UNIVERSIDAD DE SALAMANCA

Departamento de Informática y Automática

Facultad de Ciencias

Integración de robots móviles en el servidor Multi-Robot '*Player*'

Alejandro Morales Sánchez

Julio 2004

D. **Vidal Moreno Rodilla**, Prof. Titular de Ingeniería de Sistemas y Automática de la Universidad de Salamanca y Dña. **Belén Curto Diego**, Pfa. Titular de Ingeniería de Sistemas y Automática de la Universidad de Salamanca

CERTIFICAN:

Que el trabajo de grado que se recoge en la presente memoria, titulada **Integración de Robots móviles en el servidor multi-robot Player** y presentada por Don **Alejandro Morales Sánchez** para optar al título de Grado por la Universidad de Salamanca, ha sido realizado bajo su dirección en el Area de Ingeniería de Sistemas y Automática del Departamento de Informática y Automática de la Universidad de Salamanca.

Salamanca, 20 de Julio de 2004

D. **Vidal Moreno Rodilla**
Prof. Titular de Universidad
Área de Ingeniería de Sistemas y Automática
Universidad de Salamanca

Dña. **Belén Curto Diego**
Profa. Titular de Universidad
Área de Ingeniería de Sistemas y Automática
Universidad de Salamanca

Agradecimientos

Quisiera agradecer a mis tutores Doña Belén Curto Diego y Don Vidal Moreno Rodilla su paciencia y colaboración. Tambíen quiero agradecer a Andrés Vicente Lober el gran trabajo realizado en la construcción de los robots móviles y al resto de personas que me han ayudado a realizar este trabajo.

Así mismo, quiero agradecer a mi família su inestimable apoyo y a Noe su paciencia y el ánimo que he recibido de ella.

A mi abuelo Andrés

Índice general

Índice de figuras

1

Introducción

En este capítulo se va a realizar una introducción a los conceptos clave relacionados con el objetivo de este trabajo, como son la Robótica y con más profundidad la Robótica Móvil, sus aplicaciones reales, los sistemas multi-robot y las distintas posibilidades que existen de desarrollar programas de control para sistemas robóticos y multi-robot.

La Robótica tiene una historia reciente. El término Robot proviene de la palabra checa *robota* que significa "servidumbre, trabajos forzados", y se comenzó a utilizar a partir del estreno de la obra de teatro "Los Robots Universales de Rossum"(1923) de Karen Capek.

Gracias al desarrollo de las tecnologías de la información así como de la mecánica y la electrónica, se posibilitó el desarrollo de los primeros robots en

la década de los 50. Su intruducción en la industria a principios de los 60 como elementos manipuladores, suscitó el interes de los investigadores para lograr manipuladores más rápidos, precisos e incluso móviles. Posteriormente en la década de los 70 se amplía el campo de actuación de los robots, continuando la evolución de los mismos y viéndose especialmente beneficiada por la investigación en Inteligencia Artificial, que combinada con la Robótica permite la creación de robots autónomos con movimientos avanzados en espacios interiores y exteriores, visión artificial e inteligencia.

1.1. Robótica móvil

El desarrollo de robots móviles surge en los años 70 como banco de pruebas para estudiar técnicas de Inteligencia Artificial y ante la necesidad de ampliar el campo de actuación de la robótica en el ámbito de la industria, ya que inicialmente estaba limitado al uso de estructuras ancladas en uno de sus extremos. En esa década comenzaron a utilizarse industrialmente vehículos autónomos para el transporte de distintos elementos en la cadena productiva, especialmente en la industria del automóvil. La gran mayoría de los vehículos, incluso a día de hoy, son filoguiados. Éstos circulan siguiendo caminos fijos predispuestos en la planta, es decir el recorrido de este tipo de vehículos está restringido a trayectorias preestablecidas, lo que hace que sean poco flexibles tanto para la elección de caminos alternativos como para la evitación de posibles obstáculos. Como contrapartida a los vehículos afiloguiados, se comenzó a investigar la posibilidad de realizar vehículos con más autonomía, capaces de desplazarse desde un punto (origen) a otro (destino) evitando los obstáculos que encuentren a su paso (navegación). Este tipo de vehículos son conocidos como robots móviles autónomos.

La arquitectura básica de un robot móvil está compuesta por un sistema de actuadores de locomoción que le permiten desplazarse, un sistema de sensores que le permiten percibir aspectos de su entorno y un módulo controlador que alberga el software de control del robot.

Una definición correcta de robot móvil plantea la capacidad de movimiento sobre entornos no estructurados, de los que se posee un conocimiento incierto, mediante la interpretación de la información suministrada a través de sus sensores y del estado actual del vehículo.

Los robots autónomos son sistemas mecánicos que pueden interactuar con un ambiente obteniendo información de él, y tienen un comportamiento individual aunque éste sea socializado. Su comportamiento se define por sus percepciones y no requiere de intervención humana.

Los robots móviles autónomos son capaces de percibir el entorno e incluso de realizar una representación de él. Se caracterizan por una conexión inteligente entre las operaciones de percepción y acción, las cuales definen su comportamiento y le permiten llegar a la consecución de los objetivos programados sobre entornos con cierta incertidumbre.

El grado de autonomía depende en gran medida de la facultad del robot para abstraer el entorno y convertir la información obtenida en órdenes, de tal modo que, aplicadas sobre los actuadores del sistema de locomoción, garantice la realización eficaz de su tarea. De este modo, las dos grandes características que lo alejan de cualquier otro tipo de vehículo se relacionan a continuación (Lozano-Pérez, 1.990):

- La percepción, que determina la relación del robot con su entorno de trabajo, mediante el uso de los sensores de a bordo.
- El razonamiento, que determina las acciones que se han de realizar en cada momento, según el estado del robot y su entorno, para alcanzar las metas asignadas.

De este modo, la capacidad de razonamiento del robot móvil autónomo se traduce en la planificación de unas trayectorias seguras que le permitan la consecución de los objetivos encomendados.

Los robot móviles deben tener características de maniobrabilidad, controlabilidad, capacidad de tracción, estabilidad, eficiencia y consideraciones de navegación como la odometría.

Pueden tener multitud de sistemas de tracción como son la configuración diferencial, en triciclo, Ackerman, síncrona, ruedas omnidireccionales, cadenas, patas, reptantes, etc, según el medio en el que se vayan a mover.

El uso de robots está justificado en aplicaciones en las que se realizan tareas molestas o arriesgadas para el trabajador humano. Entre ellas, el transporte de material peligroso, las excavaciones mineras, la limpieza industrial o la inspección de plantas nucleares son ejemplos donde un robot móvil puede desarrollar su labor y evita exponer, gratuitamente, la salud del trabajador. Otro grupo de aplicaciones donde este tipo de robots complementa la actuación del operador lo componen las labores de vigilancia, de inspección o asistencia a personas incapacitadas. Así mismo, en aplicaciones de teleoperación donde existe un retraso sensible en las comunicaciones, resulta interesante el uso de vehículos con cierto grado de autonomía. No obstante y como se muestra en la sección 1.1.3, hoy en día existen robots móviles para uso lúdico, competitivo,...

Una de sus principales dificultades es determinar su posición dentro de una zona determinada. Se pueden realizar estimaciones según una posición inicial por métodos odométricos, o bien utilizar sistemas de medidas absolutas respecto a una referencia externa, como balizas, satélites, etc.

El mecanismo de posicionamiento se basa en el sistema de percepción de que disponga. Existen multitud de sistemas sensoriales como sonar, laser, infrarrojos, radares, etc, aunque no es el único fin de estos, ya que la percepción del robot le ha de servir también para interaccionar con los elementos del ambiente.

Todo robot móvil se debe plantear las dos preguntas basicas: ¿dónde estoy? ¿cómo llego al objetivo? [JBF96]. Sin duda para resolver estas cuestiones se debe recurrir a la planificación de movimientos por medio de los actuadores del sistema y a técnicas de planificación, manteniendo quizá una representación del ambiente por el que se mueve.

1.1.1. Arquitectura de un Robot Móvil

Todo robot móvil debe estar provisto de una serie de elementos físicos que le doten de la capacidad suficiente para realizar las tareas para las que sea programado. Estos elementos pueden agruparse en:

- Sistema de actuadores.
- Sistema sensorial.
- Controladores de los dispositivos sensores y actuadores.
- Módulo controlador o Computador de control.

A continuación pasamos a describir cada uno de estos grupos con mas detalle.

El **Sistema de Actuadores** lo constituyen una serie de elementos físicos que interactuan con el entorno, denominados efectores (p.e: ruedas, patas, brazos, cepillo...) y un conjunto de dispositivos eléctricos, hidráulicos o neumáticos, que van conectados a los efectores y se encargan de actuar sobre ellos (p.e: motores eléctricos, dispositivos hidráulicos, dispositivos neumáticos...). Normalmente, auque son conceptos diferentes, se suele denominar al conjunto de un actuador y su correspondiente efector como actuador simplemente. Todo robot móvil de tener al menos un sistema de actuadores de locomoción que le permitan desplazarse, aunque adicionalmente suelen tener otros actuadores para la realización de tareas específicas (limpiar, empujar, lanzar...) para las que ha sido diseñado el robot.

El **Sistema Sensorial** está formado por un conjunto de sensores que constituyen el sistema de percepción del robot. Los sensores son dispositivos electrónicos que miden cantidades físicas de propiedades (distancia, sonido, altitud, velocidad, inclinación...). Proporcionan señales que deben ser convertidas en valores o símbolos. Distintos sensores pueden servir para medir la misma propiedad. Los sensores son limitados ya que pueden devolver valores imprecisos debido a la naturaleza del entorno en el que trabajen, provocadas por ruidos, efectos de luz, etc. Existen varias clasificaciones de los dispositivos sensores:

- Desde el punto de vista del tipo de información que proporcionan se clasifican en internos y externos.

  Internos: proporcionan información sobre el propio robot, relativa a su posición, velocidad, aceleración, etc.

  Externos: proporcionan información sobre el entorno que rodea al robot, relativa a la proximidad, visión, presencia, etc.

- Desde el punto de vista de la capacidad de cálculo que requiera el tratamiento de la información que proporcionan, se clasifican en sencillos y complejos.

  Sencillos: el tratamiento de la información que proporcionan no implica mucha capacidad de cálculo. Necesitan cierta electónica y cierta programación (acceso a los puertos e interpretación básica de la información recibida). Ejemplos: sensores de interruptor (bumpers), sensores de luz (células fotoeléctricas), etc.

  Complejos: el tratamiento de la información que proporcionan implica un capacidad de cálculo mucho mayor. Ejemplo: sensores de visión artificial, sensores de interpretación de sonidos, etc.

- En cuanto a la técnica empleada para realizar la medición del los valores del entorno, se clasifican en activos y pasivos.

  Activos: producen un estímulo y miden su interacción con el entorno. El estimulo puede ser una señal luminosa, de ultrasonido, etc. Consumen más energía que los pasivos. Ejemplos: sensores de ultrasonido, sensores láser, sensores de infrarrojos...

  Pasivos: simplemente miden características del entorno. Ejemplos: interruptores (bumpers de contacto), sensores de luz, botones,...

Todos los dispositivos sensores y actuadores deben tener asociado un **controlador** que permita comunicarse con ellos. En el caso de los sensores transforma la información recibida de manera que pueda ser interpretada y en el de los actuadores se encarga normalmente de convertir las señales digitales en eléctricas que provocan el funcionamiento del actuador.

Los robots móviles también requieren de un **módulo controlador** o computador de control, el cual sirve para ejecutar el programa que gobierna el funcionamiento del robot. Normalmente se encuentra conectado a los controladores de los dispositivos y opera sobre ellos según ordene el programa. Los módulos controladores van acompañados de algún tipo de memoria donde se almacena el código a ejecutar y algunos valores.

Opcionalmente los robots móviles pueden incorporar elementos de comunicación con el exterior (pantalla, altavoces, conexión serie, conexión ethernet,...), que se emplean para interactuar con otra máquina (pc, robot,...), o con el hombre.

1.1.2. Navegación

La navegación [Bat95] es el proceso que permite a un robot móvil moverse de forma autónoma por un determinado entorno. Va asociado a la realización de cualquier tarea que realice un robot móvil, ya que siempre tendrá que desplazarse por un determinado entorno. Para que esto sea posible, el robot debe poseer capacidad de movimiento, percepción de su entorno, memoria, razonamiento y reacción ante eventos no previstos.

El proceso de navegación constituye la base sobre la que se sustentan el resto de tareas que realiza el robot móvil, puesto que si el robot no es capaz de desplazarse correctamente, dificilmente podrá realizar el resto de tareas que tenga programadas.

Las aplicaciones donde se están usando robots móviles (sección 1.1.3) van aumentando a medida que se optimizan los sistemas de navegación que los componen. Independientemente de los objetivos que persigan la distintas aplicaciones, existe un fin común a todas ellas que es ir de un punto a otro del entorno evitando obstáculos y demás posibles situaciones excepcionales tales como pasillos bloqueados, que no se caiga por las escaleras, etc. El proceso destinado a la consecución de este fin es lo que se conoce por navegación.

En líneas generales se puede considerar el proceso de navegación como aquél

cuyas entradas son el conocimiento específico del entorno, descripción de la posición actual, descripción del destino y observaciones del robot sobre el entorno, tal y como se muestra en la figura 1.1. Las salidas producidas son las órdenes de movimiento adecuadas para alcanzar la posición destino evitando obstáculos y otros imprevistos que puedan surgir.

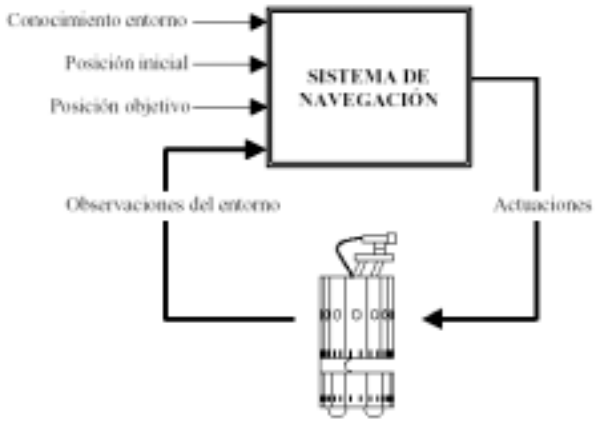


Figura 1.1: Proceso de Navegación

Debido a la complejidad que presenta el proceso de navegación, se suele dividir en subtareas cuya solución es más fácil de abordar. Esta modularidad facilita al mismo tiempo la adicción de otras tareas, así como la detección y recuperación ante posibles fallos. La mayoría de las arquitecturas presentan de forma explícita o implícita las siguientes subtareas:

- Planificación de trayectorias: se trata de crear una secuencia ordenada de posiciones (puntos, lugares, direcciones o cualquier otro tipo de referencias) por las que tiene que pasar el robot hasta alcanzar la posición destino.
- Evitación de obstáculos: es necesario un mecanismo que permita evitar los obstáculos, tanto estáticos como dinámicos, con los que se puede encontrar el robot. Algunos sistemas utilizan una planificación local que incluye replanificar cada vez que un objeto nuevo es detectado.
- Estimación de la posición: para navegar de un sitio a otro, el robot necesita

saber, con mayor o menor precisión, la posición en la que está en cada momento o, cuando menos, su posición relativa con respecto al objetivo. El robot móvil puede usar herramientas internas como sistemas odométricos, giroscopios, compases magnéticos o bien marcas de referencia externas para estimar su posición. En algunos casos se modifica el entorno situando balizas para obtener la posición del robot por triangulación o GPS (*Global Positioning System*) cuando se trata de entornos abiertos.

- Supervisión, detección y recuperación de errores: debido a que es casi imposible para el programador predecir todas las situaciones en las que se puede encontrar el robot, es necesario establecer un mecanismo para tratar situaciones distintas de las contempladas al realezar el sistema de navegación (fallos de sensores y situaciones del entorno no contempladas).

La navegación es un proceso complejo, que normalmente se divide en tareas para poder tratarlas de forma individual y luego integrarlas formando el sistema de navegación. La arquitectura de navegación define la descomposición y organización de las tareas incluidas en el sistema, así como las estructuras compartidas entre los distintos módulos de control. Existen múltiples arquitecturas de navegación que no vamos a entrar a estudiar puesto que están fuera del ámbito de este trabajo de Grado.

1.1.3. Aplicaciones de la robótica móvil

Existe multitud de documentación referente a robots móviles. Podemos realizar una revisión de acuerdo a los fines para los que han sido diseñados. Sin embargo es posible afirmar que los sistemas robóticos aplicados a algún campo se basan en una parte que proporciona la movilidad, junto con otra parte que proporciona la aplicación específica. La mayor o menor integración de las dos partes proporciona la posibilidad de reutilización o de especialización de los robots.

Transporte Existen multitud de soluciones comerciales, funcionando ya desde hace mucho tiempo. Su sistema de guiado varía mucho, pero uno de los más

utilizados es el de seguimiento de caminos fijos realizados con marcas, ya sean ópticas, inductivas, de luz, etc. En este caso se muestra en la figura 1.2 una solución de Flexitrack [FLE]. Se trata de carretillas de carga que realizan el seguimiento de una banda en el suelo. Se le pueden poner marcas en el camino para indicar puntos de parada y realizar carga o descarga. Dispone además de un radar para evitar colisiones. Existen varios modelos con distintas capacidades y tipos de carga.

Figura 1.2: Soluciones de transporte de Flexitrack

Robot de limpieza Los robots de limpieza están empezando a popularizarse en usos particulares. Existen soluciones que permiten aspirar una habitación de una forma bastante correcta. Tenemos un ejemplo claro con la aspiradora de Electrolux (figura 1.3(a)). Esta aspiradora tiene un sistema de percepción por ultrasonidos, de forma que le permite evitar obstáculos y elegir la ruta más adecuada. Otra solución bastante conocida (figura 1.3(b)), es la adoptada por el

(a) (b)

Figura 1.3: (a) Aspiradora autónoma de Electrolux (b) Robot limpiador del metro de París

metro de París. Se trata de un sistema autónomo para la limpieza de los andenes del metro. Su sistema de guiado se realiza por balizas magnéticas colocadas en el suelo.

Vigilancia y prospección Este tipo de robots se utilizan en los casos que resultan peligrosos para las personas, como zonas radioactivas, lugares con alta temperatura o alto riesgo, manipulaciones de materiales delicados, lugares con alta presión, tuberías, minas, etc. También se utilizan robots en investigación aeroespacial, como el malogrado *March Pathfinder* [Pat] o el *Mars Odyssey* [Mar].

(a) (b)

Figura 1.4: (a) March Pathfinder(NASA) (b) Groundhog, explorador de minas de la CMU

También se empiezan a ver robots de vigilancia como es el caso del mostrado en la figura 1.5. Tiene integrada una cámara de vídeo y un teléfono celular para realizar avisos en caso de detectar alguna intrusión.

Figura 1.5: Robot vigilante

Ayuda En este apartado se incluyen robots de guía, que se utilizan en centros comerciales, o como guías turísticos, etc. También existen robots de ayuda a

discapacitados, robots lazarillos, silla de ruedas inteligente, etc.

En la figura 1.6 se observa un robot guía, del museo de Valladolid, diseñado por el CARTIF. Este robot es capaz de realizar expresiones con sus rasgos faciales. Dispone de un sintetizador de voz y de un sistema de navegación. También dispone de una pantalla táctil para que los visitantes puedan interactuar con él. El sistema de guiado se basa en sonar y su movilidad está proporcionada por dos motores en disposición diferencial [DC02].

Figura 1.6: Robot guía del museo de Valladolid creado por el CARTIF

Aplicaciones especiales En este área se incluyen varios tipos de robots. Los robots de uso militar se suponen los más avanzados. Es el caso del avión espía de la figura 1.7(c). Sin embargo existen algunos falsos mitos sobre robots como los utilizados en la desactivación de explosivos, ya que la inteligencia principalmente es aportada por un operador remoto, y no incluyen prácticamente comportamientos autónomos inteligentes. También se han popularizado los llamados robot bomberos, que deben localizar y apagar un pequeño incendio. Este tipo de robots autónomos está todavía en investigación.

Uso lúdico Se incluyen robots dedicados al entretenimiento o acompañamiento. Se han popularizado enormentente desde la inclusión en el mercado del robot

(a) (b) (c)

Figura 1.7: (a) Desactivador de explosivos, (b) robot bombero y (c) avión espía

Aibo [AIB] y SDR-4X [SDR] de Sony (figura 1.8(a)). Estos robots son capaces de adaptarse al entorno y aprender de él. Disponen de múltiples sensores como cámaras de vídeo, micrófonos, infrarrojos, etc. Reconocen órdenes de voz y el entorno en tiempo real, memoriza caras, etc.

(a) (b)

Figura 1.8: (a) Aibo y (b) SDR-4X de SONY

Concursos y competiciones Al igual que ocurre con las competiciones de automovilismo o motociclismo, los concursos de robots pretenden ser el punto de evaluación y comparación de las distintas tecnologías aplicadas a la construcción de robots autónomos.

Existen multitud de competiciones de este tipo. Quizá la más representativa a nivel internacional sea RoboCup [ROB] (figura 1.9(a)). En esta competición se trata de jugar un partido de futbol con robots autónomos, los cuales son capaces de interactuar con otros robots y con el medio. Tambíen existen pruebas específicas con otros objetivos, muchas veces auspiciadas por casas comerciales.

Sin duda la competición más importante a nivel nacional es HISPABOT

[HIS] (figura 1.9(b)). En este concurso existen las pruebas de rastreadores, sumo, laberinto y velocista, en las que se valoran muchos de los aspectos básicos en Robótica, como la controlabilidad, interacción con el ambiente, técnicas de planificación, etc.

(a) (b)

Figura 1.9: (a) Imagen del concurso Robocup e (b) HISPABOT

1.2. Sistemas multi-robot

Podemos definir un sistema multi-robot como aquel en el que participan varios robots para lograr un determinado objetivo. En estos lo que se trata de conseguir es una cooperación entre los distintos robots.

Hoy en día los sistemas multi-robot se emplean en numerosos ámbitos que van desde el industrial hasta el militar pasando por el lúdico (figura 1.9(a)), pero en todos ellos se aplican para la realización de tareas que pueden descomponerse en subtareas ya sea por la dispersión física, conjunción de esfuerzos o porque la tarea principal sea de naturaleza descentralizada (p.e. coordinación espacio temporal).

El principal problema que existe a la hora de trabajar con estos sistemas es la construcción de los agentes que doten de inteligencia al sistema. Es decir un conjunto de robots por sí solos no hacen nada, necesitan tener implementado un comportamiento cooperativo que consiga el objetivo global del sistema. Normalmente las tareas u objetivos que se persiguen, implican tener que resolver una serie de aspectos un tanto complejos, como son la distribución de las tar-

eas entre los distintos robots y la creación de un mecanismo de interacción y coordinación entre ellos. En cuanto a la distribución de las tareas deben tenerse en cuenta aspectos como ¿de que forma distribuye un único robot la tarea? o ¿deben conocer los robots la tarea que se resuelve colectivamente? o ¿qué deben conocer de las tareas que realizan los otros robots del sistema?. Por otra parte, si consideramos los aspectos de interacción y coordinación entre ellos, deben plantearse preguntas del tipo ¿cuándo deben comunicarse?, ¿cómo?, ¿pueden reconocerse?. Lógicamente todos estos aspectos deben plantearse en el ámbito de un problema concreto. Pero lo que se pretende reflejar con estas cuestiones es la complejidad que se esconde tras la creación de los sistemas multi-robot.

La solución más empleada a la hora de implementar el comportamiento de un sistema de este tipo consiste en centralizar el control, de modo que uno de los robots sea el que coordine al resto, o controlar a todos los robots desde un programa alojado en una máquina del sistema. Esta solución disminuye notablemente el grado de autonomía de los robots, algo que no es deseable. Por ello las investigaciones en este área van encaminadas a la colaboración de robots lo más autónomos posibles.

Una de las herramientas más utilizadas a la hora de trabajar con sistemas multi-robot son los servidores multi-robot. Estos permiten trabajar sobre varios robots simultaneamente y constituyen una capa de abstracción que facilita la programación de los robots. Estos servidores se encargan de acceder a los dispositivos de los robots a petición de los programas clientes que interactúan con ellos, de modo que el programa cliente no tiene por qué conocer ni la estructura física del robot ni su funcionamiento a bajo nivel.

1.3. Antecedentes

Uno de los aspectos más compejos a la hora de trabajar con robots móviles es la creación de programas que doten al robot de la autonomía suficiente para realizar el objetivo propuesto con garantias, ya no sólo por la complejidad que implican los programas en sí, sino porque estos, en la mayor parte de los casos,

deben desarrollarse a medida para cada robot particular. Es decir, el programa se debe adaptar a la arquitectura física del robot y a la forma de comunicarse y de trabajar de cada modelo de dispositivo que posea el robot, lo que implica tener profundo conocimiento de los dispositivos físicos del robot, sus conexiones, protocolos, etc y lo que es peor, cada programa sólo es válido para un robot particular ya que depende de su arquitectura.

Otro aspecto bastante complejo es el trabajo con sistemas multi-robot, no sólo desde el punto de vista de la creación de programas de control de los múltiples robots, sino referido a la comunicación que debe existir entre los distintos robots, ya que en este tipo de sistemas es muy común centralizar el control en un equipo externo o en un único robot, encargándose este de tomar las decisiones y coordinar al resto. Los servidores multi-robot permiten controlar varios robots a través de ellos facilitando bastante este trabajo.

Hoy en día existen herramientas que tratan de facilitar el manejo de los robots (sus dispositivos), con el fin de poder desarrollar programas de control válidos para distintos robots y sin necesidad de conocer el funcionamiento a bajo nivel de los dispositivos. Estas herramientas permiten la coordinación y manejo de múltiples robots y son conocidas como servidores de dispositivos robóticos, servidores multi-robot o servidores robóticos.

El objetivo fundamental de este trabajo es integrar en un servidor multi-robot una serie de robots móviles no comerciales, desarrollados íntegramente en el Departamento de Informática y Automática de la Universidad de Salamanca, con la finalidad de facilitar la creación de programas de control de los robots en varios lenguajes de programación de alto nivel, y de esta manera crear una plataforma sobre la que poder integrar todo tipo de robots (comerciales o no) sobre la que investigar en técnicas tan novedosas como la navegación, colaboración robótica, etc. Para decidir si era necesario construir un servidor multi-robot nuevo o si por el contrario podíamos usar alguno de los existentes, hemos realizado un estudio sobre los servidores multi-robot existentes, sus características y las posibilidades de adaptación a distintos tipos de robots que ofrecen.

En los siguientes apartados se describen los servidores más destacados que

existen en la actualidad.

1.3.1. *Saphira*

Saphira es un sistema de control robótico desarrollado en el Centro Internacional de Inteligencia Artificial *SRI* para la empresa *ActivMedia* [ACT]. El objetivo de este software es facilitar la creación de programas que realicen tareas de navegación y localización con robots móviles. Ha sido diseñado usando técnicas de orientación a objetos y está escrito en C++.

Su arquitectura ha sido diseñada para trabajar contra un servidor ubicado en el robot móvil, que se encarga de proporcionar una serie de servicios en un formato estandard. Este paradigma cliente/servidor, abstrae a *Saphira* de las particularidades de cualquier robot y permite usarlo con cualquiera que posea un servidor que respete el protocolo de comunicación con *Saphira*. El servidor del robot es el encargado del acceso a bajo nivel de los dispositivos que posee, ya sea para establecer sobre los actuadores valores recibidos de *Saphira* o para enviarle valores solicitados de los sensores.

Tiene una gran limitación y es que se distribuye bajo licencia, únicamente con los robots de *ActivMedia* [ACT], por tanto no puede adaptarse a robots que no hayan sido fabricados por *ActivMedia*. Otro aspecto a tener en cuenta es que la versión que se distribuye normalmente sólo permite controlar un único robot simultaneamente, lo que impide el trabajo con sistemas multi-robot, aunque existe una versión multiagente, desarrollada en el año 1996.

1.3.2. *Ayllu*

Ayllu [Wer] ha sido escrito y mantenido por Barry Werger, y se define como una herramienta para el desarrollo de sistemas de control distribuido para grupos de robots móviles. Facilita la comunicación entre los componentes del sistema distribuido y la planificación de tareas. A pesar de poseer muchas características específicas para los sistemas basados en comportamiento, es muy útil para el desarrollo de una gran cantidad de arquitecturas de todo tipo, desde reactivas

a deliberativas, y proporciona los medios para la coordinación de tareas, como la planificación a alto nivel y el procesamiento de visión con el control a bajo nivel.

Proporciona una pequeña interfaz denominada *AylluLite*, que puede ser fácilmente adaptada a programas no creados con *Ayllu*, de manera que puedan ser controlados por *Ayllu*. Ha sido diseñado para ser altamente portable entre sistemas operativos y leguajes y para permitir la máxima interoperabilidad. Actualmente se encuentra disponible para plataformas UNIX, Macintosh, Windows y QNX y permite interactuar entre hosts heterogeneos de forma transparente.

Su principal objetivo es facilitar la implementación de sistemas multi-robot robustos que puedan enfrentarse a comunicaciones de alcance limitado y con posibles ruidos, situaciones del mundo real rápidamente cambiantes, variaciones en la disponibilidad de los recursos, tareas que requieran redistribución de los recursos del sistema e incluso a fallos hardware. Con el fin de conseguir su objetivo, *Ayllu* ofrece un sistema de paso de mensajes entre los elementos del dominio multi-robot, válido para una gran variedad de técnicas de control de comportamiento y que permite incluso la reconfiguración dinámica de las estructuras de comportamiento y la redistribución de tareas entre los robots del grupo.

Tiene definidos un conjunto de comportamientos estandard de control de los motores e interpretación de los sensores para los robots móviles de *Pioneer*. Entre estos se incluye control del desplazamiento basado en la velocidad o la distancia, y de todos los dispositivos de los robots *Pioneer*.

Los comportamientos pueden ser instanciados múltiples veces e interactuar a través de 'puertos', que pueden ser dinámicamente redirigidos en tiempo de ejecución. También pueden ser arbitrariamente distribuidos entre los distintos elementos del sistema sin realizar cambios de código.

Ayllu no proporciona funciones para el control de los robots, es decir no proporciona información de los sensores ni actua sobre los motores. En cambio proporciona un conjunto de comportamientos de control para los robots *Pioneer*.

Un sistema *Ayllu* consiste en un conjunto de comportamientos que se ejecu-

tan en paralelo, y un conjunto de conexiones a través de las cuales los comportamientos intercambian mensajes. Los comportamientos pueden ser instanciados en un determinado *hosts* (ordenador), o pueden propagarse a través de una red. De este modo un ordenador puede controlar un gran número de robots, un conjunto de ordenadores puede controlar un único robot o un conjunto de robots pueden ser controlados de un modo totalmente distribuido.

Un comportamiento *Ayllu* es una encapsulación de un conjunto de procesos que comparten una *interfaz*. La *interfaz* está formada por puertos, *slots* y *monostables*.

Los puertos y los *slots* son registros que contienen un valor simple de un tipo específico (integer, float...). Los *slots* sólo son accesibles por los procesos del comportamiento, mientras que los puertos pueden usarse por otros comportamientos para escribir en ellos. Los puertos son conectados dinámicamente en tiempo de ejecución y cada conexión debe ser entre dos comportamientos, que pueden estar en el mismo ordenador o distribuidos a través de una red. Las conexiones son unidireccionales y cada puerto puede tener tantas de entrada y de salida como desee. Los mensajes escritos en un puerto se propagan a través de todas las conexiones de salida de ese puerto.

Los *monostables* son valores booleanos que tienen asociado un periodo. Cuando un monostable es disparado, se pone a '*true*' durante un periodo específico de tiempo, volviendo posteriormente a '*false*'. Los *monostables* sólo son accesibls por los procesos del propio comportamiento.

El proceso de construcción de un sistema *Ayllu* es mas un direccionamiento de la información entre componentes simples que el diseño de un algoritmo. Por ejemplo, la creación de un sistema de evasión de colisiones puede llevarse a cabo por un comportamiento que simplemente escale sus entradas, las cuales estarán conectadas a los puertos de un vector de sonars y su salida al puerto del comportamiento que controla la velocidad de desplazamiento de la base del robot. Otro ejemplo podría ser un comportamiento que a través de un puerto reciba información visual como entrada (mediante el dispositivo Fast-Trak Vision System de *Pioneer*) y como salida al puerto de velocidad rotacional del controlador

del motor y se encargue de seguir los objetos que perciba visualmente. Si los dos comportamientos que hemos puesto como ejemplos estuviesen activos al mismo tiempo el robot mostraría un comportamiento muy robusto que le permitiría seguir objetos mientras esquiva los obstáculos que encuentra a su paso.

Tiene una gran limitación y es que *Ayllu* se distribuye bajo licencia por *ActivMedia* y sólo es válido para robots *Pioneer* puesto que los comportamientos estandard que definen se han realizado sólo para estos robots y no ofrece la posibilidad de adaptarlos a otros ya que su código no es abierto.

1.3.3. *CARMEN*: Carnegie Mellon Robot Navigation Toolkit

CARMEN [MMT] es una colección de aplicaciones servidoras (*open-source*) para el control de robots móviles. Las aplicaciones están escritas en lenguaje C y sus autores son Michael Montemerlo, Nicholas Roy, Sebastian Thrun. Se trata de un software modular diseñado para proporcionar las directivas básicas de navegación: control de sensores y desplazamiento, esquivación de obstáculos, localización, planificación de caminos, generación de mapas,...

El conjunto de aplicaciones que constituyen *CARMEN* son las siguientes:

- `base_services` Este programa controla el movimiento del robot y acepta entradas de los dispositivos sensores. Debe ejecutarse en el ordenador desde el que se establezcan las conexiones con el robot.
- `localize` Este programa hace uso de la información de los sensores que recibe de base_services para buscar la posición del robot en un mapa proporcionado por param_server.
- `map_editor` Programa que permite crear y editar mapas que pueden ser usados por *CARMEN*.
- `navigator` Programa que permite la navegación autónoma de un robot.
- `navigator_panel` Programa que proporciona una interfaz gráfica que

muestra la posición y el destino del robot sobre un mapa, permitiendo fijar la posición actual y la orientación y seleccionar el destino.

- `param_edit` Programa que permite al usuario cambiar ciertos parámetros como el robot que se está ejecutando. Permite guardar los cambios en un fichero "'.ini"'.
- `param_server` Este programa proporciona a otros programas información acerca del robot que está siendo usado y del entorno que rodea a este.
- `robotgraph` Este programa proporciona una simple interfaz gráfica para el manejo del robot, que permite moverlo y obtener la información actual de sus sensores.
- `simulator` Este programa proporciona generación de datos de un robot virtual. Necesita hacer uso de un mapa previamente generado.
- `vasco` Aplicación que crea un mapa a partir de los datos de los sensores y de la odometría y lo almacena en un fichero log.

La comunicación entre todos estos programas se realiza haciendo uso de un paquete independiente llamado IPC (Inter Process Communication), desarrollado en 1991 por Christopher Fedor y Reid Simmons.

Las aplicaciones que componen *CARMEN* han sido desarrolladas para ejecutarse sobre el sistema operativo Linux y comunicarse con los dispositivos de los robots a través de conexiones RS-232. Actualmente el software ha sido compilado y probado para las siguientes versiones de Linux:

- Red Hat 5.2, 6.2, 7.1, 7.3 y 8.0
- SuSe 8.0

Soporta las siguientes bases de robots móviles, aunque el código puede ser modificado para trabajar con otras:

- *iRobot* ATRV [iRo]
- *iRobot* ATRVjr [iRo]

- *iRobot* B21R [iRo]
- *ActivMedia Pioneer* I [ACT]
- *ActivMedia Pioneer* II [ACT]
- *Nomadic Technologies Scout* [Nom]
- *Nomadic Technologies* XR4000 [Nom]

y posee el código para trabajar con los siguientes sensores:

- SICK LMS (*Laser Measurment System*) [Sic]
- SICK PLS (*Proximity Laser Scanner*) [Sic]
- Garmin GPS35-LVS (*Global Positioning System*)
- Sonar (soporte preliminar)

La forma de trabajar que emplea *CARMEN* consiste en utilizar las aplicaciones servidoras necesarias, de las descritas anteriormente, de manera que unas hacen uso de otras y alguna de ellas se encarga de interactuar con el cliente (que puede ser un programa). Es posible definir aplicaciones servidoras para el control de nuevas bases de robots o dispositivos.

Una de las principales limitaciones que posee *CARMEN* es que no está preparado para trabajar con múltiples robots simultaneamente. Esto dificulta bastante su uso con el propósito que nosotros perseguimos (control de múltiples robots simultaneamente). Otro de los inconvenientes que posee es que sólo está preparado para trabajar mediante conexiones serie con los robots, lo que limita bastante el radio de actuación. Actualmente el software se encuentra en una versión alfa, por lo que no se puede garantizar su completo funcionamiento.

1.3.4. Player

Player [BPGH02] es un servidor multihilo de dispositivos robóticos, desarrollado por Brian Gerkey y Kasper Stoy, en el Laboratorio de Investigación

Robótica de la Universidad de California del Sur, junto con el Laboratorio de Ciencias de la Información de Laboratorios HRL. Se distribuye bajo GNU (*General Public License*) y se ejecuta en cualquier plataforma de tipo UNIX.

Player proporciona a los programas clientes, un control simple y completo sobre los dispositivos sensores y actuadores del robot a través de la red. Se puede ejecutar bien en el propio robot o en un ordenador al que estén conectados los dispositivos que se desean manejar. Soporta tanto conexiones serie como TCP/IP, permite el control de múltiples robots simultaneamente y la integración de nuevos dispositivos para ser controlados a través de él.

Player se adapta bastante a las necesidades que buscamos, por esta razón y tras realizar un profundo estudio de él (sección 2) hemos decidido usarlo como servidor sobre el que integrar y controlar nuestros robots.

1.4. Objetivos del trabajo

A continuación se van a presentar los principales objetivos de investigación y desarrollo que se han afrontado durante la realización de este trabajo de Grado.

Hasta ahora todos los programas de control sobre los robots móviles propios que realizábamos estaban hechos a medida, en el sentido de que servían únicamente para un robot, ya que desde estos se acceden y controlan los dispositivos particulares del robot. Esto provoca que un mismo programa no sea válido para dos robots con distinta arquitectura o sencillamente con distintos modelos de dispositivos. Con el fin de evitar tener que crear programas a medida para cada robot particular, tratamos de establecer una plataforma software que constituya una capa de abstracción entre los dispositivos del robot y los programas de control, de forma que éstos sean válidos para múltiples robots independientemente de su arquitectura o modelos de dispositivos. Al mismo tiempo esa plataforma software debe permitir controlar múltiples robots simultaneamente y facilitar la tarea de integración de nuevos robots para su control en ella.

Así pués, el objetivo fundamental de este trabajo es la integración de varios

robots propios, con distintas arquitecturas y dispositivos, en un servidor de dispositivos robóticos que permita, de una forma muy sencilla y abstracta, la creación de programas para el control, navegación y colaboración de dichos robots. De esta forma estableceremos un entorno sobre el que poder llevar a cabo tareas de investigación en técnicas como la navegación y la colaboración entre robots móviles.

La consecución de este objetivo obliga a realizar un análisis, estudio y valoración de las herramientas sofware para el control de dispositivos robóticos que existen actualmente, con el fin de elegir la que mejor se adecúe a nuestras necesidades. Una vez decidida la herramienta software a utilizar, se debe realizar la integración de nuestros robots en ella y si es posible adaptarla a nuestras necesidades con el fin de que la integración de nuevos robots resulte lo más sencilla posible.

Finalmente se realizarán pruebas que garanticen el correcto funcionamiento de los robots a través del servidor utilizado y haciendo uso de los cambios o adaptaciones propuestas.

Cabe destacar que no se encuentra dentro de los objetivos de este trabajo la realización de un determinado programa de colaboración, navegación o comportamiento, sino la elaboración de una capa de abstracción que permita la creación de dichos programas sin necesidad de implementar el control de los dispositivos robóticos a bajo nivel. La creación de este tipo de programas se abordará en trabajos futuros.

1.5. Organización de la memoria

Esta memoria comienza con un repaso de los aspectos clave utilizados en las distintas etapas de este trabajo, como son los robots móviles y los sitemas multi-robot.

A continuación se realiza una pequeña descripción y análisis de las herramientas que permiten la gestión y control de dispositivos robóticos, centrán-

donos posteriormente en la que mejor se adecúa a nuestras necesidades (*Player*).

En el siguiente capítulo se describe y explica la arquitectura que proponemos en este trabajo de Grado, para la integración de robots que quieran ser controlados mediante conexiones TCP/IP, en el servidor *Player*.

A continuación se ha creado un capítulo en el que se explican ejemplos reales de aplicación de la arquitectura sobre dos robots no comerciales.

Finalmente se detallan los resultados y conclusiones más relevantes de la investigación.

2

Servidor de Dispositivos Robóticos

Player

En este capítulo se va a realizar un análisis, estudio y valoración del servidor de dispositivos robóticos *Player*, concretamente de su forma de trabajar, su arquitectura, protocolo con los programas clientes que interactúan con él y la forma de integrar nuevos dispositivos. Se trata de comprobar las posibilidades de integración y control de dispositivos robóticos que ofrece, con el fin de valorar si puede emplearse como una herramienta que permita el control de robots no comerciales de un modo sencillo y completo.

Como se comentó en el apartado 1.3 el objetivo de este trabajo de Grado es integrar en un servidor multi-robot una serie de robots no comerciales con la

finalidad de facilitar la creación de programas de control de los robots en varios lenguajes de programación. En este capítulo se analiza si el servidor *Player* es válido para la integración y control de múltiples robots no comerciales, o si por el contrario es necesario desarrollar uno nuevo.

2.1. Descripción

Player es un servidor multihilo de dispositivos robóticos, desarrollado por Brian Gerkey y Kasper Stoy, en el Laboratorio de Investigación Robótica de la Universidad de California del Sur, junto con el Laboratorio de Ciencias de la Información HRL. Se distribuye bajo GNU (*General Public License*) y se ejecuta en cualquier plataforma de tipo UNIX.

Player proporciona a los programas clientes, un control simple y completo sobre los dispositivos sensores y actuadores del robot móvil, a través de la red. Se puede ejecutar bien en el propio robot, siempre y cuando este posea un módulo controlador con un sistema operativo tipo UNIX, o en el computador encargado de controlar al robot, al cual deben estar conectados los dispositivos que se deseen manejar.

Los programas de control desarrollados por el programador, que actúan como clientes, se conectan a *Player* por medio de *sockets* TCP estándar, estableciéndose entonces una comunicación entre ambos, que permite leer información de los sensores, enviar ordenes a los actuadores y configurar los dispositivos en tiempo real.

Un ejemplo sencillo del funcionamiento de *Player* es el mostrado en la figura anterior 2.1. El cliente establece una conexión con el servidor (*Player*) por medio de un *socket* TCP, y envía un conjunto de mensajes al servidor para abrir los dispositivos a los que desea acceder. Tras este proceso, *Player* enviará continuamente datos desde los dispositivos al cliente y éste, por su parte, realizará el control sobre el robot, enviado las órdenes apropiadas al servidor *Player*.

El programa cliente puede ejecutarse en una máquina que tenga una conexión

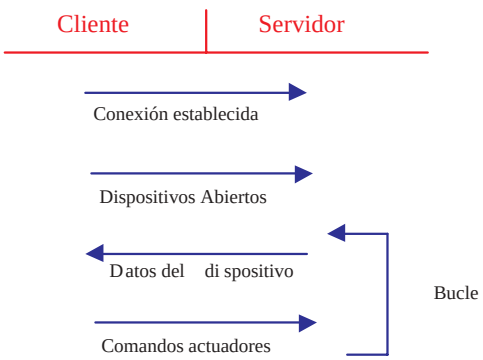


Figura 2.1: Interacción entre cliente y *Player*

de red con la máquina en la que se ejecuta *Player* y puede estar escrito en cualquier lenguaje que pueda abrir y controlar un *socket* TCP. Actualmente, es posible desarrollar clientes en C++, Tcl, LISP, Java y Python.

Otra característica muy interesante es que no tiene limitación en el número de clientes a los que puede dar servicio y permite que varios clientes accedan simultáneamente a dispositivos comunes de uno o varios robots. Esto hace que la creación de clientes multi-robot sea muy sencilla.

Player permite estructurar los programas de control del robot tal como el desarrollador desee, sin imponer limitaciones. De esta forma es posible diseñar un programa que nos permita controlar el robot de forma interactiva, mediante las teclas del ordenador, o bien por medio de sentencias de procesamiento independientes y secuenciales.

2.2. Dispositivos, *Interfaces* y *Drivers*

En *Player* se utiliza el concepto de dispositivo para referirse a una entidad abstracta que proporciona una *interfaz* estándar con algún servicio, de modo que es posible leer y escribir de y en ellos. Un dispositivo puede ser un sonar o un láser. *Player* no implementa dispositivos cerrados, sino que varios clientes pueden acceder de forma concurrente a un dispositivo dado, de modo que el

sistema es más flexible.

Por otra parte, *Player* establece una distinción entre la *interfaz* de un dispositivo y el *driver* del mismo. La *interfaz* de un dispositivo especifica el formato de los datos, órdenes y posibilidades de configuración que el dispositivo soporta. El *driver* de un dispositivo implementa el control a bajo nivel sobre el dispositivo. Según esto más de un *driver* puede soportar una única *interfaz* de dispositivo, pero no al contrario.

Todas las interacciones entre los clientes y el servidor (*Player*) se realizan a través de las *interfaces*, sin hacer referencia al *driver* que se está ejecutando por debajo. La asociación entre los dispositivos que va a controlar *Player* y el *driver* que va a utilizar con cada uno de ellos se hace en un fichero de configuración que *Player* lee al arrancar. Por ejemplo podríamos configurar *Player* para usarse con un dispositivo `position`, al que asociamos el índice 0 y el *driver* `p2os_position`, y otro dispositivo del mismo tipo al que asociamos el índice 1 y el *driver* `rwi_position`. De esta forma un programa cliente podría controlar la posición de dos robots distintos mediante una única *interfaz* sin preocuparse del *driver* asociado al dispositivo de cada robot, simplemente debe decidir si accede al dispositivo con índice 0 ó 1.

En la versión 1.3 de *Player* se encuentran implementadas *interfaces* para los siguientes dispositivos:

- **player.** Representa el servidor en si mismo. Se utiliza para configurar el comportamiento del mismo. Este dispositivo puede ser leído pero no escrito. Las peticiones de configuración incluyen apertura y cierre de dispositivos, etc.

- **misc.** Proporciona información como puede ser el nivel de batería del robot y, si se dispone, el estado del parachoques del robot. No puede ser escrito ni configurado.

- **gripper.** Controla la pinza del robot Pioneer 2 de ActivMedia [ACT]. Se puede leer el estado de la pinza y ordenar que se abra o cierre. No puede ser configurado.

- **position.** Controla la posición de robot. Se pueden obtener las coordenadas y el ángulo del robot, su velocidad angular y tangencial. Se emplea para indicar nuevas velocidades.

- **sonar.** Proporciona las lecturas realizadas por los sonars del robot. Solo puede ser configurado para activar o desactivar los sonars.

- **laser.** Controla el láser SICK LMS-200 [Sic], obteniendo las lecturas del mismo. Puede ser configurado para cambiar la apertura de lectura y activar o desactivar la lectura de los valores de reflexión. No puede escribirse.

- **vision.** Permite interactuar con el dispositivo de visión en color ACTS ActivMedia. Sólo permite lectura.

- **ptz.** Controla la cámara Sony EVID30 [Son]. Al leer de este dispositivo obtenemos la panorámica actual, inclinación y zoom. Estos parámetros pueden modificarse escribiendo en el mismo.

- **laserbeacon.** Procesa las lecturas del láser, buscando la señales emitidas por un conjunto de balizas. Al leer del dispositivo se obtiene el identificador, el rango, la posición y la orientación de las balizas.

- **broadcast.** Permite a los clientes comunicarse con varios servidores *Player*. Si un cliente escribe en este dispositivo todos los servidores recibirán lo escrito. Al leer se obtendrían los últimos datos escritos por cada cliente.

- **gps.** Sólo funciona en el simulador, proporciona información de la posición y el ángulo del robot en coordenadas globales.

- **bps.** Proporciona la misma información que el gps pero basándose en el dispositivo laserbeacon.

Es posible implementar nuevas *interfaces* para otros dispositivos, o múltiples *interfaces* para un dispositivo, aunque esto último no es aconsejable ya que es preferible modificar una *interfaz* existente ampliando su funcionalidad que definir varias cada una para realizar ciertas tareas sobre un mismo dispositivo. Es importante tratar de mantener una única *interfaz* para cada tipo de dispositivo

ya que esto pemite a los usuarios el manejo de distintos modelos de dispositivos de un mismo modo.

En la versión 1.3 de *Player* se encuentran implementados *drivers* para el control de los siguientes sistemas y dispositivos robóticos:

- Robots Pioner 1, Pioneer 2, AmigoBot, PeopleBot de ActivMedia [ACT]: posición (motores), sonars, *bumpers*, voltaje de la batería y brújula.
- Robots RWI de la serie B: posición, sonars, *bumpers*, voltaje de la batería y láser.
- Robots K-Team [Kte], Kameleon 376SBC: posición, infrarrojos y voltaje de la batería.
- Cámara Sony EVID30 PTZ [Son].
- Láser SICK LMS-200 [Sic].
- Tarjetas de red Wireless.
- Tarjetas de sonido.

Para poder controlar desde *Player* los dispositivos de un robot que no se encuentre en la lista de los implementados, es necesario crear los *drivers* para el manejo de los mismos, adaptándolos a las *interfaces* existentes (siempre y cuando exista una asociada a ese tipo de dispositivos).

2.3. Arquitectura de *Player*

Player ha sido diseñado con el propósito de que resulte sencillo añadir nuevos dispositivos, como un sistema asíncrono y para ser independiente de lenguaje y plataforma. Está implementado en C++ y hace uso de hilos POSIX para crear programas multihilo.

Dentro de *Player* hay un hilo servidor y un hilo por cada dispositivo abierto. El hilo servidor atiende las conexiones de nuevos clientes a través de un *socket*

TCP, recibe las órdenes de todos los clientes conectados y envía los datos y respuestas de los dispositivos a cada cliente (ver figura 2.2). Cuando *Player* recibe una petición de un cliente sobre un dispositivo con el que aún no se tiene comunicación, crea un nuevo hilo que realiza la comunicación con dicho dispositivo.

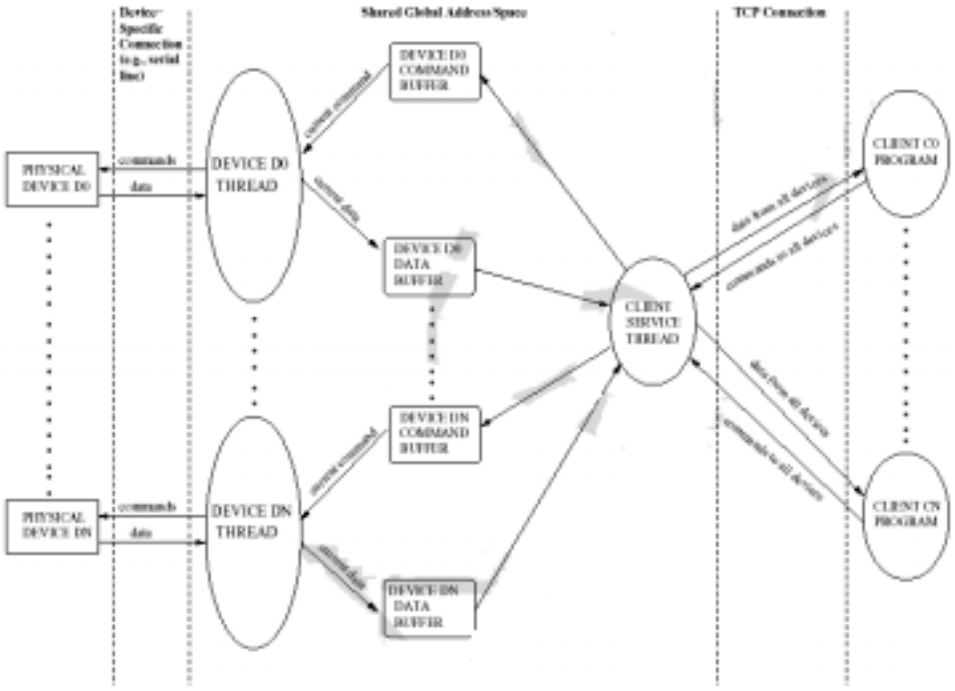


Figura 2.2: Arquitectura del Servidor *Player*

La comunicación entre los hilos se realiza a través de un espacio de direcciones global compartido. Cada dispositivo tiene asociado un *buffer* de órdenes y un *buffer* de datos, protegidos por secciones *mutex* (ver figura 2.2). Estos *buffers* proporcionan un canal de comunicación asíncrona entre los hilos de los dispositivos y el hilo que atiende a los clientes. Así cuando este hilo recibe una orden de un cliente para un dispositivo, escribe la orden en el *buffer* de órdenes correspondiente, y cuando el hilo del dispositivo está preparado para enviar una nueva orden, lo lee de su *buffer* de órdenes y lo envía al dispositivo. Del mismo modo cuando el hilo del dispositivo recibe un dato del mismo, lo escribe en el *buffer* de datos, y después cuando el hilo que atiende a los clientes esta

preparado para enviar datos, lo lee del *buffer* de datos y se lo pasa a un cliente en particular.

El cliente puede ejecutarse en el mismo *host* que *Player* o en otro *host* que tenga conexión con el de *Player* a través de red.

Cuando un cliente lee de un dispositivo, el cliente recibe el estado del dispositivo en el instante en el que es leído. Este estado suele ser enviado al cliente con una frecuencia de 10 Hz o menos (es configurable). Por su parte si un cliente escribe en un dispositivo, está enviando una orden al mismo. Esta orden suele incluir los parámetros del dispositivo que más frecuentemente cambian durante el uso del mismo. Por ejemplo en el dispositivo *position* que controla el robot móvil, la orden es un conjunto de velocidades.

Finalmente los dispositivos pueden ser configurados desde *Player*. Mientras que la lectura y escritura son asíncronas, la configuración es un proceso de petición-respuesta síncrona entre cliente y dispositivo. Cuando un cliente envía una petición de configuración al servidor, la petición se añade a una cola de entrada para el dispositivo correspondiente, y cuando el dispositivo sirve la petición genera una respuesta que se añade a la cola de salida del dispositivo. Esta respuesta será transmitida por el servidor al cliente que se encuentra esperando. Otro aspecto que lo diferencia de la lectura y escritura, es que las peticiones de configuración no se sobrescriben. Normalmente la configuración se emplea para asignar o solicitar algún aspecto del estado del dispositivo, siendo un proceso menos frecuente que la lectura y escritura.

2.4. Protocolo de comunicaciones entre los Clientes y *Player*

La comunicación entre cualquier programa cliente y *Player* se realiza a través de un *socket* TCP, por lo que un cliente que desee comunicarse con él, deberá crear un *socket* TCP y conectarse a *Player* (por defecto a través del puerto 6665). Cada vez que *Player* recibe una nueva conexión, responde con una cadena de

32 caracteres, que identifica su versión. El cliente debe recoger esta cadena de caracteres, y a partir de ahí el servidor queda a la espera de peticiones por parte del cliente.

Los mensajes que intercambian cliente y servidor están compuestos de una cabecera (ver figura 2.3) y un cuerpo. La cabecera está compuesta por 32 bytes que contienen información sobre cómo interpretar el cuerpo del mensaje.

Figura 2.3: Cabecera del Mensaje

Los campos de la cabecera tienen el siguiente significado:

- **STX** Indica el comienzo del mensaje. Siempre tiene el valor 0x5878.
- **type** Indica el tipo de mensaje que va en el cuerpo. Hay siete tipos de mensajes que se describen posteriormente.
- **device** Indica la *interfaz* a la que pertenece el mensaje (*position*, sonar, láser...).
- **index** Indica a cuál dispositivo, en caso de haber varios iguales, va dirigido el mensaje.
- **t_sec** Sólo es usado por el servidor. Indica los segundos del tiempo actual del servidor.
- **t_usec** Sólo es usado por el servidor. Indica los microsegundos del tiempo actual del servidor.
- **ts_sec** Sólo es usado por el servidor. Indica los segundos del *timestamp* fijado por el dispositivo que proporciona el dato (instante en el que el dispositivo sondeó el valor).
- **ts_usec** Sólo es usado por el servidor. Indica los microsegundos del *timestamp* fijado por el dispositivo que proporciona el dato (instante en el que el dispositivo sondeó el valor).

- **reserved** Campo reservado para uso futuro.
- **size** Tamaño en bytes del cuerpo del mensaje, sin incluir la cabecera.

El cuerpo del mensaje puede ser de alguno de los siguientes tipos:

- **Dato** (type 0x0001) Es enviado desde el servidor a un cliente y contiene el valor leido por un dispositivo. Por defecto el servidor envía valores de un dispositivo al cliente que lo solicite, con una frecuencia de 10Hz.
- **Orden** (type 0x0002) Es enviada desde el cliente a *Player* y contiene una orden de actuación sobre un dispositivo. El servidor no responde al cliente cuando éste envía un mensaje de este tipo.
- **Solicitud** (type 0x0003) Es enviado desde el cliente a *Player* y contiene una solicitud de cambio en la configuración de un dispositivo.
- **Acuse de recibo** (type 0x0004) Es enviado desde el servidor a un cliente para confirmar que un mensaje de solicitud ha sido recibido. Es interpretado y ejecutado por el dispositivo.
- **Sincronización** (type 0x0005) Es mandado desde el servidor a los clientes tras el envio de un conjunto de datos. Este mensaje indica al cliente que todos los datos de un ciclo ya han sido enviados.
- **Acuse de recibo negativo** (type 0x0006) Es enviado desde el servidor a un cliente para indicar que un mensaje de solicitud ha sido recibido, pero no ha podido ser interpretado o ejecutado por el dispositivo.
- **Acuse de recibo erróneo** (type 0x0007) Es enviado desde el servidor a un cliente para indicar que un mensaje de solicitud no ha podido ser puesto en la cola para entregarlo al dispositivo.

2.5. Diagrama de Clases

Como ya se ha comentado anteriormente *Player* forma parte de un proyecto de código abierto, se encuentra implementado en C++ y hace uso de una gran

cantidad de clases, muchas de las cuales se relacionan en una compleja jerarquía. Puesto que forma parte de un proyecto de código abierto, puede ser modificado o adaptado a las necesidades de cada usuario (crear nuevos *drivers*). Esto implica conocer las clases que lo componen y las relaciones existentes entre estas. En la siguiente figura (2.4) se muestra el diagrama de clases simplificado de *Player*, con las clases mas relevantes.

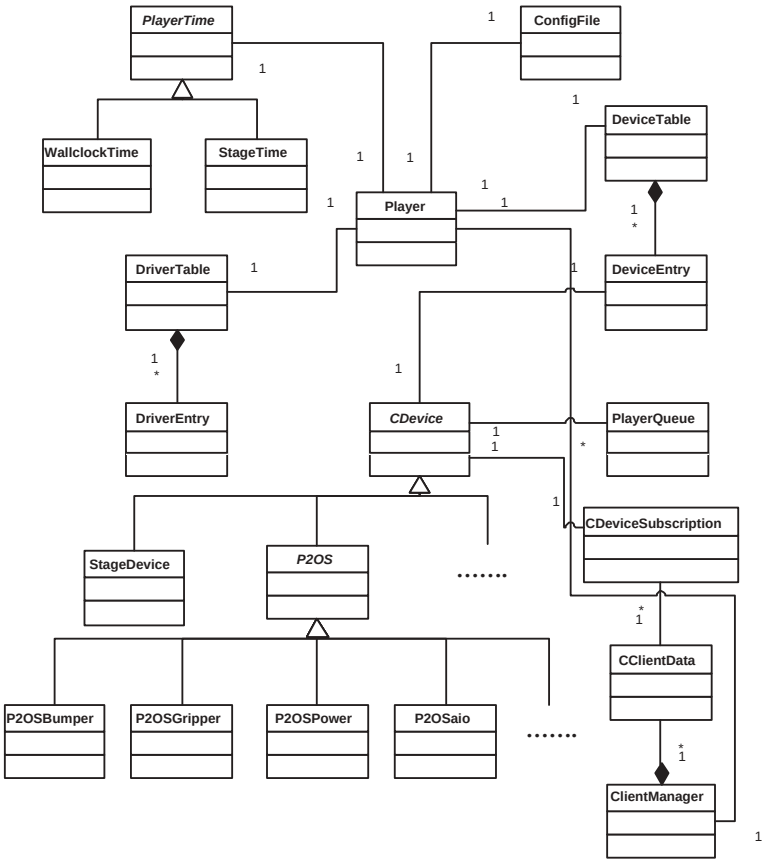


Figura 2.4: Diagrama de Clases de *Player*

A continuación pasamos a describir brevemente las clases que aparecen en el diagrama 2.4, prestando especial atención a la clase *CDevice*, puesto que la mayor parte de modificaciones que se realizan sobre *Player* son incorporaciones de nuevos *drivers* de dispositivos, y esta clase abstracta es de la que deben heredar las nuevas clases que implementen un *driver*.

- *ConfigFile*. Es una clase que encapsula los métodos encargados de leer y guardar la información de los ficheros de configuración, como por ejemplo los ficheros que *Player* lee al arrancar. En ellos el usuario especifica los dispositivos que quiere que se carguen, los *drivers* asociados a estos, sus índices...

- *PlayerTime*. Es una clase abstracta de la que heredan algunas funciones virtuales puras las clases *WallclockTime* y *StageTime*. Cabe destacar la función `GetTime` que redefinen ambas clases y que proporciona el tiempo actual, que se emplea para establecer el *timestamp* de los datos leidos de los dispositivos.

- *DeviceTable*. Es una clase que implementa una lista enlazada de objetos de la clase *DeviceEntry*. Esta segunda clase almacena información relativa al dispositivo como su identificador, un puntero a él, su nombre y su modo de acceso. Por cada instancia de *Player* se crea un objeto de la clase *DeviceTable* con un objeto de la clase *DeviceEntry* por cada dispositivo que el usuario haya especificado en el fichero de configuración.

- *PlayerQueue*. Es una clase que implementa una cola de elementos. Donde estos son estructuras con información relativa a peticiones y respuestas de configuración entre dispositivos y clientes. Entre la información de esos elementos se encuentra un puntero al cliente, el identificador del dispositivo, el tipo de mensaje, un *timestamp* y la información de solicitud o respuesta.

- *DriverTable*. Es una clase que implementa una lista enlazada de objetos de la clase *DriverEntry*. Cada objeto de la clase *DriverEntry* almacena información acerca de un *driver* de un determinado dispositivo. Entre la información que almacena se encuentra el nombre del *driver*, el tipo de acceso que permite al dispositivo, un puntero a la función encargada de iniciar el *driver*, ... Por cada instancia de *Player* que se ejecuta, se crea un objeto de tipo *DriverTable* en el que se registran todos los *drivers* disponibles para los dispositivos.

- *CClientData*. Es una clase que encapsula todos los datos de un determinado cliente ofreciendo métodos para operar sobre estos. Cabe destacar el

almacenamiento de una lista enlazada de objetos de tipo *CDeviceSubscription* con información de los dispositivos a los que un cliente está suscrito.

- *ClientManager*. Es una clase que encapsula toda la información de los clientes que están interactuando con el servidor *Player*. Posee métodos para añadir y borrar clientes, aceptar la conexión con un cliente, ...

Con respecto a las clases que heredan de la clase *CDevice*, son clases que se encargan de la implementación de *drivers* particulares de dispositivos. La gran mayoría de ellas no se han representado en el diagrama debido a la cantidad existente. Tampoco se han representado en el diagrama de la figura 2.4 otras clases menos relevantes ni las que implementan las distintas *interfaces*.

Tiene especial importancia para la creación de nuevos *drivers* (2.6), la clase *CDevice* ya que todos deben heredar de ella y redefinir ciertos métodos virtuales. Por ello en el siguiente apartado la describimos con más detalle.

2.5.1. La clase *CDevice*

Se trata de la clase base de todos los dispositivos. Define la *interfaz* que todos los dispositivos deben implementar. Entre las variables miembro que posee esta clase, y que heredan sus derivadas, cabe destacar las siguientes:

- *Buffer* de Datos. Se asocia al dispositivo y se usa como zona de memoria compartida entre el hilo que ejecuta el *driver* del dispositivo y el que atiende las peticiones de los programas clientes, de manera que el hilo del *driver* almacena en este *buffer* los valores leidos del dispositivo y el hilo que atiende las peticiones de los clientes lee de él los valores que envía a éstos. Está protegido por un *mutex* para evitar problemas de concurrencia.

- *Buffer* de Órdenes. Se asocia al dispositivo y se usa como zona de memoria compartida entre el hilo que ejecuta el *driver* del dispositivo y el que atiende las peticiones de los programas clientes, de manera que este último almacena en él las órdenes enviadas por los programas clientes y el hilo que ejecuta el *driver* lee de él las órdenes. Está protegido por un *mutex*

para evitar problemas de concurrencia.

- Cola de Solicitudes de Configuración. Se asocia al dispositivo y es un objeto de tipo cola (declarado en *Player*). Se usa como zona de memoria compartida entre el hilo que ejecuta el *driver* del dispositivo y el que atiende las peticiones de los programas clientes, de manera que este último almacena en ella las solicitudes de configuración que los programas clientes envian al dispositivo. Está protegido por un *mutex* para evitar problemas de concurrencia.

- Cola de Respuestas de Configuración. Se asocia al dispositivo y es un objeto de tipo cola (declarado en *Player*). Se usa como zona de memoria compartida entre el hilo que ejecuta el *driver* del dispositivo y el que atiende las peticiones de los programas clientes, de manera que el hilo del *driver* almacena en ella las respuestas a las solicitudes de configuración sobre el dispositivo para que el hilo que atiende a los clientes se las envie. Está protegido por un *mutex* para evitar problemas de concurrencia.

Los tamaños de los *buffers* y colas se establecen a través del constructor de *CDevice*, que debe ser invocado desde el constructor del *driver* que estemos implementando.

En cuanto a las funciones miembro que posee esta clase, y que heredan sus derivadas, cabe destacar las siguientes:

- *PutData*. Este método debe llamarse siempre que el *driver* reciba un nuevo dato del dispositivo. Su implementación debe copiar en el *buffer* de datos asociado al dispositivo, el dato recibido.

- *GetNumData*. Método encargado de devolver el número de paquetes de datos que pueden ser leidos por un cliente en ese momento. Este método se llama siempre que un programa cliente quiere consultar el valor de los datos de un dispositivo.

- *GetData*. Método llamado cuando un programa cliente solicita los datos de un dispositivo. Su funcionalidad consiste en acceder al *buffer* de datos del dispositivo y copiar los valores de estos para que sean enviados al cliente.

También hay métodos de acceso a órdenes:

- *PutCommand*. Cuando un cliente envia una nueva orden al dispositivo, el servidor invoca automaticamente a este método. Su misión consiste en copiar la nueva orden en el *buffer* de órdenes asociado al dispositivo.
- *GetCommand*. Método encargado de devolver las órdenes enviadas sobre el dispositivo. Debe acceder al *buffer* de órdenes, copiarla y devolverla.

Así como de configuración de acceso:

- *PutConfig*. El servidor invoca este método cuando recibe una nueva solicitud de configuración. Su tarea consistirá en añadir a la cola de solicitudes de configuración la nueva recibida.
- *GetConfig*. Método que sirve para comprobar si hay alguna configuración pendiente de aplicar sobre el dispositivo. Su misión consiste en extraer de la cola de solicitudes de configuración la más antigua, si es que hay alguna, y devolverla.
- *PutReply*. Método que es invocado como respuesta a la aplicación de una nueva configuración sobre el dispositivo. Su función consiste en almacenar en la cola de respuestas, la respuesta a la configuración aplicada.
- *GetReply*. El servidor controla periodicamente las respuestas a las solicitudes de configuración de dispositivos por parte de los clientes, a través de este método. El cual accede a la cola de respuestas para extraer el valor más antiguo, si es que hay alguno.

Además de los citados anteriormente existen otros tres métodos que deben ser obligatoriamente redefinidos:

- *Setup*. Se llama cuando el primer cliente solicita a *Player* información del dispositivo. En él se debe iniciar el dispositivo (comunicaciones, valores iniciales...) y crear un hilo sobre el que se ejecutará la función principal asociada al dispositivo.

- *Shutdown*. Cuando el último cliente se desconecta de *Player*, se invoca a este método. En él se deben realizar todas las tareas de finalización del dispositivo y la finalización del hilo.
- *Main*. Es la función principal asociada al dispositivo, se ejecuta al crearse el hilo asociado al dispositivo y su implementación normalmente consiste en un bucle que interactua continuamente con el dispositivo.

2.6. Integración de Dispositivos en *Player*

Uno de los aspectos interesantes que presenta *Player*, es que el conjunto de *drivers* que tiene implementados, permiten controlar dispositivos de algunos robots comerciales (posee *drivers* para controlar todos los dispositivos de: Pioneer 1, Pioneer 2, AmigoBot, RWI B-series robots...). Esto implica que, si quiere usarse con robots de fabricación propia (como ocurre en nuestro caso), u otros que no estén entre los implementados, es necesario desarrollar un conjunto de *drivers* para los dispositivos de estos robots. *Player* ofrece la posibilidad a sus usuarios de añadir al proyecto nuevos *drivers* de dispositivos, de modo que dichos dispositivos puedan pasar a ser controlados a través de él.

De un modo simplificado y teniendo presente la estructura de clases descrita en el apartado 3.3.3, pasamos a describir cómo se integra [BPGH02] un nuevo dispositivo en *Player*. El primer paso a la hora de crear un nuevo *driver* es decidir si el dispositivo se adapta plenamente a alguna de las *interfaces* existentes (consultar apartado 2.2), o si por el contrario va a ser necesario modificarla o crear una nueva. Es menos costoso adaptar el *driver* a una *interfaz* existente que crear una nueva, pues de lo contrario será necesario implementar una nueva clase para la *interfaz* del dispositivo que estemos añadiendo. En las clases que implementan las *interfaces* se definen, además de los métodos que permiten a los clientes interactuar con los dispositivos, los tipos de datos con los que esos métodos trabajan y los que proporcionan a los clientes. Es importante tener presente los tipos de datos definidos en la clase de la *interfaz*, ya que el *driver* que implementemos deberá proporcionar a la *interfaz* los datos con esa estructura y tipo, y la *interfaz* le enviará los datos con esa estructura y tipo.

A continuación debe crearse una nueva clase que herede de *CDevice* (2.5.1), que es la clase base de todos los dispositivos, y define la *interfaz* que todos los dispositivos deben implementar. La nueva clase debe implementar las funciones virtuales puras heredadas y redefinir aquellas que considere necesarias, así como añadir las variables miembro que pueda necesitar.

Una vez redefinidas las funciones miembro heredadas es necesario implementar dos funciones adicionales que no van a pertenecer a la clase del *driver*, pero que son exclusivas de cada uno. La primera es una función que crea una nueva instancia del *driver* y que es invocada desde *Player* cada vez que un cliente solicita trabajar con un dispositivo usando ese *driver*. Con respecto a la segunda es una función que registra el nuevo *driver* en un objeto de tipo *driverTable*, que es una tabla global que posee *Player* con los *drivers* que pueden ser instanciados. La llamada a esta función se realiza desde otra denominada `register_devices`, la cual es invocada desde *Player* nada mas arrancar. Debemos ser nosotros los que incluyamos en la función `register_devices` la llamada a la función de registro de nuestro *driver*, para que *Player* sepa de su existencia.

Finalmente, se debe crear una biblioteca estática que contenga el código del *driver* creado para el nuevo dispositivo. Para ello se deben modificar los *makefiles* (`Makefile.in`) y ficheros de configuración (`configure.in`), ya que *Player* hace uso de las herramientas *Automake* [Foub] y *Autoconf* [Foua] para la generación y compilación del código, de forma que se incluya la nueva biblioteca creada. Normalmente los *drivers* se agrupan en bibliotecas atendiendo a criterios razonables, como por ejemplo que sean de un mismo robot o de un mismo dispositivo. Por lo tanto es aconsejable agrupar los *drivers* de los dispositivos de un robot en una misma biblioteca.

2.7. Valoración

Podemos decir que *Player* es un servidor multi-robot que constituye una capa de abstracción entre los dispositivos robóticos y los programas clientes que acceden a ellos. De modo que facilita en gran medida la creación de programas

clientes, en varios lenguajes de programación, para el manejo de los robots. Presenta numerosas ventajas, como son:

- *Opensource* permite adaptar el código a nuestras necesidades, integrar nuevos dispositivos, crear nuevas *interfaces*...
- Gratuito.
- Constituye una herramienta muy útil para la investigación en robótica móvil, ya que permite desarrollar programas clientes sin tener que adaptarlos a dispositivos o arquitecturas particulares.
- Permite acceder a múltiples dispositivos simultaneamente, incluso de robots distintos.
- Soporta múltiples programas clientes, los cuales pueden estar escritos en distintos lenguajes de programación.
- Está preparado para trabajar con un simulador robótico denominado *Stage* también *opensource*.
- Ha sido usado en varios proyectos de investigación, por lo que su fiabilidad y eficiencia están más que probadas.

No obstante creemos que también posee algún aspecto un poco menos ventajoso, como por ejemplo el hecho de que esté pensado para trabajar con dispositivos a través de conexiones serie. Es decir, *Player* contiene *drivers* para el control de determinados modelos de dispositivos, de modo que en la implementación de esos *drivers* se establece una comunicación directa con el dispositivo, el cuál posee su *interfaz* o protocolo de comunicación, que es respetado por el *driver*.

Esta forma de trabajar permite que dicho *driver* sirva para cualquier ejemplar de ese modelo de dispositivo, siempre y cuando exista una conexión serie con él. Pero desde nuestro punto de vista, implica dos limitaciones que son: cuando *Player* se ejecuta en un PC, la cantidad de dispositivos que pueden ser controlados a través de él es bastante escasa, ya que se limita a un dispositivo por conexión serie del PC. La segunda es que el radio de acción para una

conexión serie es bastante limitado, sobre todo si lo comparamos con conexiones TCP/IP inalámbricas.

Creemos que el uso de *Player* para el control de robots (sus dispositivos) a través de conexiones TCP/IP inalámbricas puede ser muy ventajoso, ya que con una única instancia de *Player* podrían controlarse varios robots, lo que facilitaría la investigación en temas como la colaboración robótica, y a través de este tipo de conexión se puede alcanzar un mayor radio de acción. El control de dispositivos a través de conexiones TCP/IP presenta el problema de que a través de este tipo de conexiones no se puede acceder directamente a los dispositivo, sino que se necesita que el robot tenga una aplicación que atienda las peticiones que *Player* realiza sobre los dispositivos del robot. Después de pensar en esto durante algún tiempo, se nos ocurrió el modo de aprovechar esta limitación. La idea es que, como ya que no van a ser los *drivers* de *Player* los que accedan directamente a los dispositivos, sino que ahora se deben comunicar con una aplicación que se ejecute en el robot, no tenemos porque hacer que cada *driver* sea particular para un modelo de dispositivo. Podemos crear una serie de *drivers* genéricos para cada tipo de dispositivo, de forma que sea la aplicación que se ejecuta en el robot la que se adapte a estos *drivers*. Esta forma de trabajar evitaría tener que crear un *driver* específico para cada modelo de dispositivo e incluso permitiría crear un conjunto de *drivers* genéricos que evitasen tener que modificar una sola línea del código de *Player*.

3

Arquitectura propuesta para integrar dispositivos en *Player*

Podemos decir que *Player* es un servidor multi-robot que abstrae y facilita la tarea de crear programas clientes, en varios lenguajes de programación, para el manejo y control de robots. Ofrece una API (*Aplication Programming Interface*) que permite acceder desde un programa cliente a los dispositivos, de una forma sencilla y sin necesidad de conocer su implementación a bajo nivel. Pero tiene algunos aspectos complejos, ya comentados en el apartado 2.7, como son el hecho de tener que crear un *driver* específico para cada dispositivo particular que se quiera controlar a través de *Player*, y que esté fundamentalmente orientado al control de dispositivos a través de conexiones serie, dejando un poco al margen la integración de robots que puedan ser controlados a través de conexiones

TCP/IP.

El hecho de tener que crear un *driver* en *Player* por cada dispositivo de un robot que quiera controlarse a través de él, implica crear y modificar las clases que lo componen y por tanto un profundo conocimiento de esa estructura de clases. Con el fin de facilitar la integración de dispositivos que puedan ser controlados a través de conexiones TCP/IP y evitar tener que crear un *driver* para cada dispositivo en particular, proponemos una arquitectura en tres capas que permita integrar en *Player* nuevos dispositivos sin necesidad de elaborar un *driver* específico para cada uno de ellos, es más sin siquiera modificar una línea de código en *Player*.

Nuestra propuesta consiste en añadir una capa más a la arquitectura cliente-servidor que emplea *Player*, de manera que en el robot se ejecute una aplicación servidora que se encargue de acceder a los dispositivos a petición de *Player*, el cual hará de intermediario entre los programas clientes y la aplicación servidora del robot. De esta forma es posible crear una serie de *drivers* genéricos (uno por cada tipo de dipositivo: uno para sonar otro para infrarrojos, ...) que se comunicarán con la aplicación servidora del robot, en lugar de con los propios dispositivos. Para que los *drivers* genéricos sean válidos con cualquier aplicación servidora de cualquier robot, será necesario establecer un protocolo de comunicación entre ambas partes, que deberá respetar la aplicación servidora del robot que se quiera integrar en *Player*. Así se evitará tener que crear un nuevo *driver* por cada modelo particular de dispositivo

Nuestro objetivo es facilitar la integración de robots en *Player* con el fin de que se puedan crear programas clientes sin necesidad de conocer el funcionamiento interno de los dispositivos del robot ni de *Player*.

En este capítulo analizamos la arquitectura que proponemos y su funcionamiento.

3.1. Descripción de la Arquitectura Propuesta

Como ya se ha comentado anteriormente, se trata de una arquitectura en tres capas (ver figura 3.1) en la que *Player* actúa como servidor de los programas clientes y al mismo tiempo como cliente de una aplicación servidora que se ejecuta en el robot y que se encarga de acceder a los dispositivos de éste.

Con respecto a la capa más baja está formada por una aplicaión servidora necesaria para acceder desde los *drivers* de *Player* a los dispositivos, ya que no es posible acceder a través de comunicaciones TCP/IP directamente a los dispositivos.

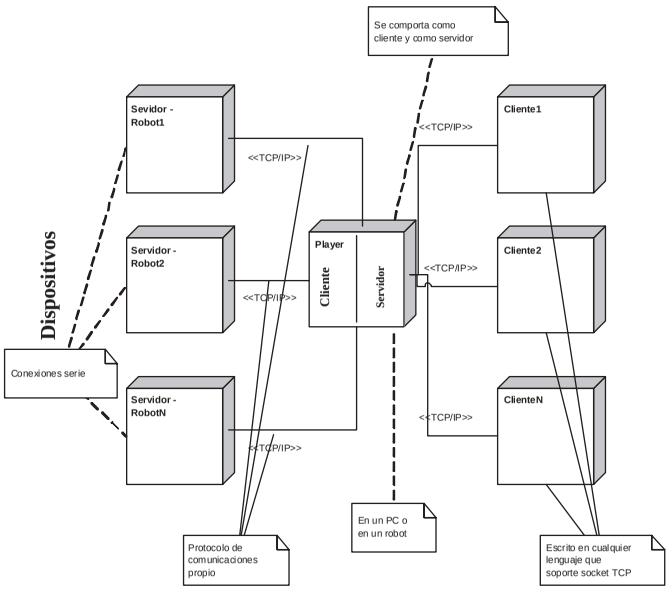


Figura 3.1: Arquitectura Propuesta

La aplicación servidora que compone esta capa accederá por un lado a los dispositivos a través de conexiones serie y por otro se comunicará a través de un protocolo propio sobre TCP/IP, con los *drivers* genéricos que forman parte de *Player*.

En cuanto a la capa intermedia, la constituye *Player* que además de comportarse como servidor atendiendo las peticiones de los programas clientes, ahora se comportará como cliente de la capa más baja (aplicación servidora). Es decir el

acceso a los dispositivos ya no se realizará directamente a través de los *drivers* implementados en *Player*, sino que se crearán unos *drivers* genéricos, que se comportarán como clientes de la aplicación servidora que se ejecuta en el robot. La comunicación entre los *drivers* genéricos (clientes) y el servidor se realizará a través de un protocolo propio. La creación de una serie de *drivers* genéricos (clientes) para cada tipo de dispositivo (uno para sonar, otro para infrarrojos, otro para *position*, otro para *batery...*), evita tener que crear un *driver* específico para cada dispositivo concreto del robot y permite usar estos sin necesidad de modificar código de *Player*. Es decir un robot para cuyos dispositivos existan *drivers* genéricos ya implementados en *Player* y que posea el servidor que proponemos, podrá controlarse a través de *Player* sin modificar su código.

Lo que pretendemos es dejar establecido un modelo de servidor y de *drivers* genéricos (clientes), de manera que un robot que incorpore este modelo de servidor pueda ser integrado y controlado desde *Player* sin necesidad de crear nuevos *drivers*.

La capa más alta está formada por los programas clientes, los cuales pueden estar escritos en cualquier lenguaje que soporte socket TCP y hacen uso de la API (*Aplication Programming Interface*) que *Player* les ofrece para acceder a distintos tipos de dispositivos. Esta capa no se verá afectada por nuestra propuesta.

Cabe destacar que esta arquitectura es para el control de robots desde *Player* a través de conexiones TCP/IP, lo que permite su uso con redes inalámbricas, mejorando de esta forma el radio de acción entre cliente y servidor, con respecto a las comunicaciones a través del puerto serie.

Tanto las comunicaciones entre *Player* y los clientes como entre *Player* y el servidor se realizan sobre TCP/IP, y permiten que *Player* atienda a múltiples clientes al mismo tiempo, así como que acceda a varios servidores de distintos robots, lo cual facilita el trabajo con sistemas multi-robot.

3.2. El Servidor-Robot

Se trata de una aplicación servidora, que debe ejecutarse en el módulo controlador del robot. Su tarea consiste en acceder a los dispositivos del robot, ya sea para obtener sus valores o actuar sobre ellos, a petición de los clientes (en este caso *Player*).

3.2.1. Requisitos

Debido a que *Player* puede realizar peticiones simultáneas sobre distintos dispositivos del robot y éstas deben ser atendidas con la mayor rapidez posible, se ha decidido que el servidor admita múltiples conexiones y sea multiproceso, de manera que cada proceso atienda las peticiones que realice un cliente sobre un determinado dispositivo.

Así mismo debe ser capaz de gestionar un gran número de dispositivos robóticos y acceder a ellos de la forma más rápida posible y evitando problemas de concurrencia por parte de varios procesos. También debe permitir integrar nuevos dispositivos de la forma más sencilla posible.

3.2.2. Diseño Lógico y Diagrama de Clases

El servidor ha sido diseñado utilizando técnicas de orientación a objetos y siguiendo un ciclo de vida en espiral, de forma que sea fácilmente implementable para su integración en cualquier tipo de módulo controlador.

Desde el punto de vista lógico, se pueden diferenciar dos capas en el diseño del servidor (ver figura 3.2). La capa superior es la encargada de gestionar los distintos procesos que constituyen el servidor y de atender las peticiones de los clientes (*drivers* genéricos de *Player*), siguiendo un sencillo protocolo de comunicaciones propio. En cuanto a la capa inferior, es la que realiza el acceso a los dispositivos ya sea para su consulta o actuación. Entre ambas capas debe existir una comunicación en los dos sentidos: la capa superior debe enviar a la inferior los datos de actuación sobre los dispositivos actuadores (se realiza

a través del buffer de tramas de actuadores) y la inferior debe mandar a la superior el valor leido de los dispositivos sensores (se realiza a través del buffer de tramas de sensores).

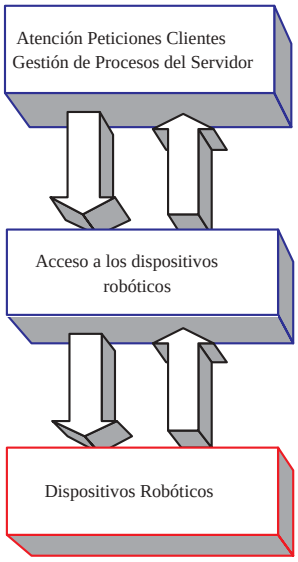


Figura 3.2: Capas de la Arquitectura del Servidor

Esta división en capas del servidor se ha realizado con el fin de independizar el acceso a los dispositivos, de la gestión de procesos y peticiones de los clientes y se ha tenido muy en cuenta a la hora de realizar la descomposición en clases del sistema. De esta forma los usuarios que deseen integrar su robot en *Player* aplicando la arquitectura que proponemos, únicamente deberán adaptar la capa más baja a los dispositivos concretos de su robot, ya que es la que realmente depende de los dispositivos que posea el robot. Esto se traduce en que tendrán que modificar o adaptar sólo algunas clases de los dispositivos concretos.

En cuanto a la estructura de clases del servidor, el diagrama 3.3 representa las clases que lo componen y las asociaciones entre estas. Como puede apreciarse en este diagrama el servidor hace uso de dos listas enlazadas (*Lista*), una formada por los dispositivos sensores del robot (*sonar*, *infrarrojo*, *bateria* y *bumper*) y la otra por los actuadores (*motor* y *relé*). Así mismo utiliza un buffer para almacenar tramas (*BufferTramaSensores*) que contienen las lecturas de los dispositivos sensores, el cual es consultado cada vez que se recibe una solicitud

para conocer su estado. También emplea un buffer de tramas de actuadores (*BufferTramaActuadores*) que sirve para almacenar en él los valores enviados por los clientes para los dispositivos actuadores.

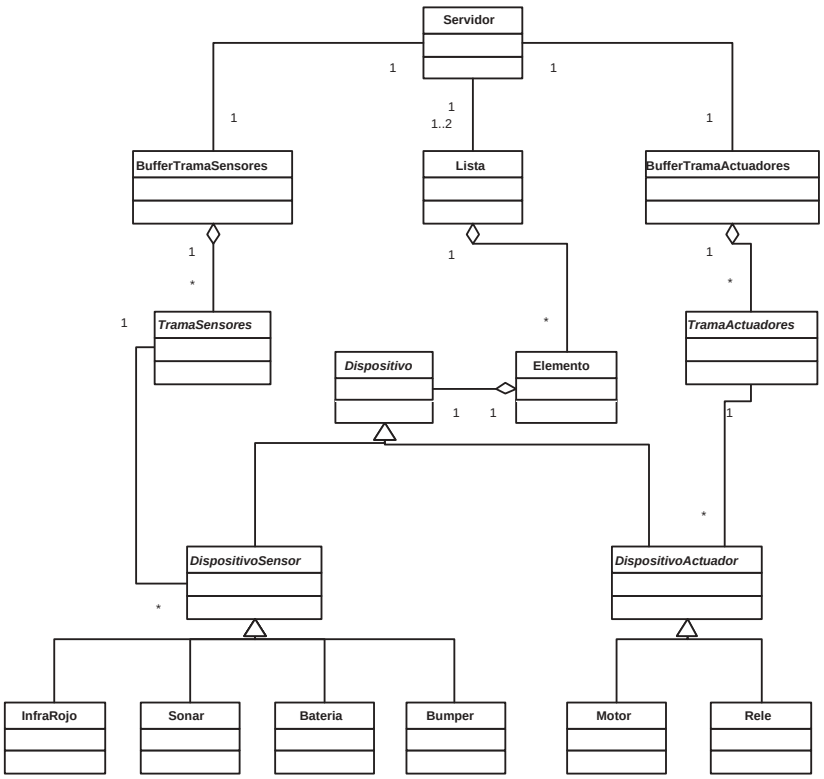


Figura 3.3: Diagrama de Clases del Servidor

En cuanto a las clases que heredan de '*DispositivoSensor*' y '*DispositivoActuador*' representan a distintos tipos genéricos de dispositivos y son las encargadas de comunicarse y entenderse con su correspondiente *driver* genérico integrado en *Player*, así como de acceder a los correspondientes dispositivos físicos.

3.2.3. Procesos

Con respecto a los procesos de los que consta el servidor, se dividen en tres tipos según la tarea que realizan, así son necesarios:

- Un *proceso principal* que se encarga de crear y finalizar al resto de proce-

sos, así como de atender las nuevas conexiones solicitadas desde la parte cliente (*Player*).

- Un *proceso informador/actuador* encargado de acceder a los dispositivos físicos del robot. Su tarea consiste en realizar lecturas periódicas de los dispositivos sensores y depositarlas en una zona de memoria compartida (*buffer de tramas de sensores*) para que el resto de procesos puedan acceder a ellas, asi como en operar sobre los dispositivos actuadores con los valores que los clientes hayan depositado en el *buffer de tramas de actuadores*.

- Un conjunto de *procesos auxiliares* encargados de atender las peticiones de lectura y actuación de una determinada conexión (cliente), a través de un socket TCP. Estos consultan la información de los dispositivos sensores a través del buffer de tramas de sensores y almacenan los valores para los dispositivos actuadores en el buffer de tramas de actuadores.

El motivo por el que sólo un proceso se encarga de acceder a los dispositivos físicos, es para evitar problemas de concurrencia, ya que en la mayoría de las arquitecturas de robots móviles no es posible acceder a varios dispositivos al mismo tiempo. También podría realizarse haciendo uso de secciones críticas, pero como lo que prima en este tipo de aplicaciones es que sean lo más rápidas posibles es más eficiente realizarlo de ese modo. Este proceso se encarga de ejecutar el código asociado a la capa lógica que cubre el acceso a los dispositivos físicos, y que deberá modificarse cada vez que se incorpore un nuevo dispositivo al robot o que el servidor se adapte a un nuevo robot.

Los diagramas de actividades 3.4, 3.6, 3.7 asociados a cada tipo de proceso, reflejan el funcionamiento del servidor.

El Proceso Principal

Como se muestra en el diagrama de actividades del *proceso principal* (figura 3.4), cuando el servidor se arranca sólo existe el *proceso principal* que se encarga de iniciar las listas dinámicas con los dispositivos sensores y actuadores que va a manejar. Estos dispositivos deben poderse leer de un fichero de configuración

o establecerse en el código del programa. A continuación se encarga de crear el *proceso informador/actuador*, que es el encargado de acceder a los dispositivos físicos, ya sea para leer el valor de los sensores o para dar valores a los actuadores.

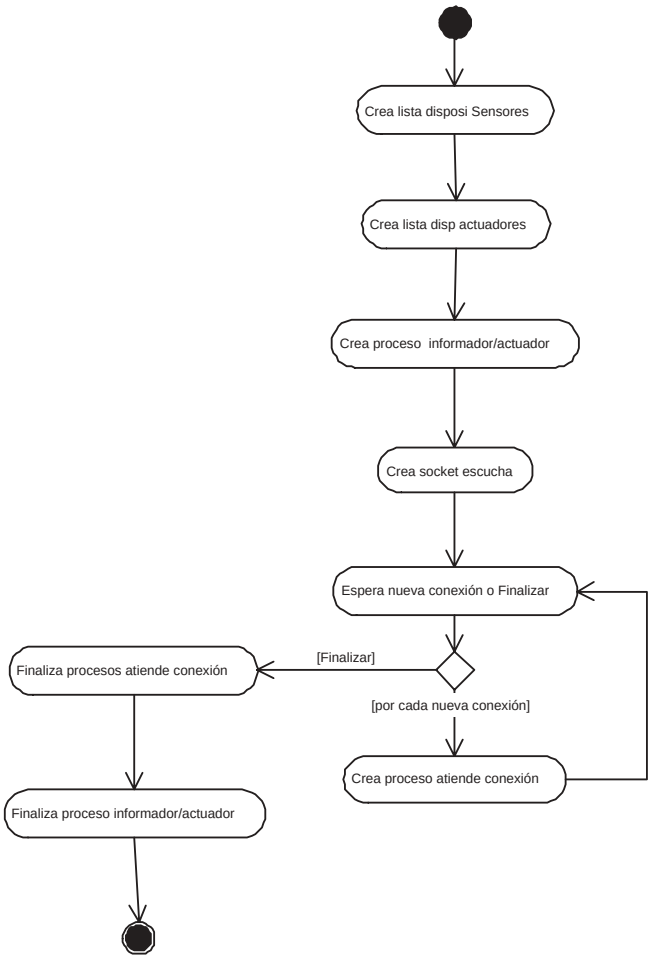


Figura 3.4: Diagrama de Actividades del proceso principal

Una vez hecho esto el *proceso principal* crea un socket TCP y se queda a la espera de conexiones por parte de los clientes. Por cada conexión que se produce desde un cliente (normalmente *drivers* genéricos de *Player*) crea un proceso que se encarga de atender todas las peticiones que el cliente realice sobre un determinado dispositivo (ver figura 3.5).

Si uno de los procesos encargados de atender las peticiones de los clientes,

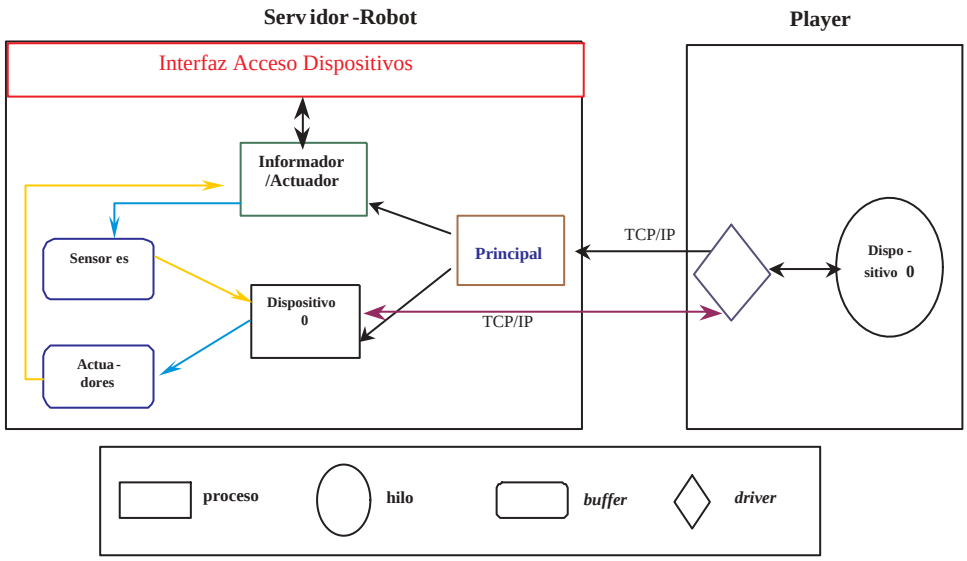


Figura 3.5: Servidor con un único proceso atendiendo peticiones

recibe la orden de finalizar el servidor, se encarga de comunicarlo al *proceso principal*, el cual finaliza el resto de procesos ordenadamente y el servicio.

El Proceso Informador/Actuador

En cuanto al *proceso informador/actuador*, como ya se ha comentado anteriormente, es el único que accede a los dispositivos, debido fundamentalmente a evitar problemas de concurrencia y mejorar la eficiencia, en cuanto a la velocidad de respuesta, del servidor.

El acceso a los dispositivos por parte de este proceso se realizará a través de las clases que se asocian a cada tipo de dispositivo. Como cada robot puede tener una forma distinta de acceder a estos, deberá crearse una clase por cada tipo de dispositivo que quiera ser controlado, que contenga todas las funciones que le permitan operar sobre él.

El diagrama de actividades de la figura 3.6 resume el funcionamiento del *proceso informador/actuador*.

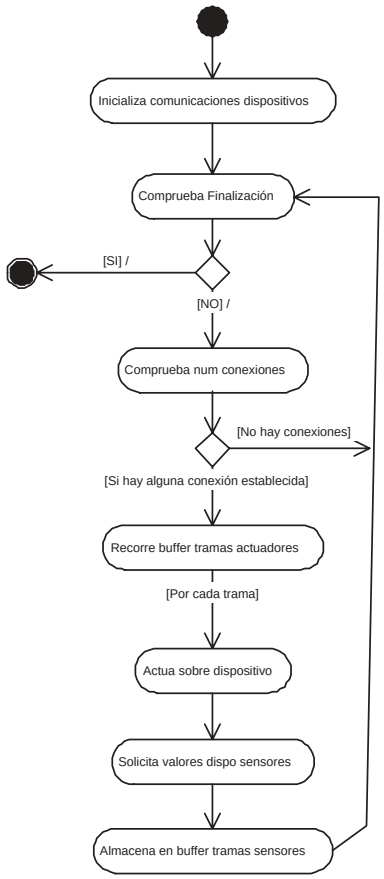


Figura 3.6: Diagrama de Actividades proceso Informador/actuador

Como puede observarse, una vez creado este proceso por el *proceso principal*, lo que hace es iniciar las comunicaciones con los dispositivos físicos, normalmente a través de conexiones serie. A continuación comprueba el número de conexiones que hay establecidas con clientes (normalmente *drivers* de *Player*) y si hay alguna establecida realiza las siguientes acciones. En primer lugar debe leer del buffer de actuadores las órdenes que los otros procesos, a petición de los clientes, depositan y llevarlas a cabo sobre los dispositivos. Cabe destacar que el buffer de actuadores se debe comportar como una cola, es decir las órdenes almacenadas tienen que irse extrayendo por orden de llegada de modo que todas se ejecuten. Por cada orden que extrae y ejecuta, el proceso solicita a los dispositivos sensores sus valores y los almacena en el buffer de sensores, con el fin de tener la información más reciente posible. El buffer de sensores será consultado

por los procesos que atienden a los clientes para proporcionar el valor de los dispositivos sensores.

Cuando el *proceso informador/actuador* recibe la orden de finalizar desde el *proceso principal*, cierra ordenadamente las comunicaciones con los dispositivos físicos y acaba.

Un aspecto interesante que hemos considerado a la hora de diseñar el servidor, ha sido la posibilidad de asignar distintas prioridades a los procesos. De manera que se pueda dar más o menos prioridad a un proceso encargado de atender las peticiones sobre un determinado dispositivo o al proceso encargado del acceso a los dispositivos. Cuanta más prioridad tenga un proceso mayor tiempo de CPU se le asignará, por tanto las peticiones que atienda se llevarán a cabo más rápidamente. Esto permite, por ejemplo, que el servidor de más prioridad a las órdenes que recibe de movimiento del robot que a otras. Las prioridades se asignarán en el momento de la creación de los procesos, aunque sería interesante poder cambiarlas en tiempo de ejecución. La política de asignación de prioridades dependerá de las necesidades de cada robot en particular, aunque inicialmente todos los procesos deberían tener la misma prioridad.

Por otro lado hay que destacar que el servidor se comunica con *Player* (con los *drivers* genéricos) a través de un protocolo propio (descrito en el apartado 3.4) sobre TCP/IP y que puede interactuar no sólo con *Player* sino con cualquier otro programa cliente que respete el protocolo de comunicación establecido.

A la hora de realizar la implementación de esta arquitectura hay algunos aspectos que pueden ser modificados o adaptados a determinadas necesidades particulares. Por ejemplo que el buffer de sensores sólo almacene los últimos valores de los dispositivos, o que el *proceso informador/actuador* ejecute más de una orden antes de leer el valor de los sensores.

Los Procesos Auxiliares

Son un conjunto de procesos que se crea dinámicamente y se encargan de atender las peticiones de los clientes. La arquitectura está pensada para crear

uno por cada nuevo cliente que se conecte al servidor, pero es conveniente tener en cuenta posibles limitaciones relativas al número de procesos que soporte el módulo controlador donde se ejecute el servidor.

Estos procesos se comunican directamente con los clientes y lo hacen a través de un protocolo propio que se describe en el apartado3.4. Los clientes serán los *drivers* genéricos de dispositivos integrados en *Player*. Aunque es posible que cualquier otro programa cliente que respete el protocolo se comunique con ellos.

A continuación reflejamos, también con un diagrama de actividades (figura 3.7), el funcionamiento de los procesos encargados de atender las peticiones de los clientes.

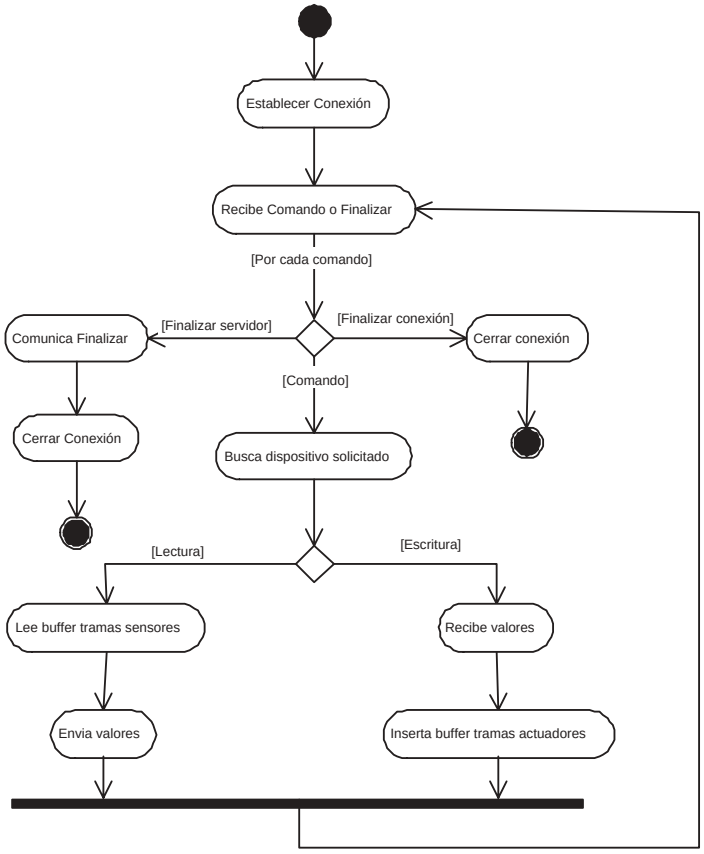


Figura 3.7: Diagrama Actividades proceso atiende conexión

Estos procesos, cuando son creados por el *proceso principal*, deben recibir

el descriptor de socket a través del cual deben atender al cliente y una vez establecida la conexión con este, deben esperar las solicitudes que realice. Estas pueden ser de varios tipos:

- Orden: dependiendo del tipo de orden (apartado 3.4), el cliente solicita el valor de un determinado dispositivo sensor o la actuación sobre un dispositivo actuador. En el segundo caso el cliente debe enviar los valores que quiere establecer en el dispositivo.

- Finalizar conexión: el cliente decide finalizar sus solicitudes y el proceso, tras cerrar las conexiones acaba.

- Finalizar servidor: el cliente solicita la finalización del servidor. En este caso el proceso debe comunicárselo al *principal* para que este acabe con los otros procesos. Una vez comunicado cierra las conexiones y acaba.

Cuando estos procesos reciben una orden de solicitud de lectura de los valores de un dispositivo sensor, consultan dichos valores en el *buffer de tramas de sensores*, que contiene valores muy recientes, y lo envian automáticamente al cliente. De este modo evitamos que cada proceso de este tipo tenga que acceder a los dispositivos para realizar la consulta y que su respuesta sea mucho más lenta.

Si la orden es de actuación, lo que debe hacer es almacenarla en el *buffer de tramas de actuadores* para que el *proceso informador/actuador* lo ejecute con la mayor rapidez posible.

Es importante destacar que el buffer de actuadores debe estar protegido con un *mutex* o algo similar para evitar problemas de acceso concurrente. No ocurre lo mismo con el de sensores, puesto que sólo un proceso escribe en él mientras que el resto sólo leen.

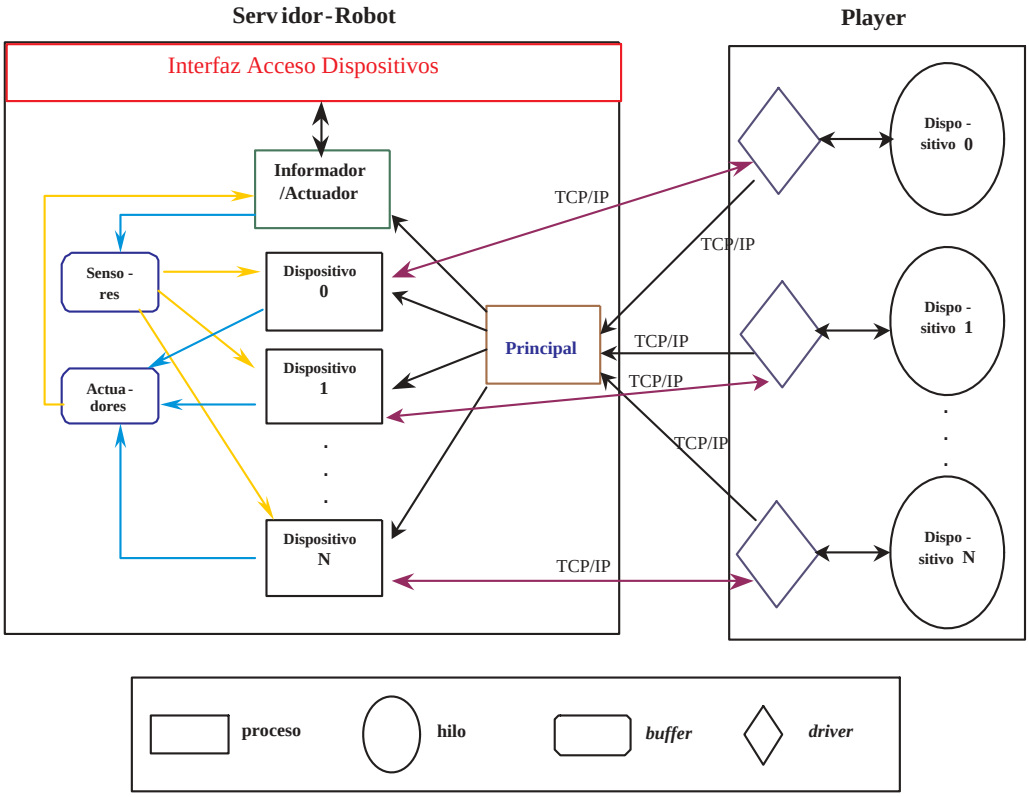


Figura 3.8: Representación del servidor y sus procesos

3.3. El Cliente

Debido a la arquitectura que se ha propuesto (figura 3.1), el cliente puede ser un programa escrito en cualquier lenguaje que soporte socket TCP, con el único requisito de que respete el protocolo de comunicaciones (sección 3.4) que impone el servidor. Puede establecer una única conexión con el servidor y a través de ella acceder a distintos dispositivos de una forma secuencial, o puede establecer múltiples conexiones y acceder simultaneamente a varios dispositivos del robot.

En nuestra arquitectura, teniendo en cuenta que buscamos integrar múltiples robots en *Player*, convertimos a este en cliente. De manera que solicite al servidor (que se encuentra en el robot) mediante comunicaciones inalámbricas (TCP/IP), información o actuación sobre los dispositivos que posea el robot.

3.3.1. Los *Drivers* Genéricos

Para convertir a *Player* en cliente, debemos respetar su estructura, su forma de trabajo y las interfaces que ofrece a sus clientes. Pues bien, puesto que *Player* está preparado para integrar en él *drivers* de nuevos dispositivos, nuestra propuesta consiste en integrar una serie de *drivers*, uno por cada tipo genérico de dispositivo, que se comporten como clientes. Cada uno de estos *drivers* es ejecutado por *Player* en un hilo independiente, y es desde el propio *driver* desde donde se establece una conexión con el servidor que se ejecuta en el robot, para enviar peticiones y recibir datos del correspondiente dispositivo.

Cuando hablamos de un *driver* por cada tipo genérico de dispositivo, lo que pretendemos es que cada uno de ellos sirva para cualquier dispositivo de ese tipo. Por ejemplo, que el *driver* genérico de los sonars sirva para cualquier número y tipo de sonars que pueda incorporar un robot móvil. De esta forma evitamos tener que crear un nuevo *driver* por cada dispositivo robótico particular que se quiera integrar en *Player* y permitimos la integración de robots, que incluyan el modelo de servidor propuesto, sin tener que realizar ninguna modificación sobre el código de *Player* (siempre que existan *drivers* genéricos para sus dispositivos).

El comportamiento de los *drivers* genéricos debe ser distinto dependiendo de si es de un dispositivo sensor o actuador, puesto que los primeros se limitan a pedir los valores mientras que los segundos envian los valores cada vez que *Player* se los proporcione. También hay que tener en cuenta el número de instancias que pueda tener y el tipo de valores que devuelva o reciba el dispositivo, los cuales vienen impuestos por *Player*, ya que debemos respetar las interfaces que proporciona a sus clientes. Así pues, no es posible construir un único *driver* genérico para dispositivos sensores y otro para actuadores, sino que habrá que crear uno por cada tipo genérico de dispositivo.

3.3.2. Diseño Lógico

Como ya se comentó en el apartado 2.3 cuando un cliente se conecta a *Player* y realiza la primera solicitud sobre un dispositivo, *Player* crea un hilo (uno por cada dispositivo), sobre el que se ejecuta el *driver* asociado a ese dispositivo. En nuesto caso estará creando un nuevo cliente que se comunica con el servidor del robot, realizando las peticiones oportunas. Cuando el dispositivo no es utilizado por ningún cliente, *Player* se encarga de finalizar su hilo correspondiente y por tanto nuestro cliente.

Los diagramas de actividades 4.11 y 4.12 reflejan el funcionamiento de un *driver* genérico de un dispositivo sensor y otro de un actuador.

Como puede observarse en la figura 4.11, lo primero que hace cualquier dispositivo sensor es establecer la conexión con el servidor. A continuación solicita el número de ejemplares de dicho dispositivo que posee el robot, con el fin de poder reservar memoria.

Los *drivers* genéricos de los dispositivos sensores no solicitan los valores a petición de los programas clientes, sino que continuamente estan recibiendo los valores de los dispositivos sensores. Si estos valores cambian respecto a los últimos recibidos, se encapsulan en unos tipos válidos para *Player* y se escriben en el buffer de datos asociado al dispositivo, de modo que cuando el cliente los solicite serán entregados por *Player*. De esta forma los programas clientes disponen de unos valores más precisos, puesto que si los valores no cambian

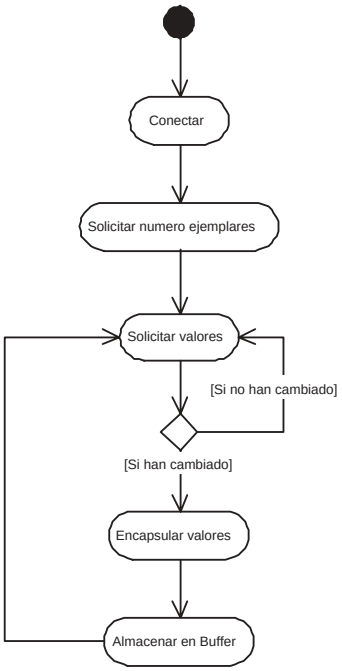


Figura 3.9: Diagrama de actividades de un sensor

reciben los mismos y en cuanto cambian reciben los nuevos.

En cuanto a los *drivers* genéricos de los dispositivos actuadores, cabe destacar que, como se muestra en la figura 4.12, leen los valores que los clientes envían al dispositivo de un buffer de datos asociado a él y que sólo hacen peticiones al servidor cuando hay valores que mandar. Con el fin de evitar comunicaciones innecesarias, cuando los valores a enviar son iguales que los últimos mandados no se envian.

3.3.3. Diagrama de Clases

Como ya se explicó en la sección 2.6, *Player* se ha desarrollado siguiendo un enfoque orientado a objetos y está formado por una compleja jerarquía de clases que determinan su funcionamiento y que permiten, haciendo uso de la herencia, crear nuevas clases que constituyen *drivers* para nuevos dispositivos que quieran manejarse a través de él. Para que la creación de los *drivers* genéricos sea lo más

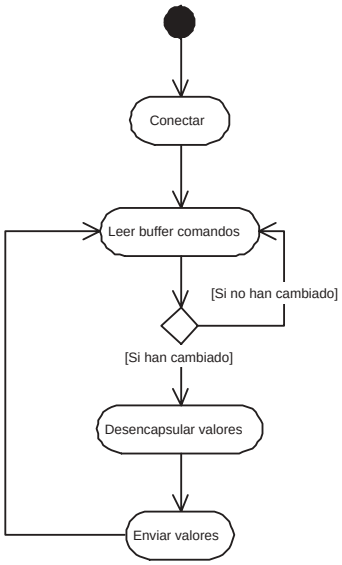


Figura 3.10: Diagrama de actividades de un actuador

sencilla posible e implique desarrollar la menor cantidad de código, hemos creado una jerarquía de clases que permite heredar muchos de los métodos necesarios ya implementados. La figura 3.11 muestra un diagrama con la estructura de clases que proponemos para la integración de los *drivers* genéricos.

La clase '*CDevice*' es abstracta, propia de *Player* y contiene todos los métodos que deben redefinir los *drivers* que se implementen, para adaptarse al modo de trabajar de *Player*. Contiene, entre otros, métodos virtuales (deben redefinirse), para leer valores de los buffers, para escribir valores en ellos, para configurarlos, un método principal...

En cuanto a la clase '*Dispositivo Genérico*' también es abstracta, pero esta es propuesta por nosotros. Hereda de la anterior y contiene los métodos y atributos comunes tanto a los *drivers* de dispositivos sensores como a los de actuadores, propios de cualquier cliente, como conexión y desconexión con el servidor, enviar orden...

Las clases '*Dispositivo Generico Sensor*' y '*Dispositivo Generico Actuador*' heredan de las dos anteriores y encapsulan los métodos y atributos propios de cada tipo de *driver* (sensores y actuadores). De estas clases debe heredar

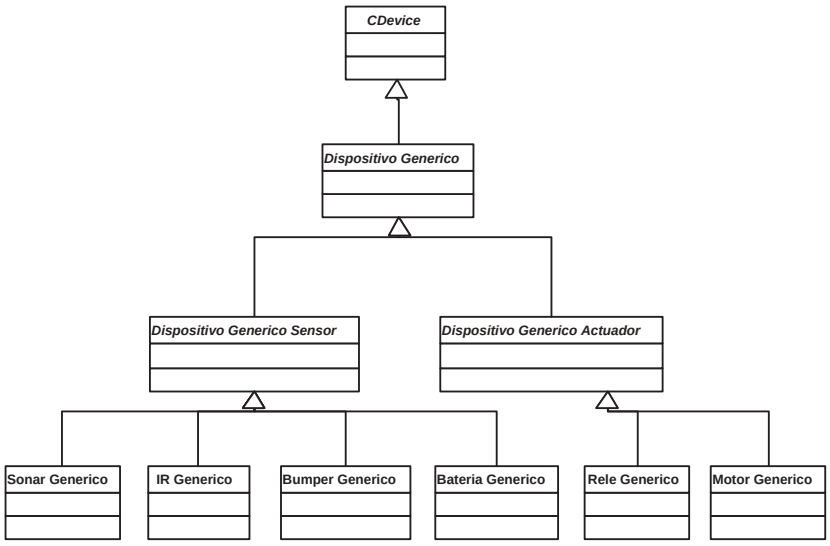


Figura 3.11: Diagrama de clases del cliente

cualquiera que implemente un *driver* genérico válido para nuestra arquitectura. En el diagrama, heredando de estas, aparecen una serie de clases que representan *drivers* genéricos para los dispositivos indicados. Cuantas más haya, más dispositivos se podrán integrar sin modificar el código de *Player*.

3.4. Protocolo entre cliente y servidor

Como se ha comentado con anterioridad la comunicación entre cliente y servidor se realiza a través de socket TCP, siguiendo un sencillo protocolo (ver figura 3.12) que se describe a continuación.

En primer lugar el cliente envía un *byte* a modo de orden indicando a qué dispositivo quiere acceder o si quiere finalizar la conexión o el servidor.

- 0 Sonars.
- 1 Infra-rojos.
- 2 Bateria.

- 3 Bumpers.
- 4 Motores.
- 5 Relés.
- 6 Finalizar conexión.
- 7 Finalizar servidor.

Si el dispositivo es sensor, el servidor responde con un *byte* indicando el número de unidades de ese dispositivo que posee el robot. A continuación, manda el conjunto de valores de todos ellos. Cada uno de esos valores irán en uno o dos *bytes*, dependiendo del tipo de dato que sean (caracter, entero...)

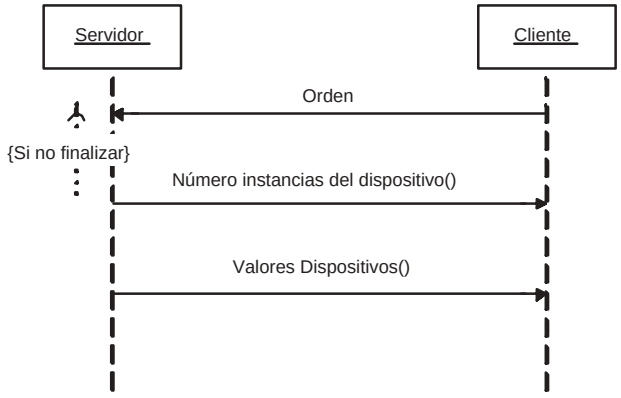


Figura 3.12: Protocolo entre los *drivers* genéricos de dispositivos sensores y el servidor

En caso de tratarse de un dispositivo actuador (ver figura 3.13), lo que se hace es mandar, después del primer byte con la orden, otro indicando sobre cuántas unidades de ese dispositivo se va a actuar y a continuación sus valores. Si se produce un error en el envío de los valores, estos vuelven a ser reenviados.

El tipo de dato de los valores de los dispositivos viene impuesto por *Player*, ya que tiene que ajustarse a las interfaces que ofrece a los programas clientes.

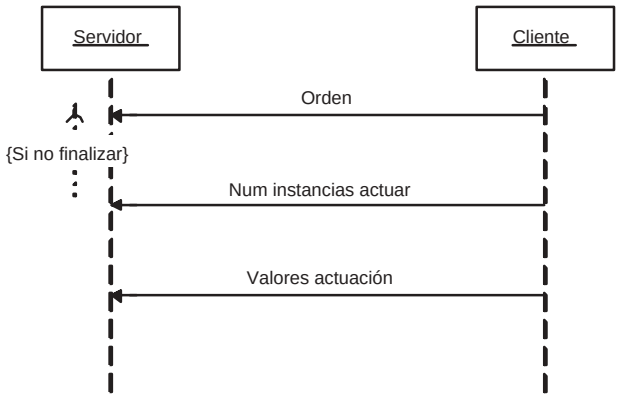


Figura 3.13: Protocolo entre los *drivers* genéricos de dispositivos actuadores y el servidor

4

Aplicación de la Arquitectura

Para probar el correcto funcionamiento de la arquitectura descrita en el capítulo anterior, esta se ha implantado sobre dos robots móviles con soporte de comunicaciones TCP/IP, desarrollados íntegramente en el Departamento de Informática y Automática de la Universidad de Salamanca.

En este capítulo se describe la estructura física de los robots que se han integrado en *Player* siguiendo nuestra arquitectura, así como la implementación realizada de la misma.

4.1. Robot *Sabina*

Sabina es un robot móvil, creado para la limpieza de superficies planas donde existen objetos (obstáculos) y cuya distribución es desconocida e incluso cambiante.

4.1.1. Estructura Física

Sabina posee una base cilíndrica hecha de aluminio sobre la que se levanta una estructura también cilíndrica, de menor diámetro y de PVC. El diámetro de nuestro robot es de 50 cm y su altura de 40 cm. Se apoya en el suelo a través de dos ruedas que le permiten avanzar, retroceder y girar alrededor de un eje vertical, gracias a dos motores dispuestos en configuración diferencial.

En su parte inferior delantera cuenta con unos sensores de infrarrojos y una pequeña caja donde se encuentra el módulo controlador (DK-40) en el que se ejecuta el servidor, y en la inferior trasera con un cepillo eléctrico.

Figura 4.1: Vista General del Robot *Sabina*

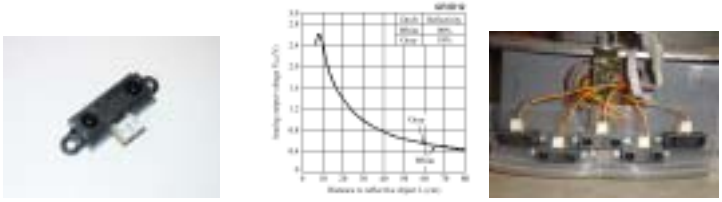
4.1.2. Dispositivos Electrónicos

El robot *Sabina* posee una serie de dispositivos que pueden agruparse en dos módulos: uno sensorial y otro de actuación. Los dos son totalmente independi-

entes, es decir se puede trabajar sobre ellos simultáneamente.

El módulo sensorial tiene como objetivo detectar los posibles obstáculos que el robot pueda encontrar en su camino y que impidan su movimiento. Constituye su sistema de percepción, y debe permitirle detectar obstáculos a cualquier altura que le impidan avanzar. Está formado por los siguientes dispositivos:

- Conjunto de 5 sensores medidores de distancia por infrarrojos (Sharp GP2D12), cuya misión es detectar obstáculos en el suelo, puesto que se encuentran colocados en la parte frontal de *Sabina* a 5cm del suelo (ver figura 4.2(c)). Como características más importantes de este modelo de dispositivo, cabe destacar que indican, mediante una salida analógica entre 0 y 3V, la distancia medida. El voltaje de salida varía de forma no lineal cuando se detecta un objeto en una distancia entre 10 y 80 cm y su valor es actualizado cada 40 ms. Funciona con una tensión de alimentación próxima a los 5V.



(a) (b) (c)

Figura 4.2: Sensor de infrarrojos GP2D12, salida analógica que proporciona y conjunto de infrarrojos de *Sabina*.

- Conjunto de 8 sensores (Polaroid 6500) medidores de distancia por ultrasonidos (sonars), cuyo objetivo es detectar objetos alrededor de *Sabina*, pues se encuentran ubicados en todo el diámetro de la estructura cilíndrica (ver figura 4.1). Necesita dos señales digitales de control, una para lanzar el ultrasonido y otra para recibir el eco en formato TTL (Transistor-Transitor Logic). Cada uno de ellos lleva asociado un transductor que se encarga del envío y la recepción de las señales de ultrasonido. El alcance de este modelo de dispositivo va de 15cm a 11m aproximadamente y funciona con una alimentación que va de 4.5V a 5.8V.

(a) (b)

Figura 4.3: Sensor de ultrasonidos Polaroid 6500 y Transductor

- 1 microcontrolador (PIC 16F876) a cuyos pines se conectan los dispositivos anteriores, y que contiene un sencillo programa que permite leer sus valores. Este dispositivo posee una memoria Flash de 8 KBytes, 28 pines y 3 puertos de E/S (2 de 8 bits y 1 de 6 bits con 5 lineas de entrada en convertidor A/D).

En cuanto al módulo de actuación, su tarea consiste en permitir que el robot se desplace y que realice su tarea objetivo (cepillar). Está formado por los siguientes dispositivos:

- 2 motores con reductora y control PWM (*Pulse Width Modulation*) [Bar] que le permiten moverse libremente, ya que pueden girar tanto hacia adelante como hacia atrás independientemente (figura 4.4). Funcionan con un voltaje de 12V y son capaces de dar un máximo de 106rpm. La conexión al PIC se realiza a través de un *driver* de potencia (L298).

Figura 4.4: Motores que usa *Sabina*

- 1 cepillo eléctrico, con una batería propia (figura 4.5).
- 2 relés de doble contacto, uno de los cuales se emplea para la activación y desactivación del cepillo (el otro de momento no se usa). Funcionan con

un voltaje de 6V. La conexión al PIC se realiza a través de un *driver* de potencia (L293).

- 1 microcontrolador (PIC 16F876) a cuyos pines se conectan, por medio de *drivers* de potencia, los relés y motores y que posee un sencillo programa que se encarga de controlar los dispositivos. Este dispositivo posee una memoria flash de 8 KBytes. Tiene 28 pines y 3 puertos de E/S (2 de 8 bits y 1 de 6 bits con 5 lineas de entrada en convertidor A/D).

Figura 4.5: Vista del cepillo y motores de *Sabina*

Tanto el PIC que controla los dispositivos sensores como el de los actuadores, se encuentran ubicados dentro del cilindro de PVC, en una pequeña placa de prototipos con un cristal de cuarzo a 4Mhz y una serie de conectores que permiten la conexión de los pines a los distintos dispositivos (figura 4.7).

Además de los dispositivos descritos anteriormente, *Sabina* también cuenta con un módulo controlador de la casa BECK. Se trata del Dk-40 (figura 4.20) con un chip IPC@CHIP SH12 [Ipc], cuyas características hardware más importantes son:

- Procesador a 186Mhz.
- Memoria RAM (256 Mb).
- Flash Disk.
- Dos puertos serie.
- Tarjeta de red.

A través de los puertos serie se realiza la conexión con los PIC del módulo sensorial y de actuación.

En cuanto a las características software más relevantes del DK-40 destacamos:

- Servidor Web.
- Servidor FTP.
- Servidor Telnet.
- Permite conexiones por socket TCP y UDP (hasta 64).
- Posee una API (*Aplication Programming Interface*) que ofrece funciones para el control de los puertos serie, los *sockets* y la creación de procesos (hasta 35).
- Soporta semáforos.
- Incorpora una *shell* de órdenes tipo DOS y permite ejecutar aplicaciones compiladas para MS-DOS.

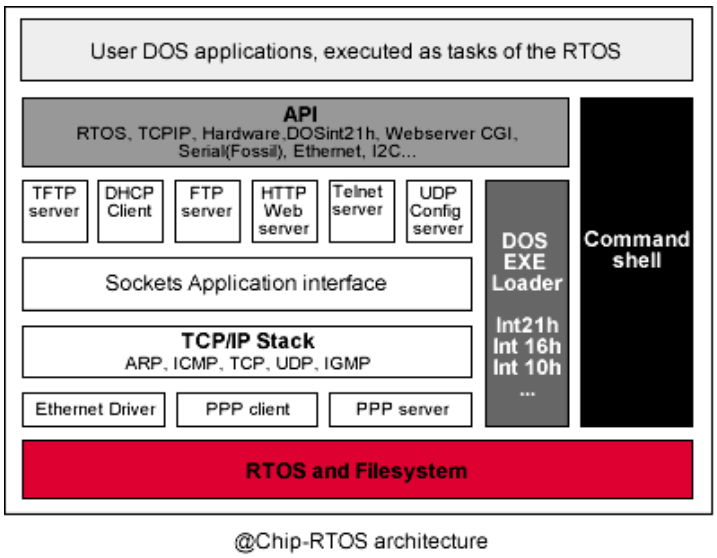


Figura 4.6: Arquitectura interna del Dk40

En este módulo controlador es donde se ejecuta el programa servidor que accede, a través de los PIC, a los distintos dispositivos a petición de Player.

La alimentación de todos los dispositivos y del módulo controlador se realiza a través de una batería de plomo de 12V y 7A que *Sabina* lleva incorporada, y puesto que no todos los dispositivos funcionan al mismo voltaje se emplean una fuente de 12V y otra de 5V. El único dispositivo que lleva alimentación propia es el cepillo eléctrico que lleva una batería incorporada, debido al alto consumo que produce. El voltaje de la batería también es controlado a través del PIC de sensores, conectando uno de sus pines a un controlador de voltaje.

La figura 4.7 muestra el esquema de conexiones entre los distintos dispositivos de *Sabina*.

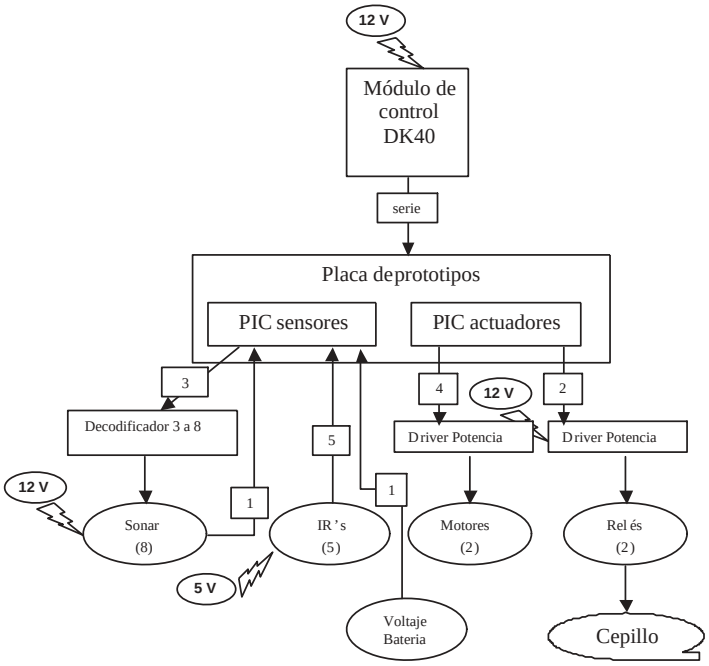


Figura 4.7: Esquema de conexiones de *Sabina*

4.2. Integración del Robot *Sabina*

La integración de *Sabina* en *Player* para ser controlado a través de él usando comunicaciones TCP/IP, constituye la primera prueba que vamos a realizar de la arquitectura propuesta. Por tanto, aún no hay implementado en *Player* ningún *driver* genérico de dispositivo, ni un servidor robótico que podamos adaptar a los dispositivos concretos de *Sabina*.

Así pues el primer paso consiste en crear una serie de *drivers* genéricos para los dispositivos que posee *Sabina*, que son: sonars, infrarrojos, relés, baterías y motores. Cada uno de estos *drivers* genéricos servirá para cualquier dispositivo de ese tipo que posea un robot que se quiera integrar. Así mismo, cada uno de ellos estará asociado a una *interfaz* siendo, en algún caso, necesario crear una nueva asociada a un nuevo tipo de dispositivo.

Una vez implementados los *drivers* genéricos para los dispositivos que posee *Sabina*, el segundo paso será desarrollar una aplicación servidora, que se ejecutará en el módulo controlador de este, y que también puede servir como base para la creación de futuras aplicaciones servidoras, ya que la capa encargada de la gestión de procesos y peticiones de los clientes (figura 3.2) debe ser igual en todas las aplicaciones servidoras.

4.2.1. Creación de los *drivers* genéricos para los dispositivos de *Sabina*

Como ya se comentó en el capítulo 2, *Player* se encuentra implementado en C++ y hace uso de una compleja jerarquía de clases que debe conocerse para añadir un *driver*. Para integrar un nuevo *driver* es necesario crear una clase que herede de '`CDevice`' (sección 2.5.1).

Las clases de nuestros dispositivos genéricos, además de heredar de '`CDevice`', lo harán de una clase base que hemos creado llamada '`DispositivoGenerico`' (figura 4.8) que implementa los métodos y atributos comunes a cualquier dispositivo sensor y actuador, como son:

- `Conecta` método encargado de establece la conexión con el servidor. Recibe como argumento un puerto y la IP del servidor. Devuelve 1 si todo ha ido bien.
- `Desconecta` método encargado de finalizar ordenadamente la conexión con el servidor. Devuelve 1 si todo ha ido bien.
- `EnviaComando` método encargado de enviar una orden al servidor. Recibe como argumento el orden a enviar. Devuelve un 1 si todo ha ido bien.

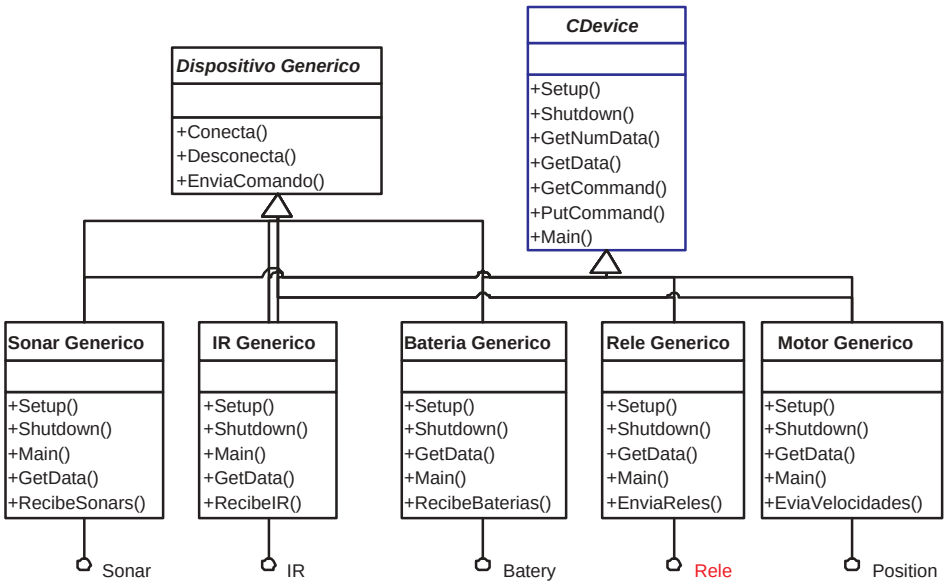


Figura 4.8: Diagrama de Clases implementación *drivers* genéricos

De esta forma las clases que implementan los *drivers* genéricos de los dispositivos heredarán estos métodos y no necesitarán implementarlos. No obstante, hay una función que es propia de cada dispositivo genérico. En el caso de los dispostivos actuadores es la que se encarga de enviar los valores, mientras que en los dispositivos sensores es la que se encarga de recibirlos (figura 4.8). Esta función debe ser propia de cada dispositivo ya que unos envían valores al servidor y otros los reciben de él. Pero además de esto el tipo y número de esos valores variará de unos dispositivos a otros.

También deben implementar, las clases de los *drivers* genéricos, las funciones

virtuales puras heredadas de la clase '`CDevice`' y aquellas otras que aún no siendo puras deban tener otro comportamiento distinto del definido por esa clase (las funciones de esta clase han sido explicadas en el apartado 2.6). Los métodos virtuales puros que obliga a redefinir la clase '`CDevice`' son:

- `Setup` Se llama cuando el primer cliente solicita a *Player* información del dispositivo. En él se establece la conexión con el servidor y se crea un hilo sobre el que se ejecutará la función principal (`Main`) asociada al dispositivo.
- `Shutdown` Cuando el último cliente se desconecta de *Player*, se invoca a este método. En él se desconecta del servidor y se finaliza el hilo.
- `Main` Es la función principal, se ejecuta al crearse el hilo asociado al dispositivo y su implementación es muy parecida para todos los dispositivos sensores y para todos los actuadores.

  Una vez redefinidas las funciones miembro heredadas es necesario implementar dos funciones adicionales que no van a pertenecer a la clase del *driver*, pero que son exclusivas de cada uno. La primera es una función que crea y devuelve una nueva instancia del *driver* (figura 4.9). Es invocada desde *Player* cada vez que un cliente solicita trabajar con un dispositivo usando ese *driver*. No tiene por qué llamarse de ningún modo, aunque suele denominarse '`NombreDispositivo_Init`' (figura 4.9).

```
CDevice* Position_Sabina_Init(char* interface, ConfigFile* cf, int section)
{
  if(strcmp(interface, PLAYER_POSITION_STRING))
  {
    PLAYER_ERROR1("driver \"position_sabina\" does not support interface \"%s\"\n", interface);
    return(NULL);
  }
  else
    return((CDevice*)(new CPositionSabina(interface, cf, section)));
}
```

Figura 4.9: Implementación de la función 'Init' del *driver* genérico de Position

Con respecto a la segunda es una función que registra el nuevo *driver* en un objeto de tipo *driverTable*, que es una tabla global que posee *Player* con los *drivers* que pueden ser instanciados. Suele denominarse '`NombreDispositivo_Register`' (figura 4.10).

```
void Position_Sabina_Register(DriverTable* table)
{
  table->AddDriver("position_sabina", PLAYER_ALL_MODE, Position_Sabina_Init);
}
```

Figura 4.10: Implementación de la función 'Register' del *driver* genérico Position

Finalmente y para que *Player* conozca la existencia de nuestros nuevos *drivers*, es necesario incluir en la función `register_devices` del fichero '`deviceregistry.cc`' la llamada a la función de registro de nuestro *driver*. De esta forma cada vez que se crea una instancia de *Player* se invoca a la función `register_devices` y se cargan en un objeto de tipo '`DriverTable`' todos los *drivers* disponibles.

Una vez creadas las nuevas clases y modificadas algunas de las existentes, hay que crear una biblioteca estática que contenga el código de los *drivers* creados para los nuevos dispositivos. Para ello se deben modificar los *makefiles* (`Makefile.in`) y ficheros de configuración (`configure.in`), ya que *Player* hace uso de las herramientas *Automake* [Foub] y *Autoconf* [Foua] para la generación y compilación del código, de forma que se cree la nueva biblioteca y se enlace con el ejecutable de *Player*.

Funcionamiento de los *drivers* genéricos

El funcionamiento de los *drivers* genéricos de los dispositivos sensores puede resumirse del siguiente modo. Primero se pide el número de instancias que posee el robot de ese dispositivo (para reservar memoria) y luego se entra en un bucle que solicita continuamente los valores del dispositivo al servidor. Tras adaptar los valores recibidos a los tipos de datos que *Player* define para cada dispositivo, los almacena en el buffer de datos asociado a ese dispositivo (ver figura 4.11).

En el caso de los actuadores, también es un bucle en el que se espera a que haya alguna orden en el buffer de órdenes asociado a ese dispositivo. Cuando llega una nueva orden se lee, se desencapsula y se envía al servidor. Tras enviarla se vuelve a encapsular en el tipo definido por *Player* para ese dispositivo y se almacena en el buffer de datos asociado a él. De esta forma

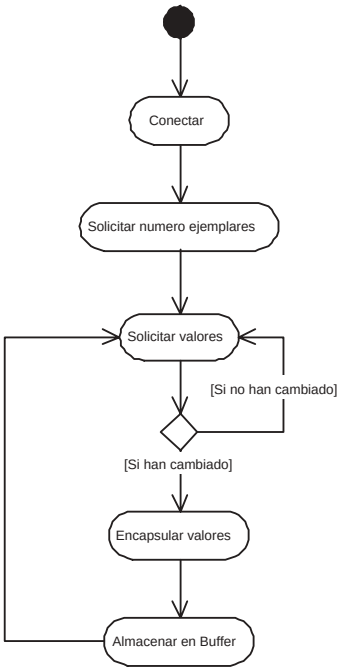


Figura 4.11: Diagrama de Actividades de un dispositivo genérico sensor

si el cliente solicita los valores de ese dispositivos no es necesario pedírselos al servidor ya que los últimos enviados se encuentran en el buffer de datos (ver figura 4.12). En el caso de los actuadores se almacena el último valor enviado al servidor para evitar mandar valores duplicados, ya que si el siguiente valor a mandar es igual que el último enviado no se reenvia.

4.2.2. Creación de una *interfaz* para el dispositivo Relé

Es necesario que todo *driver* se adapte a un *interfaz*, ya que *Player* ofrece a sus clientes las *interfaces* para el manejo de los dispositivos, haciendo uso del *driver* que el cliente quiera. Esto permite abstraer a los clientes de las particularidades de cada *driver*, ya que independientemente del que se use, el manejo del dispositivo se hará siempre mediante las mismas funciones. Así un programa cliente que, por ejemplo, evite obstáculos haciendo uso de un dispositivo sonar y otro position, será válido para cualquier robot que incorpore estos dispositivos

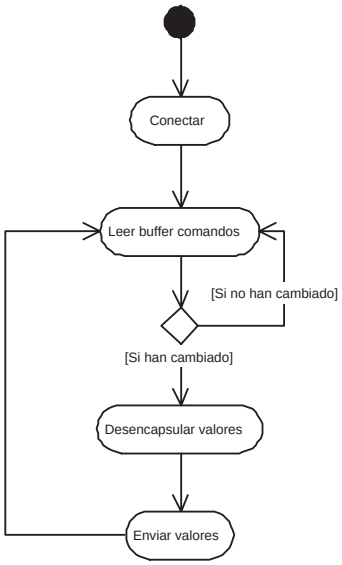


Figura 4.12: Diagrama de actividades de un dispositivo genérico actuador

independientemente del *driver* que emplee cada uno.

Existen unas cuantas *interfaces* definidas por *Player* (consultar apartado 2.2), las cuales se implementan mediante una estructura de clases (4.13), y a las que es conveniente adaptar nuestros *drivers*. En este caso ha sido posible adaptar a *interfaces* existentes todos los *drivers* genéricos excepto el de los relés. Para este dispositivo hemos tenido que crear una nueva *interfaz* ya que no había ninguna definida para este tipo de dispositivos. (figuras 4.8 y 4.13).

En cuanto a los otros, el *driver* genérico para el dispositivo sonar se ha adaptado a la *interfaz* Sonar, encapsulada en la clase '`SonarProxy`' (figura 4.13), el del dispositivo infrarrojos a la *interfaz IR*, encapsulada en la clase '`IRProxy`', el del dispositivo batería a la *interfaz Power*, encapsulada en la clase '`PowerProxy`' y el del dispositivo motor a la *interfaz Position*, encapsulada en la clase '`PositionProxy`'.

La *interfaz* de un dispositivo ofrece un conjunto de métodos, que pueden ser usados en un programa cliente, para realizar operaciones sobre el dispositivo. Es importante destacar que un *driver* no tiene por qué adaptarse íntegramente a la *interfaz*. Es decir, no todos los modelos de dispositivos asociado a la *interfaz*,

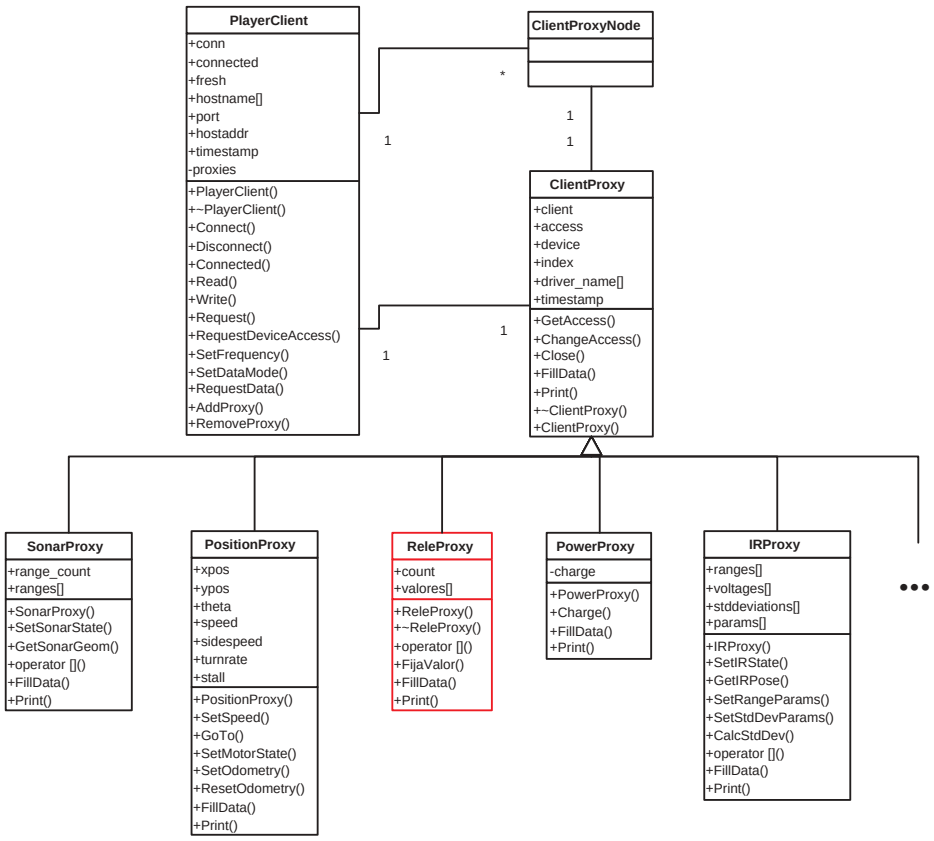


Figura 4.13: Diagrama de Clases de las *interfaces* asociadas a los dispositivos genéricos

podrán realizar todas las operaciones que esta ofrece, por tanto el *driver* particular de cada modelo sólo podrá adaptarse a las operaciones que este permita realizar. Por ejemplo, la *interfaz* `position` ofrece métodos para mover un motor hacia adelante y hacia atrás. Como pueden existir motores que sólo se muevan hacia adelante, estos no soportarán el método de movimiento hacia atrás.

La creación de una nueva *interfaz* implica crear una clase que herede de '`ClientProxy`' (figura 4.13). Esta es la clase base de todas aquellas que encapsulan una *interfaz* asociada a un dispositivo. Declara una serie de atributos y define un conjunto de métodos que todas ellas heredan. Entre los atributos que posee (figura 4.13) y que heredan sus clases derivadas cabe destacar:

- `client` Se trata de un puntero a un objeto de la clase '`PlayerClient`' (figura 4.13), que sirve para indicar a qué cliente de *Player* pertenece la *interfaz*. La clase '`PlayerClient`' se usa para controlar cada conexión con el servidor *Player*. Contiene una lista (`proxies` figura 4.13) para almacenar los objetos de las *interfaces* que tiene creados cada cliente. Implementa una serie de métodos que permiten la conexión y desconexión con *Player*, establecer la frecuencia del envio de datos, fijar el modo en el que Player enviará los datos, ...

- `access` Almacena el modo de acceso al dispositivo. Existen tres modos de acceso lectura ('r'), escritura ('w') y total ('a').

- `device` Almacena el nombre por el que se identifica este tipo de dispositivo.

- `index` Contiene un valor numérico que identifica cada dispositivo particular de ese tipo.

- `driver_name` Almacena el nombre del *driver* asociado al dispositivo que se está usando en ese momento. Es posible que el cliente acceda a varios dispositivos del mismo tipo a través de distintos *drivers*.

- `timestamp` Contiene el tiempo en el que el dato ha sido generado por el dispositivo.

En cuanto a los métodos que posee (figura 4.13) y que también heredan sus clases derivadas se encuentran:

- `GetAccess` Se trata de un método que proporciona el modo actual de acceso que tiene el dispositivo.

- `ChangeAccess` Método que permite cambiar el modo de acceso al dispositivo.

- `Close` Este método permite impedir el acceso al dispositivo.

- `FillData` Se trata de un método que debe ser redefinido en las clases derivadas y que es usado internamente para extraer y desencapsular de los buffers los nuevos valores recibidos.

- `Print` Este método también debe ser redefinido en las clases derivadas y sirve para mostrar en un formato comprensible el valor actual del dispositivo.

La nueva clase, creada como derivada de '`ClientProxy`', para la *interfaz* del dispositivo relé ha sido denominada '`ReleProxy`' (figuras 4.13 y 4.14) e incorpora los siguientes atributos y métodos propios:

- `operator[]` Se trata de la sobrecarga del operador '[]' con el fin de permitir consultar mediante un índice el valor de un determinado relé, siempre que haya más de uno.

- `FijaValor` Este método recibe dos argumentos que son la posición dentro del vector de relés, a la que queremos acceder y el valor que queremos fijar en ella. Logicamente sirve para dar valor a un relé (los posibles valores son 0 y 1).

Además de estos métodos la clase '`ReleProxy`', redefine los métodos '`FillData`' y '`Print`' heredados de '`ClientProxy`'.

Una vez implementada la clase es necesario definir las estructuras de datos y órdenes que serán almacenadas en los buffers de datos y órdenes respectivamente. También es necesario definir un nombre y un código para la nueva *interfaz*. Todo esto se lleva a cabo en el fichero de cabecera '`Player.h`'.

Finalmente es necesario registrar la nueva *interfaz*, para que *Player* conozca de su existencia. Esto se realiza añadiéndola a un vector de *interfaces* que posee *Player*, denominado `'interfaces'`, el código de esta, su nombre y el *driver* por defecto que debe usar. El vector de *interfaces* se encuentra en el fichero '`DeviceRegistry.cc`'.

Al igual que con los *drivers*, la compilación de la nuevas clases creadas y las modificadas se debe realizar modificando los *makefiles* (`Makefile.in`), puesto que *Player* hace uso de las herramientas *Automake* [Foub] y *Autoconf* [Foua].

```
class ReleProxy : public ClientProxy
{

public:
   /** Constructor.*/
   ReleProxy (PlayerClient* pc, unsigned short index,
              unsigned char access = 'c')
         : ClientProxy(pc,PLAYER_RELE_CODE,index,access)
     {count=2;}

   ~ReleProxy(){}

   /// Numero de relÃ©s.
   char count;

   /// Valores de los relÃ©s (en el caso de sabina 0 o 1).
   char valores[PLAYER_RELE_MAX_SAMPLES];

   // Sobrecarga del operador [] para leer directamente el
   // valor de los relÃ©s.
   unsigned short operator [](unsigned int index)
   {
     if(index < count)
        return(valores[index]);
     else
        return(0);
   }

  // FunciÃ³n para activar o desactivar un relÃ©.
  int FijaValor(int pos, int valor);

  // interface that all proxies must provide
  void FillData (player_msghdr_t hdr, const char* buffer);

  /// Print out the current digital input state.
  void Print ();
};
```

Figura 4.14: Declaración de la clase '`ReleProxy`'

4.2.3. Creación del Servidor para el robot *Sabina*

Una vez integrado en *Player* los *drivers* genéricos de los dispositivos y creada la nueva *interfaz* del dispositivo relé, hay que implementar el servidor que se ejecutará en el módulo controlador (Dk-40) de *Sabina*. El diseño y la implementación han sido realizadas utilizando técnicas de orientación a objetos, para que pueda ser facilmente adaptado a otros robots con sólo modificar las clases de acceso a los dispositivos. El lenguaje empleado ha sido C++.

Antes de comenzar con el diseño se han considerado las restricciones que vienen impuestas por el módulo controlador donde se va a ejecutar el servidor.

En nuestro caso hemos tenido que usar la API (*Aplication Programming Interface*) (RTOS [Rto]) que ofrece el módulo DK-40, para la creación y manejo de procesos, conexiones serie y TCP/IP. El resto del desarrollo se ha realizado utilizando las bibliotecas estandar de ANSI C++.

La figura 4.15 representa las clases que hemos usado para la implementación, así como los métodos más importantes de cada una de ellas. En ella puede observarse que faltan algunas clases ('`TramaActuadores`' y '`BufferTramaActuadores`') de las propuestas en el apartado 3.2, el motivo se explica a continuación, pero me gustaría destacar que la estructura del servidor que proponemos en el capítulo anterior es orientativa y en ocasiones no va a poder, o no va a interesar implementarla tal cual.

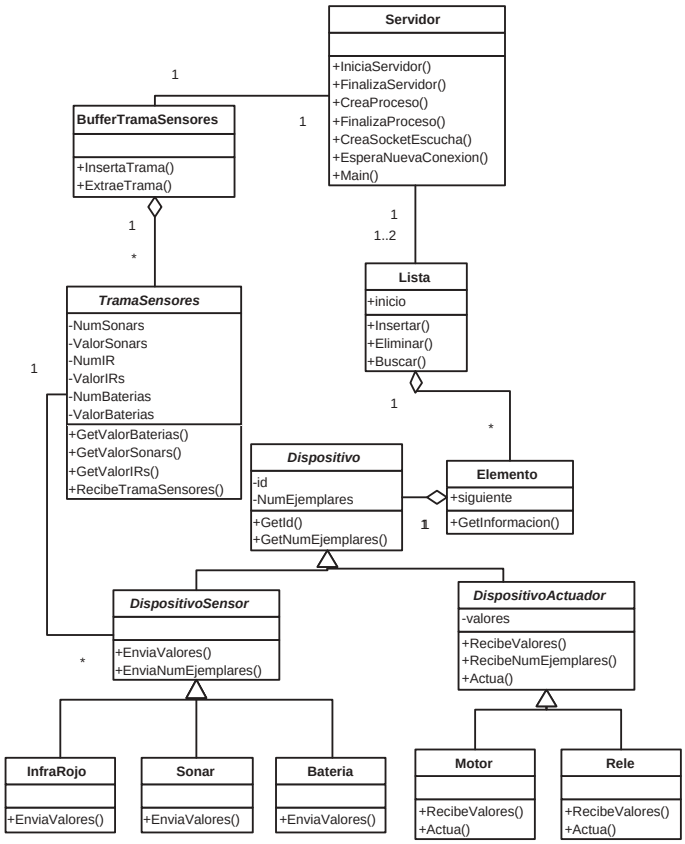


Figura 4.15: Diagrama de Clases del servidor de *Sabina*

Empezaremos explicando la clase '`Lista`', que como su propio nombre in-

dica representa una lista enlazada dinámica formada por elementos, los cuales contienen como información cualquier dispositivo que herede de las clases abstractas '`DispositivoSensor`' o '`DispositivoActuador`'. El servidor va a usar dos objetos de tipo lista, en una se insertarán los dispositivos sensores que posea el robot y en la otra los actuadores.

La clase '`Dispositivo`' implementa las funciones comunes a todos los dispositivos, como son obtener el identificador de un dispositivo o el número de ejemplares del mismo. Sin embargo las clases '`DispositivoSensor`' y '`DispositivoActuador`' poseen funciones propias de cada tipo de dispositivo e implementan alguna común a todos los ejemplares de su tipo como el envío o la recepción del número de ejemplares. Estas dos clases dejan sin implementar los métodos que dependen de cada dispositivo en particular, como el envio y la recepción de los valores. Así pués las clases asociadas a cada dispositivo particular sólo deben implementar una (sensores) o dos (actuadores) funciones.

Las clases que heredan de '`DispositivoSensor`' (figura 4.15), encapsulan sólo las operaciones relativas a la comunicación con los *drivers* genéricos de *Player* y no acceden a los dispositivos físicos para obtener sus valores. El motivo por el que se ha realizado de esta manera es para reducir el número de accesos a los dispositivos y evitar problemas de concurrencia, puesto que la clase '`TramaSensores`' almacena el valor de todos los dispositivos sensores del robot y es ella la que solicita periodicamente al PIC de los sensores el valor de todos ellos, depositándolos en el buffer de tramas de sensores. De esta manera todos los procesos que atienden peticiones sobre dispositivos sensores sólo tienen que acceder al buffer de tramas de sensores para obtener los valores que necesitan y enviarlos rápidamente al cliente.

En cuanto a las clases que heredan de '`DispositivoActuador`' (figura 4.15), encapsulan las operaciones relativas a la comunicación con los *drivers* genéricos de *Player* y además implementan una función que actúa sobre el dispositivo físico asociado con los valores que reciben del *driver* correspondiente. De esta forma no es necesario hacer uso de un buffer que almacene los valores que deben dársele a cada dispositivo actuador. El motivo por el que se ha establecido esta distinción entre los dispositivos sensores y los actuadores, es que según las car-

acterísticas de *Sabina* nos interesa que se ejecute la última orden recibida y no pasa nada si se deja de ejecutar una intermedia. Por esta razón hemos prescindido del uso de buffers para guardar las órdenes de los dispositivos actuadores, y por tanto de las clases '`TramaActuadores`' y '`BufferTramaActuadores`'.

No obstante, a pesar de que el acceso a los dispositivos se encuentre repartido entre las clases '`TramaSensores`', '`Motor`' y '`Rele`', sólo un proceso será el que ejecute esas funciones de acceso. De esta forma evitamos problemas de acceso concurrente y hacemos que el acceso a los dispositivos sea lo más rápido posible.

En cuanto a los procesos usamos un `proceso principal`, un `proceso informador/actuador` y creamos un `auxiliar` por cada conexión recibida de un cliente. Cabe destacar que hemos dado la misma prioridad a todos los procesos, puesto que nos interesa que todos los dispositivos sean atendidos por igual, aunque es posible cambiarla. El algoritmo de asignación de CPU que seguimos es el *Round Robin*.

Para comprobar el correcto funcionamiento de la arquitectura hemos creado una serie de programas clientes que a través de *Player* y usando conexiones TCP/IP inalámbricas, nos permitan acceder y controlar al robot *Sabina*. Concretamente se han hecho un programa de esquivación de obstáculos (tanto estáticos como dinámicos) y otro de rastreo de una superficie. Los resultados han sido plenamente satisfactorios en ambos casos.

4.3. Robot *Castaño*

Castaño es un robot móvil, creado originalmente para recorrer laberintos sobre superficies planas que no contengan obstáculos en el suelo. Fué creado para participar en la prueba de laberinto de *Alcabot* 2004 [Alc].

4.3.1. Estructura Física

Castaño posee una base rectangular hecha de metacrilato sobre la que se levanta una estructura formada por otras dos planchas de metacrilato. La dis-

tancia de la plancha más baja al suelo son 5'5 cm, mientras que la separación entre esta y la intermedia es de 4 cm y entre la intermedia y la más alta es de 3 cm. Sobre la plancha más baja se sustentan un móduo controlador (Dk-40), y sobre las otras dos el resto de dispositivos (ver figura 4.16).

Mide 18cm de largo, 11cm de ancho y 15cm de alto. Se apoya en el suelo a través de tres ruedas, dos de las cuales tienen asociado un servo y la tercera sirve de guía. Puede avanzar, retroceder y girar alrededor de un eje vertical, gracias a los dos servos dispuestos en configuración diferencial.

En su parte inferior delantera cuenta con dos bumper de contacto que le sirven como medida de seguridad por si llegase a colisionar.

Figura 4.16: Vista General del Robot *Castaño*

4.3.2. Dispositivos Electrónicos

El robot *Castaño* posee dispositivos sensores y actuadores, pero a diferencia de *Sabina*, todos los dispositivos se agrupan en un único módulo sensorial-actuador, ya que todos ellos se controlan a través de un único microcontrolador.

Este módulo sensorial-actuador se encarga tanto de detectar los elementos que rodean al robot como de moverse. Constituye el sistema de percepción-actuación del robot y está formado por los siguientes dispositivos:

- Conjunto de 4 sensores medidores de distancia por infrarrojos (Sharp

GP2D12), colocados en la plancha intermedia de metacrilato, en forma de rectángulo (cada uno apuntando al exterior de una cara). Su misión consiste en proporcionar información sobre el entorno del robot con el fin de poder detectar obstáculos a media altura o seguir paredes. Como características más importantes de este modelo de dispositivo, cabe destacar que indican mediante una salida analógica entre 0 y 3V la distancia medida. La tensión de salida varía de forma no lineal (ver figura 4.2(b)) cuando se detecta un objeto en una distancia entre 10 y 80 cm y su valor es actualizado cada 40 ms. Funciona con una tensión de alimentación próxima a los 5V.

- 2 bumpers (conmutador de dos posiciones) de contacto que le sirven como medida de seguridad por si se encuentra obstáculos bajos que no detecten los infrarrojos. Están colocados en la parte frontal de la plancha inferior.

Figura 4.17: Bumper

- 1 brújula digital (CMPS03) que le permite conocer en cada momento su orientación y facilita las tareas de navegación. Es un sensor de campos magnéticos que una vez calibrado ofrece una precision de 3-4 grados y una resolución de décimas. También se encuentra ubicado sobre la plancha inferior.

- 3 servos (MX400) trucados para rotación, dos de los cuales están asociados a las ruedas haciendo la función de motores con reductora y permitiendo el movimiento del robot. El tercero está asociado a un disparador, colocado sobre la plancha superior, para el lanzamiento de objetos. Funcionan con un voltaje de 4.8V. La conexión al PIC se realiza a través de un *driver* de potencia (L293).

Figura 4.18: Servo MX400

- 1 microcontrolador (PIC 16F876) a cuyos pines se conectan los bumper, los infrarrojos y la brújula y mediante *drivers* de potencia los servos. Posee un sencillo programa que se ha desarrollado para controlar los dispositivos. Este dispositivo posee una memoria flash de 8 KBytes. Tiene 28 pines y 3 puertos de E/S (2 de 8 bits y 1 de 6 bits con 5 lineas de entrada en convertidor A/D).

Figura 4.19: Microcontrolador PIC 16F876

El microcontrolador (PIC 16F876) que controla los dispositivos, se encuentra en una pequeña placa de prototipos con un cristal de cuarzo a 4Mhz y una serie de conectores que permiten la conexión de los pines a los distintos dispositivos (figura 4.21).

Además de los dispositivos descritos anteriormente, *Castaño* también cuenta con un módulo controlador de la casa BECK. Se trata de un Dk-40 (ver figura 4.20) con un chip IPC@CHIP SH12 [Ipc], cuyas características más importantes son:

- Procesador a 186Mhz.
- Memoria RAM (256 Mb).

- Flash Disk.
- Dos puertos serie.
- Una tarjeta de red.

A través de uno de los puertos serie se realiza la conexión con el microcontrolador (PIC 16F876) que controla los dispositivos.

En cuanto a las características software más relevantes del DK-40 destacamos:

- Servidor Web.
- Servidor FTP.
- Servidor Telnet.
- Permite conexiones por socket TCP y UDP (hasta 64).
- Posee una API (*Aplication Programming Interface*) que ofrece funciones para el control de los puertos serie, los socket y la creación de procesos (hasta 35).
- Soporta semáforos.
- Incorpora una *shell* de órdenes tipo DOS y permite ejecutar aplicaciones compiladas para MS-DOS.

Figura 4.20: Módulo controlador Dk-40

En este módulo controlador es donde se ejecuta el programa servidor que accede, a través del microcontrolador, a los distintos dispositivos a petición de *Player*.

La alimentación de todos los dispositivos y del módulo controlador se realiza a través de una batería de 12V y 1,6A formada por 8 pilas de 1.5V, que *Castaño* lleva incorporada. Puesto que no todos los dispositivos funcionan al mismo voltaje se emplean una fuente de 12V y otra de 5V. El voltaje de la batería también es controlado a través del microcontrolador, conectando uno de sus pines a un controlador de voltaje. La figura 4.21 muestra el esquema de conexiones entre los distintos dispositivos de *Castaño*.

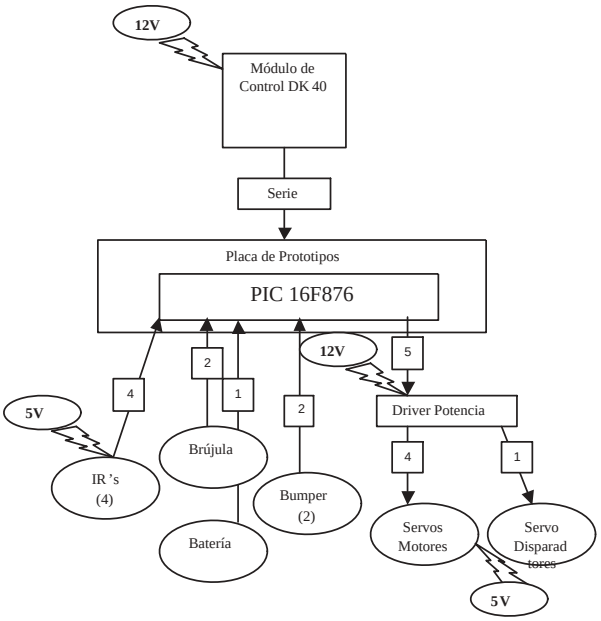


Figura 4.21: Diagrama de conexiones de los dispositivos de *Castaño*

4.4. Integración del Robot *Castaño*

Para la integración de *Castaño* en *Player* se han seguido los mismos pasos que en la integración de *Sabina*. En primer lugar se ha comprobado si existen *drivers* genéricos en *Player* para todos los dispositivos que posee *Castaño*. Si fuese así no haría falta crear nuevos *drivers* genéricos, pero como en este caso posee tres dispositivos nuevos (bumper, brújula y servo) ha sido necesario crear los *drivers* genéricos correspondientes.

No hay que olvidar que cada *driver* debe estar asociado a una *interfaz*, por tanto siempre hay que comprobar si los nuevos *drivers* se pueden adaptar a una existente (consultar apartado 2.2), o si es necesario crearla nueva. En este caso el *driver* del dispositivo servo se ha adaptado a la *interfaz* `Position`, el del *bumper* a la *interfaz* `Bumper` y para el del dispositivo brújula ha sido necesario crear una nueva *interfaz*.

Una vez realizadas las modificaciones oportunas sobre *Player*, el siguiente paso ha sido la creación de la aplicación servidora basada en el modelo descrito en la sección 3.2.

La integración de *Castaño* ha resultado bastante más rápida y sencilla que la de *Sabina*, puesto que ya se disponía de los *drivers* genéricos para algúnos dispositivos de *Castaño* (infrarrojos y batería) y de un modelo de aplicación servidora que poder adaptar, teniendo en cuenta que la capa encargada de la gestión de procesos y peticiones de los clientes (figura 3.2) debe ser bastante parecida en todas las aplicaciones servidoras.

Lógicamente, cuantos más *drivers* genéricos haya integrados en *Player* menos modificaciones habrá que realizar sobre él y más sencilla será la integración de nuevos robots.

4.4.1. Creación de los *drivers* genéricos para los dispositivos de *Castaño*

En el caso de *Castaño* ha sido necesario crear tres nuevos *drivers* genéricos, uno para el dispositivo *bumper*, otro para la brújula y otro para el servo. Como se ha explicado en la sección 4.2.1, es necesario crear una nueva clase, que herede de '`CDevice`' (sección 2.5.1) por cada *driver* genérico. La clase '`CDevice`' es la clase base de todos los *drivers* de los dispositivos. Esta clase declara los métodos que deben ser implementados para la lectura y almacenamiento de información en los buffers de datos y órdenes, así como los que se ejecutan al crearse y finalizarse un objeto de la clase. Todos ellos son heredados y algunos deben ser redefinidos pues se trata de una clase abstracta.

La creación de las clases asociadas a los *drivers* genéricos de estos dispositivos se ha realizado utilizando la estructura de clases ya descrita y explicada en la sección 4.2.1. Así pues se han creado tres nuevas clases (`BumperGenérico`, `ServoGenerico` y `BrujulaGenerico`) que heredan tanto de la clase '`DispositivoGenerico`' como de '`CDevice`' (figura 4.22). Recordar que la clase '`DispositivoGenerico`' implementa los métodos comunes a cualquier dispositivo sensor y actuador, por tanto deben ser heredados por todos y de este modo se evita implementarlos.

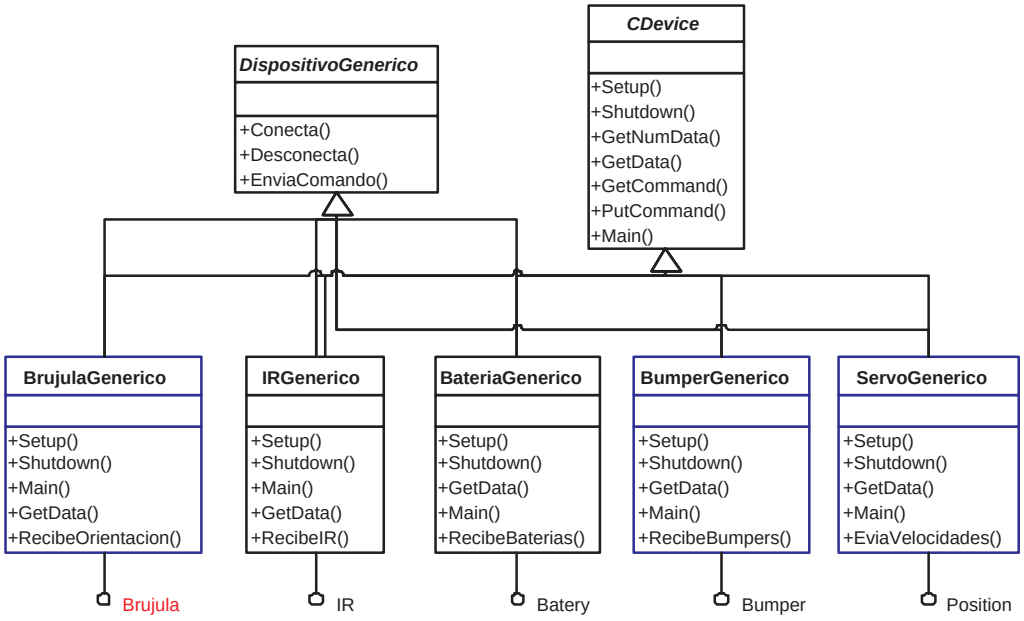


Figura 4.22: Diagrama de Clases de los *drivers* genéricos de *Castaño*

Cada una de estas tres nuevas clases, debe implementar los métodos virtuales puros '`Setup`', '`Shutdown`' y '`Main`', heredados de la clase '`CDevice`'. Por ejemplo, la implementación del método '`Setup`' en la clase '`BumperGenerico`' consistirá en establecer la conexión con el servidor y crear un hilo sobre el que se ejecutará la función `Main`. En cuanto a esta función, su comportamiento es parecido en todos los dispositivos sensores y en todos los actuadores (figuras 4.11 y 4.12).

Las nuevas clases tienen la posibilidad de redefinir otros métodos heredados de '`CDevice`'. Así por ejemplo, la clase '`BumperGenerico`' redefine la función '`GetData`'. Este método debe proporcionar a *Player* el último valor leido del

```
int BumperGenerico::Setup()
{
   if(conecta(PUERTO_SABINA,SABINA)!=0)
   {
      cout << "Error al establecer la conexiÃ³n desde el driver de " <<
              "bumper_sabina con el servidor de Sabina." << endl;
      return(-1);
   }
   /* Despues de inicializar todo arrancamos el trhead. */
   StartThread();
   return(0);
}
```

Figura 4.23: Implementación de la función virtual Setup en la clase BumperGenerico

dispositivo y es invocado cada vez que un cliente solicita el valor de ese dispositivo. Su implementación consiste (figura 4.24) en bloquear un *mutex* para acceder al buffer de datos de ese dispositivo, copiar el último dato en un puntero que se recibe como argumento, devolver, también por referencia, los segundos y milisegundos del dato y desbloquear el *mutex*.

```
size_t BumperGenerico::GetData(void* client, unsigned char* dest,size_t len,
                  uint32_t* timestamp_sec, uint32_t* timestamp_usec)
{
  Lock();

  memcpy(dest, (uint8_t*)device_data, sizeof(player_bumper_data_t));
  if(timestamp_sec)
    *timestamp_sec = data_timestamp_sec;
  if(timestamp_usec)
    *timestamp_usec = data_timestamp_usec;

  Unlock();
  return(sizeof(player_bumper_data_t));
}
```

Figura 4.24: Redefinición del método GetData en la clase BumperGenerico

Una vez implementadas las funciones virtuales puras heredadas y redefinidas aquellas que se considere oportunas, es necesario implementar dos funciones adicionales que no van a pertenecer a la clase del *driver*, pero que son exclusivas de cada uno. Se trata de las funciones '`NombreDriver_Init`' que crea y devuelve una nueva instancia del *driver* y '`NombreDriver_Register`' que registra el *driver* en la tabla de *drivers* disponibles que posee cada instancia de *Player*. Un ejemplo

de la implementación de estas funciones puede verse en las figuras 4.9 y 4.10.

Hay una función que debe definirse en la nueva clase dependiendo de si el dispositivo es sensor o actuador. En el caso de los dispostivos actuadores es la que se encarga de enviar los valores al servidor, mientras que en los dispositivos sensores es la que se encarga de recibirlos (figura 4.22). La implementación de esta función debe ser propia de cada dispositivo ya que el tipo y número de los valores enviados o recibidos variará de unos dispositivos a otros. En el caso de la clase '`BumperGenerico`', como es un dispositivo sensor esta se denomina '`RecibeBumpers`' (figura 4.25) y se encarga de recibir el valor de los *bumpers* que envia el servidor.

```
int BumperGenerico::RecibeBumpers()
{
  char *valores;
  int i,tam_dir;

   tam_dir=sizeof(struct sockaddr_in);
   valores=new char[tr.num_bumper];

   // recibo los valores
   if(recvfrom(num_s,valores,sizeof(char)*tr.num_bumper,0,
            (struct sockaddr*)&dir_serv,(socklen_t*) &tam_dir) == -1)
      return (-1);
   else // y los paso a char.
   {
      for(i=0; i<tr.num_bumper; i++)
         tr.valores[i]=(0xff)&(valores[i]);

      delete [] valores;
   }
   return (0);
}
```

Figura 4.25: Implementación de la función RecibeValores de la clase Bumper-Generico

Finalmente y para que *Player* conozca la existencia de nuestros nuevos *drivers*, es necesario incluir en la función `register_devices` del fichero '`deviceregistry.cc`' la llamada a la función de registro ('`NombreDriver_Register`') de nuestro *driver*. De esta forma cada vez que se crea una instancia de *Player* se invoca a la función `register_devices` y se cargan en un objeto de tipo '`DriverTable`' todos los *drivers* disponibles.

Una vez creadas las nuevas clases y modificadas algunas de las existentes, hay que crear una biblioteca estática que contenga el código de los *drivers* creados para los nuevos dispositivos. Para ello se deben modificar los *makefiles* (`Makefile.in`) y ficheros de configuración (`configure.in`), ya que *Player* hace uso de las herramientas *Automake* [Foub] y *Autoconf* [Foua] para la generación y compilación del código, de forma que se cree la nueva biblioteca y se enlace con el ejecutable de *Player*.

4.4.2. Creación de una *interfaz* para el dispositivo Brújula

Igual que ocurrió con el dispositivo Relé en la integración de *Sabina*, al integrar *Castaño* nos vemos obligados a crear una nueva *interfaz* para el dispositivo Brújula, ya que no existe ninguna (sección 2.2) que se adapte a este dispositivo.

En cuanto al resto de *drivers* genéricos que se han creado, ha sido posible adaptarlos a *interfaces* existentes (figura 4.26). Así el *driver* genérico para el dispositivo *bumper* se ha adaptado a la *interfaz Bumper*, encapsulada en la clase '`BumperProxy`'. El del dispositivo servo se ha adaptado a la *interfaz Position*, encapsulada enla clase `PositionProxy`.

Como ya se explicó en la sección 4.2.2, la creación de una nueva *interfaz* implica crear una clase que, haciendo uso de la herencia se adapte a una estructura de clases ya existente (figura 4.26).

La nueva clase debe heredar los métodos y atributos de la clase base '`ClientProxy`' (descritos en la sección 4.26). Estos le proporcionan la funcionalidad para obtener valores de los buffers de *Player*, gestionar el modo de acceso al dispositivo, asociar un determinado *driver*, etc. Existen dos métodos de esta clase que deben ser redefinidos. Son '`FillData`', que es usado internamente para extraer y desencapsular de los buffers los nuevos valores recibidos, y '`Print`', que debe mostrar en un formato comprensible el valor actual del dispositivo.

La clase '`BrujulaProxy`', que es como la hemos denominado, encapsula la *interfaz* del dispositivo brújula (figura 4.26) y añade los siguientes atributos y métodos propios, a los heredados de '`ClientProxy`':

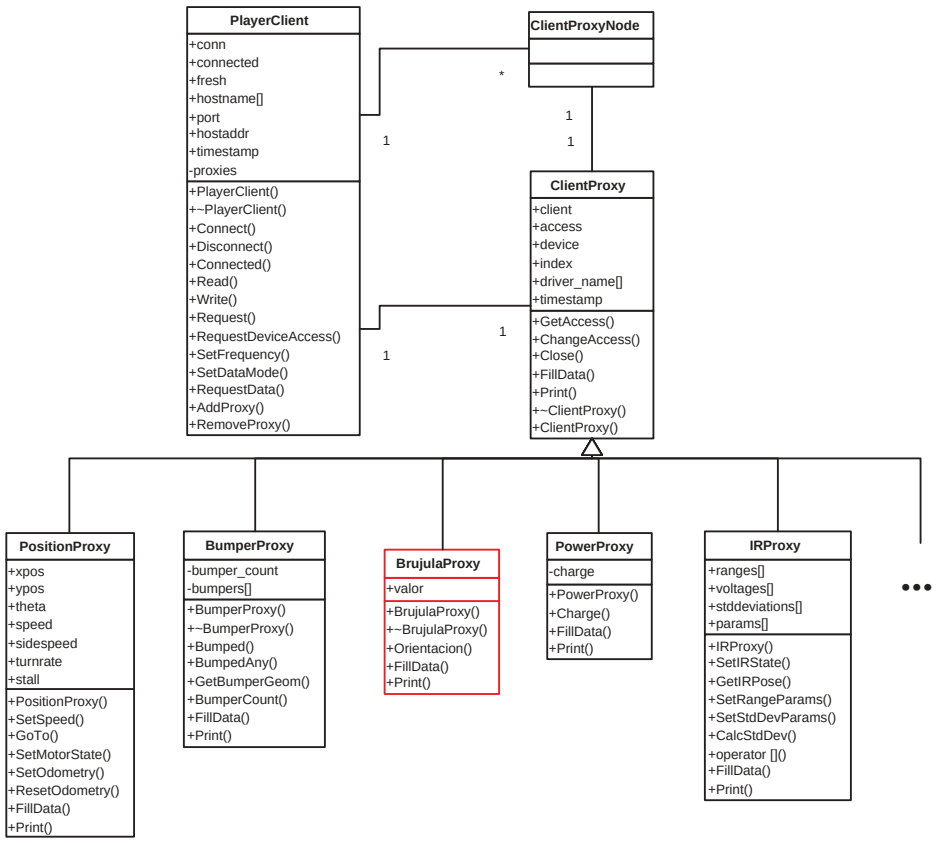


Figura 4.26: Diagrama de clases de las *interfaces*

- `valor` Atributo donde se almacena el valor (orientación) que proporciona el dispositivo.

- `BrujulaProxy` Se trata del constructor de la clase. Se encarga de iniciar el atributo '`valor`' y de invocar al constructor de la clase base '`ClientProxy`', pasándole como argumentos el código e índice de la *interfaz*, el modo de acceso inicial y un puntero a un objeto de tipo '`PlayerClient`' (cliente que ha creado el objeto de nuestra *interfaz*). Este constructor se encarga de iniciar los atributos heredados, añadir a la lista de *interfaces* del cliente, el objeto de nuestra *interfaz* y de solicitar el acceso al dispositivo en el modo demandado y haciendo uso del *driver* que el dispositivo tenga asociado.

- `Orientación` Método que devuelve el último valor (orientación) proporcionado por el dispositivo.

Además de estos métodos la clase '`BrujulaProxy`', redefine los métodos '`FillData`' y '`Print`' heredados de '`ClientProxy`'.

Una vez implementada la clase es necesario definir las estructuras de datos y órdenes que serán almacenadas en los buffers de datos y órdenes respectivamente. En este caso como el dispositivo brújula no admite órdenes sólo hay que definir la estructura de datos (figura **??**. También es necesario definir un nombre y un código para la nueva *interfaz* (figura **??**. Todo esto se lleva a cabo en el fichero de cabecera '`Player.h`'.

```
#define PLAYER_BRUJULA_CODE       ((uint16_t)27)  // Dispositivo brújula
#define PLAYER_BRUJULA_STRING      "brujula"

/** [Datos] */
typedef struct player_brujula_data
{
  uint8_t grados;
  uint8_t direccion;
} __attribute__ ((packed)) player_brujula_data_t;
```

Figura 4.27: Definición del nombre, código y estructura de datos de la *interfaz* Brújula

Finalmente es necesario registrar la nueva *interfaz*, del modo explicado en la sección 4.2.2.

La compilación de la clase creada y de las modificadas se debe realizar reflejándolo en los *makefiles* (`Makefile.in`), puesto que *Player* hace uso de las herramientas *Automake* [Foub] y *Autoconf* [Foua].

4.4.3. Creación del Servidor para el robot *Castaño*

Una vez integrados en Player los *drivers* genéricos de los dispositivos que posee *Castaño* y creada la *interfaz* del dispositivo brújula, es necesario implementar la aplicación servidora que se ejecutará en el módulo controlador de *Castaño*, para dar servicio a los *drivers* genéricos.

La creación del programa servidor se ha realizado tomando como base el creado para *Sabina* (sección 4.2.3), ya que ambos robots comparten el mismo

modelo de módulo controlador (Dk-40) que suele ser el elemento que más restricciones establece a la hora de crear el servidor. Recordar que este módulo permite la creación y manejo de procesos, conexiones serie y TCP/IP a través de una API (*Aplication Programming Interface*) denominada RTOS [Rto].

La figura 4.28 muestra la estructura de clases empleada para la implementación del servidor. Esta es prácticamente igual que la de *Sabina*, con la diferencia de que ha sido necesario crear tres nuevas clases ('`Brujula`', '`Bumper`' y '`Servo`') para dar servicio a los *drivers* genéricos asociados a estos dispositivos. También ha sido necesario modificar la clase 'TramaSensores' ya que es la encargada de leer y almacenar el valor de todos los dispositivos sensores, y los que posee *Castaño* son distintos de los de *Sabina*. El resto de las clases del diagrama se mantienen igual, su descripción y estructura puede ser consultada en la sección 4.2.3.

Las clases que heredan de '`DispositivoSensor`' ('`Brujula`', '`InfraRojo`', '`Bumper`' y '`Bateria`' figura 4.28), tienen como única tarea la comunicación con el *driver* genérico asociado a su dispositivo. Cada una de ellas debe declarar y definir una función que envíe los valores leidos de su dispositivo al *driver* genércico. Para evitar tener que acceder desde cada una de estas funciones al microcontrolador (PIC 16F876) que gestiona el acceso a los dispositivos físicos y evitar así problemas de concurrencia, se hace uso de la clase '`TramaSensores`' que almacena los valores leidos de todos los dispositivos sensores. De esta forma la función que envia los valores a su *driver* genérico, cuando recibe una petición, sólo tiene que solicitar a esta clase los valores de su dispositivo y enviarlos.

En el caso de las clases que heredan de '`DispositivoActuador`' ('`Servo`' figura 4.28) es distinto, ya que su tarea consiste en recibir las órdenes que deben realizarse sobre el dispositivo físico y ejecutarlas. Es decir, en este caso si que se accede desde una función ('`Actua`') de estas clases, al microcontralador (PIC 16F876) para actuar sobre los dispositivos físicos. El motivo por el que se ha realizado de este modo es evitar tener que hacer uso de un buffer que almacene los valores que deben dársele a cada dispositivo actuador, ya que el uso de *Castaño* pemite que deje de ejecutarse una orden intermedia.

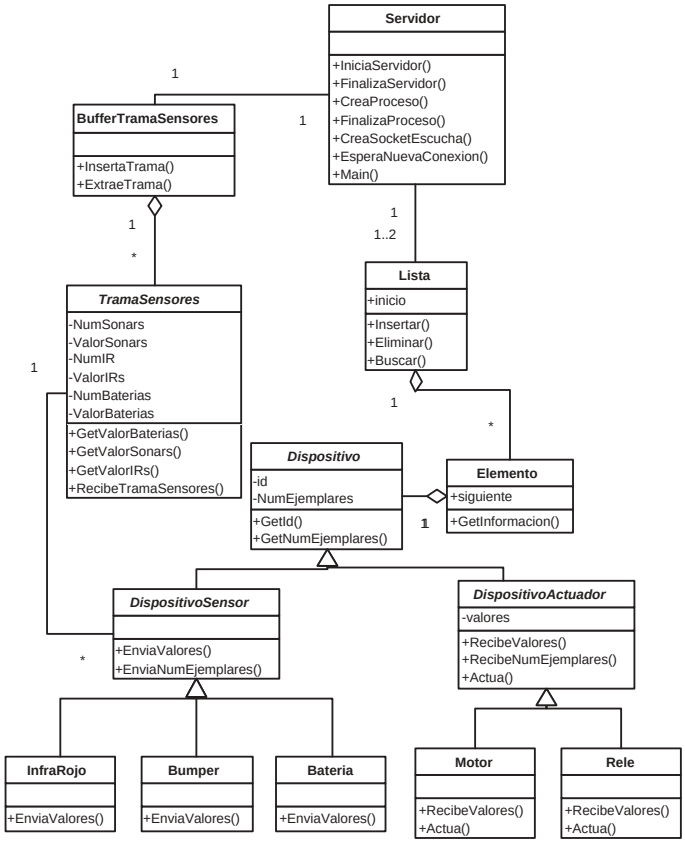


Figura 4.28: Diagrama de Clases del Servidor del Robot *Castaño*

Con respecto a los procesos, se sigue usando la misma política que en el servidor del robot *Sabina*. Un único `proceso informador/actuador` que ejecuta las funciones de acceso al microcontrolador (PIC 16F876) que controla los dispositivos físicos. Un `proceso principal` que se encarga de crear los `procesos auxiliares` y atender las conexiones con el servidor. Un conjunto de `procesos auxiliares` (uno por conexión) que atienden las peticiones de los *drivers* genéricos de *Player*.

Para comprobar el correcto funcionamiento de la arquitectura hemos creado una serie de programas clientes que a través de *Player* y usando redes inalámbricas, nos permitan acceder y controlar el robot. Concretamente se ha hecho un programa de seguimiento de paredes y otro de navegación a través de un laberinto. Los resultados obtenidos con ambos programas nos han permitido

comprobar el correcto funcionamiento de la arquitectura, así como las posibilidades que ofrece esta forma de trabajar en el ámbito de la colaboración robótica. Mantenemos la misma prioridad para todos los procesos y seguimos usando el algoritmo de asignación de CPU *Round Robin*.

5

Resultados y conclusiones

El objetivo de este trabajo de Grado ha sido integrar en un servidor multi-robot, para ser controlados a través de él (mediante conexiones serie o TCP/IP), varios robots. De manera que se establezca un entorno de trabajo en el que poder desarrollar sistemas multi-robot o investigar y trabajar más fácilmente, en áreas como la colaboración robótica. Haciendo uso de un servidor multi-robot es posible crear programas clientes que, a través de él, controlen varios robots o dispositivos simultaneamente.

Para conseguir este objetivo ha sido necesario realizar un estudio de los servidores de dispositivos robóticos más difundidos (sección 1.3), con el fin de comprobar si alguno de ellos se adecuaba a nuestras necesidades, o si por el contrario sería necesario crear un servidor nuevo.

El servidor de dispositivos robóticos *Player* resultó ser el mas adecuado, por lo que se ha realizado un estudio en profundidad de él (sección 2). Tras este estudio se advirtieron algunos aspectos un tanto complejos que presentaba, como la integración de nuevos dispositivos o el control de estos a través de conexiones TCP/IP. Por este motivo, y con el fin de adaptar el servidor a nuestras necesidades, se ha propuesto una arquitectura que facilita la integración, para ser controlados a través de conexiones TCP/IP, de todo tipo de robots (dispositivos) en *Player*. Esta arquitectura evita tener que realizar modificaciones sobre *Player* y facilita por tanto la integración de nuevos dispositivos en él, de modo que agiliza mucho su integración en el entorno que buscamos.

La arquitectura propuesta ha sido probada sobre dos robots móviles desarrollados íntegramente en el Departamento de Informática y Automática de la Universidad de Salamanca (sección 4).

A lo largo del tiempo que ha durado la realización de este trabajo de Grado se han extraido una serie de conclusiones que pasamos a detallar:

- Actualmente existen muy pocos servidores de dispositivos robóticos gratuitos o de código abierto, que sean adaptables a todo tipo de dispositivos. La mayoría de ellos pertenecen a empresas privadas y se centran en la integración de dispositivos de robots comerciales, o simplemente son simuladores robóticos.

- El servidor de dispositivos robóticos *Player* es gratuito, de código abierto y permite la integración, para ser controlados a través de él, de todo tipo de dispositivos. Facilita la creación de programas clientes válidos para un gran número de robots, puesto que constituye una capa de abstracción que permite trabajar sobre los dispositivos sin necesidad de conocer cómo funcionan a bajo nivel. Esto es algo realmente interesante ya que ofrece la posibilidad de que un programa cliente sea válido para distintos robots (con distinta arquitectura), o que se puedan crear programas clientes que controlen varios robots al mismo tiempo haciendo que estos colaboren entre si.

- A la hora de trabajar con sistemas multi-robot haciendo uso de algún

servidor, es bastante común que las comunicaciones entre el servidor y los robots se realicen a través de conexiones TCP/IP (redes inalámbricas) ya que aumentan el radio de actuación y permiten el control de múltiples robots por medio de un único punto de acceso a la red, lo que facilita el trabajo en el área de la colaboración robótica.

- Hemos propuesto y desarrollado una arquitectura para la integración en *Player* de robots que quieran ser controlados a través de él, por medio de conexiones TCP/IP. Esta arquitectura facilita la integración de los robots, ya que evita tener que modificar el código de *Player*, es sencilla de adaptar, agiliza la integración en un entorno común de un conjunto de robots y funciona correctamente pues ha sido probada sobre distintos robots obteniendo resultados satisfactorios.

- El uso de la arquitectura propuesta, junto con *Player* puede ser una buena plataforma sobre la que realizar trabajos e investigaciones en áreas novedosas como la colaboración robótica.

En lo referente a la aplicación de la arquitectura a robots reales, las pruebas realizadas nos permiten afirmar:

- La diferencia entre tiempos de reacción si se accede directamente al dispositivo a través del driver de *Player* o si se accede a través del servidor ubicado en el robot es del orden de milisegundos, aunque depende en gran medida de lo eficiente que sea la implementación del servidor.

- Es facilmente adaptable a varios tipos de robots pués gracias a su diseño sólo implica modificar la capa más baja (acceso a los dispositivos).

Bibliografía

[ACT] Activmedia robots. http://www.activrobots.com/.

[AIB] Aibo. http://www.aibo-europe.com/.

[Alc] Alcabot. concurso de microrrobots de la universidad de alcalá. http://www.depeca.uah.es/alcabot/alcabot2004/.

[Auta] Gnu autoconf. http://www.gnu.org/software/autoconf/.

[Autb] Gnu automake. http://www.gnu.org/software/automake/.

[Bar] Michael Barr. Introduction to pulse width modulation. http://www.netrino.com/Publications/Glossary/PWM.html.

[Bat95] Dr. D. Anibal Ollero Baturone. Planificación de trayectorias para robots móviles. Informe técnico, Universidad de Málaga, Julio 1995.

[BPGH02] Richard T. Vaughan Brian P. Gerkey y Andrew Howard. *Player. User Manual*. 2002.

[DC02] Salvador Dominguez y Eduardo Zalama Casanova. A mobile robot with enhanced gestual abilities. En *9th Mechatronics Congress*. 2002.

[FLE] Flexitrack. http://www.flexitrack.com.

[HIS] Hispabot. http://www.hispabot.org/.

[Ipc] Ipc@chip. http://www.beck-ipc.com/ipc/index.asp?sp=en.

[iRo] irobot. http://www.irobot.com/.

[JBF96] H. R. Everett J. Borenstein y L. Feng. *Where am I?. Systems and Methods for Mobile Robot Positioning*. J. Borenstein, 1996.

[Kte] K-tam robots. http://www.k-team.com/.

[Mar] Estados unidos regresa a marte. http://ciencia.nasa.gov/headlines/y2001/ast24oct_1.htm.

[MMT] Nicholas Roy Michael Montemerlo y Sebastian Thurn. Carmen. carnegie mellon robot navigation toolkit. http://www-2.cs.cmu.edu/ carmen/.

[Nom] Nomadic technologies. http://nomadic.sourceforge.net/.

[Pat] Mars pathfinder. http://spacelink.nasa.gov/NASA.Projects/Space.Science/Solar.System/Mars.Pathfi

[ROB] Robocup. http://www.robocup.org/.

[Rto] Ipc@chip documentation index - sc12 @chip-rtos v1.10. http://www.beck-ipc.com/files/ipc/documentation/api/SC12APIDOC0110.PDF.

[SDR] Sdr-4x. http://www.tokyodv.com/news/SonySDR-4XRobot.html.

[Sic] Sick. sensor intelligence. http://www.sick.com/us/products/en.html.

[Son] Sony evi-d30. http://picturephone.com/products/sony_evi_d30.htm.

[Wer] Barry Werger. Ayllu. distributed behavior-based control. http://robots.activmedia.com/ayllu/aylluman1_5a.pdf.