#include <Printer.h>

using namespace std;
using namespace Demo;

int
main(int argc, char* argv[])
{
    int status = 0;
    Ice::CommunicatorPtr ic;
    try {
        ic = Ice::initialize(argc, argv);
        Ice::ObjectPrx base = ic->stringToProxy(
                                "SimplePrinter:tcp -p 10000");
        PrinterPrx printer = PrinterPrx::checkedCast(base);
        if (!printer)
            throw "Invalid proxy";

        printer->printString("Hello World!");
    } catch (const Ice::Exception& ex) {
        cerr << ex << endl;
        status = 1;
    } catch (const char* msg) {
        cerr << msg << endl;
        status = 1;
    }
    if (ic)
        ic->destroy();
    return status;
}
```

Listado 3.6: *Cliente Printer*

Capítulo 4

Descripción Informática

Una vez presentados los conceptos, precedentes y líneas base en las que se enmarca este trabajo, pasaremos a describir el objeto del mismo, el desarrollo de la arquitectura software *jderobot* 5. Se hará principal hincapié en las ideas de diseño que se plasman en la implementación a fin de aclarar el porqué de cada decisión.

La versión 5 de *jderobot* viene precedida de la 4.3, última versión estable del software, comentada en la sección 2.1.2. Sin embargo, el salto conceptual que hay entre ambas viene precedido de lo que hemos denominado la versión 4.4, que corresponde a una versión experimental que propone una serie de cambios a la versión estable en busca de determinadas características, como la interoperabilidad y la mejora del API de los interfaces, ideas sobre las que se fundamenta el diseño de la versión 5. Así pues, este capítulo se divide en dos partes principales, una primera donde se habla de la versión 4.4, mostrando las ideas de diseño y los resultados obtenidos mediante un ejemplo simple que implementa un algoritmo de navegación local aleatorio. Y una segunda parte donde se habla de la versión 5, mostrando las ideas y conceptos que la definen, más los resultados obtenidos de su aplicación. En este último caso, el ejemplo elegido es la aplicación *carspeed* [Hidalgo Blázquez, 2008], que se ha portado a la versión 5, como prueba práctica de la arquitectura.

4.1 *jderobot* 4.4

Esta sección describe la versión 4.4 del software *jderobot* con la que iniciamos este trabajo. Se pretende dotar a la versión 4.3 del software de los objetivos que nos hemos marcado, principalmente la capacidad de que el sistema pueda programarse con múltiples lenguajes y que las unidades mínimas del sistema sean entidades más autocontenidas con interfaces bien definidos. El resto de objetivos se abordarán en la medida de lo posible, teniendo en cuenta que no queremos rehacer el sistema *jderobot* 4.3 completamente, sino añadir funcionalidad.

En líneas generales, las ideas tras la versión 4.4 son dos:

- Interoperabilidad entre lenguajes de programación
- Mejora en la definición de los APIs de los interfaces

En la sección 2.2 se argumenta la utilidad de usar diferentes lenguajes de programación, obteniendo como conclusión, que un sistema complejo compuesto de múltiples módulos (o esquemas como denominamos a la unidad mínima en *jderobot*) como una arquitectura software para robots resulta más fácil y flexible cuantas menos limitaciones, en cuanto al lenguaje de programación, presente. De ahí que una de las principales

ideas de la versión 4.4 persiga precisamente esto, dotar de mecanismos que permitan desarrollar e interoperar esquemas o unidades mínimas en diferentes lenguajes de programación, haciendo además posible la comunicación entre ellos.

Por otro lado, en la descripción dada de la versión 4.3 (sección 2.1.2) se introduce la idea de que los esquemas o unidades mínimas se comunican entre sí mediante interfaces, de modo que podíamos ver al esquema como un componente software. Dicho componente implementa una serie de interfaces y usa otros tantos para llevar a cabo su cometido. Todo esto se plasma de una manera un tanto *débil*, ya que en la práctica los esquemas exportan un conjunto de variables al que se denomina interfaz, pero que no lleva relacionado ninguna semántica en cuanto a que representa el valor de cada una de estas variables. Dicha semántica es algo que de alguna manera se acuerda entre las partes, pero que no se obliga mediante un API estricto. Y por ello, otra de las líneas maestras de la versión 4.4 es definir los interfaces de una manera más *fuerte*, dotándolos de las operaciones necesarias que fijen la semántica de cada interfaz, haciendo que un esquema sea una entidad autocontenida que únicamente se comunica con otros esquemas a través de sus interfaces.

4.1.1 Interoperabilidad entre lenguajes en *jderobot* 4.4

En la sección 3.2.1 se introducen una serie de técnicas que nos permiten construir software capaz de interoperar independientemente del lenguaje de programación que se haya usado. De entre ellas, la elegida como solución para dotar a *jderobot* 4.3 de interoperabilidad multi-lenguaje ha sido SWIG. En el apartado 3.2.3 se describe su funcionalidad.

El flujo de trabajo normal de SWIG consiste en describir el API de nuestro software escrito en C mediante el lenguaje de descripción de interfaces de SWIG (muy parecido a cabeceras C) y generar los diferentes recubrimientos en alguno de los diferentes lenguajes soportados por SWIG. En principio es una opción viable, dado que *jderobot* 4.3 está escrito en C, de modo que podemos crear un recubrimiento de todo el API de *jderobot*. Así, tendríamos que las llamadas relacionadas con el API de la arquitectura (mostradas en la tabla 4.1), podrían recubrirse como vemos en el listado 4.1.

put_state()	Cambia el estado de un esquema
speedcounter()	Actualiza contador de iteraciones de un esquema
myexport()	Exporta un símbolo
myimport()	Importa un símbolo
jdeshutdown()	Finaliza la ejecución
get_configfile()	Obtiene el fichero de configuración
get_schema()	Obtiene un esquema dado un identificador
struct JDESchema	Tipo de dato para un esquema
struct JDEDriver	Tipo de dato para un driver

Cuadro 4.1: *API de la arquitectura*

Sin embargo, enseguida surgen algunas complicaciones. Como se comentó en la sección 2.1.2, los esquemas exportan e importan `callbacks` o punteros a funciones, que posteriormente usan para diferentes tareas. SWIG no provee ningún mecanismo para envolver callbacks, y dado que el uso de *callbacks* es un mecanismo principal de comunicación en *jderobot* 4.3, nos vemos ante un problema para el que SWIG no está diseñado. Por otra parte, no sólo queremos que nuestros esquemas escritos en C puedan ser accedidos por otros escritos en otros lenguajes, si no que también sería deseable lo contrario. Y de nuevo, SWIG no proporciona ningún mecanismo para esto.

```
%module jde

%{
#include <jde.h>
%}

/*constants and enums*/

/*functions*/
extern void put_state(int numschema,int newstate);
extern void speedcounter(int numschema);
extern int myexport(char *schema, char *name, void *p);
extern void *myimport(char *schema, char *name);
extern void jdeshutdown(int sig);
extern char* get_configfile();
extern JDESchema* get_schema(int  id);

/*typedefs*/
typedef struct {
    /*structure fields*/
}JDESchema;

typedef struct {
    /*structure fields*/
}JDEDriver;
```

Listado 4.1: *jde.i*

Así pues, lo único que seremos capaces de obtener con SWIG es un recubrimiento que nos permita usar *jderobot* como una librería, de modo que programas escritos en python o java, podrán acceder a toda la funcionalidad implementada en *jderobot*. La versión 4.3 no es muy rica en cuanto a definición del API de la arquitectura en sí, así que un paso previo es modificar o añadir lo necesario a dicho API para obtener un recubrimiento lo más funcional posible. Las principales modificaciones realizadas al API han consistido en declarar tipos de datos y sus correspondientes operaciones (constructores, destructores, acceso, ...) para las principales entidades de la arquitectura:

JDEHierarchy : Representa una jerarquía de esquemas

JDESchema : Representa un esquema

JDEInterface : Representa un interfaz

JDEInterfacePrx : Representa un proxy al interfaz (ver apartado 4.1.2)

Como se ha dicho anteriormente, SWIG no tiene mecanismos estándar para recubrir *callbacks*, principal mecanismo de comunicación de *jderobot* 4.3, de modo que la solución quedaría a medias. Sin embargo, apoyándonos en características de algunos lenguajes para los que SWIG es capaz de generar *bindings*, es posible llegar a una solución completa. Este es el caso de Python y su módulo `ctypes` que permite crear punteros C a funciones desde métodos Python, de modo que podríamos pasar punteros a código escrito en Python, y de este modo, tener esquemas escritos en C y/o en Python interoperando de manera completa. Esta solución se ha plasmado en la clase Python `PyJDESchema`, que hereda de `JDESchema` y es capaz de generar punteros a sus métodos que posteriormente cualquier esquema del sistema puede llamar.

Utilizando características del lenguaje java, es posible que llegásemos a una solución similar. En general, para cada lenguaje deberíamos desarrollar una solución ad-hoc, lo que hace que esta solución requiera de un esfuerzo importante. Así pues, en este punto se ven las limitaciones del recubrimiento mediante SWIG, que aun siendo funcional en cierta medida, no consigue lo que se pretendía en un principio, interoperabilidad completa entre diferentes lenguajes sin necesidad de usar `hacks` ad-hoc para cada uno.

4.1.2 Interfaces

De entre todas las modificaciones aplicadas al API de *jderobot* 4.3, merecen atención las aplicadas a los interfaces. En este apartado se comentan las soluciones empleadas, a fin de obtener un API que fuese simple de recubrir con SWIG, a la vez que funcional.

Hasta el momento, los interfaces definidos en *jderobot* son en los mejores casos simples estructuras de datos (por ejemplo VarColor). En la mayoría de los casos, ni si quiera existe una estructura de datos, si no una colección de llamadas a `myexport()` que definen símbolos accesibles en un esquema y que no son mas que punteros sin tipo (`void`). Con ello, la semántica de un interfaz era un *acuerdo* entre los esquemas implicados, que no siempre se respetaba, o que en el mejor de los casos era diferente entre esquemas. El uso de punteros a memoria hace que la comunicación sea local al proceso, por lo que hasta el momento desarrollar un sistema compuesto de varios procesos no es posible.

Así pues, el primer paso es definir un tipo de datos que represente un interfaz y otro tipo de datos capaz de representar el rol de uso. De este modo, un esquema definirá sus interfaces, y los esquemas que quieran usarlos, accederán a ellos mediante sus representantes. Con ello, la semántica se fija mediante las operaciones declaradas de cada interfaz y todos los esquemas están forzados a seguirla. Los tipos de datos declarados para el efecto son `JDEInterface` y `JDEInterfacePrx`, que representan el tipo base para todo interfaz que definamos en el sistema.

Dado que el sistema *jderobot* 4.3 está escrito en el lenguaje C, no contamos con mecanismos de herencia, que simplificarían la creación de diferentes interfaces, derivados de `JDEInterface`. De modo, que la declaración de cada interfaz es un tanto tediosa, ya que tenemos que declarar una y otra vez las mismas funciones. En este punto, se decide desarrollar una serie de macros de preprocesador para aliviar dicha tarea.

Se han declarado e implementado algunos de los interfaces más utilizados en *jderobot* 4.3 y pueden encontrase dentro de la raiz de la versión 4.4 del svn en el directorio `interfaces`. Para usarlos basta usar la cabecera donde se declaran los tipos `Interfaz` e `InterfazPrx` y todas las operaciones asociadas. Si queremos usar la implementación estándar tendremos que enlazar una vez compilado nuestro código con las librerías adecuadas. Otra opción es implementar nosotros mismos las operaciones, con lo que podemos añadir funcionalidad a los interfaces de manera muy sencilla.

Así, la manera de declarar un interfaz, por ejemplo, para los *encoders* quedaría como muestra el listado 4.2. Dicha construcción de preprocesador, es capaz de generar tanto la estructura de datos adecuada, como todas las operaciones necesarias para manejarla.

En la declaración de los atributos, vemos que podemos usar diferentes tipos (`VARIABLE, ARRAY`). En concreto se han implementado los siguientes:

VARIABLE : Atributos normales que representan un valor.

ARRAY : Para atributos compuestos de colecciones.

SYNTHETIC : Para atributos calculados a partir de una expresión.

Los detalles de uso de estas macros están recogidos en la documentación doxygen contenida en el fichero `interfaces.h`.

```
#ifndef ENCODERS_H
#define ENCODERS_H
#include "interface.h"

#ifdef __cplusplus
extern "C" {
#endif

enum robot_enum {ROBOT_X,ROBOT_Y,ROBOT_THETA,
                                        ROBOT_COS,ROBOT_SIN,ROBOT_NELEM};

#define Encoders_attr(ATTR,I) \
   ATTR(I,robot,float,ARRAY,ROBOT_NELEM) \
   ATTR(I,x,float,SYNTHETIC,self->robot[ROBOT_X]) \
   ATTR(I,y,float,SYNTHETIC,self->robot[ROBOT_Y]) \
   ATTR(I,theta,float,SYNTHETIC,self->robot[ROBOT_THETA]) \
   ATTR(I,cos,float,SYNTHETIC,self->robot[ROBOT_COS]) \
   ATTR(I,sin,float,SYNTHETIC,self->robot[ROBOT_SIN]) \
   ATTR(I,clock,unsigned long int,VARIABLE,0)

INTERFACE_DECLARATION(Encoders,Encoders_attr)

#ifdef __cplusplus
}
#endif
#endif /*ENCODERS_H*/
```

Listado 4.2: *Interfaz encoders*

Tras la declaración anterior, quedan declarados los tipos `Encoders` y `EncodersPrx`, que un esquema puede implementar o usar. Para implementar un interfaz, podemos usar la implementación estándar que acompaña dicho interfaz, o bien redefinirla a nuestro gusto. Para usarlo, basta conseguir una instancia del representante (`EncodersPrx`). En el siguiente apartado se muestra un ejemplo de uso de interfaces.

4.1.3 El proyecto *jderobot* 4.4

Todo el código desarrollado para la versión 4.4 de *jderobot* se encuentra en la rama `branches/4.4` del svn oficial[1]. Su estructura de directorios es similar a la que teníamos en la versión 4.3, a la que se a añadido el directorio `interfaces` que contiene algunos de los interfaces más comúnmente usados por los esquemas de *jderobot*.

Además se ha añadido código en *base* que declara los tipos de datos y operaciones utilizados para definir el API de la arquitectura software como comentamos anteriormente.

Por último, se han añadido una serie de ejemplo que muestran el uso del nuevo código y que pueden ser usados a modo de documentación para explorar las nuevas capacidades de la versión 4.4.

4.1.4 Aplicación randomwalk

Como prueba del sistema *jderobot* 4.4 se ha implementado un esquema que utiliza otros esquemas básicos (motores y láser) mediante los mecanismos comentados ante-

[1] http://svn.jderobot.org/jderobot/branches/4.4

riormente. El objetivo del esquema es avanzar hasta que detecta un obstáculo, comento en el cual gira una cantidad aleatoria de grados que eviten dicho obstáculo y así poder seguir avanzando.

Un extracto del código se muestra en el listado 4.3. En el podemos ver como se usa un interfaz, obteniendo un representante e iniciándolo (despierta al esquema que lo implementa) y posteriormente accediendo o fijando sus valores mediante el API definido.

El mecanismo para utilizar un interfaz implementado por otro esquema es el siguiente:

1. Obtener representante (*proxy*) del intefaz indicando el nombre por el que se conoce al interfaz en el sistema y una referencia al esquema que lo va a utilizar. En el ejemplo mostrado en 4.3 esto se hace con la llamada:

   ```
   l = new_LaserPrx("laser",root_schema);
   ```

   , con la que obtenemos una referencia al representante si es que existe un interfaz llamado así.

2. Indicamos al interfaz que queremos que el esquema que lo implemente debe activarse. De esta manera podemos implementar una activación en cadena en la que los esquemas jerárquicamente superiores activan los esquemas que implementan sus dependencias. Esto se realiza usando una de las llamadas que provee el API de un representante de interfaz:

   ```
   LaserPrx_run(l);
   ```

3. Usamos interfaz a través de su representante, obteniendo o fijando los valores que necesitemos para nuestro programa. En el caso del ejemplo obtenemos los valores leídos por el láser usando su API:

   ```
   LaserPrx_laser_get(l)[i]
   ```

   , donde estamos obteniendo el *array* de valores láser, del que nos fijamos en la posición i.

4. Indicamos al esquema que implementa el interfaz que ya no vamos a seguir usándolo. Caso de ser el único esquema que usaba dicho interfaz, el esquema que lo implementa puede decidir desactivarse para dejar de consumir recursos. Para ello usamos la llamada del API:

   ```
   LaserPrx_stop(l);
   ```

Para demostrar el recubrimiento Python generado con SWIG, el mismo ejemplo se ha implementado también en Python. El listado 4.4 muestra un extracto donde, de nuevo vemos el uso de los APIs, pero esta vez con sintaxis Python. La principal diferencia es que Python es un lenguaje orientado a objetos, por lo que interfaces y representantes son objetos con sus métodos correspondientes.

Sin embargo, aunque el mecanismo multi-lenguaje nos permite hacer programas que usen *jderobot* en todos aquellos lenguajes soportados por SWIG, no resulta posible desarrollar un esquema en Java y usarlo desde otro escrito en Python dado que no resulta fácil recubrir el mecanismo de *callbacks* que hay detrás de un esquema. Además

```
1   ...
2   /*obtenemos un representante del interfaz "laser"*/
3   l = new_LaserPrx("laser",root_schema);
4   if (l == 0){
5     fprintf(stdout,"Can't get laser prx\n");
6     exit(-1);
7   }
8   LaserPrx_run(l);/*iniciamos interfaz*/
9
10  /*obtenemos un representante del interfaz "motors"*/
11  m = new_MotorsPrx("motors",root_schema);
12  if (m == 0){
13    fprintf(stdout,"Can't get motors prx\n");
14    exit(-1);
15  }
16  MotorsPrx_run(m);/*iniciamos interfaz*/
17
18  {
19    unsigned int left_laser,front_laser,right_laser,i;
20    unsigned int threshold = 500;
21    unsigned int left_obst,right_obst;
22
23    right_laser = 0;
24    /*obtenemos valor number de laser*/
25    front_laser = LaserPrx_number_get(l)/2;
26    left_laser = LaserPrx_number_get(l)-1;
27
28    while(shutdown==0){
29      left_obst = right_obst = threshold;
30      for (i=0;i<left_laser;i++){
31        if (i<front_laser)
32          /*obtenemos valor laser[i]*/
33          right_obst = fmin(LaserPrx_laser_get(l)[i],right_obst);
34        else
35          left_obst = fmin(LaserPrx_laser_get(l)[i],left_obst);
36      }
37
38      if ((right_obst < threshold) || (left_obst < threshold)){
39        if (right_obst<left_obst){
40          MotorsPrx_v_set(m,0);/*fijamos valor v*/
41
42          ....
43          ....
44        }
45      }
46    }
47  }
48  ....
```

Listado 4.3: *Random walk en C*

```
1       ...
2       myL = laser.LaserPrx('laser',myH.root_schema_get())
3       if myL == None:
4           print("Can't get laser prx")
5           sys.exit(-1);
6       myL.run()
7
8       myM = motors.MotorsPrx('motors',myH.root_schema_get())
9       if myM == None:
10          print("Can't get motors prx")
11          sys.exit(-1);
12      myM.run()
13
14      threshold = 500
15      right_laser = 0
16      front_laser = int(myL.number/2)
17      left_laser = myL.number
18
19      #from swig carray.i library
20      laserv = laser.intArray_frompointer(myL.laser)
21
22      while True:
23          right_obst = threshold
24          left_obst = threshold
25          for i in range(0,myL.number):
26              if i < front_laser:
27                  right_obst = min(laserv[i],right_obst)
28              else:
29                  left_obst = min(laserv[i],left_obst)
30
31          if (right_obst < threshold) or (left_obst < threshold):#turn
32              if right_obst < left_obst:#obstacle approaching on right
33                  myM.v = 0
34                  myM.w = 50 #turn left
35                  ....
```

Listado 4.4: *Random walk en Python*

declarar y definir las operaciones de un interfaz resulta una tarea bastante compleja, que sólo puede implementarse en C. Si bien, se han desarrollado soluciones a los objetivos que se proponían, en algunos casos parciales, resultan un tanto forzadas dado que se ha tenido que mantener la base *jderobot* 4.3 intacta para que los esquemas desarrollados en esta versión pudiesen ser compatibles con 4.4.

4.2 jderobot 5

En la sección anterior se ha descrito en detalle la versión 4.4 de *jderobot* con la que resolvimos alguno de los objetivos marcados. Pero debido a que aspirábamos a una solución mas completa y elegante seguimos avanzando, pero esta vez sin imponer una implementación base. La razón principal para decidir abrir una nueva línea de desarrollo es que la versión 4.4 aun aportando mejoras a su predecesora, éstas no llegaban a lo que se pretendía desde un principio. Sumado a los puntos flojos que la arquitectura software acumulaba en estos momentos (ver sección 2.1.2) nos hizo replantear la idoneidad de

continuar con esta línea de desarrollo, o si bien merecería la pena empezar de cero con herramientas más modernas y con todos los conocimientos acumulados hasta la fecha.

De este modo, se decide abandonar la línea de desarrollo 4 y volcar los esfuerzos en algo nuevo, que parta desde el principio con todas las ideas que se vienen buscando, más todo el bagaje positivo adquirido durante el desarrollo de todas las anteriores versiones de *jderobot*, naciendo así la línea de desarrollo 5. En el resto de esta sección veremos la base sobre la que asentamos el desarrollo de la nueva versión de *jderobot*, describiremos las características de su middleware, mostraremos algunas ideas de diseño para el desarrollo de aplicaciones basado en la versión 5, veremos una serie de componentes desarrollados y por último visitaremos la aplicación `carspeed 5`, ejemplo completo de uso de *jderobot* 5.

4.2.1 La base de *jderobot* 5

La tendencia mostrada en la sección 1.3, donde vemos cómo grupos de desarrolladores comienzan a concentrarse en torno a determinados proyectos como Player, Orca o ROS, buscando la reutilización de software. Y es esta quizá, la idea más importante a tener en cuenta, ya que el beneficio de disponer de una comunidad de usuarios y una colección de componentes reusables está fuera de toda discusión.

Así, con ello, se decide enfocar la línea de desarrollo 5 hacia alguna de las arquitecturas software principales del momento que, por supuesto, siga las ideas que pretendemos desde un principio. Además de decidir que proyecto se adapta mejor a nuestras ideas, deberemos decidir qué tipo de colaboración adoptamos, es decir, ¿vamos a utilizar ese proyecto como modelo, iniciando una nueva rama de desarrollo sobre la que aplicaremos nuestras modificaciones, o bien vamos a usarlo como base para nuestros desarrollos?.

Siguiendo la clasificación acerca del tipo de software desarrollado en una arquitectura software que encontramos en [Makarenko et al., 2007], podemos decir que nos dedicamos al primer tipo, *drivers* y algoritmos, habiendo desarrollando hasta ahora los otros dos (*middleware* de comunicaciones y *framework* de desarrollo) como un medio para conectar nuestros algoritmos, más que como un fin para nuestros intereses. Esta idea se ve reforzada en las últimas versiones de *jderobot* donde la carga cognitiva de la arquitectura se relajó ante la falta de argumentos en los beneficios de restringir la relación entre componentes o esquemas siguiendo un modelo cognitivo estricto como JDE. De este modo, apoyarnos sobre una arquitectura software que sea flexible y nos aporte tanto el *middleware* de comunicación entre componentes como un entorno de desarrollo, nos permitirá concentrarnos en la parte en la que mayor esfuerzo hemos dedicado en los últimos años.

Los tres proyectos descritos en la sección 3.1 cumplen las características que buscamos. Todos permiten desarrollar componentes en múltiples lenguajes. Componentes que de una u otra manera exportan interfaces que definen el contrato de uso de dicho componente. Además, todos ellos utilizan un *middleware* que provee los mecanismos de comunicación, en todos los casos, comunicación que puede producirse entre diferentes procesos o incluso máquinas. Pero un detalle nos hace inclinarnos por Orca, el uso de un *middleware* completamente externo al desarrollo de la arquitectura: Ice de zeroC[Henning, 2004]. De esta manera, podremos usar todas aquellas partes de Orca que nos aporten algún beneficio, pero sin ligarnos a un *middleware propietario*, pudiendo en cualquier momento prescindir de todo relación con Orca si así lo necesitásemos.

Argumentada la decisión de usar Orca como base para *jderobot* 5, veremos en más detalle que partes nos interesa utilizar. Un listado del directorio `src` donde encontramos todo el código muestra lo siguiente:

examples : Contiene una serie de ejemplos montados con varios componentes.

hydroXXX : Contienen las librerías hydro, que son un conjunto de *drivers* y algoritmos usados posteriormente por los componentes. Entre otros, hay *drivers* para acceso a hardware (lasers, gps, camaras,...).

interfaces : Contiene una colección de interfaces Ice que se implementan en diferentes componentes. Por ejemplo tenemos interfaces que describen el API de cámaras, lasers, gps, o incluso algoritmos.

libs : Contiene librerías que implementan los detalles de la arquitectura software Orca, como elementos de `logging`, depuración o configuración.

components : Contiene una colección de componentes que implementan e utilizan los interfaces comentados anteriormente. Entre otros tenemos servidores de imágenes, servidores de datos sensoriales (reales o simulados desde Stage) o incluso componentes que implementan navegación local.

utils : Contiene librerías con utilidades, como un grabador de `logs` o herramientas para describir un componente mediante un meta-lenguaje que posteriormente permite generar código de manera automática.

Del análisis de este listado sacamos la primera impresión: Orca está altamente modularizado y este hecho nos va a facilitar apoyarnos sólo en aquellas partes que realmente tengan utilidad para nuestro propósito, sin tener que ligarnos de una manera fuerte al proyecto.

De entre todo el material que contiene Orca, la parte más interesante para ser reutilizada es la definición de interfaces. Si simplemente hacemos que *jderobot* 5 implemente los interfaces propuestos por Orca automáticamente obtenemos interoperabilidad entre los componentes desarrollados entre ambos proyectos. Y esto, independientemente del diseño de nuestros componentes, el lenguaje de programación o incluso del sistema operativo donde se ejecuten. De la misma manera, el proyecto Orca podrá beneficiarse de los interfaces y/o componentes que desarrollemos.

Las librerías que implementan *drivers* y algoritmos son también interesantes, aunque en este apartado tenemos un gran bagaje y más que reutilizar, reusaremos toda la colección de *drivers* y algoritmos que poseemos. Aun así, algo muy interesante de sus librerías es la gran homogeneidad que tienen, principalmente debido a un buen diseño de clases, donde cada hay clases base para un amplio conjunto de sensores y actuadores. A esto se suma, la capacidad de carga dinámica de clases a modo de *plugins*, lo que nos permite que un componente que implementa el *driver* X resuelva en tiempo de ejecución si usará el *driver* X_1 que obtiene sus datos de un determinado hardware, o del *driver* X_2 que los obtiene del simulador. Esta característica también se utilizaba en *jderobot* 4.3, y resulta más que interesante para la versión 5. Así, resulta beneficioso adoptar la estructura y diseño de los *drivers* en Orca y refactorizar nuestros *drivers* bajo este diseño y usando las herramientas que Orca nos da, principalmente aquellos *drivers* con los que Orca no cuenta, como el *driver* para gazebo, o el *driver* para el wiimote.

Por último, la librería `orcaice`, contenida en el directorio `src/libs`, contiene una seríe de clases enfocadas a crear componentes para sistemas robóticos. Dichas clases forman la base de todos los componentes de Orca. La clase principal es `orcaice::Component`, que constituye el entorno en el que implementar un componente. Se encarga de proporcionar mecanismos de *log*, mecanismos para definir estados, mecanismos para definir y controlar subsistemas que realizan tareas, o mecanismos para obtener la configuración del componente ya sea por red, por fichero o en una BBDD centralizada. Esta clase puede ejecutarse bien como una aplicación, bien como un servicio dentro de *IceBox*.

La tarea de crear un nuevo componente es tan simple como crear una clase derivada de `orcaice::Component` que redefina aquellos métodos que implementan el comportamiento de dicho componente. En el listado 4.5 podemos ver un breve ejemplo donde se aprecia la simplicidad de crear un nuevo componente.

```
1  namespace myComponent{
2  class Component: public orcaice::Component{
3      public:
4          Component();//inicialización del componente
5          virtual ~Component();//destructor, se liberan recursos
6
7          virtual void start();//inicia tareas del componente
8          virtual void stop();//para tareas del componente
9      private:
10         ...
11         ...
12 };
```

Listado 4.5: *Declaración de un componente*

Recapitulando, tenemos que las partes más interesantes de Orca, que conformarán la base de *jderobot* 5, son:

Interfaces Ice : Nos permitirán interoperar con los componentes desarrollados en Orca.

Diseño y estructura de emphdrivers : Homogeneizará el API de nuestros emphdrivers.

Librería orcaice : Nos aporta el punto de partida para nuestros componentes.

A partir de aquí, el desarrollo de *jderobot* 5 lo basaremos en la creación de nuevos interfaces Ice, en la implementación (en principio refactorización) de *drivers* y algoritmos, y componentes que pongan todas estas partes en marcha.

4.2.2 Ice en *jderobot*

El uso del *middleware* Ice de ZeroC es, sin duda alguna, la característica más importante de *jderobot* 5. Su uso permite cumplir algunos de los requisitos planteados sin ningún esfuerzo extra. Este es el caso de la interoperabilidad entre múltiples lenguajes o el soporte multi-plataforma, que Ice trae de serie. Y por otra parte condiciona el diseño de la arquitectura, al tratarse Ice de un *middleware* orientado a objetos distribuidos.

Como comentábamos en el apartado 3.2.4, Ice es un entorno muy grande, que va desde la definición de los interfaces mediante el lenguaje `slice`, un completo `API` con infinidad de clases y un amplio conjunto de herramientas como *IceGrid*, *IceStorm*, etc., proporcionando un entorno muy flexible. Por ello, en este apartado vamos a detallar la principales partes de Ice que vamos a usar para *jderobot* y cómo vamos a hacerlo.

Definición de interfaces

Para definir nuevos interfaces, usaremos el lenguaje `slice` (Specification Language for Ice), con el que describiremos tanto los métodos como los tipos de datos que se intercambiarán en las llamadas al dicho interfaz.

Su uso seguirá las reglas que se encuentran en el manual, siendo el único detalle a destacar en este apartado que los interfaces que decidamos formen parte de la distribución estándar de *jderobot* se incluyan en el módulo `jderobot`. Esto hará que todos los interfaces se agrupen en el mismo ámbito, que en función del lenguaje de programación elegido, se plasmará en un `namespace` de C++ o un módulo de Python, por citar un par de ejemplos.

El listado 4.6 muestra un pequeño ejemplo de cómo se describiría un interfaz con el lenguaje `slice`. Podemos ver como incluir este interfaz ejemplo en el módulo `jderobot`. La implicación de esto a la hora de generar los *bindings* para C++, será que tendremos que referirnos al interfaz como `jderobot::Image`, o en Python veremos como las clases están incluidas en el módulo `jderobot`.

```
1  module jderobot{
2      ...
3      interface Image{
4          /**
5           * Returns the image source description.
6           */
7          idempotent ImageDescription getImageDescription();
8
9          /**
10          * Returns the latest data.
11          */
12         idempotent ImageData getImageData()
13             throws DataNotExistException, HardwareFailedException;
14     };
```

Listado 4.6: *Definición del interfaz Image*

Servicio de nombrado

El servicio de nombrado de Ice está incorporado en la herramienta *IceGrid* y permite registrar y buscar objetos que implementan un interfaz mediante nombres, en vez de usando sus direcciones de red físicas. Esto nos permitirá distribuir componentes a través de una red y despreocuparnos de dónde reside físicamente cada uno.

Este servicio simplificará, por ejemplo, un escenario muy típico en las aplicaciones que desarrolla el grupo de robótica como es el caso de múltiples servidores de imágenes (obtenidas generalmente de cámaras) ejecutándose en diferentes emplazamientos. Dicho escenario, hasta la fecha, requiere referirse a cada servidor por su dirección física (ip y puerto), siendo necesario modificar los ficheros de configuración cuando movemos unos de estos servidores a otro equipo. Utilizando el servicio de nombrado esto ya no será necesario, ya que nos referiremos a cada servidor por una dirección lógica, que *IceGrid* se encargará de resolver por nosotros.

La sintaxis de estas direcciones lógicas será la siguiente se puede encontrar en el apéndice E.1 del manual de Ice. Habitualmente utilizaremos algo de la forma:

```
id_objeto@plataforma/id_adaptador
```

, donde `id_objeto` habitualmente será el nombre del interfaz que implementa el objeto, `plataforma` se refiere al sistema sobre el que se está ejecutando dicho objeto, que puede englobar un nodo de cómputo o varios, e `id_adaptador` se refiere al nombre del adaptador en el que que se registro el objeto, habitualmente igual al nombre del

componente. Para clarificarlo, se muestra un ejemplo de sistema distribuido en múltiples nodos en la figura 4.1. La plataforma engloba a todos los nodos, y en el ejemplo se llama `europa`. Los nodos situados en las esquinas están ejecutando cada uno una copia del componente `cameraserver`, que implementa el interfaz `camera`. El nodo central ejecuta el componente `eldercare` que utiliza cuatro cámaras. Para referirse a ellas, dados los nombres que aparecen en la figura, tendríamos:

1. camera@europa/cameraserver1
2. camera@europa/cameraserver2
3. camera@europa/cameraserver3
4. camera@europa/cameraserver4

Este sistema de nombrado sigue el estándar de Ice, pero añade el concepto de plataforma, que proviene de Orca. En Ice simplemente se refieren a dicho elemento como categoría y tiene como fin englobar adaptadores de objetos con alguna relación. Como adición de Orca, tenemos que plataforma puede tomar el valor `local`, que en tiempo de ejecución será sustituido por el nombre del nodo donde se ejecute. Su utilidad se aplica a sistemas que no son distribuidos, donde todo el cómputo se realiza en el mismo nodo, no habiendo necesidad de definir una plataforma lógica. Este `hack` sólo está disponible si utilizamos las librerías de Orca para registrar y resolver nombres de interfaces.

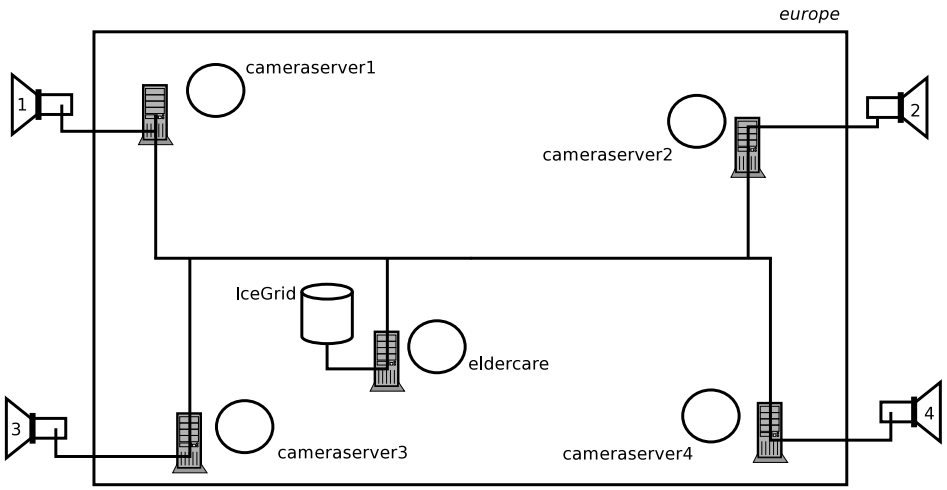


Figura 4.1: *Sistema distribuido*

Transmisión de datos

En este apartado describimos las diferentes técnicas para transmitir datos que tiene Ice. Partiendo por el mecanismo más básico de RPC síncrona, más sus diferentes variantes asíncronas, y terminando en el servicio de suscripción/publicación que provee *IceStorm*. Estudiaremos los diferentes mecanismos en busca de una conclusión sobre cúal de ellos resulta más adecuado para recibir datos desde nuestros sensores y transmitirlos a los actuadores.

Los diferentes sensores y actuadores para los que *jderobot* tendrá que dar soporte manejan una cantidad diferente de datos, y por tanto proporcionan diferentes anchos de banda. Para caracterizar el problema la tabla 4.2 muestra una relación de sensores

junto con su ancho de banda en $bytes/s$ calculado para varios muestreos. Suponemos que los sensores son capaces de entregar las muestras por segundo que indicamos. De modo que, por ejemplo, para el sensor de ultrasonido tendríamos un valor entero (4 bytes) por cada muestra lo que daría un ancho de banda a $10 muestras/s$ de $40 bytes/s$. Los datos nos muestran que una cámara, incluso funcionando a resoluciones bajas, requiere de un ancho de banda muy superior al resto de sensores. Así, dado que gran parte del esfuerzo del grupo de robótica, se centra en trabajos relacionados con la visión por computador, resulta que la cámara será uno de los sensores más habituales, por lo que tenemos que estudiar cómo se comporta Ice a la hora de manejar los grandes anchos de banda que requiere una cámara. En relación con la capacidad de transmisión, tenemos la latencia que existe entre que se requiere el dato y éste finalmente llega a su destino, parámetro que también tendrá implicaciones en la decisión de un mecanismo de transmisión.

Cuadro 4.2: *Ancho de banda de diferentes sensores*

Sensor	bytes por muestra	$10 muestras/s$	$30 muestras/s$
Ultrasonido	4 bytes	$40 bytes/s$	$120 bytes/s$
Laser 180º	720 bytes	$7200 bytes/s$	$21600 bytes/s$
Cámara@320x240 RGB	230400 bytes	$2304000 bytes/s$	$6912000 bytes/s$

Otro aspecto que condiciona la decisión de un mecanismo u otro tiene que ver con el destino de los datos transmitidos y con su capacidad de procesarlos. Es decir, no basta saber cúal es el ancho de banda necesario para transmitir un determinado tipo de datos, sino que tenemos que determinar si el destinatario es capaz de absorber dicho ancho de banda. Siguiendo en el ejemplo de las cámaras, el destinatario será con frecuencia algún tipo de procesamiento, que en unas ocasiones será capaz de trabajar con todos y cada uno de los fotogramas enviados y otras, bien por falta de capacidad de cómputo, bien por que no es necesario para dicho procesamiento, sólo usará una fracción de ellos. A esto lo denominaremos *ritmo acoplado*, cuando tanto el emisor como el destinatario trabajen al mismo ritmo, y *ritmo desacoplado* cuando cada uno trabaje al ritmo que más le conviene.

Entonces, los parámetros que usaremos para caracterizar cada uno de los mecanismos de transmisión que tiene Ice serán: ancho de banda, latencia y relación entre ritmos. Con ellos podremos realizar una decisión fundada sobre que mecanismo funcionará mejor en los escenarios que se nos plantearán con mayor frecuencia en nuestras aplicaciones.

Los mecanismos que encontramos en la versión Ice 3.4 (última disponible mientras se escribe este proyecto) son los siguientes:

1. Llamada remota
    - *a*) Llamada remota estándar
    - *b*) Llamada remota no bloqueante en el servidor
    - *c*) Llamada remota no bloqueante en el cliente
2. Suscripción/Publicación

Como vemos hay dos grupos, llamadas remotas y suscripción/publicación, con sus características propias que pasamos a detallar. La llamada remota estándar es la llamada utilizada por defecto dentro de Ice. El funcionamiento es simple: el cliente realiza la

llamada y se queda bloqueado, Ice construye el mensaje con la llamada y los parámetros y lo envía al servidor, en el servidor se procesa la llamada y se responde, la parte cliente recibe el mensaje y la llamada termina. Este sería el equivalente a una llamada local, donde el flujo de ejecución se bloquea hasta resolver la llamada a la función. En el caso de llamadas remotas, podría suceder que la comunicación se viese interrumpida, caso en el que Ice manejará la situación bien repitiendo mensajes, bien generando una excepción. La parte servidora en la configuración más simple atiende las peticiones de manera secuencial, aunque Ice permite resolver peticiones de manera concurrente, dejando al programador la responsabilidad de tratar adecuadamente las secciones críticas.

La llamada remota no bloqueante en el servidor, o *Asynchronous Method Dispatch*, funciona de manera idéntica en la parte cliente, es decir, este se bloquea hasta que la llamada se ha resuelto. Sin embargo, en la parte servidora el mensaje llega en forma de objeto. Dicho objeto es susceptible de ser almacenado para su posterior procesado, de modo que el servidor no se bloquea respondiendo, sino que responde cuando le viene mejor o cuando los datos para responder están disponibles. Un ejemplo claro de aplicación de este método es cuando la llamada desencadena cálculos costosos en tiempo. En vez de bloquearnos hasta que acaben dichos cálculos, simplemente lanzamos el cálculo y liberamos la llamada, dejando al servidor libre para hacer otras tareas. Cuando el cálculo termina, respondemos con el resultado.

La llamada remota no bloqueante en el cliente, o *Asynchronous Method Invocation*, permite a la parte cliente emitir una llamada remota y continuar su flujo de ejecución sin bloquearse hasta que la llamada se resuelva. En su lugar, se devuelve un objeto representante de la llamada, que puede ser consultado para averiguar si la llamada terminó y conseguir los valores de retorno. La parte servidora funciona igual que en una llamada estándar. En este caso, el ejemplo es un cliente que no puede bajo ningún concepto bloquearse en una llamada remota, que a priori no se sabe cuánto va a durar. De este modo, se lanza la llamada y el cliente se encarga de comprobar cuando tiene un momento si la respuesta ha llegado o no.

Por último, el mecanismo de suscripción/publicación permite que un componente publique sus datos utilizado un tercero, en nuestro caso *IceStorm*, que se encargará de distribuirlo entre todos aquellos componentes que se hayan suscrito. Este mecanismo es completamente diferente a los anteriores, en este caso tanto emisor como receptor están completamente desacoplados.

Estos cuatro mecanismos se ofreceran como parte de los mecanismos de comunicación de *jderobot*, siendo unos más adecuados en función del contexto que otros. Lo que estamos buscando en este caso, es caracterizar todos ellos con una serie de experimentos que nos den datos sobre cómo de buenos son en la tarea más exigente que podemos pensar, transmitir un flujo de imágenes. Hallar el mecanismo óptimo para este caso es de vital importancia para el éxito de la infraestructura en las aplicaciones que manejamos actualmente. Aun así, la decisión final tendrá que motivarse con el contexto, que en algunos casos exigirá uno u otro método.

Se han llevado a cabo dos experimentos para valorar por un lado el mecanismo RPC y por otro el mecanismo de suscripción/publicación. No hemos hecho distinción entre los diferentes mecanismos de RPC, dado que en el fondo todos ellos representan lo mismo. El escenario de los experimentos es el siguiente, tenemos un servidor y un n clientes que solicitan 1000 imágenes continuamente, sin pausa entre petición. En el caso del experimento para el mecanismo de suscripción/publicación, el servidor publica continuamente imágenes y los clientes se suscriben hasta recibir las 1000 imágenes. El servidor se ejecuta en una máquina y los n clientes en otra para añadir la limitación de transmisión por una red ethernet. El experimento se ha repetido para ambos mecanismos con 1,2,4,8,16,32 y 64 clientes, obteniendo en cada prueba la latencia entre la

recepción de imágenes. En el primer caso, representa el tiempo transcurrido desde que se realiza la llamada hasta que ésta retorna, y en el segundo, el tiempo transcurrido entre dos publicaciones. En la figura 4.2 podemos ver el montaje de cada uno de los experimentos con todos los elementos en juego. En el primer experimento vemos como todos los clientes atacan al servidor de manera simultánea, y en el segundo, vemos como el servidor publica sus datos en *IceStorm*, encargándose éste en última instancia de enviar los datos a cada cliente.

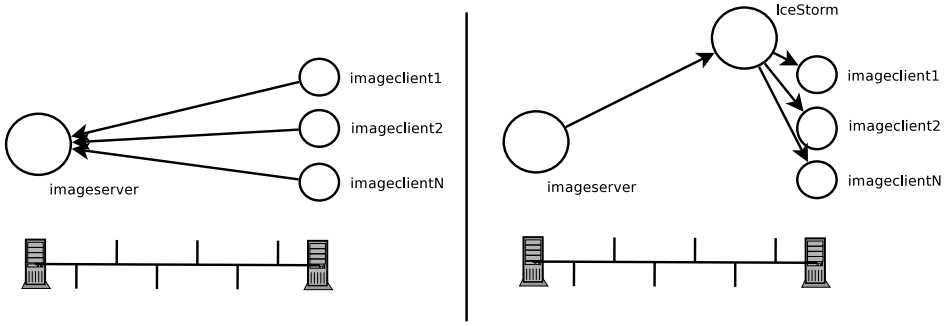


Figura 4.2: *Experimento de transmisión de datos*

Los resultados de tiempos se muestran en la figura 4.3 y muestran un crecimiento proporcional al número de clientas en la latencia de la RPC y una latencia constante en la transmisión mediante suscripción/publicación. Los datos concuerdan con lo que cabía esperar de este experimento, ya que en el primer caso el tráfico de red crece en proporción al número de clientes, mientras que en el segundo caso, siempre se transmite una sola copia de la imagen, que *IceStorm* se encarga de distribuir. Por supuesto, si *IceStorm* estuviese colocado en la misma máquina que el servidor, el comportamiento sería muy parecido al de RPC, siendo importante en el diseño de un escenario la distribución de cada componente para optimizar la transmisión de datos.

Así la conclusión que podemos sacar, es que el mecanismo de suscripción/publicación puede optimizar de manera muy notable la transmisión de grandes cantidades de datos. En un escenario real, es más que probable que no nos acerquemos a las características del experimento, pero en el momento en que tengamos más de un receptor de un mismo flujo de datos (flujos pesados principalmente) obtendremos beneficio encaminándolo por *IceStorm*.

4.2.3 Patrones e ideas para el diseño de aplicaciones en *jderobot* 5

La nueva organización del software que propone *jderobot* 5 se centra principalmente en el desarrollo de componentes que interaccionan entre sí. Sus ideas en líneas básicas se describen en las tesis que apoya la Ingeniería del Software orientada a Componentes (CBSE) y que pone su énfasis en la descomposición de sistemas ya conformados en componentes funcionales o lógicos con interfaces bien definidas usadas para la comunicación entre componentes. Internamente un componente puede desarrollarse siguiendo el diseño que mejor se adapte a su cometido. De entre los componentes que son susceptibles de ser desarrollados en plataformas robóticas podemos obtener una serie de ideas generales, que motivan el desarrollo de estrategias comunes para abordar dichas ideas.

En este apartado vamos a describir dos ideas que, con toda seguridad, serán comunes a muchos componentes. La primera es cómo dotar a un componente de una interfaz de usuario, ya sea gráfica o de texto, mediante la cual podamos comunicar el componente

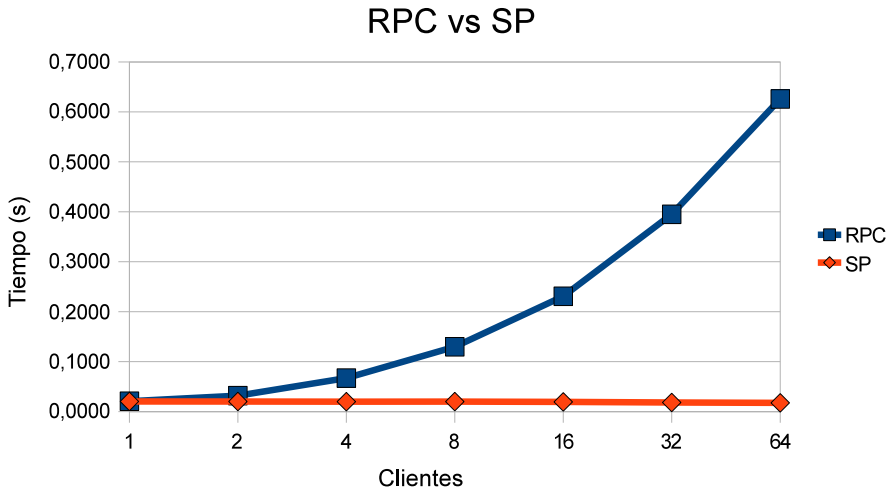


Figura 4.3: *Resultados experimento de transmisión de datos*

con el usuario, principalmente con el objeto de mostrar información de depuración. La segunda idea, propone seguir un diseño común a la hora de implementar algoritmos iterativos, los cuales son muy comunes en muchas de las soluciones desarrolladas en nuestro grupo, dado que la arquitectura cognitiva *JDE* impone a sus unidades mínimas tener flujos iterativos dado que se aplica un control reactivo.

Por supuesto, éstas no son las únicas soluciones aplicables, ya que *jderobot* 5 ante todo representa flexibilidad, pero es seguro que pueden aplicarse a muchos de los problemas que se han resuelto en trabajos anteriores y que se resolverán en el futuro.

Patrón arquitectonico Modelo-Vista-Controlador

Cuando desarrollamos un componente que gestiona internamente información y/o procesos complejos, con frecuencia se requiere algún mecanismo de depuración más avanzado que simples trazas impresas. Por ejemplo, si manejamos una imagen a la que aplicamos un filtro, o sobre la que se busca una característica, será mucho más útil contar con una ventana gráfica sobre la que se pinte dicho procesado que una serie de mensajes de texto.

De este modo, debemos pensar en patrones arquitectónicos que describan cómo solucionar este escenario. Uno de estos patrones es el de *Modelo-Vista-Controlador* [Krasner and Pope, 1988] que describe la arquitectura de aplicaciones interactivas en las que los datos se aíslan de la manera en que éstos son representados y de cómo se interacciona con ellos.

Los elementos de este patrón son el modelo, que representa los datos, la vista que los *renderiza*, y el controlador que gestiona la interacción entre los dos anteriores. En la figura 4.4 vemos un diagrama de clases con las diferentes entidades y sus relaciones. Como se puede ver, el modelo mantiene su independencia del resto de clases, dado que no genera ninguna llamada que se aplique sobre la vista o el controlador, de modo que no requiere tener conocimiento de estas clases. Su única acción es generar notificaciones indicando que sus datos han cambiado que serán usadas por la vista para actualizar su representación y por el controlador para realizar alguna acción relacionada con la lógica de la aplicación. Por otra parte, el controlador además gestiona las acciones que el usuario efectúa trasladando las llamadas necesarias al modelo.

En su aplicación a la hora de diseñar componentes para *jderobot* 5 con interfaces de usuario tendremos que, el modelo será el estado del componente, con todos los

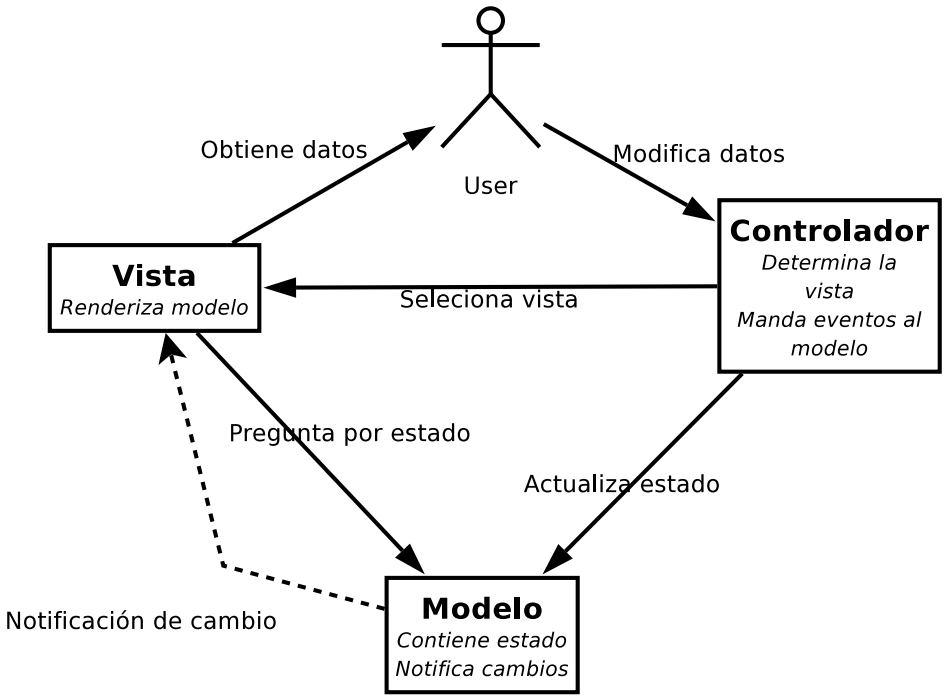


Figura 4.4: *Diagrama de clases del patrón MVC*

datos que este contenga. La vista será algún tipo de interfaz de usuario, bien gráfica, bien textual. Y el controlador manejará toda interacción entre ambos y atenderá las peticiones del usuario de la interfaz gráfica.

La implementación de las notificaciones que genera el modelo se materializa usando el patrón de diseño `Observador`, en el que el rol de sujeto observado se aplica al modelo, y el de observador se aplica a la vista, que recibirá cada notificación y actuará actualizando su representación del modelo. Dicha notificación puede ser síncrona, de modo que la vista se actualice en el mismo flujo de ejecución, o bien asíncrona si deseamos que la vista tenga un flujo de ejecución desacoplado del principal del componente. Esto es interesante si el *renderizado* tiene un coste computacional alto y no queremos que afecte al curso normal de ejecución de trabajo principal del componente. En este caso, además puede ser necesario serializar el acceso al modelo para que no aparezcan condiciones de carrera por actualizaciones/consultas simultáneas.

El uso de este patrón con *GTK+*, una de las librerías más utilizadas en las aplicaciones del grupo, sería de la siguiente manera:

Modelo : Contendría los datos de la aplicación y derivaría de la clase `Subject`.

Vista : Correspondería a la clase que representa la interfaz gráfica y derivaría de la clase `Observer`. Contiene los métodos para que el modelo pueda informar de que ha habido algún cambio en sus datos.

Controlador : Contiene la lógica de la aplicación. Sus métodos se llaman desde el flujo principal y estos desencadenan la actualización del modelo. Se encargaría de inicializar el contexto *GTK+* y de crear la vista.

GTK+ puede tener su propio flujo de ejecución, o bien integrarse en otro flujo y procesar sus eventos cuando nos interese por medio de una llamada. Ésta última es la manera más simple, dado que los eventos se procesan en algún punto de la iteración de nuestro componente, manteniéndose el acceso al modelo completamente serializado, sólo una hebra accede a él. Esta variante tiene sus limitaciones, principalmente en el

caso de que procesar los eventos de la vista resulte costoso computacionalmente, ya que el tiempo de procesado de dichos eventos se sumaría al tiempo del procesamiento que se haga en cada iteración. La alternativa es crear un flujo de ejecución propio para la vista que con *GTK+* resulta tan simple como crear una hebra y llamar a `gtk_main()` en su flujo de ejecución. Para que esta alternativa funcione correctamente debemos controlar el acceso concurrente al modelo, que ahora puede suceder tanto desde el flujo de ejecución del controlador como desde el de la vista.

Patrón de diseño Algoritmo Iterativo

Otro punto común en muchos de los componentes es que implementen algún tipo de algoritmo con el que procesan sus entradas produciendo datos de más alto nivel o que controlan algún tipo de sistema de manera iterativa. Un ejemplo podría ser un componente que recibe imágenes y devuelve características de éstas, como áreas en las que se ha producido movimiento, líneas que delimitan los objetos que contiene, o información sobre objetos de un determinado color.

La mayoría de estos algoritmos son iterativos, es decir, se ejecutan una y otra vez recibiendo datos con alguna relación temporal o espacial y producen resultados tras un número de iteraciones o en cada una de ellas. Habitualmente dichos algoritmos se configuran con determinados parámetros, que pueden modificarse o no a lo largo de las sucesivas iteraciones, y mantienen un estado interno.

Un ejemplo más detallado de un algoritmo iterativo sería la detección de movimiento a partir de imágenes en tiempo real. Las imágenes van llegando a medida que se obtienen de su fuente, por ejemplo una cámara, y se van procesando una a una. Pero para detectar movimiento necesitamos, al menos, dos imágenes consecutivas. Podemos simplemente esperar a tener dos imágenes y ejecutar el algoritmo de detección, o podemos implementar un algoritmo iterativo, que reciba imagen por imagen, almacene en su estado interno aquello que necesite, y a medida que vaya generando resultados los devuelva de alguna manera. Siguiendo este patrón podemos implementar gran parte de los algoritmos que el grupo de robótica desarrolla, con el beneficio de seguir una estructura común para todos ellos.

El patrón de diseño *Algoritmo Iterativo* se compone de las clases que se muestran en la figura. La principal de ellas es `Algorithm`, que se construye indicando una configuración concreta mediante `AlgorithmConfig`, y que permite obtener su estado (`AlgorithmState` y ejecutar una iteración del algoritmo. A dicha iteración se le suministran datos de entrada con `AlgorithmInput` y una configuración opcional, tras la ejecución devuelve datos con `AlgorithmOutput`.

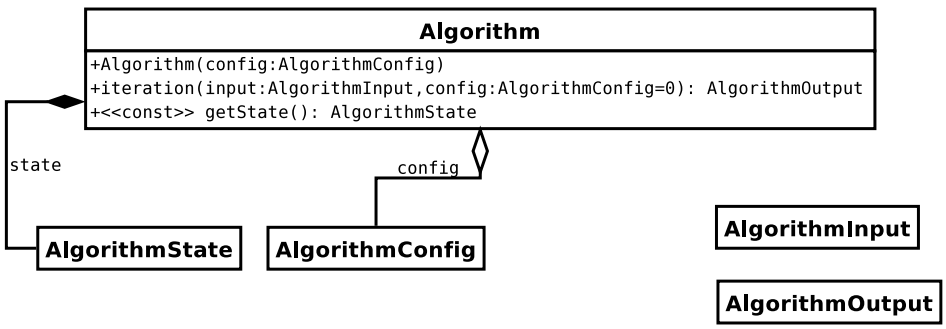


Figura 4.5: *Diagrama de clases del patrón de diseño Algoritmo Iterativo*

La aplicación del patrón de diseño *Algoritmo Iterativo* nos permitirá obtener una colección de algoritmos encapsulados de la misma manera y que a medio plazo nos

permitirá construir una biblioteca de algoritmos con una estructura similar. En los siguientes apartados veremos dos ejemplos de aplicación del patrón en componentes reales.

4.2.4 El proyecto *jderobot* 5

Hasta aquí hemos hablado de conceptos y técnicas a seguir dentro del nuevo entorno de desarrollo que abre *jderobot* 5. En este apartado describimos todo el desarrollo concreto entorno al proyecto *jderobot* 5 en el cual se han desarrollado una serie de componentes y librerías que refactorizan elementos que ya teníamos en la versión 4.3 y que forman los pilares de la nueva versión, y se ha creado toda la estructura entorno a un proyecto de software. Dicha estructura se compone del sistema de gestión de la compilación del proyecto, basado en *GNU Autotools*, el sistema de documentación basado en *MediaWiki*, un conjunto de listas de distribución para comunicación con usuarios y desarrolladores, un sistema de reporte de errores basado en *Trac* y el control de versiones basado en SVN. De éstos sólo el primero ha sido desarrollado en este trabajo fin de máster, el resto aprovechan la infraestructura ya existente en la versión 4.3.

En cuanto a los componentes desarrollados, con los que hemos ido explorando los mecanismos de Ice y aplicando las ideas que hemos descrito hasta este momento tenemos:

cameraServer : Implementa un servidor de imágenes extraídas de multitud de fuentes, como cámaras, vídeos, o recursos *streaming*. Permite la transmisión tanto por RPC, como a través de *IceStorm*.

cameraView : Implementa un cliente para `cameraServer` que muestra por pantalla las imágenes obtenidas aplicando el patrón `MVC` visto en el apartado 4.2.3.

motionDetection : Implementa un componente que aplica un algoritmo de detección de movimiento a las imágenes obtenidas desde un servidor. Es un ejemplo de aplicación del patrón `Algoritmo iterativo` visto en el apartado 4.2.3.

Además de estos componentes, se ha implementado una versión en C++ de la librería `colorspaces` denominada `colorspacesmm` que provee la clase `colorspaces::Image` capaz de manejar múltiples formatos de imágenes apoyándose en `OpenCv`. Su uso ha simplificado en gran medida la implementación de los componentes anteriormente comentados.

A continuación se hace un análisis algo más detallado de cada uno de estos desarrollos.

cameraServer

`cameraServer` es un componente capaz de exportar las imágenes extraídas de una o más cámaras mediante el interfaz `camera`. La fuente de dichas imágenes es totalmente configurable y puede provenir de:

1. Una cámara (usb o firewire)
2. Un video (mpeg, divx, ...)
3. Una fuente de prueba con múltiples patrones (ruido estático, barras, colores, ...)
4. Un recurso streaming remoto (http,https,rtsp,...)

Esto hace de `cameraServer` un componente muy versátil, permitiéndonos obtener video de casi cualquier fuente imaginable e inyectarlo en nuestras aplicaciones mediante el interfaz `camera`. Dicha versatilidad proviene del uso del framework *gstreamer* [The GStreamer team, 2010] que es la *navaja suiza* cuando nos referimos a multimedia.

gstreamer permite modelar *pipelines* con los que manejar diferentes elementos multimedia, desde su origen (cámaras, ficheros, red, ...), pasando por su procesado (codificación/decodificación, filtrado,...) hasta su destino (pantalla, fichero, red, aplicación,...). Existen multitud de elementos que pueden acoplarse a un *pipeline*, pudiendo crear complejos mecanismos de procesado multimedia.

Para el componente `cameraServer` se desarrolló un *pipeline* capaz de obtener imágenes de múltiples fuentes y adaptarlas (codificarlas, modificar tamaño, remuestrear) al formato que el usuario desee. El *pipeline* se muestra en la figura 4.6, donde podemos ver los diferentes elementos. Según la configuración se elegirá uno u otro elemento de entrada (v4l,v4l2,...), posteriormente el elemento `videocolor` ajusta el espacio de color (RGB,YUV,...), le sigue `videorate` que remuestrea el flujo de fotogramas por segundo, y por último `videoscale` ajusta el tamaño del fotograma. Los parámetros de ajuste se toman del elemento `output caps` que es un elemento de *gstreamer* que permite describir el formato que un flujo debe tener al paso de un elemento a otro. Dichos parámetros se adecuan a lo que se encuentra en el fichero de configuración. El flujo de video termina en el elemento `appsink` que no es más que un buffer que almacena los fotogramas y permite a una aplicación obtenerlos mediante una serie de llamadas.

El fichero de configuración permite indicar varios parámetros, entre otros la fuente de las imágenes y sus características (ancho, alto, tasa,etc.). Para indicar la fuente se usa una *uri*[2] en la que el campo de esquema se usa para seleccionar la fuente y el campo autoridad se usa para los parámetros específicos de cada fuente. Un pequeño extracto de este fichero de configuración con varías fuentes sería así:

```
#camera 0
CameraSrv.Camera.0.Name=cameraA
CameraSrv.Camera.0.ShortDescription=Camera pluged to /dev/video0
CameraSrv.Camera.0.StreamingUri=rtsp://192.168.1.15:8080/test.sdp
CameraSrv.Camera.0.Uri=v4l:///dev/video2
CameraSrv.Camera.0.FramerateN=25
CameraSrv.Camera.0.FramerateD=1
CameraSrv.Camera.0.ImageWidth=320
CameraSrv.Camera.0.ImageHeight=240
CameraSrv.Camera.0.Format=RGB888

#camera 1
CameraSrv.Camera.1.Name=cameraB
CameraSrv.Camera.1.ShortDescription=Camera simulated from a video
CameraSrv.Camera.1.Uri=http://www.facethewind.com/videos/may29_01.mpg
CameraSrv.Camera.1.FramerateN=15
CameraSrv.Camera.1.FramerateD=1
CameraSrv.Camera.1.ImageWidth=320
CameraSrv.Camera.1.ImageHeight=240
CameraSrv.Camera.1.Format=RGB888
```

El componente permite obtener las imágenes bien mediante el API del interfaz `camera`, o bien mediante suscripción a través de *IceStorm*.

El interfaz *camera* se ha definido utilizando el lenguaje de definición `slice` y tiene los siguientes métodos:

[2]http://tools.ietf.org/html/rfc2396

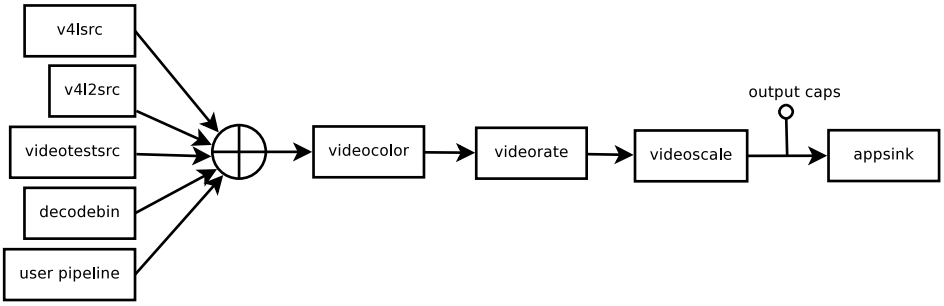


Figura 4.6: *Pipeline Gstreamer usado en* `cameraServer`

getImageDescription() : Devuelve la información sobre las imágenes que se van a servir (ancho, alto, formato,etc.).

getCameraDescription() : Devuelve la información relativa a la cámara (nombre, descripción y url para streming).

getImageData() : Devuelve la imagen e información relacionada (descripción y marca de tiempo).

startCameraStreaming() : Inicia el streaming de la cámara.

stopCameraStreaming() : Para el streaming de la cámara.

Sea cual sea el origen de las imágenes, `cameraServer` las ofrece de manera homogénea y transparente al usuario mediante los métodos indicados.

Gran parte de las aplicaciones en las que trabaja el grupo de robótica de la URJC utilizan información obtenida de cámaras por lo que este componente resulta de gran importancia. Su versatilidad a la hora de utilizar diferentes tipos de fuentes para obtener la imágenes resulta muy ventajosa en cuanto a depuración y ejecución de pruebas. Por ejemplo, utilizando una fuente de imágenes grabada en un fichero podemos obtener comparativas de diferentes algoritmos que resuelvan una misma tarea para poder elegir aquel con mejor rendimiento.

La implementación actual del componente `cameraServer` permite crear múltiples instancias de interfaz `camera`, cada uno obteniendo sus imágenes desde diferentes fuentes.

cameraView

`cameraView` es un componente muy simple que permite mostrar por pantalla las imágenes obtenidas desde un componente `cameraServer`. Su utilidad es meramente de depuración, aunque su desarrollo ha permitido experimentar las diferentes opciones de cómo integrar un interfaz gráfico en un componente a raíz del cual se investigó el patrón de diseño *MVC* comentado en el apartado 4.2.3.

El funcionamiento es simple: una vez obtenida la referencia al interfaz `camera` se obtiene imágenes de manera continua que se insertan en el modelo. Este a su vez genera una orden de actualización que todas las vistas (en este caso una ventana) reciben y pasan a re-pintar. La comunicación entre ambos es gestionada desde el controlador. Hay que destacar que el modelo no es actualizado únicamente por un usuario humano, que manejando la interfaz gráfica pulse algún botón, sino que un proceso encargado de obtener las imágenes de `cameraServer` se encarga de insertarlas en el modelo. Esta interacción es una modificación de la aplicación estricta del modelo y la gestiona el controlador, que es el que conoce la lógica de la aplicación.

Dado que usamos Ice para comunicar ambos componentes, `cameraServer` y `cameraView` podrían ejecutarse en máquinas diferentes conectadas por una red. Por el momento `cameraView` sólo es capaz de conectarse a una fuente de imágenes, pero ampliar esto a más resulta una tarea trivial.

La interfaz gráfica es muy simple y tan sólo se muestra la imagen obtenida e información sobre la tasa de transferencia. Para la configuración de este componente utilizamos una sintaxis similar a la vista para `cameraServer`, siendo en este caso únicamente necesario indicar la dirección del objeto que implementa el interfaz `camera` en el que estamos interesados. Un ejemplo sería así:

```
Cameraview.Camera.Proxy=cameraA:tcp -h 127.0.0.1 -p 9999
```

motionDetection

`motionDetection` es un componente muy parecido a `cameraView`. Utiliza los patrones de diseño *MVC* y *Algoritmo Iterativo* para realizar su trabajo. El objetivo es detectar movimiento en el flujo de video que se recibe desde un `cameraServer`.

El funcionamiento es idéntico al del componente `cameraView`, pero en este caso el modelo además incorpora una instancia de la clase `MotionDetectionAlgorithm` dentro del modelo que es actualizada iterativamente cada vez que llega una imagen y que es capaz de devolver información sobre las áreas de la imagen en las que se ha producido movimiento. El controlador por su parte, cuando detecta que ha habido movimiento genera eventos que son enviados al componente `recorder` parte del proyecto *Surveillance*[Calvo Palomino, 2010]. La figura 4.7 muestra un diagrama de las clases usadas en el componente.

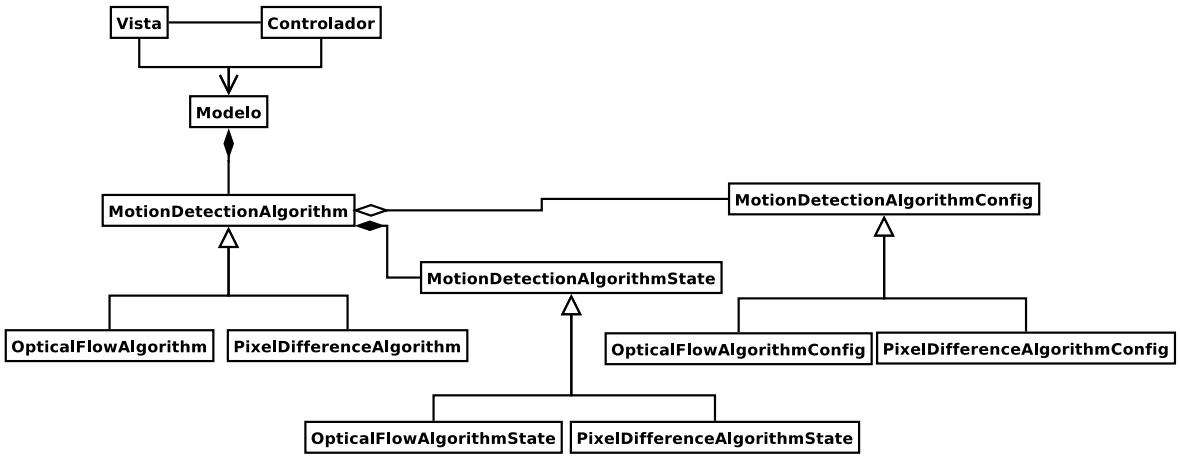


Figura 4.7: *Interfaz gráfica del componente* `motionDetection`

La clase `MotionDetectionAlgorithm` sigue el patrón de diseño *Algoritmo Iterativo*. Es una clase abstracta de la que derivan dos implementaciones para detectar movimiento diferentes, una por diferencia de píxels y otra por calculo de flujo óptico, ambas técnicas típicas para detección de movimiento en un flujo de video. La diferencia de píxels consiste en restar los valores de cada píxel en imágenes consecutivas con formato en escala de grises, de modo, que aquellos píxels que se han desplazado producirán valores diferentes de cero. Concluiremos que ha habido movimiento en aquellos píxels en los que el valor absoluto de la diferencia supere un cierto umbral parametrizable. El cálculo de flujo óptico, por su parte, permite calcular no sólo que ha habido movimiento en un píxel sino que además calcula la magnitud y dirección del movimiento. Es un algoritmo mucho más complejo y computacionalmente más costoso. Se ha utilizado la implementación que encontramos en la librería *OpenCv*.

En la figura 4.8 se puede ver la interfaz gráfica del componente. En ella se aprecia cómo se ha detectado el movimiento causado por la mano, en este caso, usando flujo óptico.

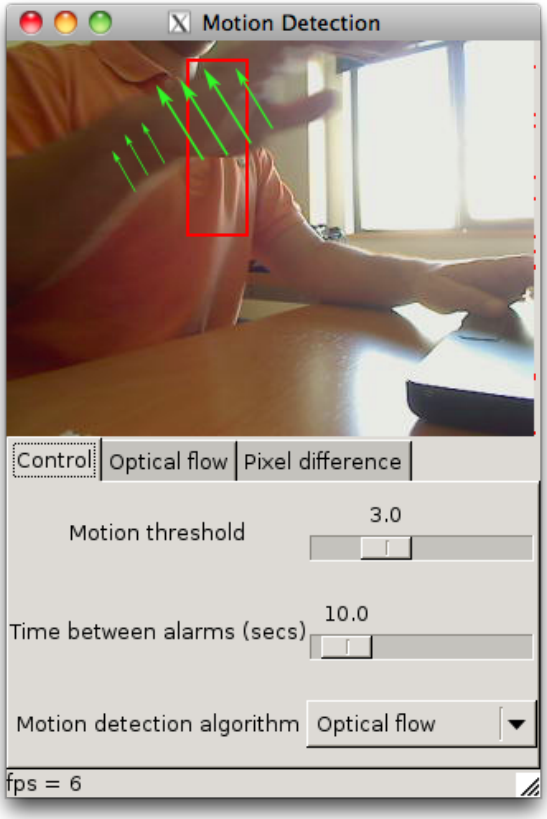


Figura 4.8: *Interfaz gráfica del componente* `motionDetection`

colorspacesmm

Procesar imágenes requiere manejar diferentes tipos de formatos. Unas veces trabajaremos en espacios de color RGB, donde la información de cada pixel se expresa en función de sus componentes de color básico (rojo, verde,azul), otras trabajaremos en espacios HSV, donde el pixel se expresa en función de tono, saturación y brillo, y otros simplemente en un espacio de escala de grises. Cada uno tiene características que lo hacen apropiado en función de la aplicación que tratemos.

Cuando desarrollamos código que puede manejar múltiples formatos, como por ejemplo un visualizador como `cameraView`, la parte encargada de decodificar termina siendo una maraña de `if` y `else` donde se procesa cada posible formato por separado. El ejemplo del visualizador lo aclara; tenemos que la ventana sobre la que se pinta la imagen sólo acepta RGB en el que cada componente se expresa con un natural de 8bits, así, para cada variante posible de RGB,HSV,YUV, tenemos una rama que traduce al RGB de 8 bits que se espera. Esto lleva a programas complicados de mantener y duplicación de código.

Para solucionar en cierta medida este problema, se ideó la librería `colorspaces` incluida en la version 4 de *jderobot*. Ésta ofrece un API con el que se puede traducir de modo eficiente entre un conjunto de espacios de color. Algunas de estas traducciones están altamente optimizadas usando diferentes técnicas. Su uso permite no replicar todo este código en cada esquema que requiere de transformaciones en espacios de

color, pero no soluciona el problema de tener que añadir condiciones para seleccionar qué formato tratar.

Con la adopción del software `OpenCv` en el desarrollo de nuestros algoritmos de visión, se empezaron a usar las funciones que su API proveía para movernos entre espacios de color. Su implementación está altamente optimizada y cubre gran parte de los formatos más habitualmente usados, pasando `colorspaces` a un segundo plano para aquellas aplicaciones que requerían transformaciones y optimizaciones muy concretas. De nuevo, `OpenCv` no soluciona el problema de las condiciones.

Así, con la idea de abordar este problema, surge la idea de crear `colorspacesmm`. Una implementación en C++ de un envoltorio para la clase base que se utiliza en `OpenCv` para manejar imágenes. Dicha clase es `cv::Mat` y no es más que una clase que representa una matriz n-dimensional de un tipo de dato especificado por el usuario (int, float,...). En el caso de imágenes dicha matriz es de dimensión $ancho*alto*numcanales$, donde ancho y alto representan el tamaño de la imagen y número de canales representa el número de componentes del formato concreto, por ejemplo, para RGB sería 3 y para escala de grises 1. Más allá de esta información `cv::Mat` no almacena nada relacionado con el formato que almacena, teniendo que usar un dato externo a la clase para representarlo. Por esta razón nos decidimos a crear la clase `colorspaces::Image` que es una especialización de `cv::Mat` a la que se añade información sobre el formato de los datos que almacena.

En la figura 4.9 se presenta un diagrama de clases con las clases en torno a `Image`. La primera en entrar en juego es `Format`, que define un formato mediante una serie de parámetros (nombre, tipo, id,...). Esta clase aplica el patrón de diseño *Singleton* modificado para permitir la creación de una sola instancia por formato que se comparten entre las imágenes. El resto de clases son especializaciones que implementan formatos específicos (RGB888, YUY2 y GRAY8), cada una de ellas definen una instancia de `Format` con los parámetros adecuados para ese formato.

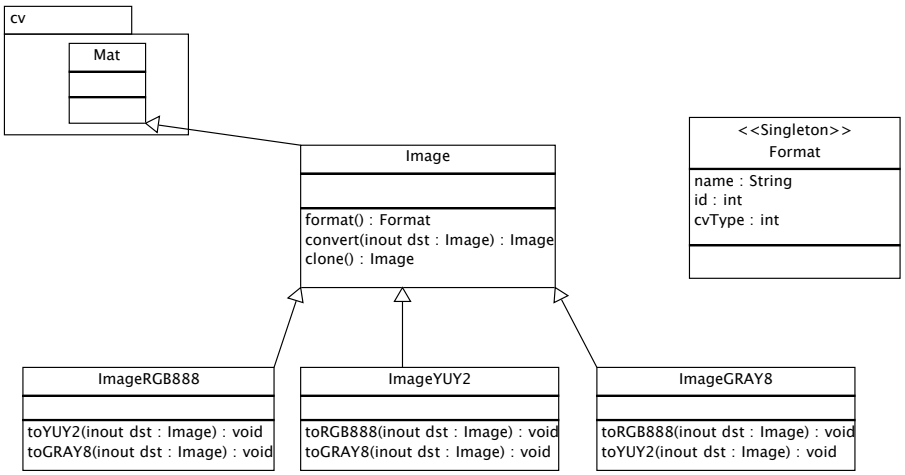


Figura 4.9: *Diagrama de clases de colorspacesmm*

Así, para manejar imágenes trabajaremos creando instancias de formatos concretos y bien utilizando las conversiones explícitas `img.toXXX`, bien haciendo *castings* a otra clase. Podemos ver un ejemplo de uso en el listado 4.7.

De este modo, podemos trabajar con imágenes sin conocer su formato y por tanto sin escribir código que requiera condiciones a la hora de manejar los diferentes formatos soportados, simplificando en gran medida la complejidad. Esta librería se ha utilizado en todos los componentes mencionados anteriormente.

```
1  ...
2  colorspaces::ImageRGB888 img1(320,240);
3  ...
4  colorspaces::ImageYUY2 img2(img1);//conversión RGB888 a YUY2
5  ...
6  ...
7  colorspaces::Image img(img1);//instancia Image independiente del formato
8  colorspaces::ImageGRAY8 img3(img1);//conversion RGB888 a GRAY8
9  ...
10 img3 = img2;//conversion YUY2 a GRAY8
11 ...
```

Listado 4.7: *Ejemplo de conversión entre formatos*

Gestión de la compilación

La gestión de la compilación del sistema se lleva a cabo con *GNU Autotools*, que son un conjunto de herramientas que permiten definir las dependencias, objetivos de compilación y reglas de distribución de las unidades de compilación. Dicho conjunto de herramientas son una constante entorno a multitud de proyectos de software libre. Su uso es bastante complejo, debido en parte a la grandísima colección de macros que se tiene que usar para definir cada acción. Sin embargo, una vez superado este primer obstáculo resulta un sistema muy flexible que permite infinidad de soluciones.

En la última década han surgido varias alternativas que parecen estar ganando puesto en la comunidad de software libre. Una de ellas es *cmake* que aporta un sistema portable a múltiples plataformas (autotools está basado en sistema Unix) y que utiliza un configuración bastante más simple. La elección de *GNU Autotools* frente a *cmake* se realizó meramente por motivos de continuidad, dado que la versión 4.3 de *jderobot* ya usaba dicho sistema. Aun así se han planteado dudas de si sustituir *GNU Autotools*, más cuando además *Orca* utiliza ya *cmake*.

Detallando muy por encima el sistema de compilación, tenemos que lleva a cabo las siguientes tareas:

Configuración del entorno : Se detectan las librerías de las que depende el proyecto y se declaran las opciones de compilación adecuadas. También en esta fase se seleccionan las rutas de instalación y los elementos a compilar.

Generación de reglas de compilación : Una vez obtenida la configuración para el proyecto, dependiente del sistema sobre el que se quiere compilar, se pasa a generar las reglas de compilación o *Makefiles*.

Compilación : Siguiendo las reglas generadas en el anterior apartado, se compila el sistema.

Instalación/Distribución : Dependiendo del objetivo, podemos bien instalar el sistema compilado o empaquetarlo para su distribución (paquetes tgz, paquetes debian,etc.).

Todo este sistema simplifica en gran medida el uso del software, ya que el usuario final que desea usarlo simplemente debe seguir una serie de pasos, comunes a todo proyecto software, que le permiten compilar e instalar el sistema sin requerirle conocer cada una de las dependencias o herramientas de compilación.

4.2.5 Aplicación carspeed 5

Para terminar con la descripción del proyecto *jderobot* 5 se presenta la aplicación *Carspeed*, que es uno de los proyectos implementados sobre la arquitectura junto con *ElderCare* [Marugán Alonso, 2010] y *Surveillance* [Calvo Palomino, 2010]. *Carpeed* se implementó inicialmente en la versión 4.3 [Hidalgo Blázquez, 2008] y el trabajo realizado en este proyecto ha sido portarlo a la nueva arquitectura software, aplicando las nuevas herramientas y refactorizando el código para poder usar sus algoritmos en otros trabajos. La implementación en C++ se ha rehecho desde cero debido a que la versión original está programada en C y altamente acoplada con el sistema *jderobot* 4.3. De este modo se ha podido reutilizar una parte mínima del código, cuidando en todo momento de seguir lo más fielmente dicha implementación original.

Carspeed es una aplicación capaz de calcular la velocidad de los coches que circulan por una carretera únicamente con imágenes obtenidas desde una cámara. Para ello se utilizan técnicas de rectificación, para calcular distancias reales a partir de las imágenes en perspectiva obtenidas con la cámara, y técnicas de algoritmos genéticos que permiten inferir la velocidad de los vehículos detectados a partir de un conjunto de velocidades hipótesis. El algoritmo completo de *Carspeed* se encuentra detallado en el proyecto fin de carrera que lo implementó [Hidalgo Blázquez, 2008] por primera vez, aunque aun así se van a detallar aquí aquellas partes más destacables.

El primero de los problemas a solventar es encontrar la relación, en términos de coordenadas, entre la imagen y la realidad. Una cámara aplica una transformación al proyectar la imagen captada sobre la superficie que la registra (sensor, película fotográfica). Una vez aplicada dicha transformación, ya no somos capaces de calcular distancias reales sobre la imagen proyectada dado que la proyección en 2D elimina la profundidad 3D. Para poder llevar a cabo la medición debemos deshacer la transformación. En el caso que nos ocupa, medir velocidades, requiere conocer distancias reales, al menos sobre el plano carretera. Y esto es justo lo que vamos a hacer, vamos a establecer una correspondencia punto por punto, entre la carretera que vemos en la imagen y la carretera real, también denominada homografía. Dicha correspondencia se calcula con un conjunto de operaciones matemáticas descritas en [Kachach, 2008]. La figura 4.10 muestra el efecto. La rectificación hace una correspondencia entre los puntos marcados (A,B,C,D) en la imagen real y en la rectificada. Una vez calculamos la relación entre imagen y realidad, ya somos capaces de medir distancias reales, y con ello podremos calcular velocidades. Este cálculo se realiza durante la inicialización del algoritmo y parte de datos que el usuario debe suministrar como parte de la configuración (puntos, largo y ancho).

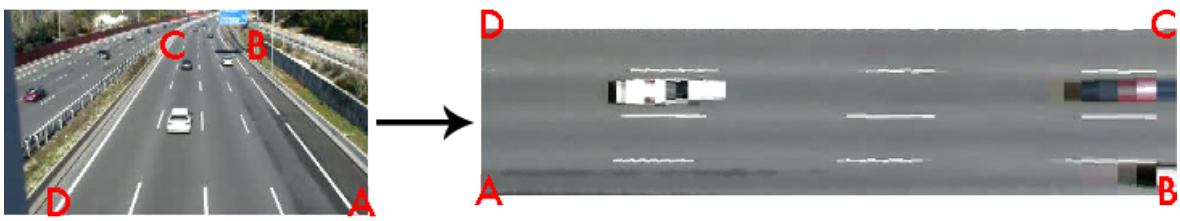


Figura 4.10: *Efecto de la rectificación de una imagen*

Lo primero es detectar el movimiento en la escena, que suponemos producido por los coches. Para ello se usa el mismo mecanismo explicado en el apartado 4.2.4, salvo que esta vez la diferencia se calcula sobre un fondo aprendido. De este modo, calculamos un fondo que suponemos es la carretera al que restamos la imagen actual. Sobre aquellas

áreas en las que aparece un coche, que por estar en movimiento no aparecen en el fondo aprendido, resultarán diferencias con valor absoluto mayor que cero. Usando un umbral, reduciremos a un valor nulo en aquellos píxeles inmóviles y a un valor diferente de cero en aquellos en los que se ha detectado movimiento. El mecanismo es muy simple y computacionalmente muy barato. En la figura 4.11 podemos ver un detalle del fondo aprendido y del resultado obtenido una vez se ha calculado la diferencia. En la izquierda vemos la imagen real, y sobre la diferencia contra el fondo podemos ver una mancha blanca que representa el movimiento causado por el vehículo. Esta será la información que reciba el algoritmo.

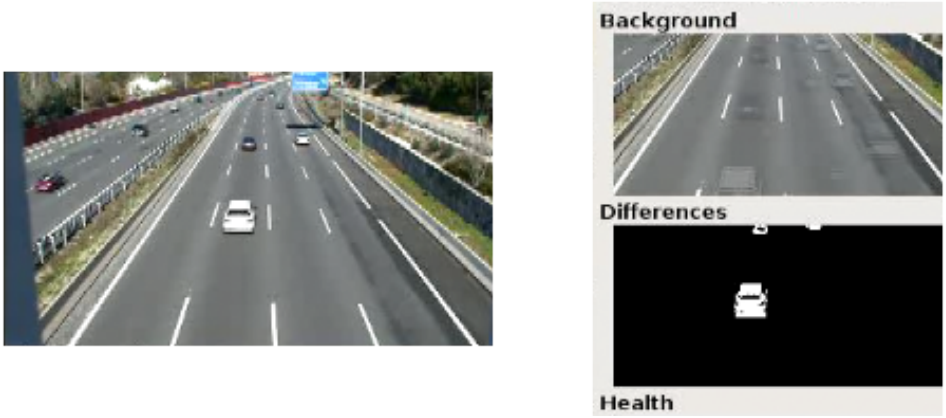


Figura 4.11: *Detección de movimiento en carspeed*

El algoritmo seguido consiste en detectar los vehículos a medida que aparecen en escena, elaborar un conjunto de hipótesis que contemplen las posibles velocidades a las que el vehículo podría estar moviéndose y a medida que avanza por la escena, corroborar o desmentir esas hipótesis hasta llegar a una conclusión: la velocidad actual del vehículo en ese tramo. En términos de algoritmos genéticos tenemos que cada vehículo detectado, genera una raza. Dicha raza tiene un número de individuos, cada uno de ellos representando una posible velocidad. En cada iteración se calcula la salud de los individuos, que será alta si representan la velocidad real del vehículo o baja en caso contrario. A medida que la salud de un individuo disminuye este muere. Así, al final sólo quedan aquellos individuos *fuertes* que resultan ser aquellos que representan de manera más exacta la velocidad del vehículo.

El punto clave del algoritmo es la función salud o función encargada de calcular como de bueno es un individuo. Esta función debe ser lo suficientemente discriminante para que todos los individuos que no cumplan las condiciones marcadas sean descartados. En [Hidalgo Blázquez, 2008] se describen diferentes soluciones para esta función, habiéndose llegado a la función salud *Cuerpo-Falda* como la más óptima. Aplicando esta función sobre la información de movimiento que obtenemos de la sucesión de imágenes, podemos calcular como de posible es que en un punto concreto haya un vehículo.

Para la aplicación de cada una de las acciones descritas, se divide la carretera en dos zonas. Una primera de detección, donde se crean la nuevas razas cuando se detectan un vehículos, y una segunda zona de seguimiento, donde se hacen evolucionar las razas aplicando la función salud. Al final de la zona de seguimiento, se concluye que un vehículo, representado por una raza, se mueve a la velocidad de el individuo que mejor salud promedio tiene.

Uniendo todos estos elementos, tenemos los siguientes pasos que se ejecutan de manera iterativa:

1. Detección de nuevos vehículos que generan nuevas razas

2. Actualización de la posición de los individuos de las razas existentes
3. Cálculo de la salud de los individuos de cada raza
4. Descarte de aquellos individuos con una salud inferior al umbral marcado
5. Conclusión de la velocidad de aquellas razas que han llegado al final de la zona de seguimiento (individuo con mejor salud promedia)

Desde el punto un punto de vista de diseño, la aplicación se enmarca en un componente llamado `carspeed` que utiliza las imágenes obtenidas por el interfaz `camera` de otro componente que las suministre, en este caso concreto una instancia de `cameraServer`. Dentro del componente `carspeed` se han aplicado los patrones *MVC* y *Algoritmo Iterativo*. El primero, para gestionar todo lo relacionado con la interfaz gráfica de la aplicación. El modelo contiene todos los datos relacionados con el algoritmo, la vista es una implementación *GTK+* de la interfaz gráfica y el controlador gestiona la lógica de la aplicación y la comunicación entre las partes. Su aplicación es idéntica a las vistas en apartados anteriores. El patrón de diseño *Algoritmo Iterativo* se ha aplicado para la implementación del algoritmo, que en este caso tiene una ejecución iterativa, encajando perfectamente en dicho patrón. La figura 4.12 muestra el diseño descrito en UML.

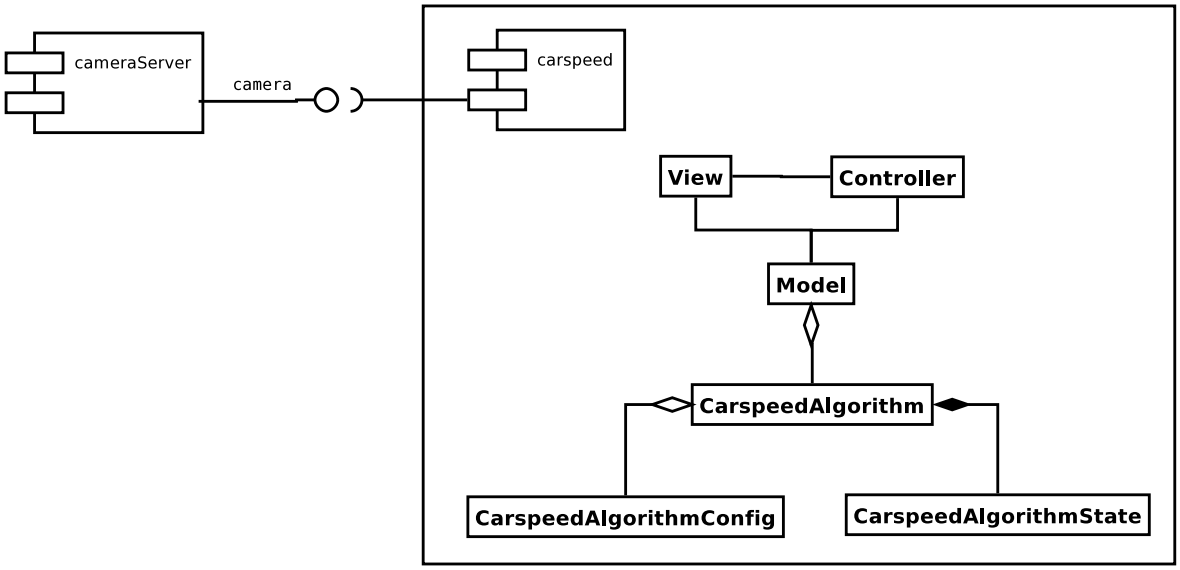


Figura 4.12: *Diseño de Carspeed*

Como aporte, además de haber re-implementado todo el sistema en C++, se ha pasado a usar *OpenCv* en lugar de las librerías IPP de Intel. De este modo, podemos decir que *Carspeed* se ha desarrollado completamente con software libre.

En la figura 4.13 vemos una ejecución típica de *Carspeed*. El interfaz gráfico consta de tres ventanas, de las cuales sólo la ventana 1 es visible siempre. Las otras dos permiten obtener información que se usa para depurar la aplicación. La ventana número 2 muestra los cálculos realizados para detectar los coches en movimiento, y la ventana número 3 muestra la carretera una vez se ha deshecho la proyección. Sobre la ventana 1 podemos ver como se ha detectado la velocidad en varios vehículos.

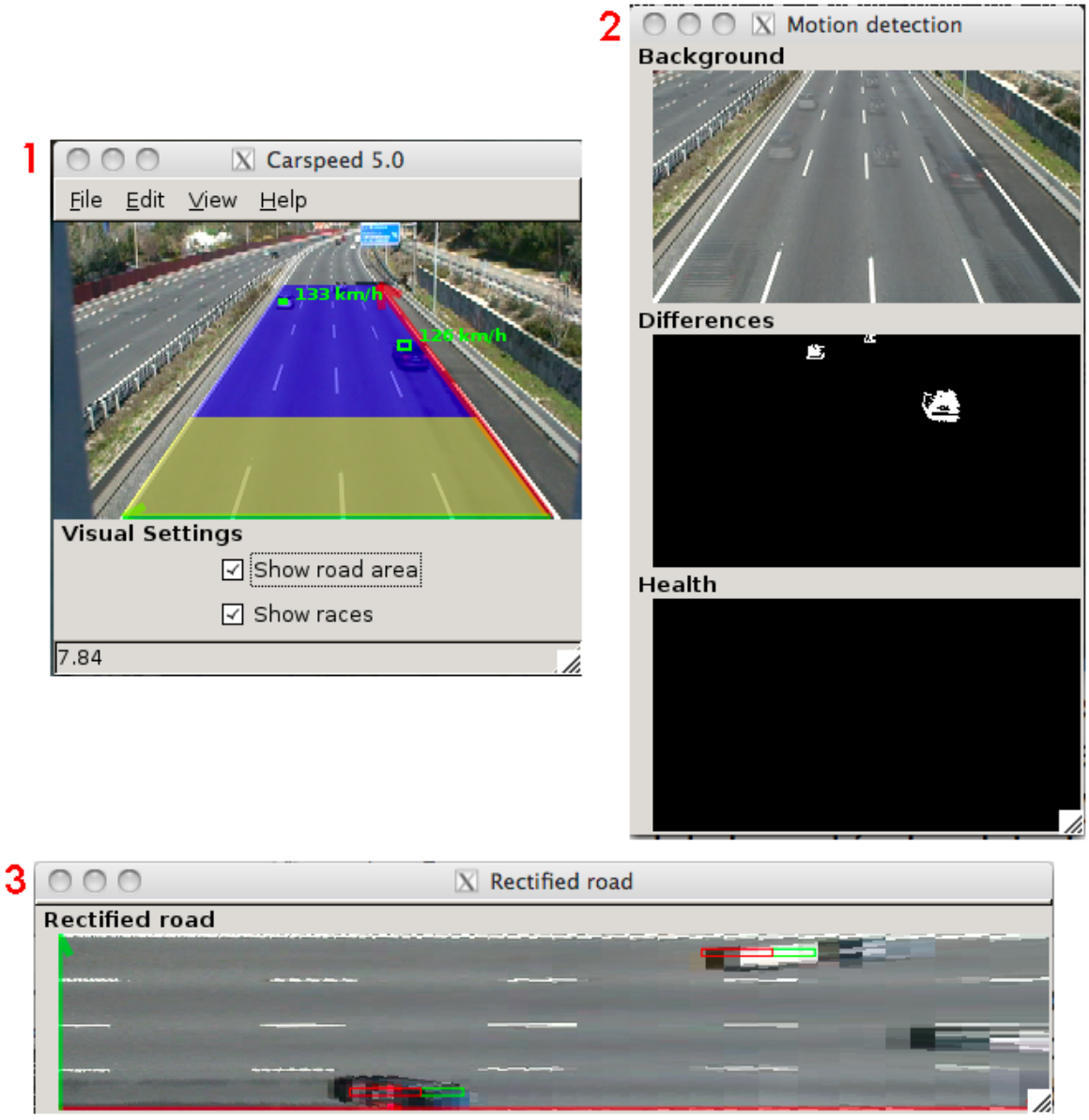


Figura 4.13: *Ejecución de Carspeed*

Capítulo 5

Conclusiones y trabajos futuros

En este último capítulo se muestran las conclusiones extraídas del desarrollo de este proyecto, comentando los puntos más relevantes que han permitido el logro del trabajo, y las principales dificultades encontradas. También hablamos de las líneas de trabajos futuros que pueden derivarse de este proyecto para su mejora y ampliación.

5.1 Conclusiones

La conclusión principal que extraemos del trabajo descrito en este proyecto es que se ha iniciado con éxito una nueva línea de desarrollo del entorno *jderobot* que cumple los objetivos y requisitos que marcamos en el capítulo 2. El sistema es flexible ya que no requiere ninguna organización concreta de sus elementos ni de cómo deben colaborar entre sí, aporta mecanismos para distribuir la carga computacional en diferentes nodos, puede ser programado en los lenguajes más populares actualmente y sigue las técnicas de la ingeniería basada en componentes con hincapié en la facilidad de reutilización de los elementos desarrollados. El diseño del sistema ha aunado tanto el bagaje aportado por todas las experiencias de las versiones anteriores de *jderobot*, como las ideas provenientes de las últimas tendencias para entornos de desarrollo para sistemas robóticos en la comunidad científica internacional, de manera que hemos reorganizado y refactorizado elementos ya existentes siguiendo los nuevos diseños. *jderobot* 5 no es sólo una implementación, es un conjunto de librerías, de reglas de diseño, de herramientas y de documentación, que lo enmarcan dentro de un proyecto de software libre con una comunidad de usuarios y desarrolladores activa. Todo el contenido del proyecto es accesible bajo licencia GPL a través del sitio web *http://jderobot.org*.

Se ha conseguido un sistema flexible capaz de adoptar múltiples organizaciones, tanto desde el punto de vista interno de sus componentes, como para la organización y comunicación de estos. Un componente puede implementarse con una o múltiples hebras, puede usar la librería gráfica que mejor le convenga sin necesidad de soluciones *ad-hoc* como en la versión 4.3, y puede aplicar los patrones de diseño que faciliten el desarrollo de la tarea que tenga encomendada. Simplemente debe mostrar al exterior el interfaz por el cual comunicarse con él, interfaz que representa el contrato por el cual el componente exporta su funcionalidad. Los interfaces explícitos facilitan la tarea de reutilización ya que no en necesario conocer nada de la implementación interna del componente. A nivel de organización de componentes, de nuevo, flexibilidad absoluta, pudiendo hacer que unos componentes dependan de otros, que unos modulen a los otros o cualquiera de las posibles organizaciones que se nos ocurran. Este objetivo se ha conseguido gracias al uso del *middleware* Ice y su lenguaje de descripción de interfaces `slice`.

El sistema *jderobot* 5 permite la distribución de sus componentes en una red de

nodos heterogéneos. Esto abre la puerta a nuevas aplicaciones, bien porque requieran mayor capacidad de cómputo, que ahora puede ser repartida entre varios nodos, bien porque el medio natural de dicha aplicación sea un entorno distribuido. Este objetivo se ha conseguido empleando el *middleware* Ice, de modo que el esfuerzo de implementar estos mecanismos ha sido nulo.

El abanico de lenguajes con los que podemos programar componentes para *jderobot* 5 es muy amplio, contando con todos los mas populares del momento como C++, Python, Java o PHP. Cada componente de *jderobot* podrá ser programado en el lenguaje que mejor se adapte al contexto y/o a las capacidades y gustos del desarrollador. De nuevo esta característica la aporta el *middleware* Ice, con el consiguiente ahorro de esfuerzo.

Para el diseño del sistema hemos seguido las guías de la ingeniería basada en componentes, sin que ello entre en contradicción con el objetivo de *Flexibilidad*. Simplemente se prefiere a la hora de empaquetar funcionalidad, hacerlo en componentes autocontenidos con un objetivo bien definido que facilitarán su uso y reutilización. Para ello se han definido algunos interfaces para el acceso al hardware robótico, siendo otros reutilizados de la plataforma Orca que tiene una colección muy rica.

En términos de eficiencia se considera que el sistema cumple los objetivos marcados, ya que las limitaciones las fija el medio de transporte (red TPC/IP) y no tanto la eficiencia del código. Aun así, ha resultado que los mecanismos de comunicación que implementa Ice tienen un coste computacional mínimo. La aplicación *Eldercare* [Marugán Alonso, 2010] portada al sistema *jderobot* 5 ha sido capaz de cumplir sus objetivos, siendo esta una aplicación que requiere de unas características en términos de rendimiento medio-altas, dado que en su aplicación mas típica habitualmente requiere el acceso a cuatro flujos de video en tiempo real con tasas cercanas a los 20 fotogramas por segundo. Para ello se han probado los diferentes mecanismos de comunicación disponibles y se ha estudiado cómo aplicarlos a nuestras aplicaciones.

Todo el software desarrollado funciona tanto para plataformas GNU/Linux como para dispositivos móviles Android, sobre los que ejecuta parte de la aplicación de video vigilancia [Calvo Palomino, 2010] desarrollada sobre *jderobot* 5. La interoperabilidad entre ambas plataformas ha sido perfecta gracias, de nuevo, a las capacidades multiplataforma de Ice y su versión para dispositivo móviles Android.

Además de la integración de las diferentes herramientas, se ha desarrollado una serie de componentes básicos que han permitido explorar las capacidades del sistema y guiar algunas de las ideas de diseño que se pretendían implementar. Con ellos se han validado los mecanismos de comunicación que aporta Ice, y se han desarrollado algunas pruebas y aplicaciones sencillas.

Para terminar con el repaso de los objetivos cumplidos con este proyecto, se ha portado la aplicación Carspeed [Hidalgo Blázquez, 2008] de dificultad media y requisitos de eficiencia medio-altos, para terminar de validar la arquitectura software. El resultado ha sido la primera aplicación funcionando sobre *jderobot* 5, utilizando sus mecanismos e ideas, con la que se ha verificado que el *middleware* de Ice es capaz de trasmitir imágenes al ritmo requerido por la aplicación (>15fps). También se ha comprobado que la aplicación sigue funcionando aun colocando el componente emisor de imágenes en otra máquina y haciendo toda la transmisión por una red *fast ethernet*.

Las principales dificultades halladas durante el desarrollo de este proyecto las encontramos durante el desarrollo de la versión 4.4, con la que intentamos alcanzar algunos de los objetivos marcados usando como base la versión 4.3. La razón es que intentamos implementar mecanismos complejos como la interoperabilidad multi-lenguaje entre esquemas utilizando herramientas que no están diseñadas para ello, de modo que obtuvimos soluciones parciales y poco escalables. Por ello se decidió dar un giro y rediseñar el sistema para utilizar herramientas que nos permitiesen alcanzar lo que nos

habíamos propuesto.

Otra de las dificultades, común en este tipo de sistemas, es la verificación de los componentes desarrollados. Aquellos con un funcionamiento simple pueden ser verificados de manera sencilla, pero cuando empezamos a tener diferentes flujos de ejecución, la cosa se complica, debiendo desarrollar técnicas que nos permitan probar cada parte por separado y en conjunto, de modo que podamos sacar alguna conclusión sobre si funcionan de manera correcta. Es complicado aplicar mecanismos genéricos, y cada problema requiere de un diseño minucioso de las pruebas a realizar. En este contexto, la reutilización de software probado por una amplia comunidad de usuarios, o la posibilidad de que el software que desarrollemos lo pruebe un grupo grande de usuarios, facilita en gran medida la tarea de verificación.

5.2 Trabajos futuros

El proyecto descrito en este trabajo fin de máster pretende ser el entorno de desarrollo a utilizar como base para los siguientes trabajos de investigación del grupo de robótica de la URJC y para la docencia de asignaturas impartidas por el grupo. Es por ello que se debe seguir desarrollando, ampliando sus herramientas, componentes y documentación, para llevarlo hasta un estado estable. Algunas de las tareas concretas para enriquecer *jderobot* 5 son:

Añadir más *drivers* y portar los ya existentes : De esta manera enriqueceremos el soporte de hardware y añadiremos soporte para aquel que ya controlamos. Debemos prestar atención a no re-implementar cosas que ya existan, bien en Player, bien en Orca. Despierta especial interés la actualización de los *drivers* para acceso a los simuladores `Stage` y `Gazebo`, para los que *jderobot* 4.3 tiene soporte para versiones bastante antiguas.

Añadir más algoritmos y portar los ya existentes : Al igual que con los *drivers*, los trabajos más interesantes realizados hasta la fecha deben añadirse a la nueva versión, y continuar desarrollando otros nuevos.

Probar otras librería gráficas : Hasta el momento sólo hemos usado *GTK+* en *jderobot* 5, así que resultará interesante implementar componentes que utilicen otras librerías (QT, OpenGL,etc.) para poder compararlas.

Acercarnos mas a Orca : Orca se ha utilizado como base para *jderobot* 5, aunque en algunos casos simplemente se ha reutilizado parte de su código en vez de utilizarlo como librería y enlazarlo en nuestra plataforma. Por ello, debemos acercarnos más y usar Orca como una librería, apoyando nuestro desarrollo sobre ella.

Poner el sistema en producción : Esto significa que debemos hacer un esfuerzo para documentar todo el sistema y generar los paquetes instalables una vez tengamos una versión estable (por el momento para la distribución GNU/Linux Debian). De esta manera el sistema podrá empezar a usarse en las asignaturas que imparte el grupo de robótica.

Al igual que *jderobot* 5 ha utilizado todas las herramientas necesarias obtenidas de la comunidad de software libre, se impone una obligación moral de devolver algo a la misma. Por ello se pretende revisar y refactorizar donde sea necesario, para hacer que toda la funcionalidad de calidad que el grupo de robótica ha implementado en lo últimos años pueda ser reutilizada por la comunidad.

Y por último, comentar el interés existente por parte de algunas empresas en varios de los proyectos desarrollados por el grupo (ElderCare y Carspeed). Proyectos apoyados

en *jderobot* 5 como entorno de desarrollo y que permitirán invertir mas esfuerzo y recursos en él.

Bibliografía

[Agre and Chapman, 1987] Agre, P. E. and Chapman, D. (1987). Pengi: an implementation of a theory of activity. In *Proceedings of 6th AAAI National Conference on Artificial Intelligence*, pages 268–272, Seattle, WA. Morgan Kaufmann.

[Agre and Chapman, 1990] Agre, P. E. and Chapman, D. (1990). What are plans for? In Maes, P., editor, *Designing Autonomous Agents: theory and practice from Biology to Engineering and Back*, pages 17–34. MIT Press.

[Arkin, 1995] Arkin, R. C. (1995). Reactive robotic systems. In Arbib, M., editor, *Handbook of the brain theory and neural networks*, pages 793–796. MIT Press.

[Arkin, 1998] Arkin, R. C. (1998). *Behavior based robotics*. MIT Press.

[Beazley, 1996] Beazley, D. M. (1996). Swig: an easy to use tool for integrating scripting languages with c and c++. In *TCLTK'96: Proceedings of the 4th conference on USENIX Tcl/Tk Workshop, 1996*, pages 15–15, Berkeley, CA, USA. USENIX Association.

[Beazley and Fulton, 2008] Beazley, D. M. and Fulton, W. (2008). Simplified wrapper and interface generator. http://www.swig.org/.

[Brooks et al., 2007] Brooks, A., Kaupp, T., Makarenko, A., Williams, S., and Orebäck, A. (2007). Orca: A component model and repository. In Brugali, D., editor, *Software Engineering for Experimental Robotics*, volume 30 of *Springer Tracts in Advanced Robotics*. Springer - Verlag, Berlin / Heidelberg.

[Brooks, 1986] Brooks, R. A. (1986). A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23.

[Brooks, 1991] Brooks, R. A. (1991). Intelligence without reason. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 569–595.

[Calvo Palomino, 2010] Calvo Palomino, R. (2010). Sistema de video vigilancia distribuido y android. Trabajo fin de máster, Universidad Rey Juan Carlos.

[Cañas and Matellán, 2002] Cañas, J. M. and Matellán, V. (2002). Dynamic schema hierarchies for an autonomous robot. In *IBERAMIA 2002: Proceedings of the 8th Ibero-American Conference on AI*, pages 903–912, London, UK. Springer-Verlag.

[Capek, 1923] Capek, K. (1923). *RUR: Rossum's universal robots*.

[Collett et al., 2005] Collett, T. H., MacDonald, B. A., and Gerkey, B. P. (2005). Player 2.0: Toward a practical robot programming framework. In *Proc. of the Australasian Conf. on Robotics and Automation (ACRA)*, Sydney, Australia.

[Coste-Maniére and Simmons, 2000] Coste-Maniére, È. and Simmons, R. G. (2000). Architecture, the backbone of robotic systems. In *Proceedings of the 2000 IEEE International Conference on Robotics and Automation*, pages 67–72, San Francisco, CA (USA).

[Firby, 1987] Firby, R. J. (1987). An investigation into reactive planning in complex domains. In *Proceedings of the 6th AAAI National Conference on Artificial Intelligence*, pages 202–206, Seattle, WA.

[García Pérez, 2004] García Pérez, L. (2004). *Navegación autónoma de robots en agricultura: un modelo de agentes*. PhD thesis, Universidad Complutense de Madrid.

[Gerkey et al., 2003] Gerkey, B. P., Vaughan, R. T., and Howard, A. (2003). The Player/Stage project: tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics ICAR'2003*, pages 317–323, Coimbra (Portugal).

[Henning, 2004] Henning, M. (2004). A new approach to object-oriented middleware. *Internet Computing, IEEE*, 8(1):66–75.

[Henning and Spruiell, 2010] Henning, M. and Spruiell, M. (2010). *Distributed Programming with Ice*. ZeroC, http://www.zeroc.com/doc/Ice-3.4.1/manual/, revision 3.4 edition.

[Hidalgo Blázquez, 2008] Hidalgo Blázquez, V. M. (2008). Detección visual de la velocidad en vehiculos sobre jdec. Technical report, Universidad Rey Juan Carlos.

[Kachach, 2008] Kachach, R. (2008). Calibración automática de cámaras en la plataforma jdec. Technical report, Universidad Rey Juan Carlos.

[Konolige and Myers, 1998] Konolige, K. and Myers, K. L. (1998). The Saphira architecture for autonomous mobile robots. In Kortenkamp, D., Bonasso, R. P., and Murphy, R., editors, *Artificial Intelligence and Mobile Robots: case studies of successful robot systems*, pages 211–242. MIT Press, AAAI Press. ISBN: 0-262-61137-6.

[Krasner and Pope, 1988] Krasner, G. E. and Pope, S. T. (1988). A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49.

[Lobato Bravo, 2005] Lobato Bravo, D. (2005). jde+: Una plataforma de desarrollo para aplicaciones robóticas. Technical report, Universidad Rey Juan Carlos.

[Makarenko et al., 2006] Makarenko, A., Brooks, A., and Kaupp, T. (2006). Orca: Components for robotics. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 163–168.

[Makarenko et al., 2007] Makarenko, A., Brooks, A., and Kaupp, T. (2007). On the benefits of making robotic software frameworks thin. In Prassler, E., Nilsson, K., and Shakhimardanov, A., editors, *IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'07) Workshop on Measures and Procedures for the Evaluation of Robot Architectures and Middleware*.

[Marugán Alonso, 2009] Marugán Alonso, S. (2009). People 3d tracking using volumetric primitives. Technical report, Universidad Rey Juan Carlos.

[Marugán Alonso, 2010] Marugán Alonso, S. (2010). Eldercare en la plataforma jderobot 5.0. Trabajo fin de máster, Universidad Rey Juan Carlos.

[Mataric, 2002] Mataric, M. J. (2002). Situated robotics. In Nadel, L., editor, *Encyclopedia of Cognitive Science*. Nature Publishers Group, London UK.

[Murphy, 1998] Murphy, R. R. (1998). Dempster-shafer theory for sensor fusion in autonomous mobile robots. *IEEE Transactions on Robotics and Automation*, 14(2):197–206.

[Murphy, 2000] Murphy, R. R. (2000). *Introduction to AI robotics*. MIT Press.

[Nilsson, 1969] Nilsson, N. (1969). A mobile automaton: an application of artificial intelligence techniques. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence IJCAI*, pages 509–520, Washington, (USA).

[Payton, 1990] Payton, D. W. (1990). Internalized plans: a representation for action resources. *Robotics and Autonomous Systems*, 6:89–103.

[Plaza, 2003] Plaza, J. M. C. (2003). *Jerarquía Dinámica de Esquemas para la generación de comportamiento autónomo*. PhD thesis, Universidad Politécnica de Madrid.

[Quigley et al., 2009] Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R., and Ng, A. Y. (2009). Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*.

[Ruiz-Ayúcar Vázquez, 2007] Ruiz-Ayúcar Vázquez, J. (2007). jdeneo.c: Una plataforma para desarrollo de aplicaciones robóticas. Technical report, Universidad Rey Juan Carlos.

[Schmidt, 1999] Schmidt, D. C. (1999). Why software reuse has failed and how to make it work for you. *C++ Report*.

[Schneider Fontán, 1996] Schneider Fontán, M. (1996). *Planificación basada en percepción activa para la navegación de un robot móvil*. PhD thesis, Universidad Complutense de Madrid.

[Serradilla, 1997] Serradilla, F. (1997). *Arquitectura cognitiva basada en el gradiente sensorial y su aplicación a la robótica móvil*. PhD thesis, Universidad Politécnica Madrid.

[Simmons et al., 1997] Simmons, R., Goodwin, R., Zita Haigh, K., Koening, S., and O'Sullivan, J. (1997). A layered architecture for office delivery robots. In *Proceedings of the ACM International Conference on Autonomous Agents*, pages 245–252, Marina del Rey (USA).

[Simmons, 1994] Simmons, R. G. (1994). Structured control for autonomous robots. *IEEE Journal of Robotics and Automation*, 10(1):34–43.

[Stentz et al., 2002] Stentz, A., Dima, C., Wellington, C., Herman, H., and Stager, D. (2002). A system for semi-autonomous tractor operations. *Autonomous Robots*, 13:87–104.

[The GStreamer team, 2010] The GStreamer team (2010). Gstreamer open source multimedia framework.

[Turing, 1950] Turing, A. M. (1950). Computer machinery and intelligence. *Mind*, 59(236):433–460.

[Vaughan et al., 2003] Vaughan, R. T., Gerkey, B. P., and Howard, A. (2003). On device abstractions for portable, reusable robot code. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*, volume 3, pages 2121–2127.

[Vega Pérez, 2008] Vega Pérez, J. (2008). Navegación y autolocalización sobre un robot guia. Technical report, Universidad Rey Juan Carlos.