Programación de robots móviles

José María Cañas Plaza
Universidad Rey Juan Carlos

2 de agosto de 2004

Resumen

El comportamiento de un robot autónomo viene determinado por el programa que gobierna sus actuaciones. La creación de programas para robots debe cumplir con ciertos requisitos específicos frente a la programación en otros entornos más tradicionales, como el ordenador personal. En este artículo planteamos que el software de los robots se estructura actualmente en tres niveles: sistema operativo, plataforma de desarrollo y aplicaciones concretas. En los últimos años, han surgido con fuerza plataformas de desarrollo con la idea de facilitar la construcción incremental de estas aplicaciones robóticas. Más allá del acceso básico a los sensores y actuadores, las plataformas suelen proporcionar un modelo para la organización del código y bibliotecas con funcionalidades comunes. En este artículo se identifican y analizan esos tres niveles, describiendo desde esta perspectiva los entornos de desarrollo de los robots reales más extendidos, y las tendencias más recientes.

Índice

1. Introducción

El principal objetivo de la robótica es la construcción de máquinas capaces de realizar tareas con la flexibilidad, la robustez y la eficiencia que exhiben los humanos. Los robots son potencialmente útiles en escenarios peligrosos para el ser humano, aburridos, sucios o difíciles. En este sentido los brazos robóticos que se emplean en las fábricas de coches para soldar y pintar, los robots móviles que se envían a Marte, o los que se utilizan para limpiar centrales nucleares son varios ejemplos de aplicaciones reales en las cuales se utilizan robots hoy día.

Hay muchos tipos de robots: con ruedas, con patas, brazos manipuladores, con orugas, de forma humanoide, de forma cilíndrica, etc. Sin embargo, la morfología no es una característica esencial, lo que identifica a cualquier robot es que combina en la misma plataforma a sensores, actuadores y procesadores. Los *sensores* miden alguna característica del entorno o propia (e.g. cámaras, sensores de obstáculos, etc.). Los *actuadores* permiten al robot hacer algo, llevar a cabo alguna acción o simplemente desplazarse (e.g. los motores). Los *procesadores* hacen los cómputos necesarios y realizan el enlace lógico entre sensores y actuadores, materializando el comportamiento del robot en el entorno en el cual se encuentra inmerso.

Generar comportamiento es programar

La existencia de robots que realicen autónomamente tareas de modo eficiente depende fundamentalmente de su construcción mecánica y de su programación. *Una vez construido el cuerpo mecánico del robot, conseguir que realice una tarea se convierte en la práctica en un problema de programación.* La generación de comportamiento en un robot consiste entonces en escribir el programa que al ejecutarse

en el robot *causa* ese comportamiento cuando éste se encuentra en cierto entorno. La autonomía y la "inteligencia" residen en ese programa. Por ejemplo, en los robots móviles el comportamiento principal es su movimiento. Los programas que se ejecutan en el robot determinan cómo se mueve éste por el entorno, reaccionando ante obstáculos percibidos por los sensores, acercándose a algún destino, etc. y para ello tienen que enviar continuamente las órdenes pertinentes a los motores.

El mercado de los robots y la autonomía

Muchos de los robots que se venden en la actualidad se compran como productos cerrados, programados por el fabricante e inalterables, por ejemplo la aspiradora robótica Roomba de iRobot[1] o el perrito Aibo[2] de Sony en sus inicios[3]. En contraste, otra parte relevante del mercado son los robots programables. Los principales clientes de estos robots son los centros de investigación, que normalmente están interesados en programarlos ellos mismos. En estos casos, el fabricante vende los robots con un software que permite su programación por el usuario.

La tendencia actual del mercado de robots es de crecimiento real: cada vez son más los productos robóticos que se venden y más frecuentes los movimientos empresariales alrededor de la tecnología robótica. Por ejemplo, recientemente Sony compró la licencia del software de visión a la pequeña empresa Evolution Robotics[4], para usarla en sus robots. Los robots de LEGO como juguetes imaginativos y los perritos Aibo vendidos como mascotas son éxitos de ventas significativos, con centenares de miles de unidades cada uno.

Aunque está en vías de consolidación, en términos generales el mercado de los robots móviles está inmaduro y todavía no se ha abierto al gran público. Esto se debe en parte a la fragilidad de los resultados (sobre todo comparados con las expectativas) y las limitaciones en autonomía que tienen los robots conseguidos hasta la fecha. De hecho, el grado de autonomía y de fiabilidad que seamos capaces de conseguir en los robots será una cuestión determinante en la evolución futura de ese mercado. El crecimiento en autonomía abre la puerta a nuevas aplicaciones comerciales robóticas, como los robots de servicio y los de entretenimiento. Avances recientes en esta línea son los ya mencionados de la aspiradora robótica Roomba y la mascota Aibo.

Sin embargo la autonomía es difícil, y salvo casos contados, el desarrollo de aplicaciones con robots sigue siendo materia de investigación. Al día de hoy, los proveedores de robots hacen negocio vendiendo los componentes hardware (pla-

[1] `http://www.irobot.com`

[2] `http://www.aibo-europe.com`

[3] En el verano de 2002 Sony hizo pública la interfaz de programación de sus perritos, Open-R, en parte forzados por un *hacker* que había conseguido desvelar algunas de sus interfaces

[4] `http://www.evolution.com/`

taformas físicas, motores, sensores, etc.) y con algunas soluciones a medida para clientes específicos. También venden el software de desarrollo, más o menos completo, e incluso algunas aplicaciones de ejemplo sencillas y/o más maduras, como la construcción de mapas o el seguimiento de personas.

Este artículo está organizado en seis partes contando con esta introducción. En la segunda sección se reseñan las peculiaridades que tiene la programación de robots y se presentan los tres niveles identificados en ese software, la evolución histórica que ha desembocado en ellos y las tendencias actuales. En la tercera sección se describe el nivel más bajo: el hardware que conforma típicamente un robot y los sistemas operativos que permiten acceder a él, repasando dos ejemplos concretos (ROBIOS y Linux). En el cuarto apartado se detallan las características principales de las plataformas de desarrollo que han aparecido en los últimos años y se analizan varios ejemplos particulares. A modo de ilustración, en la quinta sección se describen con cierto detalle los entornos de programación de cuatro robots comerciales muy difundidos (LEGO, Aibo, Pioneer y B21) enmarcándolos en los tres niveles mencionados. Finalizamos con las conclusiones principales de este trabajo.

2. Software de robots

En el manejo de robots conviene diferenciar entre el usuario final y el programador, pues el robot ofrece interfaces muy distintas a uno y a otro. El usuario es quien aprovecha las aplicaciones que ha desarrollado el programador. Para él la interfaz de uso suele ser extremadamente sencilla, máxime si el usuario es el público en general. Por ejemplo, la interfaz de uso de la aspiradora Roomba es un simple botón. Por el contrario, la interfaz de programación requiere de ciertos conocimientos por parte del desarrollador, y su objetivo es permitir la creación de programas para el robot. A lo largo de este artículo se analizan las diferentes posibilidades de programación de los robots móviles que se venden en la actualidad.

En los últimos años los avances en aplicaciones de robots móviles autónomos han sido significativos. Hoy en día, hay resultados fiables en aplicaciones como la construcción de mapas, la navegación automática, la localización, la detección de caras o el procesamiento de visión estéreo. Por ejemplo, en algunas fábricas hay robots que se mueven solos de un punto a otro remoto sin chocar con ningún obstáculo, transportando piezas pesadas. La funcionalidad de los prototipos de investigación construidos es cada vez mayor: guías de museos, cuidadores de ancianos, etc. No hay magia: detrás de esas aplicaciones hay programas que determinan el comportamiento de esos robots, programas que han escrito los diseñadores y que consiguen que el robot se mueva como lo hace.

A la hora de crear esas aplicaciones, una primera separación distingue a la

programación automática de la programación manual [BM03]. En la programación automática se engloban los sistemas con aprendizaje y la programación basada en demostración. Este enfoque es posible en brazos industriales robotizados donde la ejecución deseada es el recorrido por una secuencia de puntos, los cuales se pueden introducir en el robot simplemente conduciendo externamente al brazo por las posiciones deseadas. Sin embargo en robots móviles, con escenarios más variables, ese planteamiento es inviable y normalmente el diseñador escribe a mano el programa que determina su comportamiento.

Entre los programadores de robots móviles figuran los propios fabricantes (como Sony, ActivMedia, iRobot, etc.), algunas empresas dedicadas (como Evolution Robotics), y los centros de investigación. Gran parte del software existente hoy día para robots ha sido desarrollado por grupos de investigación, lo cual está en consonancia con la inmadurez ya comentada del mercado robótico. Afortunadamente ese software suele estar disponible libremente en Internet, reflejo del deseo de difusión del conocimiento en esos grupos.

2.1. Requisitos

Escribir programas para robots es una tarea complicada, porque los robots son sistemas complejos. De hecho, la programación de robots móviles suele ser más exigente que la creación de programas tradicionales como aplicaciones de ofimática, bases de datos, etc. En esta sección enumeramos algunos condicionantes específicos diferenciadores.

En primer lugar, los programas de robots móviles están empotrados en la realidad física, a través de sensores y actuadores. Con los sensores miden alguna característica de la realidad y con los actuadores pueden provocar cambios en ella. Por un lado, esta situación implica que el software de robots móviles debe ser ágil, pues debe tomar decisiones con vivacidad para controlar correctamente ciertos actuadores. Por lo tanto se tienen requisitos de tiempo real, si no estricto, al menos blando. Por otro lado, hay una enorme variedad de dispositivos sensoriales y de actuación, así como de interfaces, que un programador debe dominar si quiere escribir programas para robots.

En segundo lugar, una aplicación de robots móviles típicamente ha de estar pendiente de varias fuentes de actividad y objetivos a la vez. El programa de un robot tiene que atender a muchas cosas simultáneamente: recoger nuevos datos de varios sensores, refrescar la interfaz gráfica, enviar periódicamente consignas a los motores, enviar o recibir datos por la red a otro proceso de la aplicación, etc. Por ello las aplicaciones de robots suelen ser concurrentes, lo cual les añade cierta complejidad. En este sentido los sistemas operativos de robots más avanzados incorporan mecanismos multitarea y de comunicación interprocesos. Con ello permiten la distribución de las tareas en diferentes hebras que se ejecutan con-

currentemente y una programación modular que se adapta a la naturaleza de las aplicaciones de robots mejor que la estrictamente secuencial.

Tercero, otra cuestión relevante que deben contemplar las aplicaciones que corren a bordo de los robots es su interfaz gráfica. Aunque la interfaz gráfica no es indispensable para generar y materializar el comportamiento autónomo en el robot, normalmente resulta muy útil como herramienta de depuración. Además de las posibilidades de interacción con el usuario, la interfaz gráfica permite la visualización en tiempo de ejecución de estructuras internas (e.g. representaciones del mundo), las trazas y el análisis de variables internas del programa que pueden afectar a la conducta observable del robot. El código de la aplicación deberá encargarse de actualizar esa interfaz y de atender la interacción por parte del usuario.

Cuarto, el software de robots es cada vez más distribuido, tal y como apuntan Woo et al.[WMT03]. Es usual que las aplicaciones de robots tengan que establecer alguna comunicación con otros procesos ejecutando en la misma o en diferente máquina. Para generar comportamiento en un único robot esta distribución no es imprescindible, pero ofrece posibilidades ventajosas como ubicar la carga computacional en nodos con mayor capacidad o la visualización remota; y en sistemas multirrobot, hace posible la integración sensorial, la centralización y la coordinación. Por ejemplo, las comunicaciones permiten el acceso remoto a los sensores del robot, muy útil en aplicaciones de teleoperación. Esta distribución implica que el código de la aplicación debe encargarse de mantener la comunicación por red con los otros procesos remotos.

En quinto lugar, las aplicaciones de robots no cuentan con un marco estable, no hay estándares abiertos que propicien la colaboración [USEK02, MRT03, CLM+04], la reutilización de código y la integración. En otros campos de la informática hay bibliotecas que un programador puede emplear para construir su propio programa, ganando en fiabilidad y acortando el tiempo de desarrollo. Por el contrario, en robótica cada aplicación prácticamente ha de construirse de cero para cada robot concreto. No hay una base sólida y firme para el desarrollo de nuevas aplicaciones. Esa falta se debe en parte a la heterogeneidad endémica en robótica, tanto en hardware como en software, y en parte a la inmadurez del mercado de robots programables. En cualquier caso, esa ausencia está frenando las posibilidades de crecimiento del mercado robótico.

En la actualidad hay avances hacia la estandarización, como las plataformas de desarrollo que veremos en la sección 4. La aparición de esas plataformas supone un paso más hacia la madurez, tratando de estabilizar una interfaz sobre la que construir aplicaciones de robots, pero queda aún mucho camino por recorrer.

En sexto lugar, y ligado con el condicionante anterior, el conocimiento sobre cómo generar y descomponer el comportamiento artificial es muy limitado. Cómo dividir el comportamiento de robots en unidades básicas sigue siendo materia de

investigación, y no hay una guía universalmente admitida sobre cómo organizar el código de las aplicaciones de robots para que sea escalable y se puedan reutilizar sus partes. Cada desarrollador programa su aplicación combinando ad hoc los bloques de código que puedan existir en su entorno.

2.2. Sistemas operativos, plataformas y aplicaciones

El modo en que se programan los robots ha ido evolucionando con el paso de los años. Históricamente los robots eran desarrollos únicos, no se producían en serie, y los programas de control se solían construir empleando directamente los *drivers* para acceder a los dispositivos sensoriales y de actuación. El *sistema operativo* era mínimo, básicamente una colección de *drivers* con rutinas para leer datos de los sensores y enviar consignas a los actuadores. En este contexto, el programa de aplicación leía las medidas obtenidas por los sensores y escribía las órdenes de movimiento a los motores invocando directamente las funciones de librería que ofrecía el fabricante en sus *drivers*. Como muestra la figura 1(a), en este caso la aplicación se situaba directamente encima del sistema operativo.

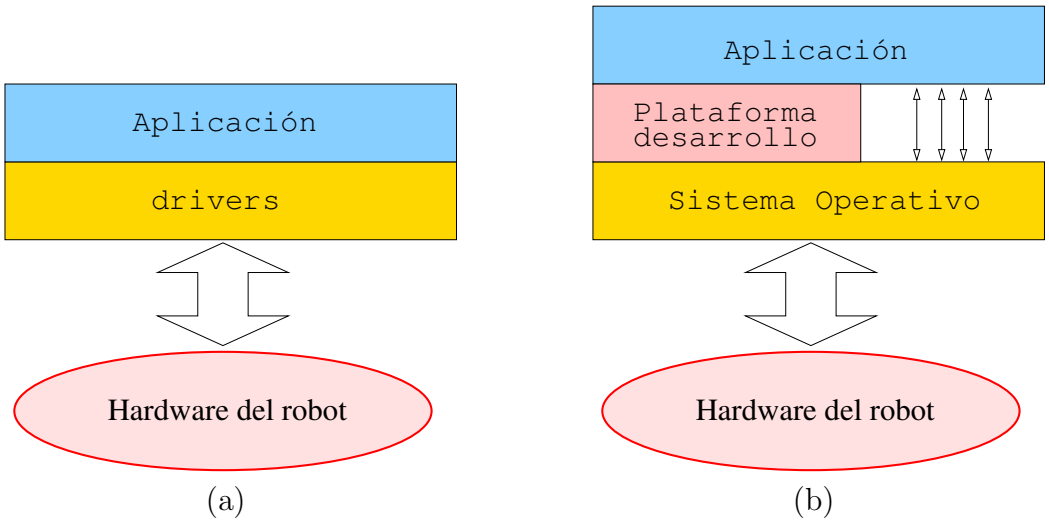


Figura 1: Programación de robots: (a) sobre *drivers* específicos de sensores y actuadores. (b) sobre una plataforma de desarrollo

Con el asentamiento de los fabricantes, el trabajo de muchos grupos de investigación y la evolución de los años han ido apareciendo *plataformas de desarrollo* que simplifican la programación de aplicaciones robóticas. Estas plataformas ofrecen acceso más sencillo a sensores y actuadores, suelen incluir un modelo de programación que establece una determinada organización del software y permite manejar la

creciente complejidad del código cuando se incrementa la funcionalidad del robot. El diseñador programa sus aplicaciones robóticas finales sobre esa plataforma de desarrollo.

Las aplicaciones suelen resolver un problema concreto e incluir técnicas específicas: de extracción de información, de toma de decisiones, etc. A la hora de su programación el usuario tiene hoy dos alternativas: la tradicional de programar directamente sobre el sistema operativo, y la más reciente de programar dentro de la plataforma de desarrollo, utilizando sus abstracciones. La primera interfaz suele ofrecer más flexibilidad, con el coste de mayor complejidad en la programación, pues es el propio programador de la aplicación quien debe cuidar que su código realice las tareas de las que se podría encargar la plataforma. La figura 1(b) ilustra estas dos posibilidades. En ambos casos las aplicaciones se apoyan en una infraestructura de programación, bien sea básica como el sistema operativo, o más abstracta como la plataforma de desarrollo. En las secciones 3 y 4 se detallan ampliamente las características de ambas infraestructuras.

2.3. Herramientas

El procedimiento de creación de aplicaciones para robots no difiere especialmente del de otras aplicaciones. El programador tiene que escribir la aplicación en cierto lenguaje, compilar y enlazar su código con las librerías de la plataforma y/o del sistema operativo, y finalmente ejecutarla en los computadores a bordo del robot. Esa ejecución genera el comportamiento deseado cuando el robot se encuentra en el entorno adecuado. Para todos estos pasos el diseñador cuenta con varias herramientas: editores para escribir el programa fuente en el lenguaje elegido, compilador, simuladores, depuradores específicos, compilador cruzado, etc.

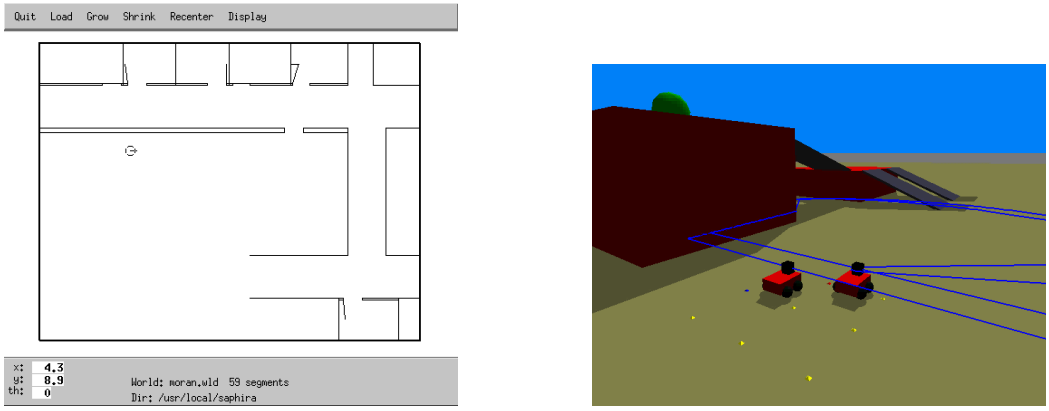
Muchos robots tienen recursos computacionales y de almacenamiento limitados: carecen de disco duro, pantalla suficiente, etc. En estos casos se utiliza el PC como entorno de desarrollo y se emplean compiladores cruzados para generar en el PC el programa ejecutable que correrá sobre el procesador a bordo del robot. Es el caso de la mayoría de robots pequeños, como el RCX de LEGO, Aibo de Sony, EyeBot[5], Kephera de K-Team[6], Pekee de Wany Robotics[7], etc.

Una herramienta particular, muy útil en la programación de robots son los *simuladores*. Los simuladores ofrecen un entorno virtual en el que emulan las observaciones de los sensores y los efectos de las órdenes a los actuadores. Sirven como evaluación, depuración y ajuste del programa de control antes de llevar la aplicación al robot real.

[5] `http://www.ee.uwa.edu.au/~braunl/eyebot`
[6] `http://www.k-team.com`
[7] `http://www.wanyrobotics.com`

Los buenos robots no son baratos y necesitan mantenimiento. Los simuladores son una opción económica para quienes quieren investigar programando robots pero no cuentan con el dinero o el material necesario. Esto se hace aún más patente en el caso de investigar con poblaciones grandes de robots. Aunque actualmente han perdido credibilidad como demostrador de resultados experimentales, los simuladores no han perdido valor como herramienta útil.

Los primeros simuladores eran poco realistas y almacenaban mundos planos donde había obstáculos bidimensionales. Con los años se ha ido ganando en realismo, incorporando ruido en los sensores y en las actuaciones. Hoy día se tienen simuladores tridimensionales, como Gazebo[8] (en la figura 2(b) se puede observar un mundo simulado con Gazebo) y JMR [LOIBGL01], capaces de simular sensores tan complejos como la visión. También se ha incluido en los simuladores más potentes la capacidad de representar un *conjunto* de robots operando en el mismo escenario, simultáneamente.



(a) (b)

Figura 2: Simuladores de robots: (a) SRIsim ©(b) Gazebo

Muchos fabricantes de robots incluyen un simulador para sus robots (e.g. `EyeSim` para el robot EyeBot, `Webots` para Kephera y Koala) aunque también hay muchos desarrollos libres (`Stage`, `Gazebo`, `JMR`). Los simuladores libres tratan de incluir soporte para los robots de varios fabricantes. La configuración concreta de el/los robot/s a simular, la disposición y parámetros de sus sensores se suele especificar en un fichero de configuración. Algunos simuladores relevantes son el `SRIsim`[9] de Kurt Konolige, que se emplea en los robots de ActivMedia (en la figura 2(a) se puede ver un mundo simulado con `SRIsim`), los simuladores `Stage` y

[8] `http://playerstage.sourceforge.net`
[9] `http://www.ai.sri.com/~konolige/saphira`

`Gazebo` de software libre orientados a multirrobot (`Stage` soporta 2D y colonias numerosas de robots, y `Gazebo` soporta 3D). También incluye visión y realidad tridimensional el simulador `JMR` [LOIBGL01], que está escrito en Java.

2.4. Lenguajes

En cuanto a los lenguajes que se emplean para programar robots, no hay diferencias significativas con los utilizados en otras aplicaciones más tradicionales. Si bien hay casos de programación funcional y programación lógica de robots, sin ninguna duda los más utilizados son los lenguajes genéricos procedimentales. Normalmente los lenguajes utilizados son genéricos, es decir, se usan en otras aplicaciones informáticas, y la parte específica de robótica se encapsula en bibliotecas u objetos particulares. También ha habido intentos de establecer lenguajes específicos para robots, como *Task Description Language* (TDL) de Simmons [SA98] o *Reactive Action Packages* (RAP) de Firby [Fir94], que tratan de incluir en la propia sintaxis del lenguaje primitivas ventajosas para la programación de robots, como la descomposición de tareas, la monitorización de ejecución, la sincronización, etc. Estos lenguajes llevan asociada una semántica que es una apuesta cognitiva sobre cómo generar el comportamiento en los robots.

En los brazos robotizados industriales es frecuente el uso de lenguajes de bajo nivel, prácticamente ensamblador [BM03] del microprocesador de turno. Por el contrario, en los robots móviles quedan lejos los tiempos de programación en ensamblador para hacer un código eficiente en extremo. La incorporación creciente del ordenador personal como procesador principal ha abierto el paso a toda suerte de lenguajes de alto nivel. Hay robots programados con JAVA, con Python, con C, con C++, Visual Basic, etc. Incluso para robots con microprocesadores suele haber compiladores cruzados específicos que permiten la programación en lenguajes de alto nivel. Por ejemplo, el robot EyeBot que incorpora el microcontrolador Motorola 68332, se programa en C.

Normalmente la plataforma de desarrollo suele determinar el lenguaje en el que se escribe la aplicación, pues se obliga a ello para reutilizar la funcionalidad ya resuelta en la plataforma. Del mismo modo, un conjunto amplio de grupos de investigación realiza sus desarrollos en C/C++, y la reutilización e integración de su software es más sencilla en este lenguaje común. También hay plataformas que no imponen un lenguaje determinado a las aplicaciones. Por ejemplo, en `Player/Stage/Gazebo` (PSG) [GVH03, VGH03] la funcionalidad se ofrece a través de mensajes de red a un servidor, la aplicación puede estar escrita en cualquier lenguaje siempre que respete el protocolo, y existen librerías hechas en C, C++, Tcl, Python, Java y Common Lisp que encapsulan ese diálogo en el cliente. En la plataforma Miro [USEK02] la funcionalidad se ofrece a través de objetos CORBA que exportan sus interfaces, las aplicaciones pueden escribirse en cualquier

lenguaje en el que exista soporte para el estandard CORBA.

Sin duda los lenguajes más utilizados en la programación de robots son los basados en texto, por su flexibilidad y potencia expresiva. Sin embargo, un caso curioso de lenguaje es el que incluye LEGO para que los niños programen sus robots RCX [BM03], código-RCX, que es completamente visual. En código-RCX un programa consiste en una columna que se forma encadenando en secuencia bloques gráficos que son instrucciones. Estas instrucciones son de espera, de actuación, de suma a una variable, etc. Se incluyen bloques condicionales, con dos ramas salientes, que hacen avanzar el flujo de programa por una de ellas, dependiendo de alguna condición sensorial. Se pueden incluir varias columnas, a modo de hilos de ejecución concurrentes. También se utiliza un lenguaje visual en los entornos Robolab[10] y RobotFlow/FlowDesigner [CLM+04], que permiten encadenar bloques gráficos funcionales con entradas y salidas. El programa del robot se materializa en una red de bloques que configura un determinado flujo de la información desde los sensores a los actuadores.

Un lenguaje que tuvo mucha presencia en los primeros años de la robótica, y que rezuma la influencia de los trabajos en Inteligencia Artificial es Lisp. Actualmente uno de los lenguajes más extendidos para programar robots es C. Este lenguaje supone un buen compromiso entre potencia expresiva y rapidez. Como lenguaje compilado su eficiencia temporal es superior a otros lenguajes interpretados. Hasta hace unos años gran parte del software proporcionado por los fabricantes de robots estaba codificado en C, como las librerías de Saphira [Act99] con que se vendían los robots de Activmedia, o el entorno de desarrollo de los robots de RWI [RWI96]. Muchos robots pequeños se programan en C, como el robot EyeBot [Brä03] y el robot LEGO RCX, que se puede programar con una variante recortada de C llamada NQC [Bau00].

Recientemente se puede observar un crecimiento en los desarrollos y aplicaciones en C++. Por ejemplo, el entorno ARIA [Act02] para programar robots de ActivMedia, el entorno OPEN-R [Son03a, Son03b] de programación del perrito Aibo de Sony, la plataforma Miro [USEK02], la plataforma Mobility [RWI99], etc. El principal avance sobre C es que proporciona la abstracción de orientación objetos, con los mecanismos de herencia y polimorfismo asociados. Potencialmente también simplifica la reutilización de componentes. Una ventaja más es que la portabilidad desde C a C++ es relativamente sencilla. Este crecimiento refleja la tendencia en la programación de robots a la orientación a objetos, a encapsular la funcionalidad en forma de objetos con métodos que se pueden invocar.

[10] `http://www.ni.com/company/robolab.htm`

3. Sistemas operativos

Como hemos mencionado, la primera opción para el desarrollador de aplicaciones robóticas consiste en escribir su código sobre la funcionalidad que le proporciona el sistema operativo de la computadora a bordo del robot. La misión principal de ese sistema operativo es ofrecer a los programas acceso básico al hardware del robot, permitir su manipulación y uso desde los programas. Fundamentalmente debe permitir recoger lecturas de los sensores y enviar órdenes a los actuadores. Para ésto el sistema operativo incorpora *drivers* que dan soporte software de bajo nivel a los dispositivos físicos. Por ejemplo, en el B21 [RWI94] el kernel de Linux incorpora el *driver* de la tarjeta ISA a la que se conectan los sensores de ultrasonidos (*Access Bus Card*), y que permite al programa obtener sus medidas.

Si el robot tiene hardware de comunicaciones, típicamente inalámbrico para mayor movilidad, el sistema operativo ofrece el soporte para utilizarlo desde el programa de aplicación. Por ejemplo, el perrito Aibo incorpora en sus últimos modelos una tarjeta wireless 802.11 para comunicación inalámbrica. Su sistema operativo permite al programa el envío y recepción de datos a través de esa tarjeta, implementando una pila de TCP/IP.

En cuanto a la multitarea, el sistema operativo y las librerías que los acompañan suele ofrecer la posibilidad de programar con distintos procesos y/o con varias hebras, ya sean con o sin desalojo. También suele incluir mecanismos de comunicación interprocesos como la memoria compartida, las tuberías (*pipes*), etc. y recursos típicos de concurrencia (como los semáforos y los monitores) para la sincronización, para evitar las condiciones de carrera y los interbloqueos, garantizar exclusión mutua, etc.

Los sistemas operativos y las librerías que los acompañan suelen incluir también soporte para los elementos de interacción del robot, ya sean botones físicos, pantallas pequeñas, pantallas normales de ordenador, etc. Este soporte permite la creación de interfaces gráficas, lo que combinado con las comunicaciones, puede facilitar la visualización remota en un ordenador externo.

Además de las interfaces de programación, el sistema operativo de un robot ofrece siempre los medios para cargar distintos programas en él y ejecutarlos en los procesadores de a bordo. Esta función es necesaria porque los robots programables, por su propia definición, no tienen un código inalterable sino que pueden ejecutar diferentes programas.

3.1. Sistemas operativos dedicados y generalistas

En el mercado de robots móviles programables gran parte de los robots pequeños tienen su propio sistema operativo dedicado, muy ligado al procesador, a los sensores y a los actuadores concretos. El sistema operativo ROBIOS en el robot

EyeBot [Brä03] (en la figura 3), Aperios en el Aibo (en la figura 7), o BrickOS[11] [Nie00] para el LEGO RCX (en la figura 8) son ejemplos relevantes.

Con la irrupción de los ordenadores personales (portátiles o no) como procesadores principales en los robots, los sistemas operativos de propósito general como Linux o MS-Windows han entrado en el mundo de los robots. Estos sistemas, además de los *drivers* del hardware, incluyen abstracciones y un conjunto muy extenso de bibliotecas genéricas, que también son útiles para la programación de robots. Por ejemplo, bibliotecas e interfaces para la multitarea, las comunicaciones y las interfaces gráficas.

Figura 3: Robot EyeBot en el corre el sistema operativo ROBIOS

ROBIOS

ROBIOS es una muestra significativa de sistema operativo dedicado. El robot EyeBot [Brä03] es un robot de pequeño tamaño que viene equipado con un microprocesador Motorola 68332 a 35 MHz. Al robot se le conectan dos motores con *encoders*, tres sensores de infrarrojos y una cámara digital. Su hardware también incluye una pequeña pantalla, cuatro botones y un radioenlace. El sistema operativo permite la carga de programas a través de un puerto serie y su ejecución pulsando los botones. Como acceso básico al hardware desde los programas, ROBIOS incluye una API (*Application Programming Interface*) con funciones para recoger las lecturas de los sensores de infrarrojos, para obtener las imágenes de la cámara y para hacer girar los motores a cierta velocidad.

En cuanto a comunicaciones, ROBIOS incluye dos funciones que le permiten enviar y recibir bytes a través del radioenlace, a otros EyeBots o un PC que se

[11] El sistema operativo libre para el RCX se llamaba inicialmente LegOS, posteriormente cambió su nombre oficial a BrickOS

encuentren dentro del alcance radio de su antena. En cuanto a la interfaz gráfica, ROBIOS incluye primitivas para muestrear si los botones están pulsados, pintar y refrescar la pantalla.

Con respecto a la multitarea, ROBIOS tiene primitivas para crear hebras, detenerlas durante un tiempo, matarlas, etc. Además ofrece dos modos de repartir el tiempo de procesador entre ellas: con desalojo (en rodajas de tiempo) y sin desalojo (con un testigo que va pasándose de una hebra a la siguiente). Correspondiéndose a las hebras de kernel o de usuario típicas de sistemas operativos tradicionales. También ofrece semáforos para coordinar la ejecución concurrente de esas hebras y el acceso a variables compartidas.

GNU/Linux

Un sistema operativo generalista que ha ganado gran aceptación en la comunidad robótica es GNU/Linux [RWI99, GVH03, MRT03]. Este sistema es *software libre* y cuenta con una comunidad muy activa y amplia de desarrolladores, lo cual garantiza en la práctica la incorporación al sistema del soporte de los nuevos dispositivos hardware que van ofreciendo los avances tecnológicos. Este sistema operativo es muy potente y robusto, y ofrece sus servicios en forma de llamadas al sistema y de funciones de biblioteca. Hay llamadas para crear procesos, hebras, controlar su coordinación, comunicaciones, etc. Además incluye un amplio abanico de herramientas de desarrollo, como el compilador de C/C++ `gcc`, el editor `emacs`, etc.

Respecto a la multitarea, GNU/Linux incluye la biblioteca *Pthreads* como referencia para materializar las hebras de las aplicaciones. Este paquete genera hebras POSIX de kernel que se pueden comunicar a través de memoria compartida. GNU/Linux también ofrece semáforos para controlar el acceso a esas variables compartidas o resolver problemas de concurrencia que puedan aparecer. Estos mecanismos se suman a los que ya ofrece GNU/Linux como creación de procesos, tuberías, `fifos`, memoria mapeada, etc., que siempre están disponibles.

En cuando a comunicaciones GNU/Linux ofrece los *sockets* para la comunicación intermáquina entre varios programas, una abstracción que permite olvidarse de los detalles de red, protocolos de acceso al medio, etc. En concreto soporta IP para el nivel de red, y los protocolos del nivel de transporte TCP y UDP. De esta manera el programador de la aplicación no debe preocuparse de localizar la máquina destino, ni modificar su código cuando, por ejemplo, el robot se conecta a la red por ethernet en vez de red inalámbrica.

GNU/Linux ofrece también las librerías necesarias para crear y manejar interfaces gráficas en `X-Window`, que es el sistema más extendido en el mundo Unix. Encima de las librerías básicas (`Xlib` y `Xt-intrinsics`), que son flexibles pero complejas, GNU/Linux dispone de varias librerías que simplifican el manejo de

la interfaz gráfica, como `Qt`, `XForms`, `GTK`, etc. Una ventaja natural de las interfaces en `X-Window` es la visualización remota: es muy sencillo que un programa ejecutándose en un ordenador GNU/Linux (e.g. el ordenador a bordo del robot) vuelque su interfaz gráfica a otro ordenador GNU/Linux (e.g. el PC de trabajo del programador).

Una de las desventajas del uso de GNU/Linux en robótica es que no es un sistema de tiempo real, pues no permite acotar plazos. No ofrece *garantía* de plazos (tiempo real *duro*) porque su sistema de planificación de procesos no implementa esas garantías. En este sentido contrasta con sistemas operativos similares como QNX, o la variante RT-Linux. Sin embargo, GNU/Linux resulta suficientemente ágil para los requisitos de aplicaciones no críticas. Por ejemplo, permite detener hebras durante milisegundos.

3.2. Procesadores, sensores y actuadores

El hardware de los robots consiste principalmente en un conjunto de procesadores, sensores y actuadores, y puede incorporar también elementos de comunicaciones e interacción. Además de ser muy heterogéneo, este hardware evoluciona rápidamente, porque los avances tecnológicos proporcionan nuevos y mejores dispositivos a un ritmo alto. Este dinamismo se contagia al software, que debe evolucionar continuamente para asimilar el soporte a las prestaciones de los nuevos dispositivos.

En cuanto a sensores, la tendencia reciente más marcada es la incorporación de la visión y del láser al equipamiento sensorial de los robots. En primer lugar, las cámaras son cada vez más frecuentes en los robots, tanto monoculares como pares estéreo. Hasta ahora las más utilizadas eran analógicas, las cuales solían ser caras y necesitaban de tarjetas capturadoras (Matrox, BTTV, etc.) con *drivers* específicos para digitalizar la imagen. En los últimos años la tecnología ha proporcionado cámaras digitales a un precio razonable (principalmente destinadas a aplicaciones multimedia), que se han introducido en los robots como otro sensor más. Las prestaciones de esas cámaras siguen mejorando rápidamente, hacia mayor calidad de imagen, mayor frecuencia, etc. Para dar soporte a estas cámaras y las anteriores en Linux se ha estandarizado la interfaz `video4linux`. También hay soporte para el bus IEEE-1394 (*firewire*), con el cual se pueden tener dos cámaras digitales capturando a 30 fps sin apenas consumo computacional. La principal dificultad asociada a las cámaras es que resulta difícil extraer información útil del enorme flujo de datos que proporcionan. En segundo lugar, desde finales de los años 90 se ha difundido enormemente el empleo del sensor láser (por ejemplo de la casa SICK) como sensor de obstáculos en los robots de interiores. Este dispositivo mide la distancia a obstáculos en un haz de 180^o, con una resolución de 1-2 cm y precisión inferior a 1 grado. El láser proporciona información muy precisa de obstáculos, con

una interpretación directa para la navegación.

En cuanto a los actuadores, se ha observado un crecimiento en el número de robots con patas, tanto en productos comerciales (los Aibo de Sony) como en los prototipos bípedos (Asimo de Honda, Qrio de Sony). Sin embargo sigue siendo mayoritario el uso de robots con ruedas, que obvian los problemas de equilibrio y ofrecen una interfaz más sencilla para el movimiento. También se aprecia una tendencia a la miniaturización: mientras a principios de los años 90 los robots más exitosos en entornos de investigación eran el B21 o los Nomad, en los últimos años son más frecuentes robots menores como el Pioneer o incluso los Aibo.

En cuanto a los procesadores, antes de 1970 los cómputos se realizaban fuera del robot. Los ordenadores eran demasiado grandes y pesados para pensar en ponerlos sobre un robot móvil. Por ejemplo, el robot pionero Shakey utilizaba una placa de control lógico en el vehículo y un computador SDS-940 fuera para realizar el análisis de imágenes y los cálculos necesarios. Con el avance de la tecnología (e.g. baterías más pequeñas) y la reducción en el tamaño de los ordenadores se tendió a incluir todo el cómputo necesario a bordo del robot móvil. Por ejemplo, en los años 80 el robot Hilare ya utilizaba una red de procesadores específicos dentro de su estructura mecánica: IBM 30/33, Intel 8085/86, Sel 32 77/80, etc.

Desde comienzo de los años 90 los ordenadores personales (PC) se han insertado como elementos principales de cómputo en los robots de investigación, primero los PCs de sobremesa y después los PC portátiles. El precio relativamente asequible y el crecimiento exponencial de su capacidad de cálculo han favorecido esa irrupción. Los procesadores específicos son cada vez más escasos y se han dejado para tareas de bajo nivel, muy concretas. Por ejemplo, los procesadores digitales de señal dedicados al procesamiento específico de imágenes; o los microcontroladores para controlar los motores y recoger medidas de sensores con poco ancho de banda (como los ultrasonidos, los odómetros y los de contacto) que incorporan robots como el Pioneer o el B21 (figura 4).

La configuración de un PC y varios microcontroladores es muy frecuente hoy día. En los microcontroladores se ejecuta un sistema operativo de bajo nivel, como el `P2OS` en los Pioneer [Act03], o `rFLEX` en los B21 [RWI99], y la comunicación con el PC se realiza a través del puerto serie RS232, respetando cierto protocolo estable. Esta configuración concentra la mayor parte del cómputo en el PC, que es el procesador más poderoso, y ofrece la ventaja de que actualizar el PC cuando se queda obsoleto es muy fácil.

En cuanto a la conexión de sensores y actuadores con el procesador del PC, el puerto serie es una buena opción. Es el caso de los cuellos mecánicos (*pantilt*), los escáneres láser de SICK, sensores de GPS, etc. Otras interfaces estandarizadas son el bus USB o el bus IEEE-1394. Una opción más de conexión son las tarjetas especiales dedicadas, que se enganchan a algún bus del PC (ISA, PCI o incluso

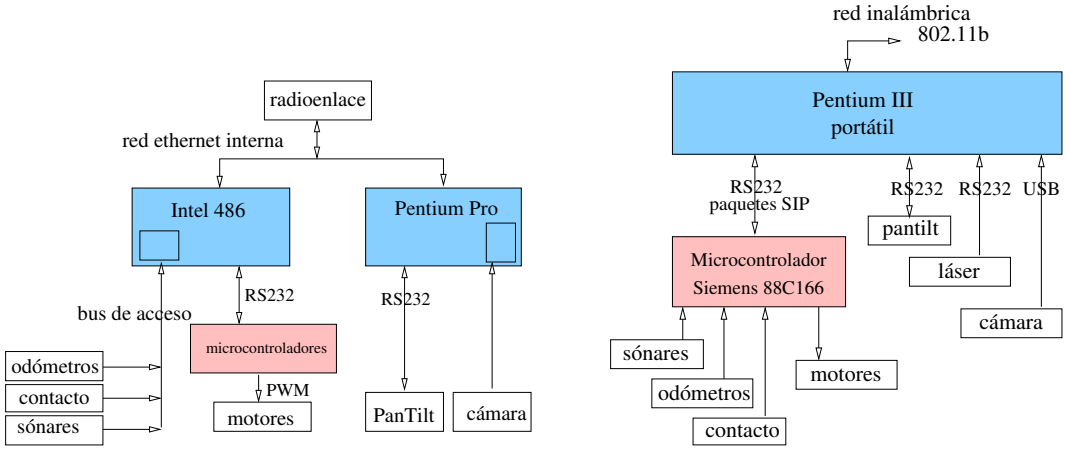


Figura 4: Arquitectura de computadores (a) de un B21 (b) de un Pioneer

PCMCIA). Es el caso de la tarjeta que hay en el B21 para sónares, las tarjetas digitalizadoras de imágenes, etc. Una tercera opción son las tarjetas de adquisición de datos que incorporan conversores A/D, D/A, entradas y salidas tanto digitales como analógicas. Estas tarjetas permiten integrar sensores a muy bajo nivel.

4. Plataformas de desarrollo

Las aplicaciones con robots presentan cada vez de mayor complejidad y ofrecen mayor funcionalidad. Como hemos argumentado en la sección 2, escribir programas para robots es complicado. En muchos campos del software se ha ido implantando *middleware* que simplifica el desarrollo de nuevas aplicaciones en esos campos; por ejemplo, Matlab para aplicaciones de ingeniería, CORBA o ACE para aplicaciones distribuidas, J2EE para aplicaciones y servicios web, etc. Este *middleware* proporciona contextos nítidos, estructuras de datos predefinidas, bloques muy depurados de código de uso frecuente, protocolos estándard de comunicaciones, mecanismos de sincronización, etc. Del mismo modo, a medida que el desarrollo de software para robots móviles ha ido madurando han ido apareciendo diferentes plataformas *middleware* [USEK02].

Hoy día los fabricantes más avanzados incluyen plataformas de desarrollo para simplificar a los usarios la programación de sus robots. Por ejemplo, Activmedia ofrece la plataforma ARIA [Act02] para sus robots Pioneer, PeopleBot, etc.; iRobot ofrece Mobility [RWI99] para sus B21, B14, etc.; Evolution Robotics vende su plataforma ERSP; y Sony ofrece OPEN-R [MGCCM04] para sus Aibo. Otros fabricantes menos avanzados carecen de plataforma de desarrollo u ofrecen una

muy limitada. Por ejemplo, para programar los robots RobuCar y RobuLab de Robosoft el desarrollador sólo dispone de unas bibliotecas con primitivas en C que ofrecen una interfaz básica, de sistema operativo.

Además de los fabricantes, muchos grupos de investigación han creado sus propias plataformas de desarrollo. Varios ejemplos son la suite de navegación CARMEN[12] [MRT03] de Carnegie Mellon University, Orocos [Bru01], PSG [GVH03], Miro [USEK02], JDE [CM02], etc. La mayoría de estas plataformas se distribuyen como software libre por los propios creadores. Al igual que ocurre con los simuladores, mientras los fabricantes buscan que su plataforma sirva para todos sus modelos, los grupos de investigación aspiran a mayor universalidad y sus plataformas tratan de incluir soporte para los robots de sus laboratorios, típicamente de diferentes fabricantes.

El objetivo fundamental de estas plataformas es hacer más sencilla la creación de aplicaciones para robots. Hemos identificado varias características que estas plataformas presentan para lograrlo: uniforman y simplifican el acceso al hardware, ofrecen una arquitectura sofware concreta y proporcionan un conjunto de bibliotecas o módulos con funciones de uso común en robótica que el cliente puede reutilizar para programar sus propias aplicaciones. Dedicaremos esta sección a describir estas tres características.

Aunque hay plataformas más y menos completas, en general aumentan la independencia de las aplicaciones respecto del hardware y de los sistemas operativos subyacentes. Las plataformas establecen un suelo más o menos firme para el desarrollo de aplicaciones con robots. Son ellas las que asimilan los cambios de bajo nivel, tratando de mantener la misma interfaz abstracta para las aplicaciones, tanto en el acceso a sensores y actuadores como a los mecanismos multitarea o funcionalidades ya asentadas en los robots.

4.1. Abstracción del hardware

En primer lugar, la plataforma uniforma y simplifica el acceso al bajo nivel. Normalmente ofrece un acceso a sensores y actuadores más abstracto y simple que el que proporciona el sistema operativo. Por ejemplo, si se dispone de un robot Pioneer equipado con un sensor láser SICK, la aplicación puede acceder a sus medidas a través de las funciones de la plataforma ARIA o pedirlas y recogerlas directamente a través del puerto serie. Utilizando ARIA basta invocar al método `getRawReadings` sobre un objeto de la clase `ArSick`, la plataforma se encarga de mantener actualizadas las variables con las lecturas. Utilizando directamente el sistema operativo la aplicación debe solicitar y recoger periódicamente las lecturas al sensor láser a través del puerto serie, y conocer el protocolo del dispositivo para

[12] `http://www-2.cs.cmu.edu/~carmen`

componer y analizar correctamente los mensajes de bajo nivel.

El acceso abstracto también se ofrece para los actuadores. Por ejemplo, en vez de ofrecer comandos de velocidad para cada una de las dos ruedas motrices de un robot Pioneer, la plataforma Miro [USEK02] ofrece una sencilla interfaz de V-W (velocidad de tracción y de giro) para la actuación motriz a través de la clase `position`. Ella se encarga de hacer las transformaciones oportunas, de enviar a cada rueda las consignas necesarias para que el robot consiga esas velocidades comandadas de tracción (V) y de giro (W).

Hattig et al. [HHB03] apuntan que la uniformidad en el acceso al hardware es el primer paso para favorecer la reutilización de software dentro de la robótica. Esta característica está presente en todas las plataformas que hemos estudiado, aunque cada una lo haga a su manera. En la plataforma `ERSP` de Evolution Robotics el acceso al bajo nivel recibe el nombre de *Hardware Abstraction Layer*, (figura 5)y en Miro, *Service Layer*. En `OPEN-R`, en Mobility y en `ARIA` la API de acceso a los sensores y actuadores viene dada por los métodos de un conjunto objetos. En `JDE` [CM02] y en PSG el acceso abstracto al hardware lo marca un protocolo, con vocación de estándard, entre las aplicaciones y los servidores.

Con la interfaz abstracta de acceso a sensores y actuadores, más o menos estable, las plataformas favorecen la portabilidad de las aplicaciones entre robots distintos. Ellas se encargan de materializar la misma interfaz para cada diferente robot soportado. Por ejemplo, una misma aplicación de movimiento escrita sobre Miro [USEK02] puede gobernar con mínimas adaptaciones robots diferentes como el B21, el Pioneer o el robot casero Sparrow porque todos ellos están soportados en la plataforma. Otro ejemplo más, en Miro la clase de `sensor_distancia` vale para dispositivos diferentes como los sónares, los infrarrojos o el láser, cada uno de los cuales proporciona información de proximidad a objetos, con sus peculiaridades.

4.2. Arquitectura software

En segundo lugar, la plataforma ofrece una arquitectura software determinada. Con ello fuerza a escribir las aplicaciones de cierta manera, recortando en flexibilidad, pero ganando en simplicidad y en portabilidad.

La arquitectura software fija la manera concreta en la que el código de la aplicación debe acceder a las medidas de los sensores, ordenar a los motores, o utilizar la funcionalidad ya desarrollada. Muchas son las alternativas software para ello: llamar a funciones de biblioteca, leer variables, invocar métodos de objetos, enviar mensajes por la red a servidores, etc. Por ejemplo, la plataforma CARMEN [MRT03] presenta interfaces funcionales, la plataforma Miro la invocación de métodos de objetos distribuidos, la plataforma TCA [SGFB97] requiere del paso de mensajes entre distintos módulos, la plataforma JDE [Cañ03] requiere la activación de procesos y la lectura o escritura de variables. Esta apariencia de las

interfaces depende de cómo se encapsule en cada plataforma la funcionalidad ya desarrollada.

Al escribirse dentro de la arquitectura software de la plataforma, el programa de la aplicación toma una organización concreta. Así, se puede plantear como una colección de objetos (como en OPEN-R), como un conjunto de módulos dialogando a través de la red (como en TCA), como un proceso iterativo llamando a funciones, etc. Esta organización está influida por la arquitectura cognitiva, y supone un modelo de programación, una estructura determinada para escribir las aplicaciones. Hay plataformas muy cerradas, que obligan estrictamente a cierto modelo. Por ejemplo, en la plataforma RAI [RWI96] los clientes han de escribirse forzosamente como un conjunto de módulos RAI (hebras sin desalojo), con ejecución basada en iteraciones. Otras arquitecturas software son deliberadamente abiertas, restringen lo mínimo, como PSG o CARMEN.

En cuanto al modelo de programación, una opción es organizar la aplicación como un único hilo de ejecución que se encarga de todo. Este modelo monohilo es muy sencillo para la programación reactiva: muchas iteraciones por segundo comprobando las condiciones relevantes. Sin embargo, cuando la aplicación robótica crece en complejidad, la programación secuencial se queda corta, y entonces resulta más natural la programación concurrente. Las arquitecturas software de las plataformas más avanzadas establecen mecanismos concretos para que la aplicación se distribuya en varias unidades concurrentes. Por ejemplo, la plataforma ARIA ofrece tareas síncronas y asíncronas (`ArPeriodicTask`, `ArASyncTask`, `ArThread`), los objetos distribuidos de Miro y de Mobility pueden ser objetos activos, con hilos de ejecución en su interior, y pueden notificar eventos de manera asíncrona.

El mecanismo multitarea que ofrece la plataforma envuelve y simplifica la interfaz del kernel subyacente para la multiprogramación, en la cual se apoyan siempre. Al igual que ocurre con la interfaz abstracta de acceso al hardware, la interfaz abstracta de multitarea facilita la portabilidad. Por ejemplo, la de ARIA está soportada sobre Linux y sobre MS-Windows, es exactamente la misma en ambos casos.

La distribución de las aplicaciones robóticas en procesos concurrentes es una tendencia firmemente confirmada en los últimos años. La distribución es doble, tanto de un proceso compuesto de varias hebras, como de varios procesos ejecutando en máquinas diferentes, comunicándose a través de la red. Todos los hilos de ejecución interactúan y cooperan para el fin común de la aplicación. Los módulos de RAI y las tareas asíncronas de ARIA son ejemplos de hebras, con y sin desalojo respectivamente, dentro del mismo proceso. Los servidores y clientes de RAI, una aplicación sobre PSG, los objetos distribuidos de CARMEN, de Mobility o de Miro son muestras de la aplicación compuesta de varios procesos ejecutándose en máquinas potencialmente diferentes. Esta tendencia a la distribución tiene sus

raíces cognitivas en las arquitecturas basadas en comportamientos, en contraste con el desarrollo monohilo secuencial que había predominado hasta principios de los años 90.

La distribución en varias hebras o procesos lleva obligatoriamente a la necesidad de comunicación y coordinación entre ellos. Para los hilos ejecutando en la misma máquina la plataforma ARIA ofrece variables comunes y varios objetos de sincronización (`ArMutex` y `ArCondition`). Para procesos corriendo en máquinas diferentes PSG y JDE ofrecen un protocolo sencillo de mensajes de texto a través de *sockets*. PSG incluye unas bibliotecas con una interfaz funcional para el envío y recepción de mensajes hacia/desde el servidor acorde a ese protocolo. Miro y Mobility resuelven la comunicacion entre objetos distribuidos con CORBA; CARMEN con TCX [Fed94]. ARIA ofrece la clase `Arsockets` con primitivas de apertura, etc. para que el programa maneje explícitamente las comunicaciones.

4.3. Funcionalidades de uso común

En tercer lugar, las plataformas más completas incluyen funcionalidades de uso común en robótica, ya resueltas. El programador puede incorporarlas sin esfuerzo en su propia aplicación si lo necesita. Además de las librerías sencillas de apoyo, como filtros de color y demás, estas funcionalidades engloban alguna técnica robótica concreta, relativamente madura, ya sea de percepción o de algoritmos de control: localización, navegación local segura, navegación global, seguimiento de personas, habilidades sociales, construcción de mapas, etc.

La ventaja de integrarlas en la plataforma es que el usuario puede reutilizarlas, enteras o por partes. Esta reutilización permite acortar los tiempos de desarrollo y reducir el esfuerzo de programación necesario para tener una aplicación. Al incluir funcionalidad común, el desarrollador no tiene que repetir ese trabajo y puede construir su programa reutilizándola, concentrándose en los aspectos específicos de su aplicación. Además, suele estar muy probada, lo cual disminuye el número de errores en el programa final.

La forma concreta en que se reutilizan las funcionalidades depende nuevamente de la arquitectura software y de cómo se encapsulen en ella: módulos, objetos distribuidos, objetos locales, funciones, etc. Por ejemplo, en PSG se incluye la localización como un nuevo sensor y a su funcionalidad se accede intercambiando mensajes con el servidor Player. En Miro la funcionalidad se encapsula en objetos distribuidos, con sus propios métodos que se ponen a disposición de los demás objetos. Por ejemplo, en el robot EyeBot las funcionalidades se ofrecen en forma de librerías. El sistema operativo ROBIOS ofrece primitivas para leer la imagen que está capturando la cámara, y para girar cada rueda motriz a cierta velocidad. Sobre esas primitivas hay una librería que ofrece control V-W y otra con varios filtros para las imágenes (filtro Sobel, filtro Laplace, etc.). En Miro, el control V-

W se ofrece en la capa de abstracción del hardware, mientras que en el EyeBot se ofrece como librería de uso opcional.

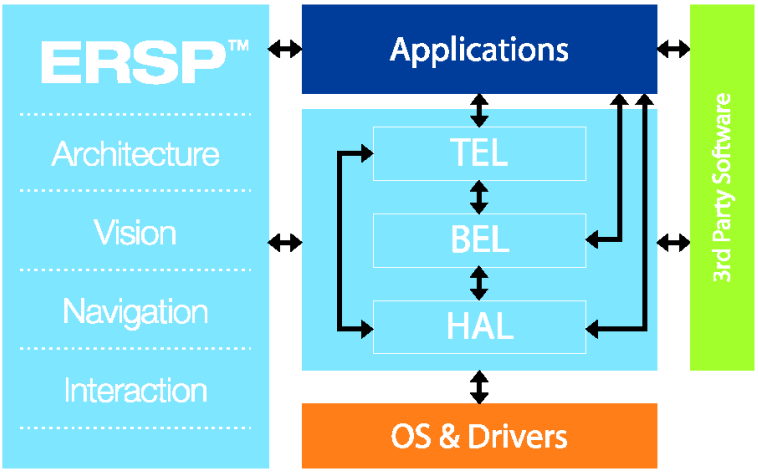


Figura 5: Módulos de navegación, visión e interacción en la plataforma ERSP

Otras funcionalidades más elaboradas son la navegación de un lugar a otro planificando trayectorias, la construcción de mapas basados en rejillas, la localización basada en correspondencia de segmentos, la localización con filtros de partículas, el procesamiento visual estéreo, etc. Normalmente estas funcionalidades nacieron como técnicas novedosas de investigación y al consolidarse van siendo integradas en las plataformas de desarrollo, tanto en aquellas de grupos de investigación como en las de los fabricantes. De esta manera, la cantidad de bibliotecas disponibles va creciendo a medida que van madurando las técnicas concretas.

La plataforma Miro incluye en su *Miro Class Framework* [USEK02] clases con algunos comportamientos, con algoritmos de localización y están trabajando para incorporar algoritmos de planificación de caminos. La plataforma CARMEN [MRT03] incluye navegación basada en planificación por descenso de gradiente, de Konolige, y módulos con las técnicas más exitosas desarrolladas en Carnegie Mellon como la construcción probabilística de mapas de ocupación con forma de rejilla y la localización dentro de esos mapas siguiendo técnicas de MonteCarlo. También incorpora la detección de personas y el seguimiento.

Los fabricantes suelen vender esas funcionalidades por separado o incluirlas como valor añadido de su propia plataforma. Por ejemplo, la empresa ActivMedia proporciona gratuitamente la plataforma de desarrollo ARIA, pero vende separadamente sus paquetes `ARNL` para navegación y localización, la utilidad `Mapper` para la construcción de mapas y `ACTS` para el seguimiento de color. La plataforma ERSP incluye tres paquetes sobre su arquitectura básica: uno de interacción, otro

de navegación y otro visión, según ilustra la figura 5. En el módulo de interacción se incluye la capacidad de reconocimiento del habla y síntesis de voz, para interactuar de modo verbal con su robot. En el módulo de navegación se incluye la capacidad de construcción automática de mapas, la localización en ellos y su utilización para planificar trayectorias.

4.4. Arquitectura cognitiva

Los comportamientos que se han conseguido en robots reales y a los que se aspira son cada vez más ricos y variados, y con ello más difíciles de generar. De manera que los programas que los causan tienden a ser cada vez más complejos. La arquitectura software ofrece un modelo de programación, una cierta manera de organizar ese código. Normalmente este modelo resulta suficiente cuando se persigue generar un comportamiento específico en el robot, pero no suele escalar cuando se quiere integrar en un único sistema un abanico amplio de comportamientos. Es decir, suele resultar insuficiente cuando la aplicación apunta a un repertorio amplio de conductas artificiales y este repertorio ha de desplegarse coherentemente en cada momento dependiendo de los objetivos del robot y del entorno. Este problema es muy complejo y al día de hoy sigue siendo un tema abierto y activo de investigación, no hay una solución universalmente admitida.

Se denomina *arquitectura cognitiva* de un robot a la organización de sus capacidades sensoriales y de actuación para generar un repertorio de comportamientos [Cañ03]. Para comportamientos simples casi cualquier organización resulta válida, pero para comportamientos complejos se hace patente la necesidad de una buena organización cognitiva. De hecho, puede llegar a ser un factor crítico: con una buena organización sí se puede generar cierto comportamiento y con una mala no, se tiene un codigo frágil y la complejidad se vuelve inmanejable. En la comunidad robótica han surgido a lo largo de su historia diversas escuelas cognitivas o paradigmas para orientar la organización del sistema en esos casos. En [Cañ03] hay una buena recopilación de los principales paradigmas con sus arquitecturas cognitivas más representativas: sistemas deliberativos, reactivos, sistemas híbridos (de tres capas, TCA,...), sistemas basados en comportamientos, basados en etología, etc.

Más allá de la interfaz estable para el acceso a un hardware diverso que proporcionan las plataformas, la estandarización del comportamiento artificial, su división en unidades reutilizables, es una cuestión muy complicada. Hattig et al.[HHB03] opinan que aún es demasiado pronto para buscar estándares en este nivel, en el modo de generar comportamientos. Recomiendan ceñirse por ahora la estandarización del acceso a los sensores y actuadores, de lo que ellos llaman bloques constituyentes.

La relación entre las arquitecturas cognitivas y las de software es múltiple. Las arquitecturas cognitivas se materializan en alguna arquitectura software, de

manera que los comportamientos generados siguiendo cierto paradigma acaban implementándose con algún programa concreto. De hecho, las propuestas cognitivas más fiables cuentan con implementaciones prácticas relevantes en arquitecturas software concretas: deliberativas (SOAR), híbridas (e.g. TCA [Sim94, SGFB97], Saphira [KM98]), basadas en comportamientos (subsunción [Bro86], JDE), etc. Una buena arquitectura cognitiva favorece la escalabilidad de la plataforma.

Hay plataformas software debajo de las cuales subyace un modelo cognitivo, pero también hay otras en las que no. No obstante, unas arquitecturas software cuadran mejor con ciertas escuelas cognitivas que con otras. Los sistemas deliberativos clásicos cuadran con la programación lógica (e.g. Prolog), con una descomposición funcional en bibliotecas (módulos especialistas) y con las aplicaciones monohilo con un sólo flujo iterativo de control (sensar-modelar-planificar-actuar). Por el contrario, los sistemas basados en comportamientos cuadran mejor con la programación concurrente, donde se tienen varios procesos funcionando en paralelo y que colaboran al funcionamiento global. Resulta natural asimilar cada unidad de comportamiento al concepto software de proceso, o incluso al de objeto activo.

La distinción entre arquitectura software y arquitectura cognitiva se hace patente en el ejemplo de Saphira [KM98]. El nombre Saphira antes designaba a la propuesta cognitiva y también la plataforma software[13] [Act99] asociada. Ahora se han separado, la plataforma se ha reescrito y rebautizado como ARIA [Act02]. Ella ofrece una capa de acceso abstracto al hardware y una arquitectura software basada en objetos. El nombre de Saphira se ha reservado para el software que materializa la arquitectura cognitiva híbrida, la cual se monta encima de ARIA. Esta separación también es clara en el caso de JDE [Cañ03], que ofrece una plataforma software de servidores por un lado, y un conjunto de esquemas que materializan la apuesta cognitiva por otro.

4.5. Plataformas de software libre

Como ya hemos comentado, existen plataformas de desarrollo creadas por los fabricantes de robots y otras creadas por grupos de investigación. La mayoría de estas últimas se publican con licencia de software libre, con la idea de contribuir al libre intercambio de conocimiento en el área y con ello al avance de la disciplina robótica. Dedicamos esta sección a enumerar las plataformas libres más importantes; en las referencias se pueden encontrar descripciones más detalladas.

[13] hasta la versión 6.2

Plataforma	*Robots soportados*	*Sistema Operativo*	*Sw libre*
Player/Stage	Pioneer, B21	Linux	sí
Miro	Pioneer, B21, Sparrow	Linux	sí
CARMEN	Pioneer, B21	Linux	sí
JMR	Pioneer		sí
Robotflow/Flowdesigner	Pioneer		sí
Webots, SysQuake	Koala, Kephera		no
ARIA	Pioneer	Linux/MS-Windows	sí
OPEN-R	Aibo	Aperios	no
ERSP	ER1	Linux/MS-Windows	no
Mobility	RWI B21	Linux	no

Cuadro 1: Tabla con plataformas de programación robots

Player/Stage/Gazebo

La plataforma Player/Stage/Gazebo[14] (PSG) [GVH03] fue creada inicialmente en la Universidad de South California. Actualmente está mantenida por Richard Vaughan, Andrew Howard y Brian Gerkey. Se compone de dos simuladores Stage y Gazebo (que ya fueron comentados en la sección 2.3), y un servidor Player al que se conecta el programa de la aplicación para recoger los datos sensoriales o comandar las órdenes a los actuadores. El soporte a robots variados como los Pioneer, los B21, etc. y los simuladores que incorpora la convierten en una plataforma muy completa. Ha ganado popularidad y usuarios, actualmente más de 20 laboratorios la utilizan en sus proyectos.

En PSG los sensores y actuadores se contemplan como si fueran ficheros [VGH03], flujos, dispositivos de modo carácter al estilo de Unix, y por ello se manipulan a través de cinco operaciones básicas: abrir, cerrar, leer, escribir y configurar. Para usar un sensor, las aplicaciones tienen que abrir el dispositivo, quizá configurarlo y leer de él, cerrándolo al final. Análogamente, para utilizar un actuador, las aplicaciones tienen que abrirlo, tal vez configurarlo y escribir datos en él, cerrándolo al final. Cada tipo de dispositivo se define en PSG con una *interfaz*, de manera que los sensores sonar del robot Pioneer y los sensores sonar del robot B21 son instancias de la misma interfaz. La única diferencia, invisible para los programas, es que las lecturas se obtienen del sensor concreto empleando *drivers* diferentes. El conjunto de interfaces posibles presenta una máquina virtual, con toda suerte de sensores y actuadores. El robot concreto instancia las interfaces relativas a los dispositivos realmente existentes.

Adicionalmente, PSG tiene un diseño cliente/servidor: las aplicaciones estable-

[14] `http://playerstage.sourceforge.net/`

cen un diálogo por TCP/IP con el servidor Player, que es el responsable de proporcionar las lecturas sensoriales y materializar los comandos de actuación. Además de permitir acceso remoto, este diseño proporciona a las aplicaciones construidas sobre PSG gran independencia de lenguaje y mínimas imposiciones de arquitectura. La aplicación puede escribirse en cualquier lenguaje, y con cualquier estilo, simplemente respetando el protocolo de comunicaciones con el servidor. El protocolo uniforma el modo en que los programas acceden al hardware. Existen librerías que encapsulan funcionalmente ese diálogo para clientes en varios lenguajes.

La plataforma PSG se orienta principalmente a una interfaz abstracta del hardware de robots, no a identificar bloques comunes de funcionalidad. No obstante, cierta funcionalidad adicional se puede incorporar como nuevos mensajes del protocolo, y añadidos al servidor Player. Por ejemplo, la localización probabilística se ha añadido como una interfaz más, `localization`, que proporciona múltiples hipótesis de localización. Esta nueva interfaz supera a la tradicional, `position`, que acarrea la posición estimada desde la odometría.

ARIA

Otra plataforma de software libre muy utilizada es ARIA (*ActivMedia Robotics Interface for Applications*) [Act02]. Esta plataforma está impulsada y mantenida por la empresa ActivMedia Robotics como interfaz de acceso al hardware de sus robots, pero la plataforma en sí se distribuye con licencia GPL y por tanto su código fuente está accesible. ARIA ofrece un entorno de programación orientado a objetos, que incluye soporte para la programación multitarea y para las comunicaciones a través de la red. Las aplicaciones han de escribirse forzosamente en C++. ARIA está soportada en Linux y en Win32 OS, por lo que una misma aplicación escrita sobre su API puede funcionar en robots con uno u otro sistema operativo.

En el bajo nivel ARIA tiene un diseño de cliente-servidor: el robot está gobernado por un microcontrolador que hace las veces de servidor. Ese servidor establece un diálogo a través del puerto serie con la aplicación escrita utilizando ARIA, que actúa como cliente. En ese diálogo se envían al cliente las medidas de ultrasonido, odometría, etc. y se reciben las órdenes de actuación a los motores.

En cuanto al acceso al hardware, ARIA ofrece una colección de clases, que configuran una API articulada según muestra la figura 6. La clase principal `ArRobot` contiene varios métodos y objetos relevantes asociados. `Packet Receiver` y `Packet Sender` están relacionados con el envío y recepción de paquetes por el puerto serie con el servidor. Dentro de la clase `ArRangeDevices`, se tienen clases más concretas como `ArSonarDevice` o `ArSick` cuyos objetos contienen métodos que permiten a la aplicación acceder a las lecturas de los sensores de proximidad, tanto si estos son sónares o escaner láser. Además de la clase `DirectMotionCommands`, `ArRobot` contiene métodos para comandar consignas a los motores del robot. Por ejemplo,

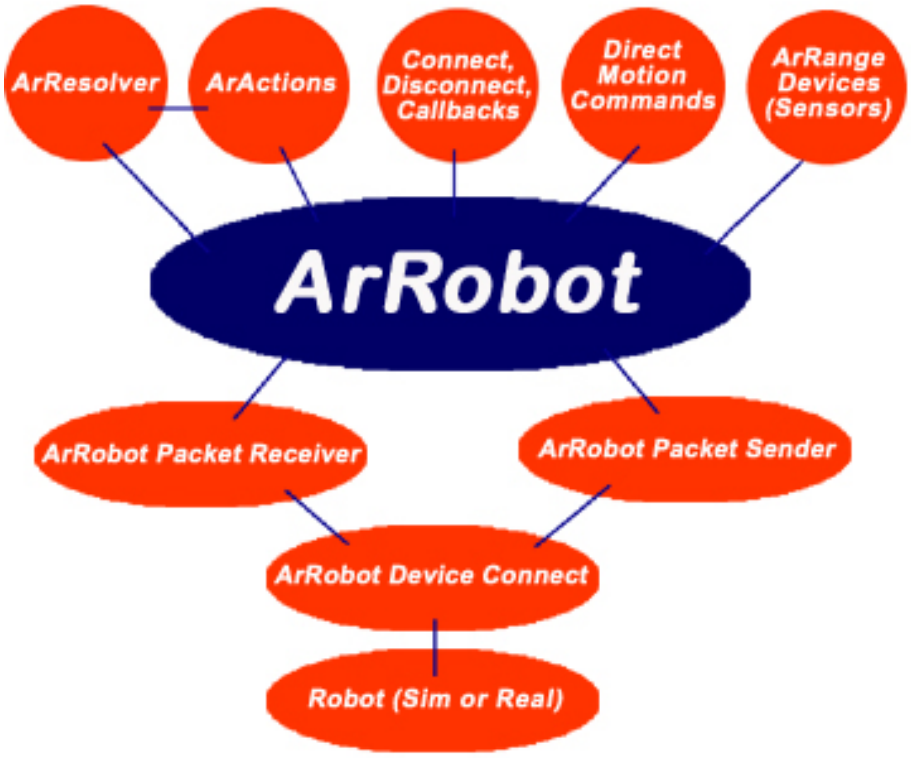


Figura 6: Estructura del API de ARIA

`ArRobot::setVel` comanda una velocidad de tracción y `ArRobot::setRotVel` permite ordenar cierta velocidad de rotación.

A diferencia de otras plataformas orientadas a objetos, los objetos de ARIA no son distribuidos, están ubicados en la máquina que se conecta físicamente al robot. No obstante, ARIA permite programar aplicaciones distribuidas utilizando `ArNetworking` para manejar comunicaciones remotas, que es un recubrimiento de los *sockets* del sistema operativo subyacente.

En cuanto a la multitarea, ARIA proporciona gran flexibilidad: las aplicaciones sobre ARIA pueden programarse monohilo o multihilo. En este último caso ARIA ofrece infraestructura tanto para hebras de usuario (`ArPeriodicTask`) como para hebras kernel (`ArThreads`). Las `ArThreads` son un recubrimiento de las `linux-pthreads` o `Win32-threads`. Para resolver problemas de sincronización y concurrencia ofrece mecanismos como los `ArMutex`, y las `ArCondition`.

ARIA contiene comportamientos básicos como la navegación segura, sin chocar contra los obstáculos. Sin embargo, no incluye funcionalidad común como la

construcción de mapas o la localización, que se venden por separado. Por ejemplo, ActivMedia vende el paquete `MAPPER` para la construcción de mapas, `ARNL` (*Robot Navigation and Localization*) para la navegación y localización, y `ACTS` (*Color-Tracking Software*) para la identificación de objetos por color y su seguimiento.

Miro

Miro[15][USEK02] es una plataforma orientada a objetos distribuidos para la creación de aplicaciones con robots. Ha sido desarrollada en la Univeridad de Ulm y está publicada bajo licencia GPL. En cuanto a la distribución de objetos, Miro sigue el estandard CORBA. De hecho utiliza la implementación TAO de CORBA, así como la librería ACE.

La arquitectura de Miro consta de tres niveles: capa de dispositivos, capa de servicios y el entorno de clases. La capa de dispositivos (*Miro Device Layer*) proporciona interfaces en forma de objetos para todos los dispositivos del robot. Es decir, sus sensores y actuadores se acceden a través de los métodos de ciertos objetos, que obviamente dependen de la plataforma física. Por ejemplo, la clase `RangeSensor` define un interfaz para sensores como los sónares, infrarrojos o láser. La clase `DifferentialMotion` contiene métodos para desplazar un robot con tracción diferencial. La capa de servicios (*Miro Services Layer*) proporciona descripciones de los sensores y actuadores como servicios, especificados en forma de CORBA IDL (*Interface Definition Language*). De esta manera su funcionalidad se hace accesible remotamente desde objetos que residen en otras máquinas interconectadas a la red, independientemente de su sistema operativo subyacente. Finalmente, el entorno de clases (*Miro Class Framework*) contiene herramientas para la visualización y la generación de históricos, así como módulos con funcionalidad de uso común en aplicaciones robóticas, como la construcción de mapas, la planificación de caminos o la generación de comportamientos.

Dentro de MIRO la aplicación robótica tiene la forma de una colección de objetos, remotos o no. Cada uno ejecuta en su máquina y se comunican entre sí a través de la infraestructura de la plataforma. Los objetos se pueden escribir en cualquier lenguaje que soporte el estándard CORBA, la plataforma no impone ninguno en concreto, aunque hay sido enteramente escrita en C++. Los objetos pueden ser tanto activos como pasivos. Por ejemplo, los objetos de sensores pueden enviar activamente sus datos (servicio activo), generando un evento cada vez que hay nuevas lecturas. Con ello supera los tradicionales métodos pasivos, que obligan a un muestreo continuo de los sensores y que no escalan bien a aplicaciones grandes.

[15] `http://smart.informatk.uni-ulm.de/MIRO`

Orocos

El proyecto OROCOS (Open RObot COntrol Software)[16], financiado por la Unión Europea, es una plataforma libre que nació en diciembre del 2000 con la aspiración de ser un "sistema operativo de software libre para robots"[Bru01]. En la actualidad se ha desgajado en dos subproyectos: *Open Realtime Control Services* y *Open Robot Control Software* propiamente dicho. El primero es un kernel para aplicaciones con varios bucles de control, como las robóticas, que cuida los aspectos de tiempo real. El kernel ofrece a las aplicaciones la posibilidad de hebras, eventos, etc. y el trabajo actual se orienta hacia una infraestructura distribuida de control basada en CORBA. El segundo subproyecto es un conjunto de clases que ofrecen funcionalidad genérica para máquinas-herramienta y robots: generación de trayectorias, cinemática y dinámica de objetos, algoritmos de control, de estimación e identificación, etc.

En conjunto, `Orocos` constituye un escenario orientado a objetos distribuidos para la programación de aplicaciones de robots. Sobre esta plataforma se ha programado, por ejemplo, un brazo manipulador de 6 grados de libertad con realimentación de fuerza.

Existen otras muchas plataformas libres para la programación de robots. Por ejemplo MARIE [CLM+04] aborda la integración de software de robots utilizando un paradigma de mediador entre aplicaciones heterogéneas. JDE [CM02] ofrece una plataforma cliente-servidor donde las observaciones y actuaciones se reciben y envían de dos servidores siguiendo cierto protocolo; y una plataforma software basada en hebras concurrentes. CARMEN [MRT03] es un conjunto de herramientas que facilitan la construcción de aplicaciones distribuidas reutilizando módulos existentes de navegación, construcción de mapas y localización.

5. Entornos de programación de robots

Dentro del mercado de robots móviles hay varios proveedores mayoritarios a nivel mundial: ActivMedia[17], iRobot, Robosoft[18] (distribuidora del robuter entre otros), K-Team[19], Sony, LEGO, etc. Cada uno de estos fabricantes incluye un entorno de programación para sus robots, y lo va actualizando a medida que aparecen nuevos diseños o nuevos dispositivos. Por ejemplo, ActivMedia comercializa los robots Pioneer, Amigo, etc.; para ellos ha desarrollado el entorno Saphira y recientemente el entorno ARIA. iRobot comercializa los robots B21, Magellan,

[16] `http://www.orocos.org`
[17] `http://www.activmedia.com`
[18] `http://www.robosoft.fr/`
[19] `http://www.k-team.com/`

Robot	*Procesador*	*Sistema Operativo*	*Plataforma*
LEGO RCX	micro Hitachi	BrickOS	no
Sony Aibo	RISC	Aperios	Open-R
ActivMedia Pioneer	micro + laptop	Linux/MS-Windows	ARIA
iRobot B21	PCs	Linux	Mobility
EyeBot	micro Motorola	ROBIOS	librerías
K-Team Kephera	micro Motorola		
Robuter	micro		librerías
Nomad	PC	Linux	librerías

Cuadro 2: Tabla resumen de robots y su entorno de programación

etc.; para ellos ha utilizado el entorno RAI, Beesoft, y recientemente Mobility. K-Team comercializa los robots Khepera y Koala, así como sus plataformas de programación y simulación.

5.1. Aibo de Sony

El robot Aibo se vende como mascota, con un programa que gobierna su comportamiento para exhibir conductas de perro (seguir pelota, buscar hueso, bailar, etc.) y le permite incluso aprender con el tiempo. A partir del verano de 2002 se abrió la posibilidad de programarlo y desde entonces se ha disparado su uso como plataforma de investigación. Por ejemplo, hay una categoría específica de la RoboCup en la que se compite al futbol robótico con estos perros. El robot tiene como sensores principales una cámara situada en la cabeza, distintos sensores de contacto (apoyo en las patas, en la cabeza y lomo) y odometría en cada una de sus articulaciones. Como actuadores principales tiene las patas, el cuello y la cola. El Aibo dispone de un procesador RISC de 64 bits, a 384 MHz (en el modelo ERS-7, 576 Mhz) y hardware de comunicaciones inalámbricas Wi-Fi.

El sistema operativo que regula los dispositivos hardware del Aibo es `Aperios` [20], un sistema operativo de tiempo real basado en objetos. Encima de ese sistema, Sony incluye la plataforma OPEN-R, que incorpora varios objetos específicos. Esos objetos establecen una interfaz de acceso al hardware del robot: hay objetos para el acceso básico a las imágenes y las articulaciones (objeto `OVirtualRobotComm`), para el manejo de la pila TCP/IP (objeto `ANT`) y para el manejo del altavoz y micrófono (objeto `OVirtualRobotAudioComm`). Por ejemplo, el objeto `OVirtualRobotComm` incluye el servicio `Effector` para ordenar movimientos a las articulaciones o encender los LEDS, el servicio `Sensor` para recibir la posición de las articulaciones y

[20] Antes conocido como Apertos, y antes conocido como Muse

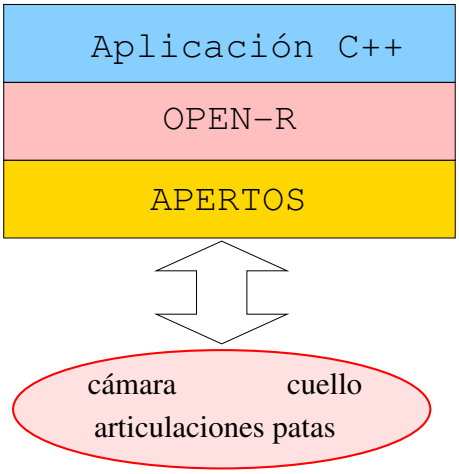


Figura 7: (a) Un robot Aibo (b) Arquitectura del software

el servicio `OFbkImageSensor` para acceder a la imagen capturada por la cámara. Del mismo modo, el objeto ANT ofrece los servicios `Send` y `Receive` para enviar y recibir paquetes a través de la conexión TCP/IP sobre la tarjeta de red inalámbrica.

La plataforma OPEN-R obliga a que la aplicación se escriba en C++. Su arquitectura software marca que la aplicación consista en uno o varios objetos, que utilizan los servicios de los objetos básicos y que pueden intercambiarse datos entre sí [MGCCM04, SB03]. Los objetos tienen un esqueleto fijo, con las funciones `DoInit`, `DoStart`, `DoStop` y `DoDestroy` [Son03a]. Adicionalmente, OPEN-R permite la multitarea y la programación orientada a eventos. Cada objeto incluye una única hebra, y todos los objetos de la aplicación ejecutan concurrentemente. Cada objeto recibe eventos y su propia ejecución viene marcada por los eventos que se generan y a los cuales va atendiendo. Por ejemplo, las comunicaciones entre objetos se materializan con memoria compartida para leer/escribir los datos y con los eventos *Notify* y *Ready* para sincronizar el acceso [Son03b].

Para programar al Aibo se utiliza el PC como entorno de desarrollo y un compilador cruzado para el procesador RISC. El programa generado se escribe desde el PC en una barra de memoria (*memory stick*), y ésta se inserta en el propio perro para su ejecución a bordo.

5.2. RCX de LEGO

Este robot se vende como juguete creativo y como plataforma educativa. Está compuesto por un procesador central (el ladrillo RCX) y un conjunto de piezas LEGO

sueltas que se ensamblan mecánicamente para montar el cuerpo. Esta presentación ofrece mucha facilidad y flexibilidad de montaje. Los sensores y los actuadores son también piezas LEGO, que se conectan a alguna de las tres entradas sensoriales y tres salidas de actuación del ladrillo RCX [FFH01]. Entre los sensores hay de luz, de choque, de rotación, etc. Los actuadores son principalmente motores. El procesador es un micro Hitachi H8/3297 de 8 bits, con 32 KB de memoria RAM, sobre el que se ejecuta un sistema operativo propio de LEGO.

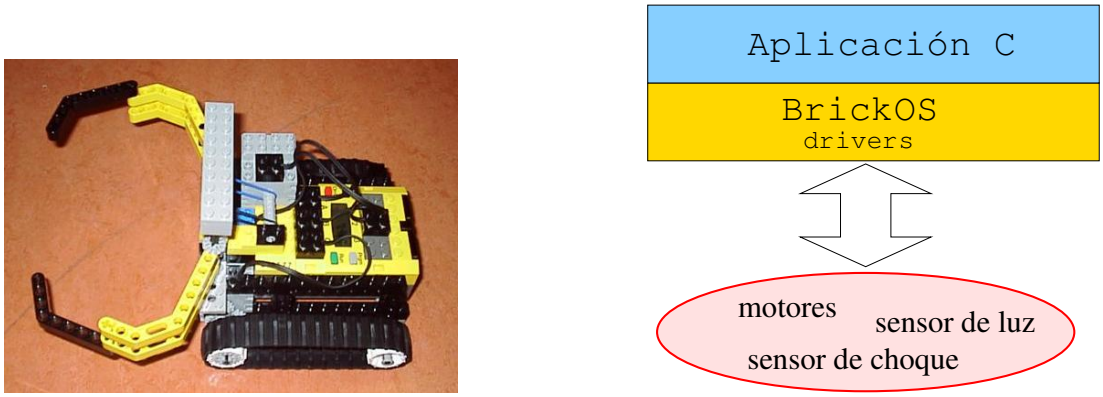


Figura 8: (a) Un robot LEGO (b) Arquitectura del software

Hay varias alternativas para programar este robot, y todas ellas utilizan el PC como entorno de desarrollo y descargan el programa en el RCX a través del enlace de infrarrojos de que dispone (bien por puerto serie o por USB). LEGO ofrece un entorno visual de programación, llamado código RCX (*RCX-code*), orientado a los niños. Como indicamos en la sección 2.4, este lenguaje posee bloques gráficos a modo de instrucciones (para leer observaciones sensoriales, para ordenar consignas a los motores) y bloques con estructuras de control (condicionales, bucles, etc.). Otro entorno gráfico con el que se puede programar es Robolab[21], una adaptación de LabView, de National Instruments. Una posibilidad más de programación es el lenguaje NQC, creado por Dave Baum [Bau00]. Este lenguaje textual es una variante recortada de C que dispone de instrucciones propias para el acceso a sensores y actuadores. Tanto los programas en código RCX como en NQC compilan a código intermedio que se interpreta en tiempo de ejecución en el propio ladrillo.

Debido a la popularidad de este robot, algunos aficionados han desarrollado sistemas operativos alternativos, que reemplazan al nativo de LEGO y exprimen mejor las posibilidades del hardware. El sistema operativo libre BrickOS[22] [23], de-

[21] `http://www.ni.com/company/robolab.htm`
[22] `http://brickos.sourceforge.net/`
[23] antes conocido como LegOS

sarrollado por Markus L. Noga [Nie00], permite la programación del ladrillo en lenguaje C. En este caso el compilador cruzado genera código no interpretado, que ejecuta directamente sobre el micro, por lo que se gana en eficiencia temporal. Del mismo modo, el sistema operativo LeJOS[24] permite la creación de aplicaciones en JAVA. Todos estos sistemas operativos, además del original de LEGO, ofrecen la capacidad de multitarea.

En todos los casos el entorno de programación es muy básico, de sistema operativo. Como es un robot muy limitado en cuanto a número y calidad de sensores y de actuadores, entonces los comportamientos generables con él tienden a ser sencillos. De ahí que no sea necesaria una plataforma de desarrollo y las aplicaciones se puedan hacer sin problemas programando directamente sobre la API del sistema operativo. Adicionalmente, no existe un simulador completo para este robot, pues la simulación correcta depende enormemente del montaje mecánico, que está muy abierto.

5.3. Pioneer de ActivMedia

El robot Pioneer es un robot de tamaño mediano, según se aprecia en la figura 9, con 26 cm de radio y unos 22 cm de alto. Consta de una base móvil con un par de motores, sensores de ultrasonido, odómetros y opcionalmente, sensores de contacto y sensor láser de proximidad [Act03]. En ella un microcontrolador (Siemens 88C166 o Hitachi H8S, según modelos) se encarga de recoger las medidas sensoriales y enviar las órdenes de movimiento a los motores. Este micro está gobernado por un sistema operativo de bajo nivel (P2OS o AROS, según modelos). En cuanto a la arquitectura de procesadores, el montaje más frecuente del Pioneer incorpora un ordenador portátil a bordo, en el que se ejecutan las aplicaciones, y en él un sistema operativo generalista como Linux. Las aplicaciones dialogan con el microcontrolador a través de puerto serie utilizando un protocolo propio llamado SIP. El robot Pioneer ha supuesto un éxito de ventas entre los robots programables y por ello se encuentra soportado en muchas plataformas software.

Para crear aplicaciones, ActivMedia ofrece la plataforma ARIA, publicada con licencia GPL. Como vimos en la sección 4.5, esta plataforma establece una interfaz abstracta de acceso al hardware y una arquitectura software orientada a objetos para las aplicaciones. ARIA obliga a que las aplicaciones se escriban en C++, que es lenguaje de los objetos de acceso al hardware. La plataforma no incluye ninguna funcionalidad robótica común, se vende por separado como una colección de paquetes software que se integran fácilmente sobre ARIA.

El fabricante ActivMedia incluye el simulador bidimensional `SRIsim` acompañando a ARIA. Como ya mencionamos en la sección 2.3, este simulador bidi-

[24] `http://lejos.sourceforge.net/`

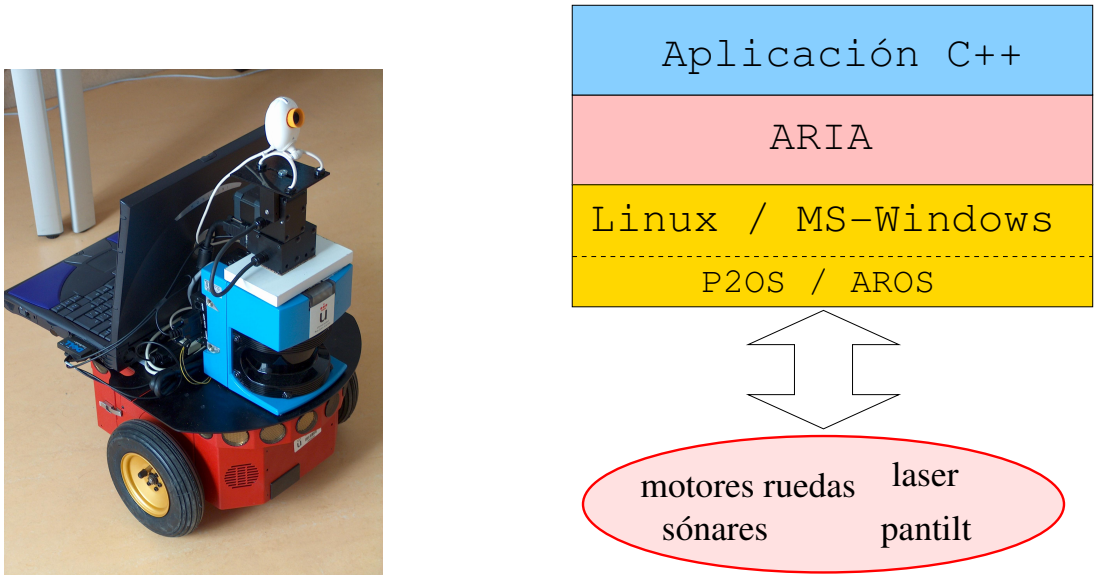


Figura 9: (a) Un robot Pioneer (b) Arquitectura del software

mensional permite emular un único Pioneer en un entorno estático y sus sensores más comunes: odometría, sónares y láser.

ARIA se ha implementado tanto sobre Linux como sobre MS-Windows, y es un claro ejemplo de portabilidad: la aplicación robótica funciona tanto si el sistema operativo subyacente en el PC a bordo es Linux o MS-Windows, pues la plataforma ofrece la misma interfaz de programación sobre ambos.

5.4. B21 de iRobot

El robot B21 lo comercializaba la empresa RWI (*Real World Interface*), que posteriormente fue absorbida por iRobot. Como se aprecia en la figura 10, el B21 es un robot cilíndrico aproximadamente 1 metro de altura, y está dotado típicamente de una base motora con varios micros y dos PCs [RWI94] ejecutando Linux. En la base motora se ejecuta el sistema operativo `rFLEX`, que gobierna los motores, recoge las medidas de algunos sensores y habla con las aplicaciones del PC a través del puerto serie. El B21 está dotado de varios sensores de contacto, tres cinturones de sensores de infrarrojo, uno de ultrasonidos y odómetros.

La plataforma de desarrollo para este robot ha ido evolucionando con el paso de los años. Inicialmente se denominaba RAI (*Robot Application Interface*) [RWI96], y estaba basada en el planificador `RAI-scheduler`, desarrollado en la Universidad de Brown. Esta plataforma sólo funcionaba en linux e imponía a las aplicaciones una estructura iterativa, debían programarse con varias hebras concurrentes sin

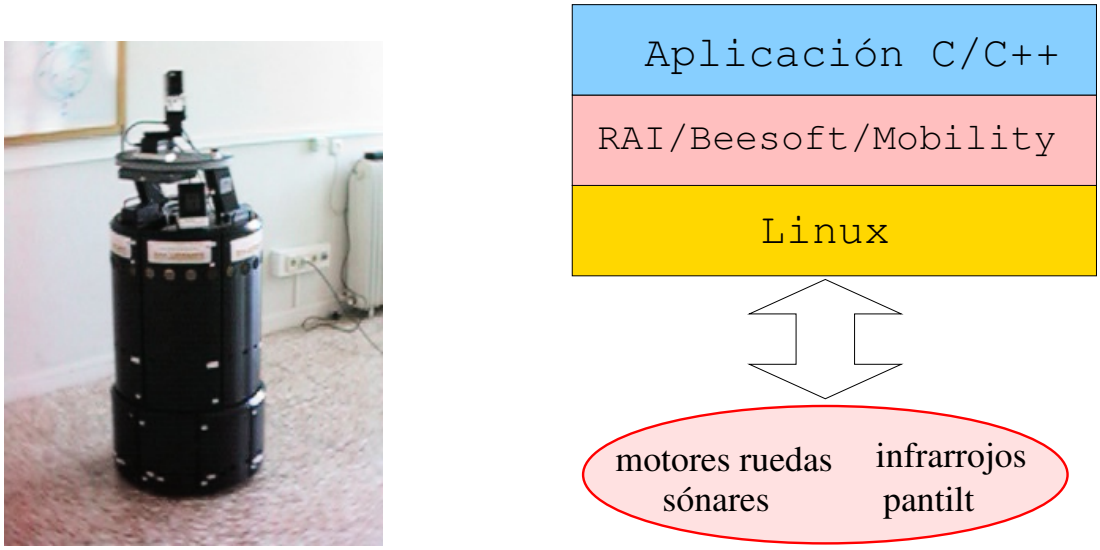


Figura 10: (a) Un robot B21 (b) Arquitectura del software

desalojo. RAI ofrecía sólo acceso hardware, no funcionalidad común. La API de acceso al hardware era la misma para las aplicaciones locales que para las remotas (a través de unos servidores y una librería para clientes). Posteriormente se reescribió completamente y pasó a llamarse Beesoft.

Actualmente, la plataforma es Mobility[25][RWI99]. Mobility ofrece un entorno de programación orientada a objetos, distribuido. Al igual que otras plataformas recientes se apoya en el estandard CORBA para la distribución de los objetos. Los interfaces de los objetos se definen en ficheros separados empleando el CORBA IDL.

Mobility incluye el modelo de objetos ROM (*Robot Object Model*), donde cada objeto es una unidad de software con una identidad, una interfaz y un estado. El modelo describe al robot como una colección de componentes. Los componentes incluyen sensores como los de odometría, de contacto, escáner láser, etc. y actuadores. Los comportamientos deseables del robot y los procesos perceptivos también se articulan como componentes. En esta línea se incluyen componentes como `deambular`, `evitar-colisión`, `seguir-paredes` y `ir-a-punto`. Los componentes pueden ser activos o pasivos. Los activos incluyen un flujo de control propio y son útiles para el envío de nuevos datos de estado, la implementación de un comportamiento con control realimentado, etc.

La plataforma incluye el marco de clases MCCF (*Mobility C++ Class Fra-*

[25]En el momento de escribir este artículo iRobot ha cesado la venta y soporte de robots de investigación como el B21 y B14, centrando su actividad en comercializar Roomba y PackBot

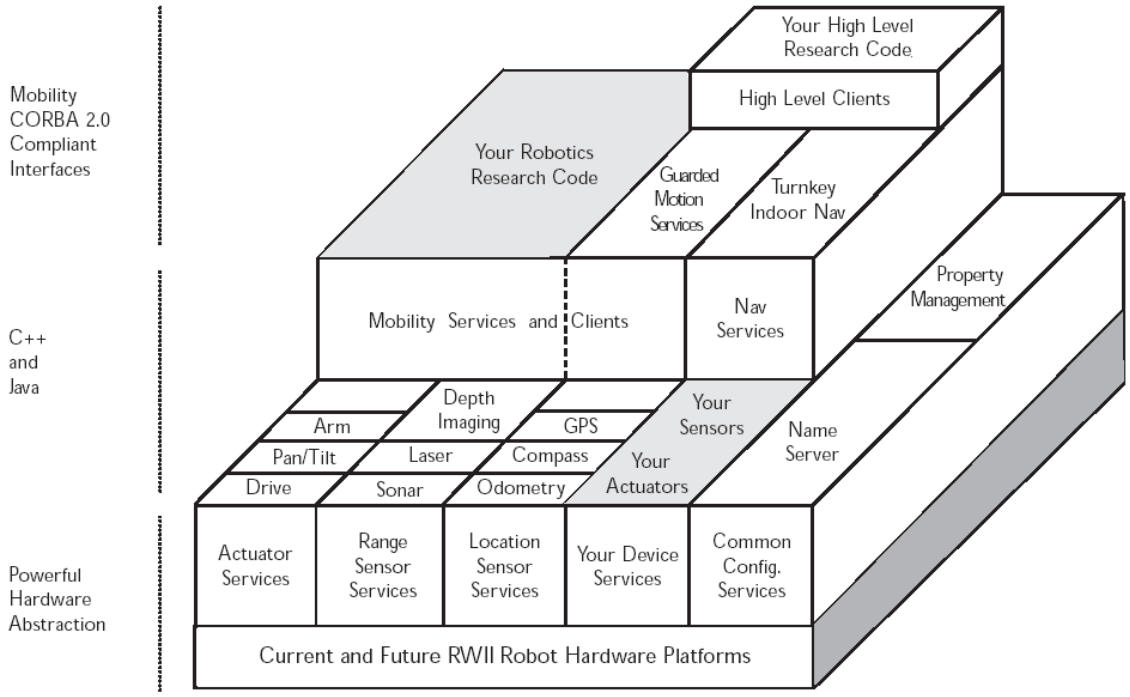


Figura 11: Entorno de programación Mobility

mework) para la programación de aplicaciones en C++, la inserción de nuevos sensores y actuadores, etc. Como muestra la figura 11, la plataforma ofrece en su capa inferior una abstracción del hardware orientada a objetos, sobre la cual se pueden ya programar aplicaciones (segunda capa). Encima del nivel intermedio, en la capa superior, se pueden programar aplicaciones de alto nivel que aprovechan ya la distribución que ofrece CORBA.

Mobility ofrece soporte para multirrobots, de manera que varios robots diferentes se pueden incorporar a la jerarquía de objetos del sistema conjunto, donde otros objetos adicionales pueden almacenar información común útil para todos ellos (e.g. mapas). La plataforma incluye la herramienta de visualización y gestión MOM (*Mobility Object Manager*). Además se pueden integrar en ella el paquete de navegación `Nav` y de simulación `Sim`, que se venden por separado como módulos opcionales.

6. Conclusiones y perspectivas

El mercado de los robots tiene actualmente una pujanza creciente, cada vez hay más movimientos empresariales alrededor de la robótica. Los robots Aibo y LEGO son sólo dos ejemplos de éxitos de ventas, superando cada uno las cien mil unidades vendidas. Los prototipos de investigación cada vez tienen mayor funcionalidad. Los robots han dejado de utilizarse exclusivamente en fábricas y se acerca su uso en el hogar, como robots de entretenimiento, o como robots de servicio.

Tener robots realizando autónomamente tareas depende de su construcción mecánica y fundamentalmente de su programación. La inteligencia y la autonomía del robot residen básicamente en su programa, que es el que determina cómo se comporta, el que decide qué consignas ordenar a los actuadores en función de las observaciones sensoriales y los objetivos del comportamiento. Como vimos en la sección 2, la programación de robots es un arte difícil que presenta ciertos requisitos genuinos frente a la programación de otras aplicaciones más clásicas. Por ejemplo, los programas de robots están empotrados en la realidad física a través de sensores y actuadores, y suelen tener requisitos de tiempo real y vivacidad. Adicionalmente, no hay estándares consolidados en la programación de robots. Esto se debe en parte a la heterogeneidad del hardware, y en parte a la inmadurez del mercado de los robots programables. Esta ausencia de estándares está frenando las posibilidades de crecimiento del mercado robótico, pues no hay una base firme y sólida sobre la que crear aplicaciones y reutilizar código.

Con el paso de los años, el modo de programar los robots ha ido evolucionando y se ha ido articulando en tres niveles diferenciados: sistemas operativos, plataformas de desarrollo y aplicaciones concretas. En la sección 5 hemos descrito el entorno de programación de cuatro robots reales muy difundidos en la comunidad: Aibo, RCX, Pioneer y B21. En los cuatro hemos corroborado los tres niveles presentados en este artículo para el software, viendo los sistemas operativos concretos y las diferentes plataformas de desarrollo que se utilizan con cada uno de ellos.

Según describimos en la sección 3, el *sistema operativo del robot* proporciona el acceso básico a los recursos hardware, entre los que se incluyen los sensores y los actuadores, así como los dispositivos de comunicaciones e interacción. El hardware de los robots evoluciona con mucha rapidez, y en los últimos años hemos observado la incorporación de la visión y del láser como sensores adicionales. Otra tendencia confirmada es la extensión del ordenador personal (bien PC de sobremesa, bien PC portátil) como procesador principal de los robots. Más allá de los sistemas operativos dedicados, esta incorporación del PC ha traido la irrupción en los robots de sistemas operativos generalistas, como Linux o MS-Windows, y el uso creciente de lenguajes de alto nivel con sus herramientas asociadas.

Las *plataformas de desarrollo* han surgido para facilitar el desarrollo de aplicaciones, tanto las creadas por los propios fabricantes, como las creadas por grupos

de investigación. Como explicamos en la sección 4, entre las distintas plataformas existentes hemos identificado varias líneas comunes: uniforman y simplifican el acceso al hardware, ofrecen una arquitectura software determinada e incorporan funcionalidad robótica común como la navegación, la construcción de mapas o la localización. A grandes rasgos, las plataformas tratan de fijar una base estable sobre la cual desarrollar aplicaciones robóticas, y eso supone un paso hacia la madurez de la programación de robots. Primero, favorecen la portabilidad de aplicaciones entre robots diferentes. Segundo, fomentan la reutilización de código, con lo cual disminuyen los tiempos de desarrollo de las nuevas aplicaciones. Y tercero, con su arquitectura software ofrecen una manera de organizar el código, lo cual permite afrontar la complejidad creciente de las aplicaciones de robots.

Las plataformas actuales recogen las tendencias recientes más firmes de la programación de robots: distribución en varios procesos concurrentes y la programación orientada a objetos. Algunas de las plataformas presentadas utilizan un modelo cliente-servidor para la distribución (PSG, JDE) mientras que la mayoría generalizan esa distribución a un conjunto de objetos remotos, que se comunican entre sí utilizando una infraestructura como CORBA (Miro, Mobility, Orocos).

La pujanza actual del mercado de los robots programables y los resultados conseguidos en los últimos años en prototipos apuntan a un futuro prometedor de la robótica. Sin embargo, dejando a un lado las expectativas idealistas, el grado de autonomía y de fiabilidad que seamos capaces de conseguir en los robots determinará fundamentalmente el crecimiento de ese mercado. Por ejemplo, la introducción de los robots en los hogares exige niveles mucho mayores de fiabilidad e independencia del entorno de los conseguidos hasta la fecha. Con mayores grados de autonomía las tareas en las que se podrán aplicar robots crecerán significativamente.

Creemos que los avances en niveles de autonomía y de fiabilidad pasan por superar la heterogeneidad actual en el software de los robots, por disponer de entornos estables de programación, y como en otras disciplinas, por la reutilización de conocimiento. La aparición de las plataformas de desarrollo supone un paso hacia adelante en ese largo camino. Sin embargo, aunque la necesidad de estándares en robótica ha sido identificada por muchos autores, en la actualidad existen *muchas* plataformas diferentes, tal vez demasiadas, reflejo de que no hay aún un estandard universalmente admitido. Hemos presentado una muestra representativa tanto de las de software libre (ARIA, PSG, Miro, etc.) como de las propietarias (Mobility, OPEN-R, ERSP, etc.). Habrá que ver cuales sobreviven de aquí a unos años, y eso dependerá enormemente de cuantos usuarios y desarrolladores sean capaces de atraer cada una de ellas.

Quizá la evolución en el corto plazo converga a la existencia de una o dos plataformas estables, al menos para uniformar el acceso al hardware. Por ejemplo,

se han definido las interfaces de movimiento y de los sensores más frecuentes, que son válidas para un conjunto amplio de robots de interiores con ruedas y no muy distintas entre las diferentes plataformas. La coincidencia está, al menos técnicamente, cerca. Esa convergencia permitiría enfocar mejor los esfuerzos en cómo organizar el código de las aplicaciones para generar comportamiento en los robots, que sigue siendo un reto de primera magnitud. Hay mucho por experimentar y crecer en esta línea, y tal vez por ella se podrán alcanzar las cotas de autonomía y fiabilidad necesarias si queremos tener robots, por ejemplo, realizando tareas automáticamente en los hogares.

Agradecimientos

El autor quiere agradecer a Vicente Matellán y a Rodrigo Montúfar las correcciones y las fructíferas discusiones alrededor de este artículo.

Referencias

[Act99] ActivMedia. Saphira operations and programming manual. Technical Report version 6.2, ActivMedia Robotics, August 1999.

[Act02] ActivMedia. ARIA reference manual. Technical Report version 1.1.10, ActivMedia Robotics, November 2002.

[Act03] ActivMedia. Pioneer 3 & Pioneer 2 H8-Series Operations manual. Technical Report version 3, ActivMedia Robotics, August 2003.

[Bau00] Dave Baum. *Dave Baum's definitive guide to LEGO MINDSTORMS*. APress, 2000.

[BFG+97] R. Peter Bonasso, R. James Firby, Erann Gatt, David Kortenkamp, David P. Miller, and Marc G. Slack. Experiences with an architecture for intelligent reactive agents. *Journal of Experimental and Theoretical AI*, 9(2):237–256, 1997.

[BM03] Geoffrey Biggs and Bruce MacDonald. A survey of robot programming systems. In Jonathan Roberts and Gordon Wyeth, editors, *Proceedings of the 2003 Australasian Conference on Robotics and Automation*, December 2003.

[Bro86] Rodney A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, 2(1):14–23, March 1986.

[Bru01] Herman Bruyninckx. Open robot control software: the OROCOS project. In *Proceedings of the 2001 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-01)*, volume 3, pages 2523–2528, Seoul (Korea), May 2001.

[Brä03] Thomas Bräunl. *Embedded robotics*. Springer Verlag, 2003.

[Cañ03] José M. Cañas. *Jerarquía Dinámica de Esquemas para la generación de comportamiento autónomo*. PhD thesis, Universidad Politécnica de Madrid, 2003.

[CLM+04] Carle Côté, Dominic Létourneau, François Michaud, Jean-Marc Valin, Yannick Brosseau, Clément Raïevsky, Mathieu Lemay, and Victor Tran. Code reusability tools for programming mobile robots. In *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-04)*, Sendai (Japan), September 2004.

[CM02] José M. Cañas and Vicente Matellán. Dynamic schema hierarchies for an autonomous robot. In José C. Riquelme y Miguel Toro Francisco J. Garijo, editor, *Advances in Artificial Intelligence - IBERAMIA 2002*, volume 2527 of *Lecture notes in artificial intelligence*, pages 903–912. Springer, 2002.

[Fed94] Christopher Fedor. TCX an interprocess communication system for building robotic architectures. Technical report, Robotics Institute, Carnegie Mellon University, January 1994.

[FFH01] Mario Ferrari, Giulio Ferrari, and Ralph Hempel. *Building robots with LEGO MINDSTORMS: The Ultimate Tool for Mindstorms Maniacs*. Syngress, 2001.

[Fir94] R. James Firby. Task networks for controlling continuous processes. In *Proceedings of the 2nd International Conference on AI Planning Systems AIPS'94*, pages 49–54, Chicago, IL (USA), June 1994.

[GVH03] Brian P. Gerkey, Richard T. Vaughan, and Andrew Howard. 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), June 2003.

[HHB03] Myron Hattig, Ian Horswill, and Jim Butler. Roadmap for mobile robot specifications. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*, volume 3, pages 2410–2414, Las Vegas (USA), October 2003.

[KM98] Kurt Konolige and Karen L. Myers. The Saphira architecture for autonomous mobile robots. In David Kortenkamp, R. Peter Bonasso, and Robin Murphy, editors, *Artificial Intelligence and Mobile Robots: case studies of successful robot systems*, pages 211–242. MIT Press, AAAI Press, 1998. ISBN: 0-262-61137-6.

[LOIBGL01] Miguel Ángel Lozano Ortega, Ignacio Iborra Baeza, and Domingo Gallardo López. Entorno Java para simulación y control de robots móviles. In *Actas de la IX Conferencia de la Asociación Española Para la Inteligencia Artificial, CAEPIA 2001*, pages 1249–1258, Gijón, November 2001.

[MGCCM04] Francisco Martín, Rafaela González-Careaga, José M. Cañas, and Vicente Matellán. Programming model based on concurrent objects for the AIBO robot. In *Proceedings of the XII Jornadas de Concurrencia y Sistemas Distribuidos*, pages 367–379. 2004.

[MRT03] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: the carnegie mellon navigation (CARMEN) toolkit. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*, volume 3, pages 2436–2441, Las Vegas (USA), October 2003.

[Nie00] Stig Nielsson. Introduction to the LegOS kernel. Technical report, 2000.

[RWI94] Real World Interface RWI. B21 robot system manual. Technical report, Real World Interface, Inc., 1994.

[RWI96] Real World Interface RWI. System software and RAI-1.2.2 documentation. Technical report, Real World Interface, Inc., 1996.

[RWI99] Real World Interface RWI. Mobility Robot Integration software, User's guide. Technical Report version 1.1, IS Robotics, Inc, May 1999.

[SA98] Reid Simmons and David Apfelbaum. A Task Description Language for robot control. In *Proceedings of the 1998 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-98)*, volume 3, pages 1931–1937, Victoria (Canada), October 1998.

[SB03] Francois Serra and Jean-Christophe Baillie. Aibo programming using OPEN-R SDK. tutorial. Technical report, ENSTA (Francia), 2003.

[SGFB97] Reid Simmons, Richard Goodwin, Christopher Fedor, and Jeff Basista. Programmer's Guide to TCA. Technical Report version 8.0, School of Computer Science, Carnegie Mellon University, May 1997.

[Sim94] Reid G. Simmons. Structured control for autonomous robots. *IEEE Journal of Robotics and Automation*, 10(1):34–43, February 1994.

[Son03a] Sony. OPEN-R SDK, Level2 reference guide. Technical Report 20030201-E-003, Sony Corporation, 2003.

[Son03b] Sony. OPEN-R SDK, Programmer's guide. Technical Report 20030201-E-003, Sony Corporation, 2003.

[USEK02] Hans Utz, Stefan Sablatnög, Stefan Enderle, and Gerhard Kraetzschmar. Miro – middleware for mobile robot applications. *IEEE Transactions on Robotics and Automation, Special Issue on Object-Oriented Distributed Control Architectures*, 18(4):493–497, August 2002.

[VGH03] Richard T. Vaughan, Brian P. Gerkey, and Andrew Howard. 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, Las Vegas (USA), October 2003.

[WMT03] Evan Woo, Bruce A. MacDonald, and Félix Trépanier. Distributed mobile robot application infraestructure. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-03)*, volume 2, pages 1475–1480, Las Vegas (USA), October 2003.