Sistema de comunicaciones de la plataforma móvil Pioneer 2-AT8

Iván Pareja Larios

20-03-2004

A mi abuela Emérita
A mis padres José Luís y Visitación y a mi hermano Jaime

Índice

Parte I

Introducción

Capítulo 1

Agradecimientos

Finalizar el proyecto de fin de carrera supone la culminación de lo que quizás sea la etapa más bella de mi vida, el periodo como estudiante. Agradezco enormemente a mi familia que hayan hecho que todo ello sea posible y que gracias a su sacrificio haya podido llegar a ser Ingeniero Industrial. Se lo agradezco especialmente a mi abuela Emérita, a la que tanto habría gustado poder presenciar estos momentos.

Agradezco a mi tutor Ricardo Sanz el apoyo prestado y la libertad que me ha dado a la hora de realizar el proyecto. Así mismo agradezco la ayuda de mis compañeros de ASLab y de Carlos Martínez. Sin ellos las presentes líneas no habrían sido posibles.

Capítulo 2

Introducción

El presente proyecto de fin de carrera se ha realizado en la División de Ingeniería de Sistemas y Automática (DISAM) de la Escuela Técnica Superior de Ingenieros Industriales. Las líneas de investigación en UPM-DISAM están dirigidas hacia el estudio y desarrollo de nuevas tecnologías en el campo del control de procesos, inteligencia artificial, robótica y visión por computador.

El proyecto tiene como objetivo la comunicación del robot móvil Pioneer 2-AT8 con la red local de computadores del departamento. El proyecto es tan sólo una pequeña parte de un proyecto padre llamado Autonomous Modular Systems (AMS), desarrollado por un grupo interdepartamental llamado ASLab (Autonomous System Laboratory).

ASLab son un conjunto de personas (alumnos, proyectantes, doctorados y profesores) que comparten intereses por la robótica, la ingeniería de sistemas de control y por el desarrollo tecnológico en general. Estas personas trabajamos juntas con el objetivo de continuar formándonos, de aprender y de aportar nuestro pequeño granito de arena a la ciencia y al avance de la humanidad.

El objetivo central del proyecto AMS es el incremento del nivel de autonomía en los sistemas de control. Esta autonomía es crítica en aplicaciones desatendidas, de alta incertidumbre o como ayuda en el manejo por humanos de sistemas industriales complejos. Para ello se diseñará una arquitectura de control modular que permita la reconfiguración en caliente de sistemas complejos de control. Así mismo se emplearán mecanismos de modelado funcional que nos permitan dotar al sistema de un conocimiento profundo sobre la planta y el propio controlador. Igualmente, se elaborarán algoritmos de reconfiguración de los controladores para adaptar el sistema a circunstancias cambiantes.

El camino hacia la autonomía pasa a través del conocimiento disponible por el controlador; a través de la capacidad de realizar acciones inteligentes, como aprender, planificar o razonar sobre el efecto de sus propias acciones. En el control

tradicional este conocimiento se limita en la práctica a la planta bajo control, pero el conocimiento que el propio controlador dispone sobre sí mismo es también crítico.

Por lo tanto una de las líneas de trabajo de ASLab y objetivo a largo plazo es la creación de sistemas de control que estén en grado de aprender a controlar cualquier tipo de dispositivo o sistema; sistemas de control con conciencia de ellos mismos, capaces de pensar, de abstraer conceptos, de aprender e incluso de sentir.

El robot Pioneer 2-AT8 sería una de las plataformas susceptibles de ser controlada por este todavía hipotético sistema de control. Para conseguir esto, el paso previo es comunicar el robot con el cerebro donde será implementado el sistema, el cluster de computadores servidores del laboratorio.

La plataforma Pioneer 2-AT8 ha sido bautizada con el nombre de LIN. Con este denominativo se hará referencia al robot a lo largo de la presente memoria.

El proyecto tiene por lo tanto dos partes bien diferenciadas. La primera sería la investigación en el mercado, adquisición y montaje de los elementos hardware pertinentes para la conexión con el robot. El sistema hardware debe permitir la realización de labores de control local de relativo bajo nivel y la comunicación de la plataforma con la red local. La segunda sería el estudio de las diferentes tecnologías software para las comunicaciones y el diseño e implementación del sistema software de comunicación con la red local. Obviamente, para que la plataforma sea autónoma, esta comunicación deberá ser inalámbrica.

La presente memoria tiene en consecuencia tres partes. La primera es introductoria y hace referencia a los objetivos del proyecto. La segunda consiste en la descripción del estudio realizado para la adquisición del sistema hardware y de las características de los distintos elementos. La tercera y principal se ocupa de la descripción del sistema software desarrollado y de las tecnologías utilizadas para ello.

Capítulo 3

Objetivos

En este capítulo se expondrán los objetivos y especificaciones del presente proyecto de fin de carrera. Los mismos han sido elaborados a partir de entrevistas preliminares con el tutor Ricardo Sanz.

3.1. Instalación en el robot Pioneer 2-AT8 de un PC de a bordo

El primer objetivo del proyecto es dotar al robot de cerebro local, es decir, se deberá incorporar un PC de a bordo que permita la comunicación con el microprocesador de LIN, la ejecución de programas inteligentes de navegación y control del robot y la conexión del mismo con la red local del laboratorio. De este modo se podrán realizar desde la misma labores de control de alto nivel que requieran una potencia de cálculo elevada. Las especificaciones concretas que deberá cumplir el hardware son las siguientes:

1. En la máquina deberá correr un sistema operativo RTAI (Real Time Linux).

2. El tamaño de la placa y del resto de componentes hardware deberá ser menor que 20 x 18 x 5 cm, que es el tamaño del espacio libre en LIN destinado a albergar el computador de a bordo.

3. La placa deberá presentar una interfaz serial RS-232 para la conexión con el microprocesador del robot.

4. La placa deberá tener las conexiones pertinentes para albergar hardware de comunicaciones inalámbricas que permitan la comunicación del sistema con la red local.

5. El robot debe estar dotado de autonomía eléctrica. Es decir, el consumo eléctrico del hardware añadido (placa, memoria, disco duro si procede, tarjetas wireless, etc.) debe ser soportado por las baterías del robot.

6. El PC de a bordo debe permitir la conexión de dos cámaras estereoscópicas y presentar una interfaz de expansión que permita desarrollos futuros del sistema.

3.2. Instalación de Hardware para la comunicación del robot con la red local

El robot deberá estar en contacto con la red local de ordenadores por medio de una conexión inalámbrica. Ésta deberá estar dotada del mayor ancho de banda posible, o al menos de un ancho de banda suficiente para garantizar la transmisión de imagen. En desarrollos posteriores la inteligencia del robot y las labores de control de alto nivel y de guiado por visión serán implementadas en el cluster del laboratorio. Por lo tanto asegurar una conexión robusta y rápida es de vital importancia.

3.3. Diseño de software para las comunicaciones entre LIN y la red local

Asímismo se deberá diseñar un sistema software que permita la transmisión de toda la información sensorial proporcionada por el interfaz electrónico del robot desde éste a la red local y la teleoperación del robot desde la misma. La información deberá estar disponible en cuaquier punto de la red y podrá ser requerida en cualquier momento. Se deberá tener en cuenta que en los futuros desarrollos del sistema éste podrá estar constituído por plataformas heterogéneas, es decir, máquinas con arquitecturas y sistemas operativos diversos.

Éste es un objetivo primordial del proyecto, y se resume en la necesidad de hacer posible el control remoto del robot desde la red local.

Parte II

SISTEMA HARDWARE

Capítulo 4

Robot Pioneer 2-AT8

En este capítulo se expondrán las características, los diversos componentes y el estado del robot Pioneer 2-AT8 antes de la realización del presente proyecto.

El robot Pioneer 2-AT8 pertenece a la familia de robots móviles de ActivMedia. Fue adquirido por ASLab en febrero del 2003. ActivMedia Robotics diseña y construye robots móviles inteligentes así como sistemas de navegación, control y de soporte a la percepción sensorial para los mismos. Esta empresa ha vendido más de 1700 robots en todo el mundo y éstos han ganado varios y prestigiosos concursos.

Figura 4.1: Robots de Activmedia

El objetivo de la adquisición es incorporar al laboratorio una plataforma móvil e implementar sistemas de control inteligentes para la misma. Con la incor-

Figura 4.2: Pioneer 2-AT8

poración del robot al laboratorio ASLab dispone de una nueva plataforma con la que comenzar a hacer pruebas de control inteligente y por tanto avanzar hacia la consecución de su objetivo a largo plazo, que recordemos que la creación de sistemas de control capaces de aprender a controlar cualquier sistema y de tener conciencia de sí mismos.

Pioneer 2-AT8 es una robusta plataforma que incorpora todos los elementos necesarios para la implementación de un sistema de navegación y control del robot en una gran cantidad de entornos del mundo real.

El tamaño del robot es pequeño en comparación con sus prestaciones. Pesa 14 kg con una batería. Su estructura es de aluminio. Estas características le permiten poder transportar hasta 30 kg sobre él mismo. El microcontralador que gobierna los diversos dispositivos electrónicos conectados es un Hitachi H8S Los principales componentes del robot son:

- Panel

  No es más que la plataforma superior del robot. Está destinado al montaje de nuevos accesorios o elementos para el robot, como podrían ser cámaras, láser o brazos articulados. El panel está dotado con diversos orificios a través de los cuales podrían ubicarse los cables de los posibles dispositivos a añadir. Además a través del panel se puede acceder al interior del robot mediante una ranura de acceso.

- Panel de control

  Consiste en un panel de acceso al microcontrolador del robot. Está constituido por varios botones de control, leds indicadores de estado y un puerto de serie RS-232 con un conector de 9 pines.

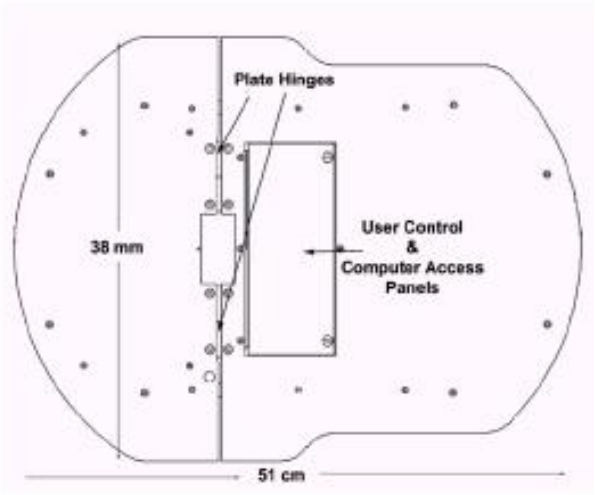

Figura 4.3: Panel superior

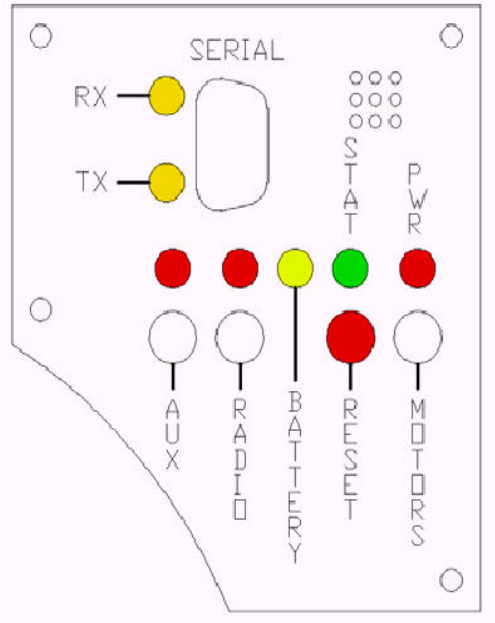

Figura 4.4: Panel de control

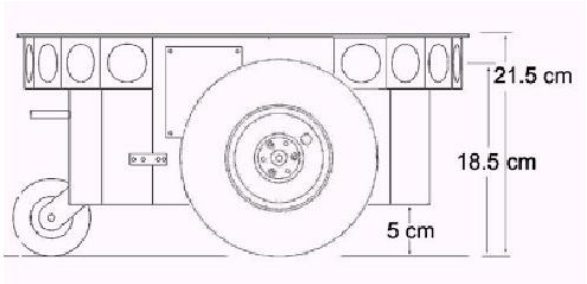

Figura 4.5: Cuerpo

El led rojo con la etiqueta *PWR* está encendido siempre que se encienda el robot. El led verde con la etiqueta *STAT* depende del modo de operación (presenta una lenta intermitencia cuando el microcontrolador espera una conexión con un cliente y rápida cuando el cliente se ha conectado). El led con la etiqueta *BATTERY* indica el estado de nuestras baterías (rojo si el votaje está por debajo de 11.5 voltios).

El conector serial incorpora dos leds que indican la entrada (*RX*) y salida (*TX*) de datos. A través de este puerto de serie se podrá establecer la comunicación con el microcontrolador del robot desde un PC externo al robot. Este puerto está compartido internamente con el puerto de serie al que se conectará el computador de abordo (on-board computer) o bien un radio modem. Mediante un circuito electrónico se deshabilita el puerto de serie interno si no existe una conexión con el computador de a bordo o un radio modem.

RADIO y *AUX* son dos interruptores que habilitan o deshabilitan respectivamente los eventuales dispositivos radio modem o el puerto de serie auxiliar. Ambos botones incorporan un led que indica el estado de los mismos.

El pulsador *RESET* resetea el microcontrolador, deshabilitando cualquier conexión activa con éste (sónares, motores, etc). Así mismo existe otro pulsador que activa o desactiva los motores.

- Cuerpo del robot

  El cuerpo de aluminio del robot aloja las baterias, los motores, los circuitos electrónicos y el resto de componentes. Además existe espacio para alojar diversos accesorios, como un PC de a bordo, un radio modem o radio ethernet o para incorporar sensores.

- Grupos de sónares

  El robot está dotado con dos grupos de 8 transductores (sónares) cada uno. Estos dispositivos permiten la detección de objetos y la determinación de

la distancia a la que se encuentran. Esta información puede ser utilizada para la elaboración de sistemas de navegación y control. Los sónares están situados en la parte frontal y trasera del robot. Cada grupo de transductores cubre un rango de 180º, de tal modo que el conjunto de sónares cubre los 360º alrededor del robot.

Cada grupo de sónares tiene su propia tarjeta electrónica controladora, lo que permite realizar un control independiente. Cada grupo de sónares está multiplexado. La frecuencia de adquisición de datos es de 25 Hz (40 milisegundos) por sonar. El rango de detección se sitúa entre los 10 cm y los 4 m. Se puede controlar qué sonar se dispara a través de software. Por defecto se disparan del 0 al 7.

Igualmente se puede ajustar la sensibilidad de los sónares y el rango detectado mediante un potenciómetro. Este hecho permite adaptar el robot al ambiente que le rodea. La configuración de los sónares con ganancias bajas reduce sus capacidad de detectar pequeños objetos. Este hecho puede ser beneficioso en el caso de que el robot se mueva en ambientes ruidosos o con superficies muy reflectantes. Por el contratio, aumentar la sensibilidad de los sónares estableciendo ganancias altas aumenta las posibilidades de detectar objetos pequeños y objetos que están a gran distancia. Esto es beneficioso si el ambiente en el que opera el robot es abierto y silecioso.

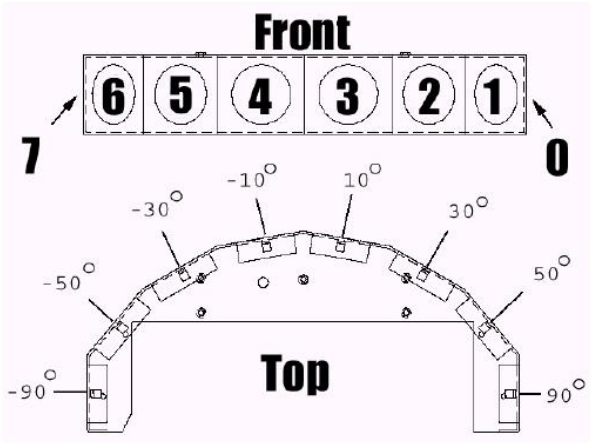

Figura 4.6: Disposición del grupo de sónares

- Motores y encoders

  El robot está dotado de 4 motores de corriente continua reversibles que pueden desarrollar altas velocidades y par de torsión. Cada uno de ellos está acompañado de un encoder óptico de gran precisión que permiten determinar la velocidad y la posición del robot.

Naturalmente el robot incorpora diversos circuitos electrónicos que gobiernan los anteriores elementos. La velocidad de los motores es regulada mediante señales del tipo PWM. Del mismo modo, este tipo de señales son las que dan los encoders al microcontrolador.

- Microcontrolador Hitachi H8S

  Los dispositivos electrónicos presentes en el robot son controlados por este modelo de microcontrolador. Su cometido es manejar los actuadores, disparar y recoger la señal de los sónares, controlar la electrónica del robot y realizar el resto de funciones de bajo nivel. Es capaz de comunicarse con otras máquinas a través de una interfaz serial RS232.

Figura 4.7: Microcontrolador Hitachi H8S

El microcontrolador, junto con su placa presenta las siguientes características:

- Opera a una frecuencia de 18MHz.
- Presenta 32K de memoria RAM y 128K de memora FLASH.
- Presenta 3 puertos de serie RS-232 configurables entre 9.6 y 115.2 kbaud.
- Ofrece 5 entradas y 2 salidas analógicas.
- Bus de 8 bits configurable como entrada o salida.

- Baterías

  El robot está alimentado con tres baterías de 12 voltios y 7 amperios-hora cada una. Son intercambiables entre sí y accesibles a través de una minipuerta en la parte trasera del robot. Por lo tanto disponemos en total de 252 watios-hora, lo que asegura varias horas de autonomía para la plataforma.

Capítulo 5

Computador de a bordo

5.1. Planteamiento del problema

Como se ha comentado anteriormente, la primera parte del proyecto consiste en dotar al robot de los elementos hardware necesarios para constituir su cerebro y el sistema de comunicaciones. En concreto, el presente capítulo se ocupa de describir las labores realizadas para la elección de un PC de bordo para el robot y de instalación del mismo.

Esta parte del proyecto es sin duda de vital importancia, pues su correcto desarrollo permite la realización del resto. La correcta elección de la placa y microprocesador a incorporar permitirá el desarrollo de la arquitectura hardware necesaria para comenzar a desarrollar software de control y de comunicaciones para el robot. Sin la existencia de esta plataforma no podría correr ningún tipo de software sobre el robot.

Comenzaron por tanto los trabajos en el proyecto con la búsqueda en el mercado de una placa y un microprocesador adecuados para LIN. Se puso especial atención en determinar los requisitos que debía cumplir y en que la solución adoptada cumpliera los mismos. A grandes rasgos la placa debía asegurar la comunicación con el microprocesador de LIN, la ejecución de programas inteligentes de navegación y control del robot y la conexión del mismo con la red inalámbrica local del laboratorio. Estos requisitos no tienen nada de extraordinario y seguramente muchas de las placas existentes en el mercado están en grado de cumplirlos. Sin embargo, como el PC debe ser incorporado a una plataforma móvil existen una serie de restricciones que hacen que la placa buscada deba tener características especiales.

La primera de ellas es el tamaño. El robot deja una pequeña cavidad de 20 x 18 cm para alojar los diversos componentes hardware. Este hecho condiciona fuertemente la elección de la placa, pues la mayoría de ellas son de un tamaño

mayor.

La segunda de ellas es el consumo eléctrico. Dado que se trata de una plataforma móvil, el sistema debe ser alimentado por las tres baterías del robot (ver capítulo 4.2). Estas baterías deben alimentar a los motores, a la electrónica de control de los dispositivos del robot, al microprocesador Hitachi H8S, al microprocesador que se incorpore y al resto de elementos (a priori un disco duro y una tarjeta wireless). Este hecho deberá ser considerado a la hora de la elección de la placa.

Por otra parte, otro condicionante lo constituyen los posibles desarrollos futuros del sistema. Está planificada la incorporación de dos cámaras de video al robot. El sistema elegido deberá tener la suficiente potencia en términos de cálculo y memoria para transmitir la señal de estas cámaras. Igualmente deben existir las conexiones necesarias para las mismas.

Este último requisito entra en conflicto directo con los dos anteriores. A menor tamaño y consumo los sistemas ofrecen menores prestaciones. Este hecho ha sido especialmente conflictivo a la hora de la adquisición de la placa, y se ha debido llegar a una solución de compromiso.

En la sección 3.1 se exponen los diversos requisitos que debe cumplir el sistema hardware que será incorporado al robot. Se recuerda de nuevo cuáles son:

- En la máquina deberá correr un sistema operativo Linux. Este hecho se debe a varios factores. Uno de ellos es que el paquete sofware ARIA (capítulo 10), aunque disponible para Windows, está fuertemente orientado a Linux. Otro de ellos es la apuesta de ASLab por este sistema operativo, que desde nuestro punto de vista ofrece más confianza y mejores prestaciones que otras plataformas. Además el sistema operativo Linux se puede descargar gratuitamente de la red.

- El espacio ocupado por la placa y del resto de componentes hardware deberá ser menor que 20 x 18 x 5 cm, que es el tamaño del espacio libre en LIN destinado a albergar el computador de a bordo. A pesar de la existencia de una plataforma en la parte superior del robot, ésta será destinada al alojamiento de cámaras y otros accesorios.

- La placa deberá presentar una interfaz serial RS-232 para la conexión con el microprocesador del robot. Éste es el único modo que existe para comunicarnos con el microprocesador Hitachi H8S (capítulo 4).

- La placa deberá tener las conexiones pertinentes para albergar hardware de comunicaciones inalámbricas que permitan la comunicación del sistema con la red local del laboratio ASLab. Este tipo de conexión consiste en un slot (ranura) para una tarjeta wireless PCMCIA.

- El robot debe estar dotado de autonomía eléctrica. Es decir, el consumo eléctrico del hardware añadido (placa, memoria, disco duro si procede, tarjetas wireless, etc.) debe ser soportado por las baterías del robot.

- El PC de a bordo debe permitir la conexión de cámaras y presentar una interfaz de expansión que permita desarrollos futuros del sistema.

- La placa debe presentar una conexión capaz de albergar memoria flash. Este requisito está estrechamente relacionado con el anterior, pues se desea la memoria flash para el posible almacenamiento futuro del sistema operativo de la plataforma y de los diversos programas de pilotaje y comunicaciones en vez de usar un disco duro.

- Presupuesto. En los proyectos de ingeniería el presupuesto es quizá la parte más importante. Sin embargo en este caso no han existido restricciones presupuestarias para la adquisición del material. Esto no quiere decir que no se hayan comparado los precios de los distintos componentes hardware y que no se hayan tenido en cuenta éstos a la hora de realizar la compra.

5.2. Placa GENE-6330

Teniendo en cuenta los diversos requisitos y tras un periodo de búsqueda en el mercado la placa que mejor se adapta a las características deseadas es el modelo GENE-6330. Ha sido adquirida a la empresa Aspid comunicaciones, que se dedica a la distribución de componentes hardware en España. Los componentes adquiridos y su precio son los siguientes:

Componente	**Precio**
GENE-6330	305 €
Memoria RAM	135 €
Convertidor DC-DC	30 €
Disco duro Fujitsu	150 €

Cuadro 5.1: Componentes hardware

La placa GENE-6330 satisface todos los requisitos enunciados anteriormente. Su tamaño (146 x 101 mm) y su reducido consumo eléctrico son sus principales características y los factores clave que nos han decantado por su adquisición. Por otra parte tanto el precio como las prestaciones ofrecidas son también muy satisfactorias en comparación con otros productos existentes (ver sección 5.3). A continuación se describen sus características y diversos componentes:

CPU
El microprocesador de la plataforma es un Transmeta Crusoe modelo TM5400 a 600 Mhz. Presenta un muy bajo consumo eléctrico, lo que le permite carecer de ventilador al no experimentar apenas calentamiento.

Memoria
La placa presenta 64 Mb de memoria SDRAM onboard. Además tiene un zócalo SODIMM en el que se inserirá una memoria RAM de 512MB.

Características físicas
El tamaño de la placa es de 146 mm x 101.6 mm x 26 mm (5.75"x 4") y su peso de 0.4 Kg.

BIOS
La memoria BIOS está constituida por una memoria FLASH ROM de 256 KB del fabricante Award.

Ethernet
La plataforma presenta un conector RJ-45 para el acceso redes ethernet.

Memoria Flash
La placa presenta una ranura que admite memoria CompactFlash de tipo II.

Interfaz de expansión
La plataforma dispone de una ranura PCMCIA en la que se podrá inserir tarjetas PCMCIA con diversos fines. En nuestro caso se usará una tarjeta PCMCIA para la comunicación inalámbrica.

Alimentación
El dispositivo necesita una alimentación de +5V y +12V de tipo AT.

Dispositivos de entrada salida

- Ranura IDE (UDMA33)
- Ranura FDD para floppy
- Puerto COM para un ratón o un teclado.
- Dos puertos de serie RS-232
- Un puerto de serie configurable RS-232/422/485
- Puerto paralelo
- Cuatro puertos USB 1.1
- Chip controlador de audio VIA VT82C686B y VT1612A Audio CODEC

Controlador gráfico
SMI LynxEM + SM712 con 4MB de memoria SGRAM

Figura 5.1: Placa GENE-6330

Como he dicho antes la principal característica de la plataforma es su bajo consumo. El microprocesador Transmeta Crusoe TM5400 consume menos de un watio cuando realiza funciones normales y menos de dos watios cuando realiza funciones que requieren elevados recursos como podrían ser la compresión y transmisión de imagen. Este hecho hace de él uno de los microprocesadores con una mejor relación prestaciones/consumo del mercado. Debido a ello el microprocesador no necesita ventilador refrigerante.

La otra principal restricción era el tamaño. Gene-6330 ha sido implementada con diversos módulos de reducido tamaño como onboard RAM o la ranura para memoria SODIMM que hacen de ella una placa de un tamaño muy reducido. De hecho en el mercado no existen placas de dimensiones menores y que a la vez presenten una interfaz (ranura PCMCIA o PCI) para la comunicación inalámbrica.

Por otra parte la plataforma presenta todos los dispositivos de entrada y salida requeridos para el proyecto. Es de especial importancia la ranura PCMCIA. La misma permitirá la comunicación inalámbrica con la red local mediante una tarjeta wireless (ver capítulo 6.4).

Igualmente se utilizará uno de los puertos de serie RS-232 para la comunicación con el microcontrolador Hitachi H8S (sección 4.7). El resto de puertos de serie serán utilizables para la comunicación con otros dispositivos. El teclado se conectará al puerto COM destinado a tal fin.

La ranura IDE se utilizará para conectar un disco duro en el que se instalará el sistema operativo. Se utilizará un disco duro fujitsu de 20Gb de capacidad de almacenamiento disponible en el laboratorio. Al igual que la placa el disco duro es de reducidas dimensiones (5 cm x 8 cm), al ser del mismo tipo que los discos utilizados en equipos portátiles.

El resto de dispositivos de entrada y salida: el puerto paralelo, las ranuras floppy, la ranura CompactFlash y los puertos USB quedan libres. Es previsible que a dos de los puertos USB sean utilizados para conectar dos cámaras estereoscópicas.

5.3. Opciones descartadas

Al explorar el mercado en busca de un producto que se ajustase a las necesidades del proyecto se encontraron varios que podían ser también adecuados, pero que fueron descartados por diversos motivos. Las opciones descartadas son las siguientes:

PCM-8500

El modelo PCM-8500, ofrecido por los mismos distribuidores (Aspid), es una placa de muy altas prestaciones, superiores a la plataforma elegida. Soporta un microprocesador Pentium 4 de hasta 2.4GHz y memoria RAM de hasta 1Gb. Además presenta todos los dispositivos de entrada y salida necesarios (ranuras PCM-CIA, puertos de seria y USB, etc). El tamaño, sin embargo, es de 203 mm x 146 mm y por tanto superior al de nuestra placa. El modelo PCM-8500 entraría a duras penas en el robot. Pero sin duda alguna el motivo principal para descartar esta placa ha sido el consumo eléctrico. Presenta una conexión para la alimentación de tipo ATX y el microprocesador consume varios watios de potencia.

PCM-6892

Este modelo también es vendido por la empresa Aspid. Es una plataforma de características muy parecidas a la elegida. Incorpora un microprocesador VIA C3 Eden a 667 Mhz de muy bajo consumo y memoria RAM de 512 Mb. Así mismo dispone de ranura PCMCIA, puertos de serie y USB. Su precio también es similar al de GENE-6330, 300 euros. Sin embargo su tamaño es de 203 mm x 146 mm. Ante iguales prestaciones hemos preferido escoger la placa más pequeña.

IB-760

Este modelo es ofrecido por la empresa OrbitMicro. Incorpora un microprocesador Transmeta Crusoe a 800MHz (muy similar al de GENE-6330) y memoria de 128 Mb. Su interfaz de expansión es muy amplia y presenta ranuras PCMCIA, PCI y PC104. Sin embargo de nuevo su tamaño (203 mm x 146 mm) y su precio de 723 euros (más del doble que los 305 euros de nuestra plataforma) han hecho

descartar esta opción.

Capítulo 6

Comunicación inalámbrica

6.1. Redes Wireless LAN

En los últimos años se ha producido un crecimiento espectacular en lo referente al desarrollo y aceptación de las comunicaciones móviles y en concreto de las redes de área local (Wireless LANs). La función principal de este tipo de redes es la proporcionar conectividad y acceso a las tradicionales redes cableadas (Ethernet, Token Ring...), como si de una extensión de éstas últimas se tratara, pero con la flexibilidad y movilidad que ofrecen las comunicaciones inalámbricas. La tecnología LAN inalámbrica permite al cliente comunicación computacional sin cables para acceder a los servicios de área local.

Los sistemas WLAN no pretenden sustituir a las tradicionales redes cableadas, sino más bien complementarlas. En este sentido el objetivo fundamental de las redes WLAN es el de proporcionar las facilidades no disponibles en los sistemas cableados y formar una red total donde coexistan los dos tipos de sistemas.

Es obvio que por la naturaleza móvil de la plataforma con la que se está trabajando, se precisa una conexión inalámbrica. En desarrollos posteriores del sistema, la inteligencia del robot y las labores de control de alto nivel y de guiado por visión serán implementadas en el cluster de computadores del laboratorio, pues de esta forma se dispone de una mayor potencia de cálculo.

Definición de WLAN

Una red de área local inalámbrica puede definirse como a una red de alcance local que tiene como medio de transmisión el campo eléctrico.

Por red de área local entendemos una red que cubre un entorno geográfico limitado, con una velocidad de transferencia de datos relativamente alta (mayor o igual a 1 Mbps tal y como especifica el IEEE), con baja tasa de errores y administrada de forma privada.

Por red inalámbrica entendemos una red que utiliza ondas electromagnéticas como medio de transmisión de la información que viaja a través del canal inalámbrico enlazando los diferentes equipos o terminales móviles asociados a la red. Estos enlaces se implementan básicamente a través de tecnologías de microondas y de infrarrojos. En las redes tradicionales cableadas esta información viaja a través de cables coaxiales, pares trenzados o fibra óptica.

Una red de área local inalámbrica, también llamada wireless LAN (WLAN), es un sistema flexible de comunicaciones que puede implementarse como una extensión o directamente como una alternativa a una red cableada.

Este tipo de redes utiliza tecnología de radiofrecuencia minimizando así la necesidad de conexiones cableadas. Este hecho proporciona al usuario una gran movilidad sin perder conectividad. El atractivo fundamental de este tipo de redes es la facilidad de instalación y el ahorro que supone la supresión del medio de transmisión cableado. Además las prestaciones de este tipo de redes se han situado a la par que las de las redes cableadas. Se ha pasado de velocidades de transmisión entre los 2 y los 10 Mbps en el año 2001 a velocidades de entre 10 y 100 Mbps en el año 2003.

6.2. Composición de una red wireless

Una red wireless es una red inalámbrica de área local típicamente conectada a un tipo de red metropolitana, normalmente localizada en un área geográfica como puede ser una ciudad o un pequeño pueblo.

Varias interfases wireless estandard existen actualmente, la más común en este momento es la familia 802.11, en la que se comprenden: 802.11, 802.11a, 802.11b y 802.11g.

Cada red wireless está compuesta por uno o varios nodos conectados juntos. Un nodo es una colección de varios PC's u otros equipos conectados juntos directamente usando la red IP y sobre un rango directo de radio.

Un nodo consiste mínimo de un router y de uno o más clientes. Los clientes normalmente requieren una pequeña configuración y solo hablan con el router, mientras el router enviará sus propios datos y los datos de los clientes al resto de la red.

El nodo participará en el intercambio de información con otros nodos garantizando siempre el alcance al resto de la red. Los nodos pueden ser conectados por radio o por otros medios.

6.3. El standard 802.11

Una vez instalado en LIN un computador de a bordo, es necesario añadir los elementos hardware necesarios para habilitar la comunicación inalámbrica con la red local.

Varias son las tecnologías disponibles en este momento para implementar una conexión de este tipo: 802.11, Bluetooth, HomeRF, IrDA y Ultra Wide Band (ver sección 6.6). Sin embargo el laboratorio ASLab cuenta ya con una red inalámbrica compuesta por varios equipos portátiles que está basada en el standard 802.11b. Sus principales características son las siguientes:

- Velocidad de transmisión de 11 Mbps, equiparable a redes convencionales de cable.
- Rango de cobertura de 100m, que proporcionará conectividad a la red en cualquier punto del edificio.
- Topología de estrella, que permitirá la administración de la red desde un único punto (al igual que los hubs en las redes cableadas), además de minimizar el impacto derivado del fallo de un equipo de un usuario final.
- Seguridad en las comunicaciones, maximizando la confidencialidad de los datos que viajan por la red.
- Interoperatibilidad con redes Ethernet, que permitirá el acceso a los recursos de la red Ethernet ya existentes, sin suponer un costo adicional en equipamiento de interconexión.
- Bajo consumo, que concede mayor autonomía a los equipos portátiles.

6.4. Tarjeta wireless Compaq wl110

Según lo enunciado anteriormente se ha decidido utilizar para el desarrollo del proyecto una de las tarjetas wireless disponibles en el laboratorio: la Compaq wl110.

A pesar de que en estos momentos existen en el mercado tarjetas del mismo tipo capaces de transmitir datos a velocidades susceptiblemente mayores, se justifica el uso de la tarjeta Compaq porque en el momento actual sus prestaciones son más que suficientes para transmitir los datos deseados. Es cierto que harán necesarias mayores prestaciones en el momento en que se transmitan imágenes. Sin embargo, aunque se sabe que el paso siguiente al presente proyecto será la

Figura 6.1: Tarjeta Compaq wl110

instalación de cámaras sobre el robot, no está claro cuándo se producirá este hecho. Por lo tanto si en estos momentos se adquiriera otro dispositivo wireless podría darse la posibilidad de que para el momento en que las cámaras estuvieran operativas ya existieran otros dispositivos o tecnologías más apropiados.

Conjuntamente con las tarjetas PCMCIA se adquirió un Access Point de Compaq. Este permite comunicar la red inalámbrica formada por las tarjetas PCMCIA con la red LAN del laboratorio.

Las pruebas realizadas demuestran que si bien la distancia operativa es aceptable, la velocidad de transmisión no alcanza los niveles esperados.

La distancia máxima a la que se ha podido mantener una comunicación entre la tarjeta y el access point es de aproximadamente 50 metros, a través de los gruesos muros exteriores del laboratorio.

Respecto a la velocidad de transmisión, si bien la máxima especificada es de 11 Mbps, en laboratorio y bajo condiciones idóneas, solo hemos obtenido una velocidad de 5.5 Mbps, con picos de 8 Mbps. A pesar de ello esta velocidad de transmisión es suficiente para cumplir con las especificaciones del proyecto y para el correcto funcionamiento del software de comunicaciones a desarrollar.

Las características específicas de la tarjeta son las especificadas en la tabla 6.1

6.5. Access Point wl410

Un Punto de Acceso, access point, o estación base, es un receptor de radio y transmisor que se conecta a tu red cableada Ethernet. A través de este dispositivo

Formato	PCMCIA
Alimentación	Fuente de alimentación 5 V / +25 mA
Velocidad	11, 5,5, 2 y 1 Mbps; Selecciones de velocidad automática (ARS)
Dimensiones	117,8 x 53,95 x 8,7 mm (Tarjeta PC)
Otras características	Difusión: Secuencia Barker de 11 chips
Compatibilidades	Canales secundarios certificados en todo el mundo (FCC/ETS/JP/FR), 11 canales
Comunicaciones	Modulación por salto de frecuencias directo (CCK, DQPSK, DBPSK)
Conexión	Alcance en metros a 11Mbps: Oficina Abierta: 160 m Oficina semiabierta: 50 m Oficina cerrada: 25m Sensibilidad del receptor dBm: -82 Retardo (a REF de <1%): 65 ns
Consumo	Modo en espera: 9 mA Modo de recepción: 185 mA Modo de transmisión: 285 mA
Frecuencias	Espectro de 2.400 - 2.483,5 MHz
Cumplimiento de normativas	Estándares IEEE 802.11b y WiFi
Interfaz	Tarjeta PC, PCI
Potencia	15 dBm
Seguridad	Encriptación WEP (Wired Equivalent Privacy) de 128 bits Tasa de error mejor que 10-5 Protocolo de Acceso CSMA/CA (evasión de colisiones) con ACK
Temperaturas de funcionamiento	0-55 o C Humedad máxima 95% (sin condensación)
Sistemas operativos compatibles	Windows 95/98/ME/NT4 (SP4)/2000 Windows CE/Pocket PC
Cobertura internacional	Europa/APA: ETS 300-328, Sello CE EE.UU.: FCC (47 CFR) Parte 15C, Artículo 15.247 Canadá: ISC RSS139 Japón: Normativas de radio MPT
Requisitos	1 ranura PCMCIA; Windows 95, Windows 98, Windows 2000, Windows NT, o Windows Me

Cuadro 6.1: Características Compaq wl110

Figura 6.2: Access Point Compaq wl110

nodos inalámbricos, como un portátil equipado con una tarjeta inalámbrica, tienen acceso a servicios de red LAN cableada. El rango operativo, administración de capacidades, seguridad de red inalámbrica y número de usuarios soportados son determinados por las capacidades del Punto de Acceso.

Consta de una conexión RJ45 donde le conectamos un cable de red. El otro extremo del cable es conectado a un hub. Dicho hub a su vez está conectado o forma parte de una red alámbrica (LAN). De esta forma, los nodos inalámbricos entran a formar parte de la propia LAN, como si no existiera una comunicación inalámbrica.

Consta de una conexión RJ45 donde le conectamos un cable de red y el otro extremo es conectado a un hub, dicho hub a su vez está conectado o forma parte de una red alámbrica (LAN) de tal forma que todos los nodos inalámbricos entren a formar parte de la propia LAN, como si no existiera una comunicación inalámbrica.

El punto de acceso que está operativo en el laboratorio es un Compaq wl410 y fue adquirido junto con las tarjetas PCMCIA. Sus características son las indicadas en la tabla 6.2:

6.6. Otras tecnologías

6.6.1. Bluetooth

Bluetooth es un protocolo punto a punto realizado para conectar inalámbricamente teléfonos celulares, portátiles, ordenadores de mano, cámaras digitales e

Formato	Sobremesa / Mural
Alimentación	Unidad de pared externa: Detección automática 100/240 VCA 47-63 Hz, 9V/1,1A Preparado para Active Ethernet (Alimentación mediante Ethernet)
Dimensiones	146 x 175 x 70 mm (incluye soporte para mesa) 130 x 175 x 46 mm (incluye soporte para pared)
Otras características	- Encendido - Actividad LAN Ethernet - Actividad LAN inalámbrica,
Comunicaciones	Conexión Ethernet 10 Base T (RJ45)
Conexión	Alcance en metros a 11Mbps: Oficina Abierta: 160m Oficina semiabierta: 50m Oficina cerrada: 25m Sensibilidad del receptor dBm: -82 Retardo (a REF de <1%): 65 ns
Especificaciones mecánicas	Cubierta de plástico Placa de montaje que permite montaje en pared o mesa Conector de antena externo y accesible
Cumplimiento de normativas	Cumplimiento IEEE 802.11b (Wi-Fi)
Interfaz	Ethernet 802.3 10Base-T (Conector RJ 45)
Peso	0,50 kg
Seguridad	Encriptación de 128 bits WEP
Temperaturas de funcionamiento	0 °C a + 40 °C Humedad máxima de funcionamiento: 90% (condensación no permitida)

Cuadro 6.2: Características Compaq wl410

impresoras. Opera en cortas distancias de unos 10 metros, eliminando la necesidad de cables e infrarrojos.

Los motivos principales para la no consideración de esta tecnología fueron la escasa velocidad (1Mbps) y su corto alcance (10m). A favor de esta tecnología cabe citar la amplia acogida en el mercado y el bajo coste en relación a otras alternativas. Además se están desarrollando puntos de acceso que permitirán ofrecer coberturas de hasta 100m y se está trabajando para que la versión 2 de este estándar soporte transmisiones a 4 Mbps.

6.6.2. HomeRF

En contraste a Bluetooth, HomeRF es un protocolo WLAN con un rango de unos 50 metros y capaz de conectar muchos aparatos simultáneamente. Ambos protocolos son optimizados de manera completamente diferentemente y diseñados para tipos diferentes de aplicaciones.

Por otra parte, mientras el estándar 802.11b era desarrollado originalmente para redes de empresas, HomeRF era ideado desde el principio para reunir las necesidades del consumidor en las aplicaciones de la red casa: asequible, robustez y facilidad de uso. Una importante diferencia de HomeRF frente a la 802.11b es la habilidad de compartir el protocolo de acceso inalámbrico (SWAP) para transportar datos asíncronos, permitiendo la integración real ordenador-telefonía.

Los motivos principales para la no consideración de esta tecnología fueron su escasa velocidad (2 Mbps) y el enfoque de dirigir los productos al hogar.

6.6.3. IrDA

Esta técnica de transmisión basada en infrarrojos es la usada en controles remotos. Es una tecnología punto a punto, muy direccional e incapaz de atravesar materiales opacos. Estas características la hacen inválida para su uso en el presente proyecto.

6.6.4. Ultra Wide Band (UWB)

La banda ultra ancha es un tipo nuevo de comunicaciones de ondas de radio que puede transmitir datos sobre una corta distancia con más flexibilidad que otras frecuencias de radio

La banda ultra ancha (UWB) es una tecnología para comunicación de radio, localización de precisión y radar portátil. Existe desde el año 1990 y ha sido principalmente usada por militares. Pero en los últimos años la Comisión Federal de Comunicaciones en los USA ha dado luz verde a la utilización de la UWB para ser usadas en aparatos comerciales.

UWB es un método barato para distribuir inalámbricamente gran cantidad de datos en una banda ancha de hasta un kilómetro de rango. Un aparato UWB trabaja emitiendo cortas series de pulsos eléctricos de baja potencia (billonésimas de segundo) los cuales no son dirigidos en una frecuencia particular en el espectro de radio pero a través del espectro entero, a través de todas las frecuencias a la vez.

Esta tecnología es la más avanzada actualmente, disponiendo de las veloci-

dades de transmisión más altas. A pesar de ello, todavía es una tecnología muy reciente y lamentablemente no se ha podido tener acceso a la misma.

Si esta nueva tecnología es aprobada (existe la preocupación de que interfiera las actuales transmisiones de GPS, tráfico aéreo y televisión) podría conducir al desarrollo de una tecnología muy superior a las LAN inalámbricas del estándar 802.11. UWB ofrece menor costo, un ancho de banda mayor y un mayor soporte al usuario que la 802.11 y Bluetooth. La señal de UWB no emplea portadora de señal, su consumo es bastante pequeño, en torno a los 50 a 70 mW.

Parte III

SISTEMA SOFTWARE PARA LAS COMUNICACIONES

Capítulo 7

Planteamiento del problema

Una vez instalados todos los componentes hardware del sistema, se abordó el el problema del diseño software del mismo. Para comenzar, se debía determinar el sistema operativo a utilizar, el lenguaje de programación con que se programaría el sistema y su arquitectura de alto nivel.

Para determinar todo ello hice un estudio de los diversos requisitos del sistema, que a continuación se expone.

7.1. Análisis de requisitos

7.1.1. Objetivos

Los objetivos del análisis de requerimientos es determinar los diversos requisitos que debe cumplir el sistema software. Se debía determinar:

- Sistema operativo que correrá en el computador de a bordo del robot.
- Lenguaje o lenguajes de programación en que será implementado el sistema
- Herramientas software que se utilizarán en esta implementación.
- Funciones principales que deberá realizar el software
- Esquema o boceto de la arquitectura de alto nivel del sistema.

7.1.2. Catálogo de requerimientos

A continuación se realiza un análisis de los diversos requisitos que el sistema software debe satisfacer. Estos requisititos fueron elaborados en virtud de progre-

sivas entrevistas con el tutor del proyecto y de la propia naturaleza del sistema. Me gustaría destacar que para la mejora del sistema los requisitos han sido sujetos a modificaciones progresivas según avanzaban los prototipos software implementados.

Requisitos funcionales

- Conexión local con el robot desde la CPU instalada en LIN. Obviamente, una de las funciones del sistema software será la de establecer una conexión local con el robot vía un interfaz serial. A través de ella se podrán enviar comandos y recibir información.
- Obtención local de la información de los sensores.
- Conexión remota con el robot (desde la red local). Así mismo, el sistema deberá permitir establecer una conexión con el robot desde la red de ASLab. Al igual que desde la conexión local, a través de la conexión remota se podrán enviar comandos al robot y recibir información.
- Transmisión de toda la información sensorial que proporcionan los sensores desde LIN a la red local. La información deberá estar disponible en cualquier punto de la red y podrá ser requerida en cualquier momento.
- Transmisión de las órdenes de teleoperación desde la red local al robot.
- Protección de LIN ante eventuales colisiones contra obstáculos por error del teleoperador.

Requisitos de usuario

El usuario de la aplicación será cualquier persona que tenga una cuenta en la red de ASLab.

Requisitos tecnológicos

Por la naturaleza del sistema físico la aplicación se ejecutará sobre un esquema cliente/servidor, con los procesos e interfaz de usuario ejecutándose en los clientes y éstos solicitando requerimientos al servidor. La máquina donde correrá el servidor será la CPU de LIN. El servidor será a su vez cliente del microcontrolador del robot (HITACHI H8S) y le requerirá la información sensorial.

El cliente o los clientes correrán sobre los terminales de la red local de ASLab. Los sistemas operativos utilizados en estos terminales son Red Hat Linux 9.0, Linux Mandrake 9.2 y Windowx XP.

El sistema operativo de la máquina donde correrá el servidor (la CPU del robot) deberá ser un sistema Linux.

El fabricante del robot proporciona una API (application programming interface) llamada ARIA que proporciona diversas funciones para la conexión e intercambio de información con el robot. ARIA es un paquete de clases programado en C++. Este hecho condiciona fuertemente a que éste sea el lenguaje de programación utilizado. La elección de otro lenguaje de programación como Java complicaría el desarrollo del sistema. Por lo tanto el sistema software será programado en C++.

Aunque en estos momentos el robot y la red local utilicen sistemas operativos Linux y tengan arquitecturas similares ("i386"), en el futuro se podrían añadir al sistema máquinas con arquitecturas y sistemas operativos diversos (como pueden ser sistemas para visión integrados o diversos microprocesadores). El software diseñado debe prever este hecho y estar preparado para la heterogeneidad del sistema. Esto lleva a la necesidad de la utilización de alguna herramienta para la programación distribuida heterogénea. Se ha pensado en CORBA como solución a este problema.

Requisitos de rendimiento

No se han especificado requisitos concretos en este sentido. Se seguirá la política de dotar al sistema del máximo rendimiento en todos sus aspectos (máximo ancho de banda en la transmisión de datos, mínimo tiempo de respuesta de los actuadores del robot, etc) .

7.2. Conclusiones

Del análisis de requisitos podemos obtener las siguientes conclusiones:

- El sistema operativo que usará el cerebro del robot es Linux Mandrake 9.2. Esta elección no solo cumple con las especificaciones, sino que además es sabido que no existen problemas de configuración con la tarjeta wireless ni con el resto de hardware utilizado. Además la licencia de uso de este sistema operativo (como el resto de sistemas operativos Linux) es gratuita.
- El lenguaje de programación utilizado será C++. Como herramienta software para el desarrollo se usará el paquete KDevelop, cuya licencia se incluye dentro del GNU Licence (es decir, gratuita).
- Por otra parte, para hacer compatible al software con un sistema distribuido y heterogéneo se utilizarán las librerías de ICa. ICa es una especificación

para CORBA, como se verá más adelante (ver capítulo 14).

- Las funciones que debe realizar el sistema son las indicadas en el apartado Requisitos funcionales.
- Como se ha dicho anteriormente, por la naturaleza del sistema físico la aplicación se ejecutará sobre un esquema cliente/servidor. La máquina donde correrá el servidor será la CPU de LIN. El servidor será a su vez cliente del microcontrolador del robot (HITACHI H8S) y le requerirá la información sensorial. El cliente o los clientes correrán sobre los terminales de la red local de ASLab.

Capítulo 8

Instalación del sistema operativo y configuración del sistema

Antes de comenzar a desarrollar cualquier tipo de sistema software es preciso instalar el sistema operativo que lo soportará. El presente capítulo describe los diversos pasos para la instalación del sistema operativo en la CPU del robot, la configuración de los diversos dispositivos y la instalación de las aplicaciones necesarias para el desarrollo del proyecto.

Sobre la CPU de LIN correrá el sistema operativo **Linux Mandrake 9.2**. El hecho de que se haya elegido un sistema operativo UNIX se debe a varios factores. Uno de ellos es la apuesta de ASLab por este tipo sistema operativo, que desde nuestro punto de vista ofrece más confianza y mejores prestaciones que otras plataformas. Otro de ellos es que el paquete software ARIA (capítulo 10), aunque presente para Windows, está fuertemente orientado a Linux. Además, y esto es, desde mi punto de vista, la principal ventaja del sistema operativo Linux, tanto el sistema operativo como las aplicaciones son gratuitas.

Se ha elegido la distribución Linux Mandrake 9.2 porque ya se utiliza tanto en los terminales de la red local del laboratorio ASLab como en equipos portátiles con resultados satisfactorios. Se sabe además que no existen conflictos entre este sistema operativo y la tarjeta wireless que será utilizada para las comunicaciones (ver sección 6.4)

Las labores de instalación del sistema no tiene mayor dificultad. La mayor parte del tiempo se deberá tan sólo seguir las instrucciones del cd-rom de instalación. Éste detecta los diversos dispositivos hardware del sistema y establece su configuración (incluida la tarjeta wireless). El administrador del sistema tan sólo deberá crear las particiones que desee e introducir los parámetros para la configuración de la red.

Se crean dos particiones en el disco duro. La primera de ellas es de tipo *swap*

y su tamaño es de 510 Mb. Este tipo de partición es utilizada por el sistema operativo para descargar en ella datos de la memoria principal que no estén siendo utilizados. La segunda partición es creada con el sistema de ficheros *ext3*. Esta partición será *montada* en el directorio raíz / al arrancar el sistema operativo. En ella residirán todos los datos y aplicaciones del sistema.

Para configurar la red no hay más que introducir los siguientes parámetros cuando son requeridos:

```
Dirección IP:       138.100.76.22
Máscara de subred:  255.255.255.0
Puerta de enlace:   138.100.76.1
DNS Primario:       138.100.4.4
```

Tras instalar el sistema operativo y configurar las diversas cuentas de usuario se procede a instalar el paquete ARIA (ver capítulo 10). Para ello tan sólo se deberá cliquear en el fichero **ARIA-1.1-11.i386.rpm** y la instalación se hará efectiva.

Capítulo 9

Sistema Operativo AROS

Como se ha indicado en capítulos anteriores, el robot Pioneer 2-AT8 es controlado en primera instancia por un microprocesador Hitachi H8S (capítulo 4). Para el desarrollo de la comunicación de éste con cualquier otra máquina es preciso utilizar una arquitectura cliente - servidor. Según este modelo, sobre el microcontrolador del robot corren los procesos servidores que se encargan del manejo y control de las tarjetas controladoras de los dispositivos electrónicos y de realizar las funciones de bajo nivel del sistema.

AROS (ActivMedia Robotics Operating System) es el conjunto de estos procesos servidores y constituye un sistema operativo que corre en el microprocesador. Es un software de bajo nivel cuyo cometido es manejar la regulación de velocidad de los motores, disparar y recoger la señal de los sonars, recoger las señales de los encoders y en general llevar a cabo todas las funciones de bajo nivel. Además es el responsable de transmitir por medio de comandos esta información a otra aplicación, la cuál será cliente de AROS. En nuestro caso, el cliente de AROS será un programa que correrá sobre la CPU de LIN y que enviará comandos y recibirá información del microprocesador Hitachi. Se aclara ya desde ahora que este proceso cliente de AROS será a su vez servidor de eventuales programas clientes de la red local.

Una de las principales características de AROS es que es actualizable y compatible con otros robots del mismo fabricante.

9.1. Protocolo de comunicación entre AROS y un cliente

AROS se comunicará con su cliente mediante una interfaz serial RS-232 usando un protocolo especial de comunicaciones para la transmisión de paquetes de

datos, de un tipo desde el cliente al servidor y de otro distinto desde el servidor al cliente. Los paquetes de datos que envía AROS se llaman SIPs (server information packets). Los paquetes que recibe son comandos a ejecutar en los actuadores (client commands).

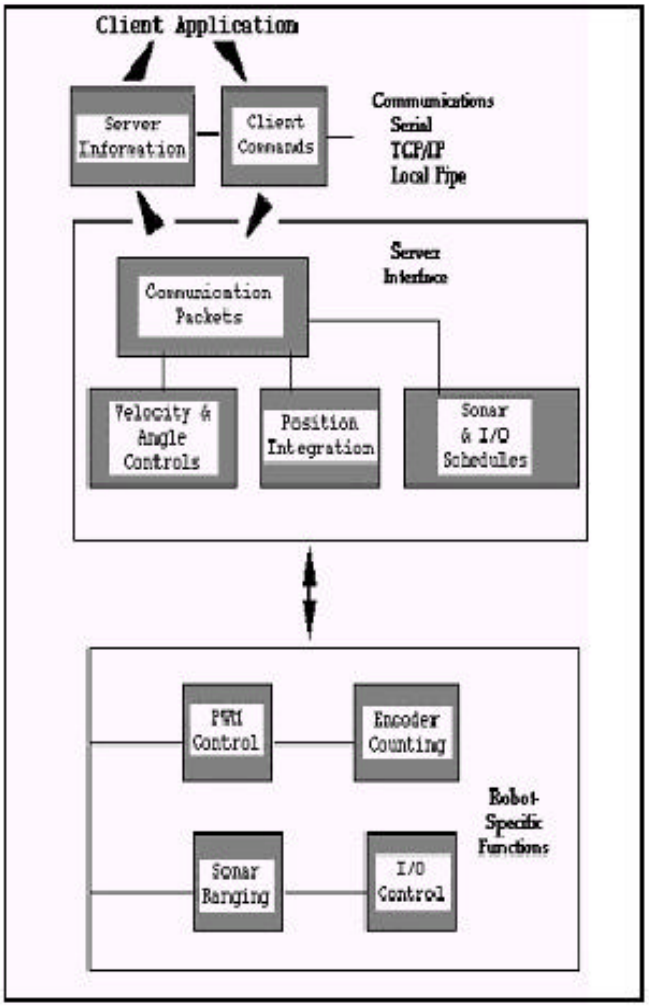

Figura 9.1: Arquitectura cliente - servidor

Ambos protocolos son un flujo de bits de un máximo de 208 bytes cada uno que consisten en cinco elementos:

- Una cabecera de dos bytes.
- Un contador de un byte de los siguientes paquetes.
- El comando del cliente o el SIP del servidor.

- Argumentos o bits de datos.
- Checksum (chequeo) de dos bytes.

AROS ignora comandos o paquetes cuyo byte contador exceda de 252 o presente un checksum erróneo. Sin embargo el interfaz cliente - servidor está provisto de métodos para reconocer paquetes defectuosos, puesto que muchos de los comandos alteran variables de estado en el servidor. Examinando el valor de estas variables se pueden detectar y reutilizar comandos defectuosos.

Al diseñar aplicaciones cliente se han de tener en cuenta las limitaciones de la comunicación serial. En términos generales, se enviará un comando o un paquete de datos entre cada tres y cinco milisegundos.

9.1.1. Paquetes de información del servidor AROS

Los paquetes que envía el servidor, SIP, informan al cliente sobre el estado del robot y ofrecen diversas lecturas sensoriales. Son los que se muestran en la tabla 9.1.

9.1.2. Comandos de un cliente

AROS posee un formato estructurado en comandos para recibir y responder a órdenes del cliente. Cada comando está determinado unívocamente por un número. Los comandos son los que muestra las tablas 9.2 y 9.3.

9.2. Conexión cliente - servidor AROS

Para ejecutar cualquier tipo de programa de control y pilotaje sobre el robot (proceso cliente) es preciso, como ya ha quedado claro, establecer una conexión con el servidor AROS a través de una interfaz serial. Después del establecimiento de la conexión el cliente enviará comandos y recibirá información del servidor.

Cuando se enciende el robot o se pulsa reset, AROS opera en un modo especial llamado de espera. En este estado AROS escucha la posible llegada de paquetes de comunicación para establecer una conexión cliente - servidor. Para establecer la misma, la aplicación cliente debe enviar una serie de tres paquetes de sincronización (los comandos SYNC0, SYNC1 y SYNC2) y recuperar las respuestas del servidor.

Después de recibir el paquete SYNC2, AROS enviará información sobre la configuración del robot, con lo que se concluye el proceso de conexión. A conti-

NAME	VALUE	DESCRIPTION
HEADER	int	Exactly 0xFA, 0xFB
BYTE COUNT	byte	Number of data bytes + 2 (checksum), not including header or byte-count bytes
STATUS/PACKET TYPE	0x3S =	Motors status
	2	Motors stopped
	3	Robot moving
XPOS	int	Wheel-encoder integrated coordinates; platform-dependent units; multiply by *DistConvFactor*‡ to convert to millimeters.
YPOS	int	
THPOS	sint†	Orientation in platform-dependent units—multiply by *AngleConvFactor*‡ for degrees.
L VEL	sint	Wheel velocities in mm/sec (*VelConvFactor*‡ = 1.0)
R VEL	sint	
BATTERY	byte	Battery charge in tenths of volts (101 = 10.1 volts, for example)
STALL AND BUMPERS	int	Motor stall and bumper indicators. Bit 0 is the left wheel stall indicator, set to 1 if stalled. Bits 1-7 correspond to the first bumper I/O digital input states (accessory dependent). Bit 8 is the right wheel stall, and bits 9-15 correspond the second bumper I/O states, also accessory and application dependent.
CONTROL	sint	Setpoint of the server's angular position servo—multiply by *AngleConvFactor*‡ for degrees
FLAGS	sint	Bit 0 motors status; bits 1-4 sonar array status; bits 5,6 M-STOP; bits 7,8 ledge-sense IRs; bit 9 joystick button; bit 10 auto—charger power-good.
COMPASS	byte	Electronic compass accessory heading in 2-degree units
SONAR COUNT	byte	Number of new sonar readings included in SIP
NUMBER	byte	If Sonar Count>0, is sonar disc number 0-31; reading follows
RANGE	int	Sonar range value; multiply by RangeConvFactor‡
... REST OF THE SONAR READINGS ...		
GRIP_STATE	byte	Gripper state byte.
ANPORT	byte	Selected analog port number 1-5
ANALOG	byte	User Analog input (0-255=0-5 VDC) reading on selected port
DIGIN	byte	Byte-encoded User I/O digital input
DIGOUT	byte	Byte-endcoded User I/O digital output
CHECKSUM	integer	Packet-integrity checksum

Cuadro 9.1: Standard Server Information Packet

COMMAND	#	ARGS	DESCRIPTION	AROS	P2OS	PSOS
			Before Client Connection			
SYNC0	0	none	Start connection. Send in sequence. AROS echoes synchronization commands back to client, and robot-specific autosynchronization after SYNC2.	1.0	1.0	3.x
SYNC1	1	none				
SYNC2	2	none				
			After Established Connection			
PULSE	0	none	Resets server watchdog	1.0	1.0	3.x
OPEN	1	none	Starts the AROS servers	1.0	1.0	3.x
CLOSE	2	none	Close servers and client connection	1.0	1.0	3.x
POLLING	3	string	Change sonar polling sequence (see text)	1.0	1.0	3.9
ENABLE	4	int	1=enable; 0=disable the motors	1.0	1.0	–
SETA	5	sint	Translational acceleration, if positive, or deceleration, if negative; mm/sec/sec	1.0	1.0	–
SETV	6	int	Sets maximum translational velocity; mm/sec	1.0	1.0	4.8
SETO	7	none	Resets local position to 0,0,0 origin	1.0	1.0	3.x
MOVE	8	sint	Translate (+) forward or (-) back mm distance	1.0	1.0	–
ROTATE	9	sint	Rotate (+) counter- or (-) clockwise degrees/sec	1.0	1.0	–
SETRV	10	int	Sets maximum rotational velocity; degrees/sec	1.0	1.0	4.8
VEL	11	sint	Translate at mm/sec forward (+) or backward (-)	1.0	1.0	3.x
HEAD	12	sint	Turn to absolute heading; ±degrees (+ = ccw)	1.0	1.0	4.2
DHEAD	13	sint	Turn relative to current heading; (+) counter- or (–) clockwise degrees	1.0	1.0	3.x
SAY	15	string	As many as 20 pairs of duration (20 ms increments) /tone (half-cycle) pairs	1.0	1.0	4.2
CONFIG	18	none	Request configuration SIP	1.0	1.4	–
ENCODER	19	int	Request one (1), a continuous stream (>1), or stop (0) encoder SIPs	1.0	1.4	–
RVEL	21	sint	Rotate at (+) counter- or (–) clockwise; degrees/sec	1.0	1.0	4.2
DCHEAD	22	sint	Heading setpoint relative to last setpoint; ± degrees (+ = ccw)	1.0		
SETRA	23	sint	Rotational (+)acceleration or (-)deceleration, in degrees/sec/sec	1.0	1.0	–
SONAR	28	int	1=enable, 0=disable all the sonar; otherwise, use bit 0 to enable (1) or disable (0) a particular array 1-4, as specified in argument bits 1-4.	1.0	1.0	–
STOP	29	none	Stops robot; motors remain enabled	1.0	1.0	–
DIGOUT	30	2 bytes	Bits 8-15 is a byte mask that selects the output port(s); Bits 0-7 set (1) or reset (0) the selected port(s).	1.7^{16}	1.2	4.2
VEL2	32	2 bytes	Independent wheel velocities; Bits 0-7 for right wheel, Bits 8-15 for left wheel; PSOS is in ±4mm/sec; AROS/P2OS in 20mm/sec increments	1.0	1.0	4.1
GRIPPER	33	int	Gripper server commands. See the Pioneer 2 Gripper or PeopleBot manual for details.	1.0	1.3	4.0
ADSEL	35	int	Selects the A/D port number for reporting Anport value in standard SIP.	1.0	1.2	–
GRIPPERVAL	36	int	Gripper server values. See Pioneer 2 Gripper or PeopleBot manual for details.	1.0		–
GRIPREQUEST	37	none	Request one (1), a continuous stream (>1), or stop (0) Gripper SIPs. See Pioneer 2 Gripper or PeopleBot manual for details.	1.0	1.E	–
IOREQUEST	40	none	Request one (1), a continuous stream (>1), or stop (0) IO SIPs	1.0	1.E	–
PTUPOS	41	2 bytes	Msbyte is port number (1-4), lsbyte is pulse width in 100µsec units PSOS or 10µsec units P2OS	–	1.2	4.5
TTY2	42	string	Sends string argument to serial device connected to AUX (AUX1 on H8S) port	1.0	1.0	4.2

Cuadro 9.2: Comandos del cliente I

TTY2	42	string	Sends string argument to serial device connected to AUX (AUX1 on H8S) port	1.0	1.0	4.2
GETAUX	43	int	Request to retrieve 1-200 bytes from the AUX (AUX1 on H8S) serial port; 0 flushes the buffer.	1.0	1.4	–
BUMP_STALL	44	int	Stall robot if front (1), rear (2) or either (3) bumps contacted. Off is 0. See BumpStall FLASH for default.	1.0	1.5	–
TCM2	45	int	TCM2 Module commands; see *TCM2 Manual* for details.	1.0	1.6	–
DOCK	46	int	Default is OFF; 1=enable docking signals; 2=enable docking signals and stop the robot when docking power sensed.	–	1.C	–
JOYDRIVE	47	int	Default is O=OFF; 1=allow joystick drive from hardware port while also connected with a client	1.0	1.G	–
SONAR_CYCLE	48	int	Change the sonar cycle time; arg in milliseconds	1.8	–	–
HOSTBAUD	50	int	Reset the HOST serial port baud rate to 0=9600, 1=19200, 2=38400, 3=57600, or 4=115200	1.8	–	–
AUX1BAUD	51	int	Resets the AUX1 serial port baud rate	1.8	–	–
AUX2BAUD	52	int	Resets the AUX2 serial port baud rate	1.8	–	–
E_STOP	55	none	Emergency stop, overrides deceleration	1.0	1.8	–
M_STALL	56	int	1 (default)=Motors stop button causes stall; 0 (P2OS default)=off	1.0	1.E	–
LEDGE	57	int	0 if inactive; 1 if stop when near-IRs triggered; 2 if impose speed control only; 3 if both stop and speed control	1.5	–	–
STEP	64	none	Single-step mode (simulator only)	1.0	1.0	3.x
TTY3	66	string	Sends string argument to serial device connected to AUX2 H8S serial port	1.0	–	–
GETAUX2	67	int	Request to retrieve 1-200 bytes from the AUX2 H8S serial port; 0 flushes the buffer.	1.1	–	–
CHARGE	68	int	1 to deploy autocharging mechanism; 0 to retract	1.7	–	–
ARM	70 - 81	int	Arm-related commands; see manual for details	1.3	–	–
ROTKP	82	int	Change working rotation Proportional PID value (not FLASH default)	1.1	1.M	–
ROTKV	83	int	Change working rotation Derivative PID value (not FLASH default)	1.1	1.M	–
ROTKI	84	int	Change working rotation Integral PID value (not FLASH default)	1.1	1.M	–
TRANSKP	85	int	Change working translation Proportional PID value (not FLASH default)	1.1	1.M	–
TRANSKV	86	int	Change working translation Derivative PID value (not FLASH default)	1.1	1.M	–
TRANSKI	87	int	Change working translation Integral PID value (not FLASH default)	1.1	1.M	–
REVCOUNT	88	int	Change working differential encoder count (not FLASH default)	1.1	1.M	
PLAYLIST	91	none	Request AmigoBot sounds playlist packet	1.0	1.E	–
SOUNDTOG	92	int	0=mute or 1=enable buzzer	1.0	–	–
SHUTDOWN	25 0	int	0=cancel shutdown; 1=simulate low-power condition; 2=initiate computer shutdown	1.6	–	–

Cuadro 9.3: Comandos del cliente II

nuación el cliente debe enviar el comando OPEN, que ordena al microprocesador Hitachi H8S la ejecución de funciones de mantenimiento y autochequeo y de inicio de diversos procesos servidores como el control de los sonar y los motores.

Además AROS incorpora un watchdog o programa de seguridad que espera que, una vez se ha conectado el cliente, se reciban al menos un paquete de comunicación cada cierto periodo de tiempo (este periodo es configurable).

Para cerrar la conexión se debe enviar el comando CLOSE.

Capítulo 10

ARIA

ARIA es un paquete software orientado a objetos programado en C++ que constituye una API (applications-programming interface) para el control de diversos robots móviles de Activmedia [1].

Como se dijo en el capítulo 9, la comunicación entre el microprocesador Hitachi H8S con cualquier máquina requiere de una arquitectura cliente - servidor. Hemos visto como los procesos servidores constituían el sistema operativo llamado AROS. Éstos se limitaban a la gestión de las tarjetas electrónicas y de las funciones de bajo nivel de la plataforma móvil.

Sin embargo en el lado del servidor AROS no se pueden llevar a cabo tareas inteligentes o de control. ARIA es el software del lado del cliente. Gracias a este paquete de clases nos podremos comunicar y controlar el robot desde aplicaciones cliente.

Mediante ARIA se podrán construir aplicaciones para el control de alto nivel del robot: desde la ejecución de simples comandos hasta la elaboración de comportamientos inteligentes (detectar y evitar obstáculos, reconocimiento de características de objetos, exploración, etc.).

ARIA está registrado bajo la GNU Public Licence, lo que significa que es un software de código abierto. Igualmente, todo el software desarrollado a partir de ARIA debe ser distribuido proporcionando el código fuente.

10.1. Comunicación con el robot

Una de las principales funciones de ARIA y el primer cometido de cualquier aplicación de control para el robot es establecer y gestionar la comunicación clien-

[1] Empresa fabricante del robot

te - servidor AROS entre los dispositivos del robot y la aplicación cliente.

ARIA suministra diversas clases para establecer esta conexión: ArDeviceConnection y sus hijos ArTcpConnection y ArSerialConnection permiten la configuración de los puertos del computador (puertos de serie o puertos IP) para la conexión a través de ellos con el robot (puerto de serie) o con un simulador del mismo (puertos IP).

Después de establecer la conexión, las funciones principales que se realizan son la lectura y escritura de datos de los dispositivos. Estas operaciones se realizan a través de las funciones ArDeviceConnection::write() y ArDeviceConnection::read(). Sin embargo el programador no tiene por qué utilizar estos métodos directamente, puesto que la clase ArRobot (sección 10.2) incorpora métodos que se apoyan en los anteriores que separan y clasifican adecuadamente los paquetes de información del servidor AROS.

10.2. ArRobot

ArRobot es la clase que constituye el corazón de ARIA. Actúa como pasarela de la comunicación cliente - servidor y gestiona la sincronización del sistema y la recolección y distribución de información referente al estado del mismo.

ArRobot gestiona los detalles de bajo nivel al construir y enviar comandos al robot y al recibir y decodificar los paquetes de información del servidor (SIPs).

Los SIPs (ver sección 9.1) son enviados desde el robot cada 100 ms a través de la conexión realizada. ArRobot provee el llamado SIP handler, lo que permite gestionar la recepción de estos paquetes de forma síncrona.

Desde la parte del cliente se pueden enviar directamente comandos o usar los diversos métodos y acciones de ARIA. Estas acciones y métodos se convierten posteriormente, de forma invisible al programador, en comandos directos.

Por otra parte, cada vez que una aplicación cliente basada en ARIA recibe un SIP ArRobot lleva a cabo una serie de cinco tareas automáticamente:

- Gestión del SIP.
- Interpretación de la información sensorial.
- Gestión y resolución de las acciones del cliente
- Examen del estado del sistema.
- Desempeño de las tareas programadas

Además el programador puede añadir más tareas a las anteriormente citadas.

10.3. Comandos y acciones

La aplicación cliente puede controlar el robot a través de comandos directos (ver capítulo 9), comandos de movimiento o a través de acciones.

Si se desea, es posible enviar comandos directos al robot a través de la clase ArRobot. Los comandos directos consisten en un byte, que constituye el número de comando a enviar. Puede estar seguido de uno o más argumentos. Las funciones que se utilizan para enviar comandos directos son: *ArRobot::com()*, *ArRobot::comInt()*, *ArRobot::comStr()*, *ArRobot::comStrN()*. Se utilizarán una u otra en función del número de argumentos del comando.

El nivel inmediatamente superior a los comandos directos son los comandos de movimiento. La clase ArRobot incorpora métodos para enviar comandos explícitos de movimiento al robot. Mediante estos métodos se puede, por ejemplo, establecer la velocidad lineal y de rotación del robot, provocar su movimiento a una distancia determinada o detenerlo.

Por último se puede controlar el robot desde un nivel de abstracción superior. Es posible definir acciones o usar las ya construidas en Aria. Las acciones serían comportamientos o pautas que debe seguir el robot. Se deben establecer prioridades entre las diversas acciones que se han añadido al robot, pues durante la ejecución del programa pueden entrar en conflicto. Es decir, dependiendo del estado del robot y de la prioridad de las acciones, se deberá ejecutar una u otra acción en cada momento. Las clases que permiten la definición de acciones y la resolución de prioridades se llaman ArAction y ArResolver.

Las acciones (alto nivel) y los comandos (bajo nivel) pueden entrar en conflicto y provocar errores en el control. Entre los objetivos del proyecto no está el control inteligente del robot. Sin embargo, en futuros desarrollos el programador deberá ser cuidadoso y tener en cuenta este hecho.

10.4. Otras características

Aria es un potente y robusto paquete compuesto por más de cien clases programadas en C++. Por tanto, además de las características enunciadas anteriormente, posee otras muchas. Existen clases para el control de dispositivos como cámaras, láser o pinzas, clases con utilidades matemáticas o para el manejo de tiempos, etc. Dados los objetivos del presente proyecto (ver capítulo 3) se destacan las siguientes clases:

- ArKeyHandler y ArJoyHandler. Permiten el control del robot a través del teclado o de un joystick respectivamente.

- ArThread. Es una clase envolvente o wrapper de la libreria pthreads. Permite la creación de subprocesos o threads dentro de una aplicación.
- ArFunctor. Es una clase que permite la creación de punteros a funciones.
- ArSocket. Es una clase envolvente o wrapper de las librerías para el manejo de sockets.

Capítulo 11

Sockets: Una solución al transporte de datos entre aplicaciones distruidas en red

Del análisis de requisitos y de la naturaleza propia del problema se ha concluido que el sistema software a desarrollar deberá ser distribuido y construido sobre una arquitectura cliente/servidor. Este hecho nos lleva a pensar en una implementación basada en sockets como método más sencillo para llegar a una solución que cumpla con los requisitos.

El desarrollo software basado en sockets constituye un mecanismo para el transporte de datos a través de redes. Esto es precisamente lo que deben hacer nuestras aplicaciones, transportar una serie de datos (comandos o datos sensoriales del robot)desde un punto de la red al punto que le sean requeridos.

Por otra parte, la programación con sockets bajo Unix (como es este caso) resulta extremadamente económica, puesto que el uso de las librerías necesarias para el desarrollo es gratuito, como la práctica totalidad del software en Linux.

A continuación se pasan a describir una serie de conceptos que servirán de base teórica al posterior desarrollo de las diversas aplicaciones.

11.1. Modelos de referencia Arpanet y OSI

El modelo de referencia utilizado en la comunicación con protocolos TCP/IP más aceptado es el empleado en las redes Arpanet, OSI y en Internet. En el modelo de redes Arpanet existen cuatro niveles diferentes, según se aprecia en la figura 11.1

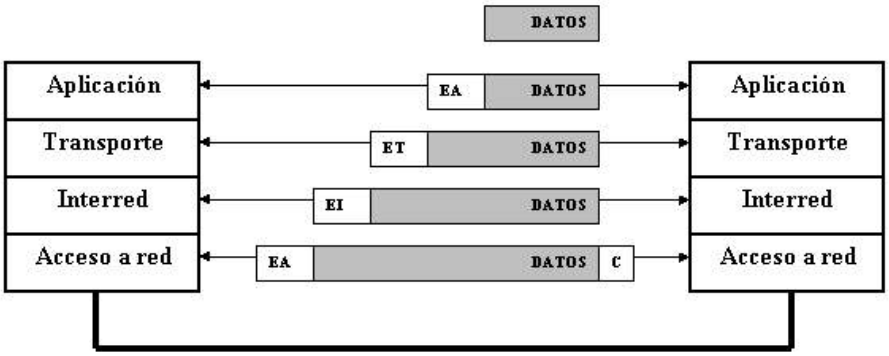

Figura 11.1: Modelo de capas Arpanet

- La capa de **acceso a red** aísla a los niveles superiores de las características del medio de transmisión empleado (Ethernet, Token Ring, líneas de serie, etc), y contiene los drivers específicos para cada medio en cuestión.
- La capa de **interred** es la encargada de crear los datagramas y aplicar un correcto encaminamiento para que estos lleguen a su destino y, en caso de que lleguen fragmentados, efectuar la reconstrucción de los datagramas originales.
- La capa de **transporte** se encarga de gestionar la entrega de datos entre las máquinas de origen y destino que intervienen en la comunicación.
- Por último, el nivel de **aplicación** contiene los procesos que utilizan la red para intercambiar datos. Desde este nivel es donde se escriben los programas y dónde se sitúan las labores principales de este proyecto.

En este esquema, los datos bajan para ser transmitidos por la red, y se les añaden cabeceras en cada nivel, que serán interpretadas por los niveles homólogos en la otra máquina. En el receptor los datos suben y se les van quitando las cabeceras.

En el modelo OSI, posterior al Arpanet, se distinguen más niveles para diferenciar más claramente y aislar los distintos problemas. Así se puede observar en la figura 11.2

Dentro del esquema OSI, el nivel aplicación del modelo Arpanet, se puede dividir en otros tres subniveles, desde los que se puede programar. La figura 11.3 muestra las equivalencias entre ambos modelos.

A nivel más bajo se programaría con sockets, apoyándonos en los servicios del nivel de transporte. En el siguiente nivel resolveríamos problemas que se

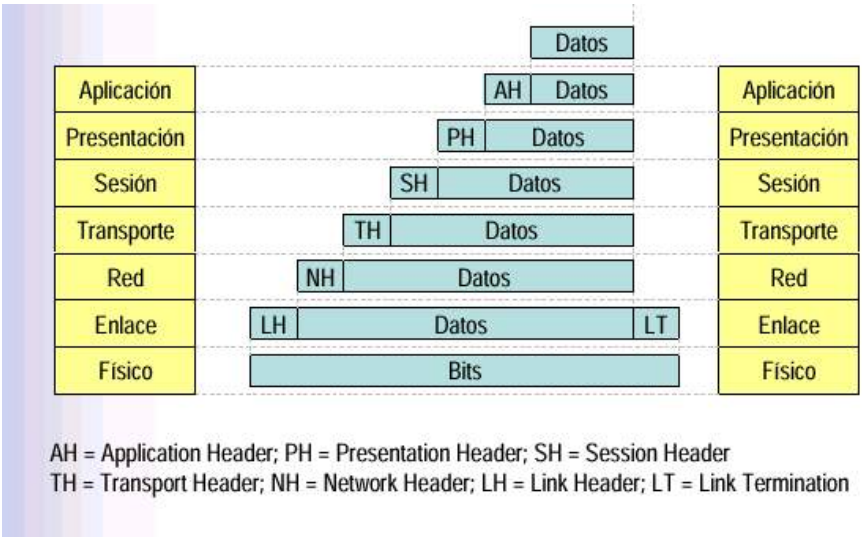

Figura 11.2: Modelo de capas OSI

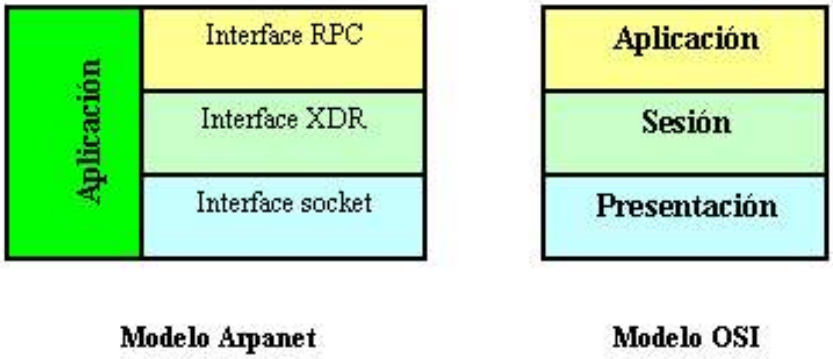

Figura 11.3: Niveles OSI y Arpanet

plantean según el modelo OSI en el nivel de presentación, a través de la programación con RPC, que se apoyaría sobre el nivel equivalente en OSI de sesión.

11.2. Aplicaciones cliente - servidor

Las aplicaciones en la Informática moderna tienden a ser distribuidas, dividiéndose como mínimo en dos partes y ejecutándose cada parte en un procesador distinto, como es el caso del sistema a desarrollar.

Una parte está en el proceso **cliente**, que emite peticiones; y otra en el proceso **servidor**, que escucha y atiende las peticiones. Como se ha dicho en repetidas ocasiones nuestro servidor correrá sobre la CPU de LIN y escuchará las peticiones del cliente, que a través de la red telecontrolará el robot.

La ventaja principal de esta arquitectura con respecto a otras es que cliente y servidor son procesos independientes, lo que implica:

- Transparencia en la localización de procesos. No es necesario saber en qué dirección, lugar o red se encuentra el proceso servidor. El proceso cliente puede estar en cualquier máquina de la red. En cuanto al servidor, también puede estar en cualquier sitio, aunque debemos saber el nombre o la dirección de la máquina en donde se esté ejecutando. Como se verá en los capítulos 13 y 14 la tecnología CORBA elimina estos requisitos.
- Transparencia en cuanto al tipo de máquina y sistema operativo. Los procesos clientes y servidores se entienden con independencia de cómo sea la máquina y el tipo de sistema operativo que tenga cargado.
- Flexibilidad para añadir más servidores y clientes. Nada impide que en se añadan a un sistema, por ejemplo, más PC's como clientes y estaciones de trabajo como servidoras.
- Fiabilidad. Podemos tener servidores redundantes y, en caso de que falle alguno de ellos, los demás entrarían en funcionamiento. Otra posibilidad es que el propio cliente, si el servidor no responde, mande peticiones a un segundo servidor, y si falla a un tercero, etc.

11.3. Asociación y sockets

Dado que entre un proceso cliente y un proceso servidor se pueden enviar datos por más de un canal , es necesario identificar los datos correspondientes a cada uno de los posibles canales. A las coordenadas que identifican de forma

unívoca a cada canal de comunicación entre dos procesos se le llama asociación (*bind*), y está compuesta por los cinco parámetro siguientes:

{protocolo, dirección local, proceso local, dirección remota, proceso remoto}

El término *protocolo* se refiere a un protocolo del nivel de transporte, la *dirección local* y *dirección remota* al número de la máquina local y remota, mientras que, por último, *proceso local* y *proceso remoto* identifican los procesos local y remoto, respectivamente.

Se define una media asociación como la terna:

{protocolo, dirección local, proceso local}

en la máquina local y como:

{protocolo, dirección remota, proceso remoto}

en la remota, conde sólo se especifican la mitad de las coordenadas de la conexión. Esta media asociación puede verse en una máquina como un punto final de comunicación, siendo acuñado el término **socket** (enchufe) en el UNIX de Berkeley. Un proceso en una máquina enviará o recibirá información a través de su socket, y lo mismo hará el de la otra máquina.

Estos parámetros para la red de Internet son, TCP o UDP como posibles valores del parámetro protocolo, la dirección de red IP como dirección local o dirección remota, y el número de puerto (entre 0 y 65355) para el proceso local y proceso remoto. En otras redes tendrá un significado similar.

No puede haber dos canales iguales entre dos procesos, ya que no se podría distinguir a qué canal se envía o reciben los datos, por lo que el sistema operativo no lo permite, generando un mensaje de error. Esto implica que las asociaciones deben ser únicas. El sistema operativo conoce las 5 coordenadas de cada asociación y entrega correctamente a cada socket los datos recibidos. También verifica que no se den dos asociaciones iguales.

De esta sección se concluye que un socket es un punto de comunicación por el cuál un proceso puede transmitir o recibir información.

11.4. Esquema de funcionamiento

En el UNIX de Bekeley el interfaz socket está pensado para trabajar de forma parecida a como se trabaja con un fichero. Así, por ejemplo, cuando se abre un

socket, se devuelve un descriptor (número entero) del mismo. Con este descriptor se pueden enviar o recibir datos de forma similar a como se leen o escriben datos de un fichero. Sin embargo, hay que destacar que existen grandes diferencias entre trabajar con un fichero y los procesos en red.

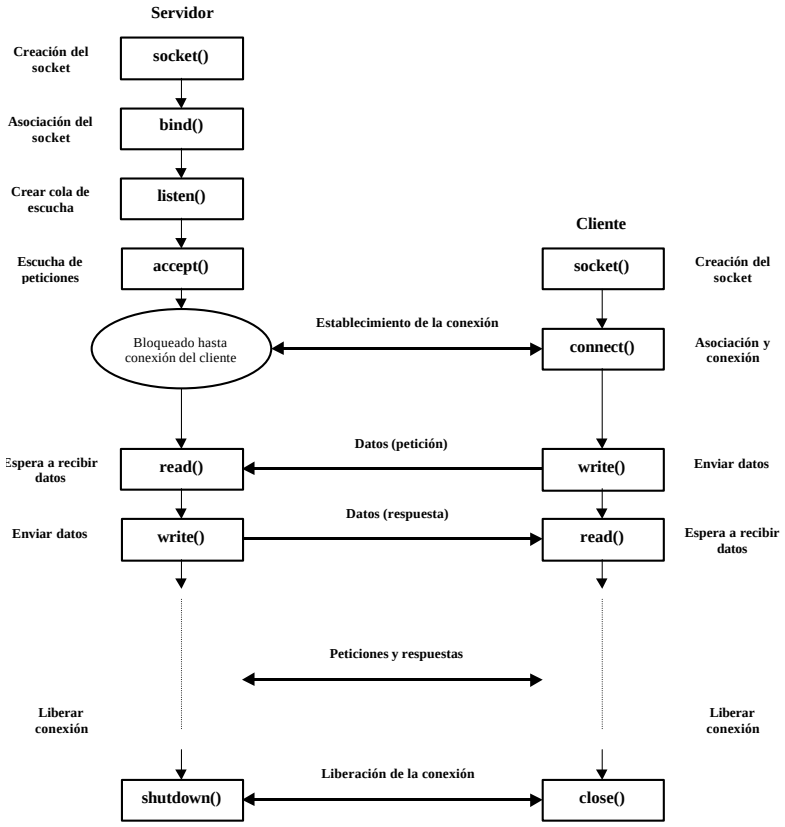

Figura 11.4: Esquema de funcionamiento de sockets

Existen dos tipos de servicios de transporte: orientado a conexión y no orientado a conexión. Un servicio orientado a conexión funciona de manera parecida a una llamada telefónica, con tres fases: establecimiento, uso y liberación de la conexión. Primero se necesita establecer la conexión antes de recibir o enviar datos.

Después establecida la conexión se intercambian datos de peticiones y respuestas y por último se libera la conexión. El servidor está esperando bloqueado a que el cliente haga una solicitud de conexión. Éste es el esquema que sigue el software desarrollado para establecer una comunicación entre el robot y la red local, pues el servicio orientado a conexión es mucho más fiable que el no orientado a conexión (donde por ejemplo, no existe información sobre la llegada del mensaje a su destino). En la figura 11.4se muestra un esquema de las llamadas cliente - servidor:

En el caso de servicios no orientados a conexión, al igual que ocurre, por ejemplo, con el sistema postal, no es necesario establecer ni liberar una conexión, simplemente se envían los datos, indicando dentro de la estructura de transmisión de los mismos la dirección de destino. El servidor, al recibir los datos, también recibe la dirección y la utiliza posteriormente para devolver la respuesta.

11.5. Dominio de un socket

El dominio de un socket indica bajo qué red se va a realizar la comunicación y, por tanto, qué tipo de direccionamiento se debe emplear. Muchas llamadas al sistema necesitan un puntero a una estructura de dirección socket que contiene las coordenadas de la asociación. La definición de dicha estructura está en el fichero <sys/socket.h>y es la siguiente:

```
struct sockaddr
{
u_short sa_family;   //Familia de la dirección, 2 bytes
char    sa_data[14]; //14 bytes de direccíón como max.
};
```

Esta estructura es genérica. Las llamadas que utilizan el direccionamiento de un socket deben utilizar esta estructura. Además, cada tipo de socket utiliza una estructura de direccionamiento que no coincide necesariamente con esta estructura genérica, lo que obliga a hacer una conversión de puntero de la estructura específica al tipo de la estructura genérica, en la propia llamada a la función.

Dominio UNIX (AF_UNIX)

Los sockets bajo este dominio comunican dos procesos dentro de una misma máquina UNIX. Están pensados para probar aplicaciones cliente - servidor sin tener que pasar por la red. Este dominio no se utiliza habitualmente, ya que con este mismo propósito está el interfaz de "loopback", cuya dirección internet es la de 127.0.0.1. Todas las máquinas tienen esta intefaz, que podría calificarse de virtual (ya que físicamente no existe, pero actúa como si lo hubiese). De este modo,

aunque la máquina no esté conectada a una red con protocolos TCP/IP, puede considerarse que está conectada a la red 127.0.0.0 a través del citado interfaz de dirección 127.0.0.1. La estructura de la dirección, definida en <sys/un.h>es:

```
struct sockaddr_un
{
short sun_family;     //Debe valer AF_UNIX
char  sun_path[108];  //Refencia de dirección
};
```

Dominio Internet (AF_INET)

En este dominio se utilizan protocolos TCP/IP, con su correspondiente direccionamiento. Las estructuras utilizadas, definidas en <netinet/in.h>son las siguientes:

```
struct in_addr
{
u_long s_addr;     //Direccion IP, en formato de red
};



struct sockaddr_in
{
short sin_family;       //Debe valer AF_INET
u_short sin_port;       //Número de puerto, 2 bytes
struct in_addrsin_addr; //Dirección Internet, 4 bytes
char sin_zerio[8];
};
```

En esta estructura especificamos el dominio (AF_INET), el puerto IP y la dirección IP, a través de la estructura *in_addr* definida anteriormente.

Las estructura para los dominios UNIX e Internet varían de longitud, por lo que en las llamadas a las funciones para la creación de sockets debemos pasar, además de un puntero, otro argumento con la longitud de las mismas. Las funciones están definidas con la estructura genérica *sockaddr*, por lo que al hacer las llamadas es preciso realizar una conversión, del tipo del puntero de la estructura específica a la citada estructura genérica. Por ejemplo para usar la llamada a *bind()* para el dominio Internet:

```
//Estructura específica para Internet
```

```
struct sockaddr_in cliente;

bind(sockfd, (struct sockaddr*)&cliente, sizeof(cliente));
```

11.6. Creación de un socket

En este apartado se verán las llamadas principales para la creación de un socket, se usa la llamada al sistema *socket()*:

```
#include <sys/types.h>
#include <sys/socket.h>

int socket(dominio,tipo, protocolo);

int dominio;     //AF_INET, AF_UNIX,...
int tipo;  //SOCK_DGRAM, SOCK_STREAM, ...
int protocolo;   //0: Protocolo por defecto
```

En caso de fallo se devuelve -1. Si todo va bien, devuelve un valor entero similar al de un descriptor de fichero. Por esto se le suele llamar *sockfd* al descriptor.

Dominio es una constante entera que designa el tipo de dominio bajo el que se hará la comunicación. El prefijo AF_ identificará la familia o tipo de direccionamiento.

En cuanto al tipo de socket podemos tener los siguientes:

SOCK_DGRAM. Están destinados a la comunicación con protocolos no orientados a conexión. No se obtiene información sobre la llegada del mensaje a su destino, es decir, la transmisión puede no ser fiable. Este tipo permite la difusión de un mensaje a más de un proceso. En el dominio Internet, el protocolo equivalente a SOCK_DGRAM es el UDP.

SOCK_STREAM. Este tipo de sockets permiten comunicaciones bajo protocolos orientados a conexión, con las siguientes características:

- Fiabilidad de la transmisión: Ningún dato se pierde.
- Conservación del orden de los datos: Los datos llegan en el mismo orden en el que fueron emitidos.
- No duplicación de datos: Sólo llega al destino un ejemplar de cada dato emitido.

- Se establece una conexión entre los dos puntos antes de la conexión. Después, en los datos que se envian no hay que especificar direcciones.
- Envío de mensajes urgentes. Existe la posibilidad de enviar datos fuera del flujo normal y por ello accesibles de forma inmediata. Se habla entonces de datos OOB o fuera de banda (out of band). Para el dominio Internet el protocolo utilizado con SOCK_STREAM es el TCP.

SOCK_RAW. Permite el acceso a los protocolos de más bajo nivel (en caso de utilizar líneas de serie).

Nuestras aplicaciones usarán el tipo de socket SOCK_STREAM, pues en ellas primará más que la velocidad, la fiabilidad de los datos. Recordemos que el objetivo de la comunicación entre el robot y las máquinas de la red local es hacer telecontrolable el mismo. Errores en el envío y recepción de los datos sensoriales pueden acarrear errores graves en el control, lo que puede llevar incluso a la colisión del robot con los obstáculos de su entorno.

La llamada a *socket()*, a partir de los valores de *dominio* y *tipo* de socket, permite deducir el protocolo a utilizar. Esta es la primera de las cinco coordenadas de una asociación, el resto serán especificadas después al realizar las distintas llamadas, y en función del tipo de protocolo.

Después de la llamada a *socket()*, lo único que tenemos es el descriptor del canal por donde se intercambiarán los datos. De esta manera, para enviar y recibir datos sólo es preciso hacer mención al descriptor del socket empleado, y no al puerto por el que se cursan.

11.7. Enlazamiento de un socket

Cuando un socket se crea con la llamada al sistema *socket()*, se crea sin ser asignada a un puerto. La primitiva *bind()* permite asociar la dirección IP y un puerto a un descriptor socket.

```
#include <sys/types.h>
#include <sys/socket.h>

int bind(desc, p_direccion, long);

int desc;                       //Descriptor del socket
struct sockaddr *p_direccion; //Puntero a la dirección
int long;                       //Longitud de la dirección
```

Si se produce un error devuelve -1; mientras que en caso de éxito devuelve 0. La llamada a *bind()* sirve para que un servidor registre su puerto antes de empezar a atender solicitudes. *bind()* informa al sistema operativo que ese puerto pertenece a dicho servidor, y todos los datos que se reciban o envíen serán para él.

El segundo parámetro, *p_addr*, es la dirección de memoria donde se encuentra la estructura *sockaddr_in* que primero será inicializada a 0, y después con los valores convenientes antes de llamar a *bind()*.

La función *bind()* proporciona dos elementos de la asociación: la dirección IP y el puerto locales. Para la mayoría de las máquinas, que actúan con un solo interfaz de red activo, es decir con una única dirección IP, basta con especificar dicha dirección y el puerto. Éste es el caso de las máquinas del laboratorio ASLab. Sin embargo existen máquinas que actúan como pasarela o gateway y disponen de más de un interfaz de red, por lo que si queremos que atiendan un servicio, venga de donde venga, no podemos llamar varias veces a *bind()* para enlazar las distintas direcciones IP de cada interfaz. En este caso deberemos utilizar la constante INADDR_ANY, con lo que indicamos al sistema que acepte peticiones que vengan por cualquier interfaz. Esta constante también se puede utilizar aunque la máquina tenga un sólo interfaz.

En cuanto al puerto, si damos el valor 0, indicamos al sistema operativo que escoja el puerto. El número de puerto será uno comprendido en el rango reservado de 1024 a 5000.

Después de la ejecución de *bind()* está casi formado el canal de comunicación.

11.8. Supresión de un socket

La llamada as sistema UNIX normal, *close()* cierra un socket. En protocolos orientados a conexión, el sistema devuelve el control inmediatamente, pero el kernel continúa enviando datos que estén todavía en cola.

```
close(desc);
int desc;     //File descriptor
```

11.9. Espera de conexión al servidor

Las solicitudes de conexión deben ser encoladas antes de ser atendidas. Con la llamada *listen()* creamos una cola en donde esperarán las peticiones de conexión que recibamos en el servidor, hasta que sean atendidas con la llamada a *accept()*. La cola escuchará peticiones de un socket local que ha de ser creado y enlazado previamente, con las llamadas *socket()* y *bind()*.

La llamada para crear la cola de peticiones tiene las siguientes características:

```
#include <sys/tipes.h>
#include <sys/socket.h>


int listen(desc,nb);
int desc;           //File descriptor
int nb;             //Número máximo de peticiones pendientes
```

donde *nb* es el tamaño de la cola, mayor que 0 y que normalmente vale 5, que es el máximo permitido. Esta llamada es válida sólo para SOCK_STREAM. Devuelve 0 si todo fue bien y 1 si hay algún error.

11.10. Aceptación de conexión en el servidor

Es utilizado con SOCK_STREAM en el servidor, tomando la primera petición de la cola, y creando un socket nuevo para la nueva conexión.

```
#include <sys/tipes.h>
#include <sys/socket.h>


int accept(desc, p_adr, p_lgadr);
int desc;              //File descriptor
struct sockaddr *p_adr;//Dirección socket conectado
int *p_lgadr;          //Puntero al tamaño de la direccion
```

Extrae la primera conexión pendiente de la cola creada en el descriptor indicado, desc. En caso de no existir alguna conexión en la cola, se bloquea hasta que aparezca una.

La función *accept()* crea un nuevo socket cuyo valor de descriptor es el entero devuelto por la llamada. Este socket está conectado con un cliente cuya dirección dejará el sistema operativo en la estructura apuntada por *p_addr*.

Por este nuevo socket se pueden enviar y recibir datos, mientras que el primer socket sigue escuchando nuevas peticiones y enviándolas a la cola.

De las 5 coordenadas de una asociación, el socket inicial proporciona las 3 primeras, que serán utilizadas para establecer una conexión en el socket nuevo. Al ejecutar *accept()* junto a las 3 primeras tuplas del servidor tomamos la dirección

IP y puerto del cliente. De esta forma podemos crear tantas conexiones como deseemos a partir del socket inicial, ya que las 2 tuplas de los clientes serán siempre distintas.

La función *accept()* presenta el inconveniente de que no puede rechazar una conexión de un cliente no deseado. Lo que debe hacer el proceso que recibe la conexión es aceptarla inicialmente, analizar su procedencia y, si no es un cliente deseado cerrar la conexión inmediatamente.

11.11. Establecimiento de la conexión en el cliente

En un cliente orientado a conexión sólo es necesario crear un socket, mediante la llamada a *socket()* antes de la fase de establecimiento de la conexión.

La llamada a *connect()* conecta al cliente con el servidor, paso necesario antes de enviar o recibir datos.

```
#include <sys/tipes.h>
#include <sys/socket.h>


int connect(desc, p_adr, plgadr);
int desc;              //File descriptor
struct sockaddr *p_adr;//Dirección socket remoto
int p_lgadr;           //Longitud de la dirección remota
```

En el proceso de establecimiento de la conexión cliente y servidor se intercambiarán los parámetros que regirán la conexión. Ambos sockets deben ser del mismo tipo y familia. La llamada retorna cuando se establece la conexión o en caso de error.

En el proceso de establecimiento de la conexión se conocen las 4 coordenadas restantes de la asociación. Por tanto, no es necesario utilizar la llamada *bind()*.

En las llamadas *connect()* y *bind()* no se especifica el dominio de comunicación, ya que este se encuentra en los dos primeros bytes de la estructura de dirección, cuyo valor es siempre de la forma AF_XXX, y también debido a que el primer argumento de dichas llamadas es el descriptor de un socket, que ha sido creado previamente con un dominio determinado.

11.12. Envío y recepción de datos

Una vez se tiene realizada la conexión entre el cliente y el servidor los dos procesos podrán intercambiar información.

Para envir disponemos de las llamada genéricas *write()* y *send()*:

```
int write(desc, msg, lg);
int desc;          //File descriptor
char *msg;         //Dirección del mensaje a enviar
int lg;            //Longitud del mensaje

int send(desc, msg, lg, opcion);
int desc;          //File descriptor
char *msg;         //Dirección del mensaje a enviar
int lg;            //Longitud del mensaje
int opcion;        //0,MSG_OOB, MSG_PEEK
```

Ambas llamadas se diferecian en que con *send()* se pueden utilizar las propiedades de los protocolos de comunicación mediante el parámetro opcion.

Para la recepción podemos utilizar las llamadas read() y recv(). Estas funciones devuelven el número de bytes recibidos si todo fue bien y -1 si hubo algún error.

```
int read(desc, msg, lg);
int desc;          //File descriptor
char *msg;         //Dirección del mensaje a enviar
int lg;            //Longitud del mensaje

int recv(desc, msg, lg, opcion);
int desc;          //File descriptor
char *msg;         //Dirección del mensaje a enviar
int lg;            //Longitud del mensaje
int opcion;        //0,MSG_OOB, MSG_PEEK
```

11.13. Limitaciones de los sockets

El problema más grave que tiene que ver con la diferencia entre las máquinas que se conectan a través de red, más concretamente con las distintas representaciones internas de datos, que pueden variar en lo siguientes aspectos:

- Los códigos de caracteres. Por ejemplo, IBM utiliza el EBCDIC, mientras que el resto utiliza notación ASCII.
- La aritmética empleada. Cyber usa C-1, el resto C-2. Así por ejemplo, una misma codificación hexadecimal puede ser interpretada como valores numéricos distintos.
- La forma de almacenar los bytes en memoria. Hay dos posibilidades: máquinas de punta delgada (little endian) como los procesadores de INTEL y DEC, que almacenan los bytes según la regla menor peso del número menor dirección. La otra posibilidad son las máquinas de punta gruesa (big endian) como los grandes ordenadores de IBM y los microprocesadores de la familia 6800, en este caso la regla es la contrario, menor peso mayor dirección.
- El tamaño de la palabra de datos: por ejemplo de la misma compañía el DEC-20 de 36 bits, y el DEC Alpha de 64 bits.

Capítulo 12

Desarrollo basado en sockets

En el presente capítulo se describirá cómo ha sido implementada la primera solución para el desarrollo software del sistema. Como ya se argumentó en el capítulo anterior, la solución más inmediata y económica para la comunicación entre dos máquinas a través de una red de área local es áquella basada en sockets, si bien somos conscientes de sus limitaciones (ver sección 11.13).

El sistema a desarrollar consta de dos partes que se comunican entre sí: cliente y servidor. El servidor enviaría información al cliente y recibiría comandos de éste. Entre los objetivos principales del proyecto (ver capítulo 3) no se encuentra el desarrollo de un cliente específico. Sin embargo, se ha pensado que un buen ejemplo es la implementación de un cliente que teleopere el movimiento del robot.

Por otra parte, aunque el cometido principal de las aplicaciones a desarrollar son el transporte de los datos sensoriales del robot y de los comandos del cliente a través de la red, este no es el único problema que se debe resolver ni la única función del sistema software. Nos debemos enfrentar, además, a los siguientes problemas:

- Cómo será la conexión con el robot.
- Cómo se adquirirán los datos sensoriales del robot.
- Cómo introducirá el usuario cliente los comandos a ejecutar en el robot.
- Qué información se debe enviar y cada cuanto tiempo.
- Cómo se protegerá el robot ante posibles impactos.
- Cómo se protegerá el robot ante fallos en la conexión.

De ello será descrito en detalle posteriormente. Nos ocupamos ahora de explicar cómo es la arquitectura física del sistema.

12.1. Arquitectura física del sistema

El sistema software consta de tres partes: AROS (ActivMedia Robotics Operating System), servidor y cliente.

Es el sistema operativo **AROS** corre en el microprocesador HITACHI H8S del robot móvil. Como ya se dijo en el capítulo 9 es un software cuyo cometido es controlar la electrónica del robot y llevar a cabo las de funciones de bajo nivel

Se comunica con la CPU de LIN (la placa GENE-6330) mediante un interfaz serial RS-232. Los detalles del protocolo de comunicaciones entre el microprocesador del robot y el computador de a bordo se pueden consultar en el capítulo 9.

Se aclara de nuevo que AROS haría por tanto las veces de servidor de nuestro servidor (que se comportaría con respecto a AROS como cliente).

El **servidor** es una aplicación que correrá en la CPU de nuestra plataforma móvil, que recordemos se trata del microprocesador de bajo consumo Transmeta Crusoe TM5400 (ver sección 5.2). Se apoyará sobre ARIA (ver capítulo 10) para comunicarse con el microprocesador de LIN. La comunicación se hará efectiva, como hemos dicho, a través de un puerto de serie. En líneas generales, nuestro servidor requerirá los datos sensoriales al robot (rango de los sonar y medidas de los encoders en forma de velocidad) y le enviará comandos para establecer la velocidad lineal y de rotación del mismo.

Por otra parte el servidor también se comunicará con el o los clientes que correrán en otros PCs de la red local. Con la instalación de la tarjeta wireless (ver sección 6.4) la plataforma móvil se convierte en un terminal más de la red ethernet.

La implementación de dicha comunicación se realizará a través de sockets de flujo SOCK_STREAM (ver capítulo 11). El protocolo de comunicaciones será TCP/IP. Como se dijo en el capítulo anterior, este protocolo asegura la no pérdida de los datos y la llegada de los mismos en el orden enviado.

El servidor enviará al cliente cualquier tipo de dato que le sea requerido cuando le sea requerido. Asimismo recibirá los comandos pertinentes de teleoperación desde el cliente y los transmitirá al microcontrolador HITACHI para su ejecución. Se puede deducir de estas palabras que nuestro servidor actúa de intermediario entre el microprocesador de LIN y las aplicaciones cliente. Se limita a recoger datos del primero y enviarlos al segundo y viceversa.

EL **cliente** será un programa que correrá en cualquier terminal de la red local de ASLab y requerirá información y enviará comandos al servidor. El cometido específico del cliente desarrollado será la teleoperación del robot. Por lo tanto será el usuario de la aplicación cliente quien controle el movimiento de LIN. Dicho

usuario se servirá del teclado para realizar dicho control. Todas las máquinas de la red local están dotadas de microprocesadores Intel con arquitecturas "i386".

La figura 12.1 muestra gráficamente la arquitectura física del sistema. En ella se pueden observar los distintos microprocesadores y dispositivos que intervienen en el sistema, así como los procesos que corren sobre ellos.

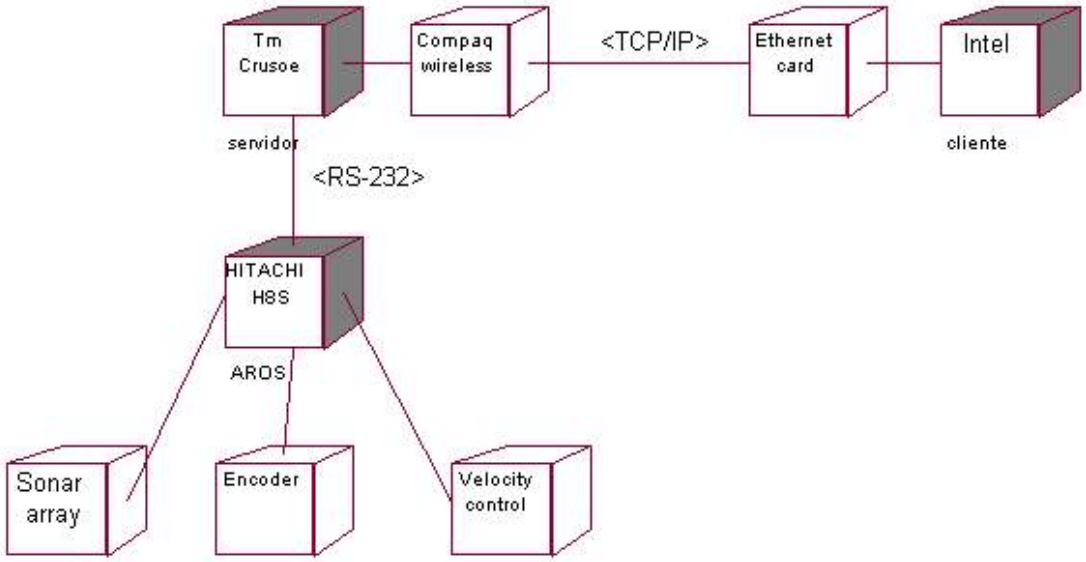

Figura 12.1: Diagrama de despliegue

12.2. Desarrollo de la aplicación servidor

En esta sección se describirá paso a paso el proceso de desarrollo del proceso servidor. Se recuerda que las funciones que debe realizar dicha aplicación son las siguientes:

- Conexión local con el robot.
- Obtención de la información de los sensores.
- Transmisión de toda la información sensorial que proporcionan los sensores desde LIN a las aplicaciones cliente. La información deberá estar disponible en cualquier punto de la red y podrá ser requerida en cualquier momento.
- Protección de LIN ante eventuales colisiones contra obstáculos.

12.2.1. Arquitectura lógica de la aplicación

En la presente sección se ofrecerá una descripción de la organización en clases de la aplicación servidor y del cometido de cada una. Con ello se pretende que el lector tenga una visión general del esquema de la aplicación para la posterior compresión de sus partes específicas.

A continuación se muestra, esquemáticamente, el diagrama UML de clases:

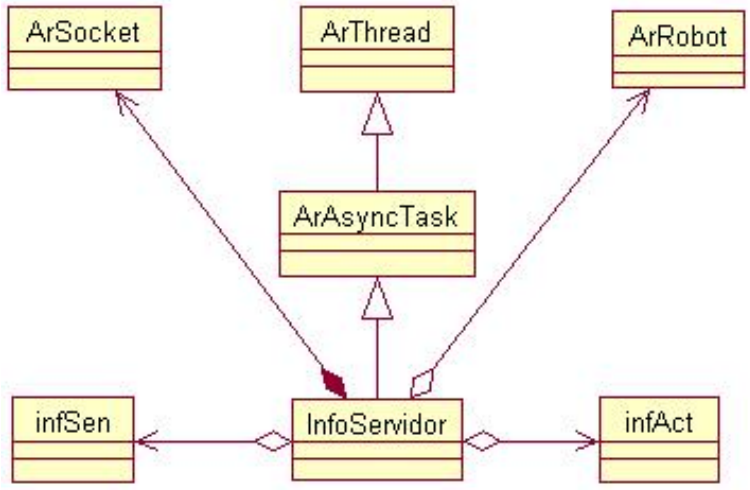

Figura 12.2: Diagrama UML servidor

Como se puede observar, siete son las clases que se usarán para el desarrollo:

- ArRobot
- ArSocket
- ArThread
- ArAsyncTask
- InfoServidor
- infAct
- infSen

Las clases ArRobot, ArSocket, ArThread y ArAsyncTask pertenecen al paquete ARIA (ver capítulo 10). InfoServidor será la clase principal de la aplicación y en ella están presentes (implementados, heredados o instanciados de otras clases) todos los métodos para la recolección de datos y comandos y transmisión

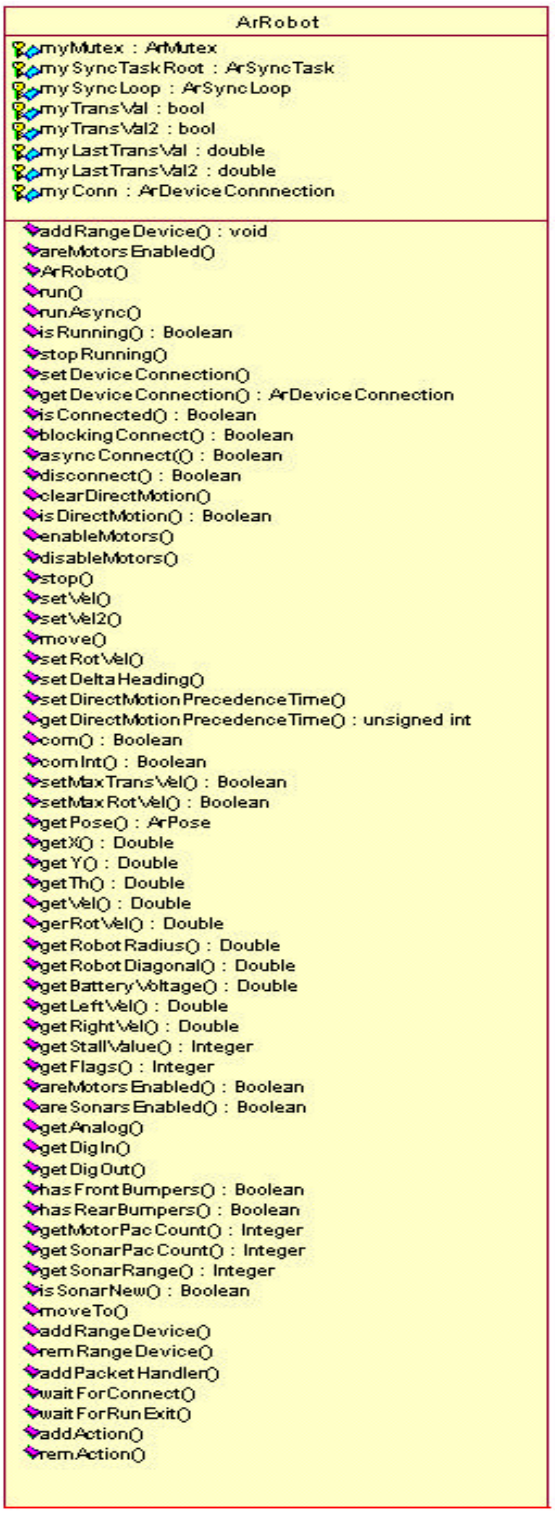

Figura 12.3: Diagrama UML ArRobot

de los mismos. Las clases infAct e infSen contienen la información transmitida o recibida de la aplicación cliente.

ArRobot actúa como pasarela de la comunicación con el microprocesador del robot. Se utilizarán sus métodos para la conexión con LIN, para la lectura del rango de los sonar, del valor de las velocidades lineal y de rotación y del resto de información sensorial que proporcione la interfaz eléctrica del robot. Así mismo, también se utilizarán sus métodos para establecer en el robot las velocidades enviadas desde la aplicación cliente. Para ello, en la clase InfoServidor existirá una instancia de la clase ArRobot. Gracias a esto hecho, cualquier objeto de la clase InfoServidor podrá intercambiar información son el robot. En la figura 12.3 se pueden observar sus métodos y atributos.

ArSocket es una clase envolvente o wrapper de las funciones del sistema para la utilización de sockets en una aplicación (ver capítulo 11). En la clase InfoServidor existen, como indica la figura 12.2, instancias de la clase ArSocket. Esto permitirá a los objetos de la clase InfoServidor comunicarse mediante sockets

Figura 12.4: Diagrama UML ArSocket

con la aplicación cliente. Los métodos y atributos de la clase ArSocket se pueden observar en la figura 12.4

Recapitulando, hemos conseguido de un modo sencillo, reunir en una sola clase (InfoServidor) métodos capaces de recoger y enviar información al robot y transmitir o recibir esta misma información a la aplicación cliente.

ArThread y **ArAsyncTask** son clases envolventes (wrapper) de la librería pthreads. Haciendo heredar InfoServidor de estas clases se podrán lanzar threads (subprocesos) desde objetos de la misma (ver sección 12.2.3). El diagrama UML de estas dos clases se puede observar en la figura 12.5

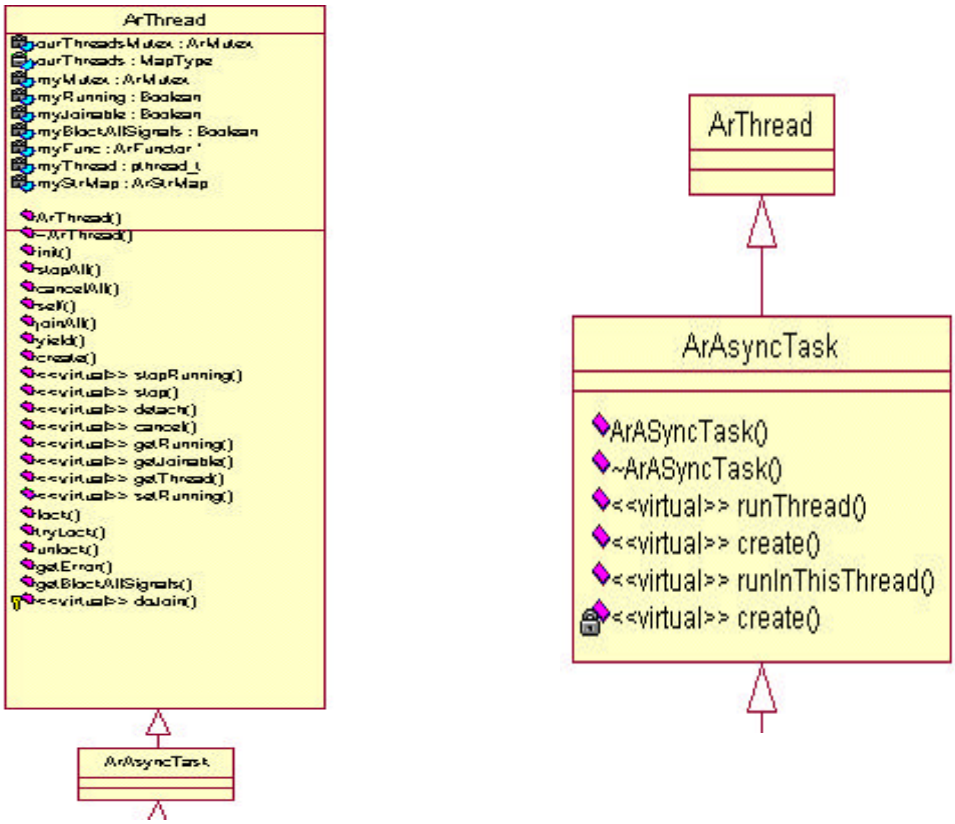

Figura 12.5: Diagrama UML de ArThread y ArAsyncTasK

Las clases **infAct** e **infSen** poseen atributos para el almacenamiento de los distintos datos a transmitir o recibir del cliente. En la clase InfoServidor existirán instancias de ambas clases. Sus atributos pueden ser observados en la figura 12.6.

Por último, **InfoServidor** es la clase que tiene como atributos instancias del resto de clases enumeradas anteriormente. Por lo tanto, a través de un objeto de esta clase se realizarán lecturas y se enviarán comandos al robot, se enviará y

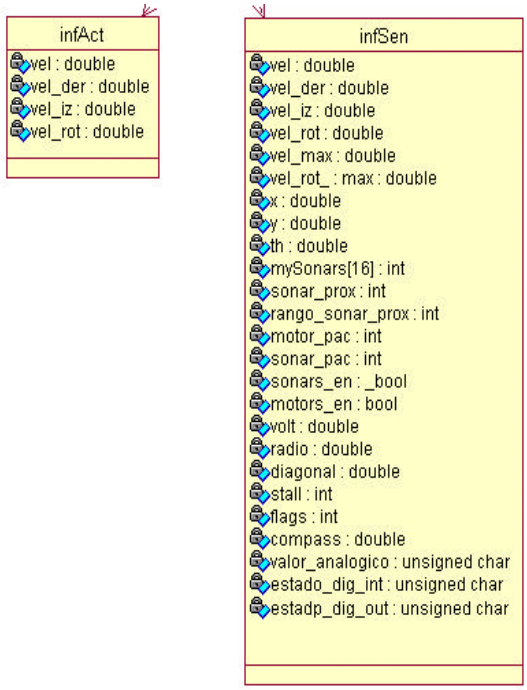

Figura 12.6: Diagrama UML de infAct e infSen

se recibirá información del cliente a través de sockets. Por otra parte, debido a que InfoServidor hereda de ArThread también gestionará cómo se realiza el intercambio de información con el cliente (mediante un thread). Su diagrama UML se puede observar en la figura 12.7.

En la figura 12.8 se puede observar el diagrama de flujo de la aplicación. Como se puede ver, después de la sección de declaraciones de los diversos objetos y variables necesarios, el programa intenta la conexión con el simulador o con el robot (ver sección 12.2.2).

De tener éxito, se procede a la activación de los motores y de los sonar. Tras esto, se abre el puerto a través del cual se intercambiará información con los clientes (ver sección 12.2.4). Una vez establecido el puerto de conexión, se procede a lanzar el thread que atenderá las solicitudes de los clientes y gestionará sus peticiones (ver sección 12.2.3).

El thread lanzado, en principio, permanecerá a la escucha de las posibles conexiones de los clientes. De este modo, el programa principal puede ser usado al

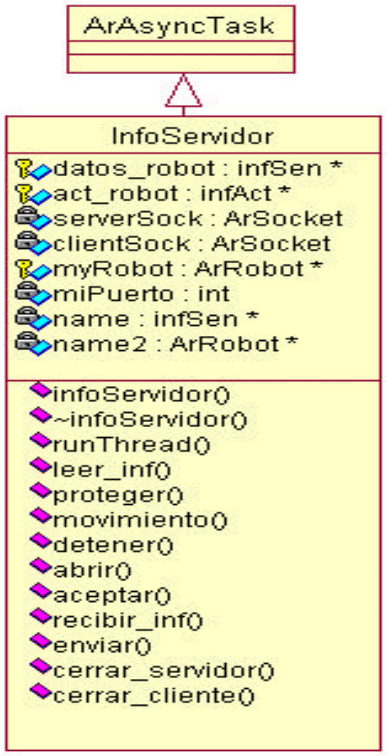

Figura 12.7: Diagrama UML de InfoServidor

mismo tiempo para ejecutar otras tareas. En el momento en que se produce una petición el servidor lee los datos sensoriales más recientes y los envía al cliente que los solicitó. Acto seguido recoge información referente a la velocidad que desea establecer la aplicación cliente en el robot. Se analiza si es posible que el robot asuma estas velocidades (se estudian entre otra cosa los posibles impactos) y se transmiten al robot (o simulador).

Después se cierra la conexión con el cliente y si los procesos anteriores fueron bien, vuelve a comenzar el ciclo. Si en estos procesos se detectaron fallos en la comunicación o bien si la variable de estado del thread (*myRunning*) así lo indica, inmediatamente se suspende dicho thread y se lanza un nuevo.

12.2.2. Conexión con el robot o con el simulador

El primer cometido de la aplicación servidor es establecer una conexión con el robot o simulador.

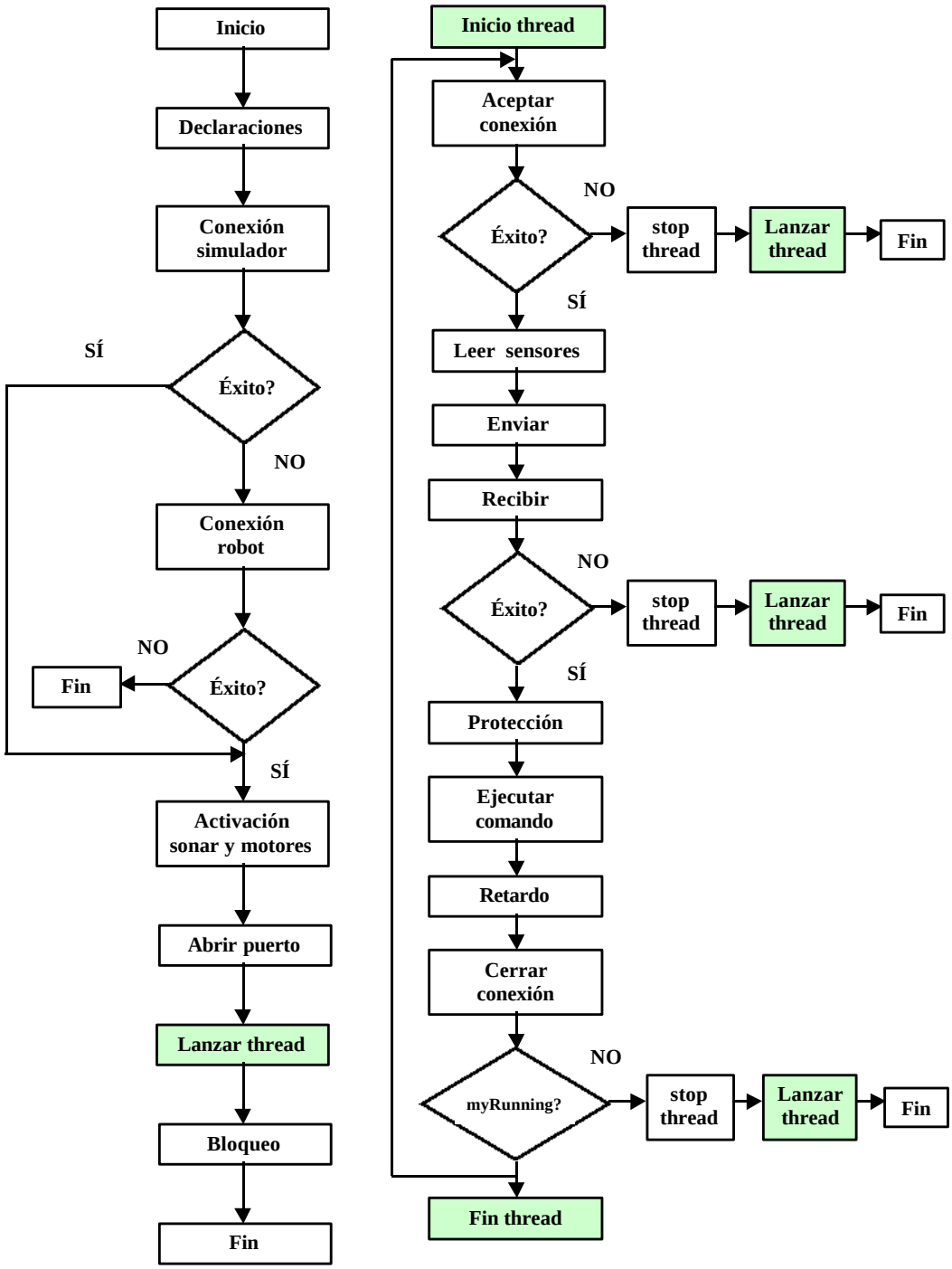

Figura 12.8: Diagrama de flujo

La conexión con el robot se realizará a través del puerto de serie COM1. Este dispositivo se corresponde con el fichero /dev/ttyS0 del árbol de directorios del sistema operativo. Se deberá configurar este puerto para la transmisión de datos (lectura y escritura) a través de él. Linux realiza estas operaciones tratando al puerto como si fuera un fichero.

Para desarrollar estas operaciones ARIA viene en socorro del programador, y dichas tareas se realizan de forma invisible, con la simple declaración de objetos ARIA y la invocación de alguno de sus métodos. En concreto se deben declarar sendos objetos de las clases ArRobot y ArSerialConnection. Con las siguientes líneas de código quedarían configurados los parámetros para establecer una comunicación a través del puerto de serie COM1:

```
#include <Aria.h>

ArRobot robot;
ArSerialConnection serConn;

//Configura puerto
serConn.setPort;
//Asocia el puerto al robot
robot.setDeviceConnection(&serConn);
```

De este modo, ya tenemos listo el puerto de serie para la comunicación. La velocidad en la transmisión de datos queda configurada a 9.6 Kbaud.

Si no queremos comunicarnos directamente con el robot, sino con el simulador del mismo que proporciona el fabricante lo deberemos hacer a través del protocolo TCP y no a través del puerto de serie. El simulador haría las veces de un servidor que deja disponible el puerto 8101 para el intercambio de comunicación. Nuestra aplicación debería conectarse a este puerto mediante las funciones disponibles en <socket.h>. Sin embargo de nuevo ARIA nos proporciona métodos que facilitan la programación. Análogamente a la conexión a través de puerto de serie, para conectarnos al simulador deberemos escribir la siguiente secuencia de instrucciones:

```
#include <Aria.h>

ArRobot robot;
ArTcpConnection tcpConn;

//Configura puerto
tcpConn.setPort;
//Asocia el puerto al robot
robot.setDeviceConnection(&tcpConn);
```

El siguiente paso consiste en decir al robot que nos queremos comunicar con él. Para ello, recordando el capítulo 9, se debían enviar los comandos SYNC1, SYNC2 y SYNC3 y atender la respuesta del robot. En nuestra aplicación servidor esta serie de comandos se envían con el método *ArRobot::blockingConnect*(). Si no se producen errores, aquí concluye el proceso de conexión. Resta tan solo habilitar los sonar y los motores mediante los comandos correspondientes.

12.2.3. Thread para las comunicaciones

Una vez que se ha efectuado la conexión con el robot, el cometido principal de la aplicación será efectuar las lecturas pertinentes de la información sensorial y enviarlas a través de la red.

Pensando en futuros desarrollos del sistema software, como pueden ser la implementación de controladores, sistemas de navegación o de control de nuevos dispositivos, etc. las labores de recolección de datos, su envío y la recepción de comandos desde el cliente se realizarán en un thread independientemente del programa principal.

Un thread (hebra) es un subproceso dentro de un proceso. La programación multihebra (con threads) permite la ejecución de diversas tareas en paralelo. Se consigue un estilo de ejecución en el que se conmuta entre distintas partes del código de un mismo programa durante la ejecución. En el caso de nuestra aplicación este hecho aporta grandes ventajas:

- Flexibilidad para la reutilización del código. Al liberar al programa principal de las tareas de recolección y transmisión de datos se podrá utilizar el mismo para realizar otro tipo de funciones.
- Facilidad para la gestión de atenciones a clientes a través de otros puertos. La gestión mediante threads permite afrontar futuras necesidades de atención a un alto número de clientes y de uso de más puertos
- Facilidad para la protección del robot ante posibles fallos en la conexión con el cliente. En el momento que se detecte un fallo en la conexión bastará con detener el thread y lanzarlo de nuevo para gestionar dicho fallo. La gestión de este tipo de fallos provocaría mayores dificultades si transmitiéramos la información en el programa principal.

Las herramientas para la programación con threads las proporciona la librería del sistema **pthreads**. De nuevo ARIA proporciona varias clases envolventes (wrapper) de dicha librería que facilitan la programación. En concreto la aplicación servidor usará la clase ArAsyncTask (que deriva de la clase ArThread).

El proceso a seguir es hacer heredar una clase de la clase ArAsyncTask y sobreescribir la función virtual *ArThread::runThread()*. En esta función se definen las diversas tareas que realizará nuestro thread. Para lanzar y detener el thread se usarán las funciones *ArThread::create()* y *ArThread::stopRunning()*.

12.2.4. Transmisión y recepción de la información: sockets

En esta sección se aborda qué información se envía y se recibe y cómo se realizan estos envíos desde la parte del servidor.

En el capítulo 11 vimos las diversas funciones disponibles para el uso de sockets en una aplicación. Todas esas funciones están presentes en la clase envolvente ArSocket.

Para la implementación del intercambio de información entre las aplicaciones cliente y servidor, el primer paso es elegir el puerto físico que se utilizará y el protocolo de transmisión. En nuestro caso se ha elegido el puerto 7777 (puerto no usado por ninguna otra aplicación elegido al azar) y el protocolo TCP/IP. Acto seguido se llama, con los argumentos pertinentes, a la funcion *open* (ver capítulo 11) desde le programa principal.

Una vez establecido el puerto de comunicaciones, desde le thread lanzado se usaran las funciones *accept*, *write* y *read* para aceptar la conexión de los clientes e intercambiar información con ellos.

En concreto, la información que se intercambian cliente y servidor son punteros a dos estructuras de datos con distintas variables que indican el estado del robot. La estructura de datos que se pasa del servidor al cliente contiene todos los datos sensoriales que ofrece la interfaz eléctrica del robot y es la siguiente:

```
struct inf_sen{
//Velocidades
double vel;
double vel_der;
double vel_iz;
double vel_rot;
double vel_max;
double vel_rot_max;
//Posicion respecto a la posicion inicial
double x;
double y;
double th;
//Sonars
int mySonars[16];
int sonar_prox;
```

```
int rango_sonar_prox;
//Resto de informacion
int motor_pac;
int sonar_pac;
bool sonars_en;
bool motors_en;
double volt;
double radio;
double diagonal;
int stall;
bool stall_der;
bool stall_iz;
int flags;
double compass;
unsigned char valor_analogico;
unsigned char estado_in_dig;
unsigned char estado_out_dig;
};
```

La estructura de datos que se pasa del cliente al servidor contiene las velocidades que se desean establecer al robot desde la parte del cliente. Es la siguiente:

```
struct inf_act{
//Velocidad lineal
double vel;
//Velocidad de rotación
double vel_der;
//Velocidad ruedas izquierda
double vel_iz;
//Velocidad ruedas derecha
double vel_rot;
};
```

Para el establecimiento de todas estas variables se utilizan diversos métodos proporcionados por la clase ArRobot(consultar el código fuente de la aplicación).

12.2.5. Protección del robot ante impactos y fallos de conexión

Para la protección del robot ante posibles impactos se ha implementado el método *InfoServidor::proteger()*. Esta función analiza cuál es el movimiento del robot y los obstáculos que se encuentran en su trayectoria. Si se concluye que se puede producir una colisión se detendrá el robot inmediatamente, no permitiendo el

movimiento en esa dirección. Esta función se ejecuta en la aplicación servidor y no en la aplicación cliente por motivos de seguridad ante errores en la conexión de ambas.

La función es llamada antes de transmitir a LIN las velocidades deseadas por el cliente.

Por otra parte se debe proteger al robot ante posibles fallos en la conexión con el cliente. Si se detecta algún error en los procesos de aceptar la conexión o recibir información, inmediatamente se detiene el robot.

La función *InfoServidor::proteger()* ha sido probada con resultados satisfactorios.

12.2.6. Compilación de los ficheros de código fuente

La aplicación consta de un sólo fichero fuente, servidor.cpp. Para compilar el mismo y realizar el linkado con las librerías ARIA y del sistema utilizadas se ha usado el fichero /usr/local/Aria/Makefile incluido en el paquete ARIA. Este fichero contiene una serie de reglas para el compilado y linkado de ficheros fuente.

Para producir el fichero ejecutable basta con situarse en el directorio /usr/local/Aria/ y teclear:

$>make servidor

De este modo habremos creado el fichero ejecutable servidor. Para ejecutar el programa basta con teclear su nombre en una línea de comandos del siguiente modo:

$>./servidor

12.3. Desarrollo de la aplicación cliente

En esta sección se describirá paso a paso el proceso de desarrollo del proceso cliente. Se recuerda que las funciones que debe realizar dicha aplicación son las siguientes:

- Conexión remota con el robot, es decir, desde la red local.
- Obtención remota de la información de los sensores. La aplicación cliente debe recibir los datos sensoriales enviados por el servidor.
- Teleoperación del robot. El usuario de la aplicación cliente podrá teledirigir el movimiento del robot a través del teclado.

12.3.1. Arquitectura lógica de la aplicación

En la presente sección se ofrecerá una descripción de la organización en clases de la aplicación cliente y del cometido de cada una. Con ello se pretende que el lector tenga una visión general del esquema de la aplicación para la posterior compresión de sus partes específicas.

En la figura 12.9 se muestra, esquemáticamente, el diagrama UML de clases de la aplicación.

Como se puede observar, siete son las clases que se usarán para el desarrollo:

- ArRobot
- ArSocket
- ArThread
- ArAsyncTask
- ArMutex
- ArKeyHandler
- InfoCliente
- infAct
- infSen
- Comando

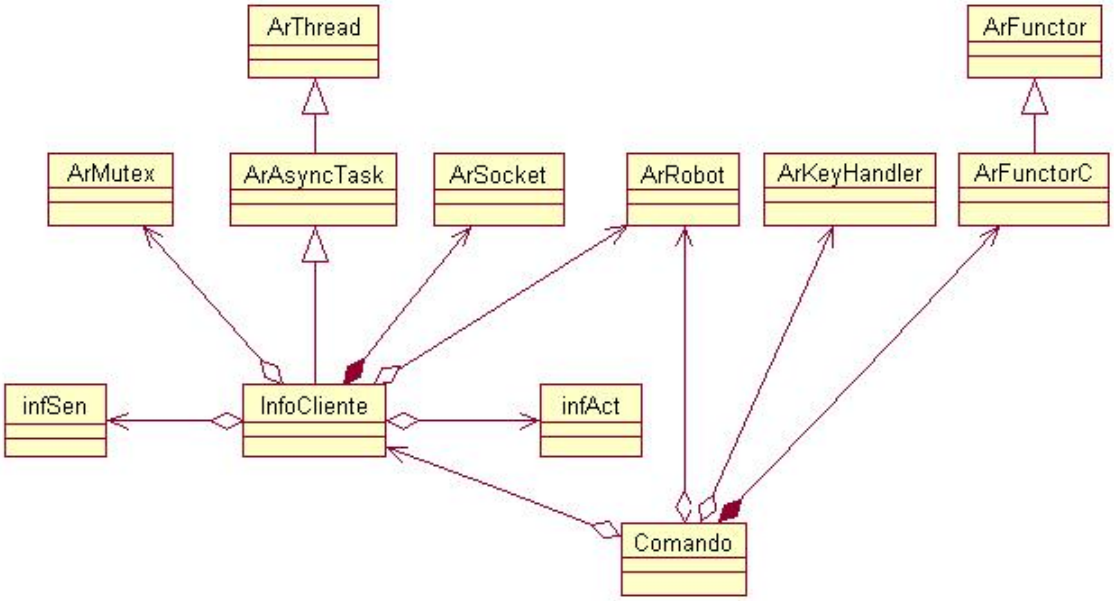

Figura 12.9: Diagrama UML cliente

Las clases ArRobot, ArSocket, ArThread, ArAsyncTask, ArMutex y ArKeyHandler pertenecen al paquete ARIA (ver capítulo 10). InfoCliente y Comando serán la clases principales de la aplicación y en ellas están presentes (implementados, heredados o instanciados de otras clases) todos los métodos necesarios para la captación de órdenes del teleoperador a través del teclado y la recepción de datos y el envío de comandos al servidor. Las clases infAct e infSen contienen la información transmitida o recibida de la aplicación servidor.

Las clases Las clases **ArRobot**, **ArSocket**, **ArThread** y **ArAsyncTask** ya fueron descritas en la sección anterior (sección 12.2) y sus diagramas UML pueden ser observados en las figuras 12.3, 12.4 y 12.5.

La clase **ArKeyHandler** proporciona diversos métodos para interceptar y manejar las entradas a través del teclado. Se podrán manejar las diversas entradas definiendo funciones que serán invocadas en el momento en que las teclas sean pulsadas. Su diagrama UML se puede observar en la figura 12.10 y su descripción detallada en la sección 12.3.2.

ArMutex es una clase envolvente de las funciones mutex del sistema operativo. Estas funciones son herramientas para la programación concurrente. Gracias a ellas podemos garantizar la exclusividad mutua de diferentes procesos o subprocesos que quieren acceder a una misma variable.

Las clases **ArFunctor** y **ArFunctorC** sirven para crear punteros a funciones que son miembros de una clase. En el desarrollo de la aplicación cliente ha sido

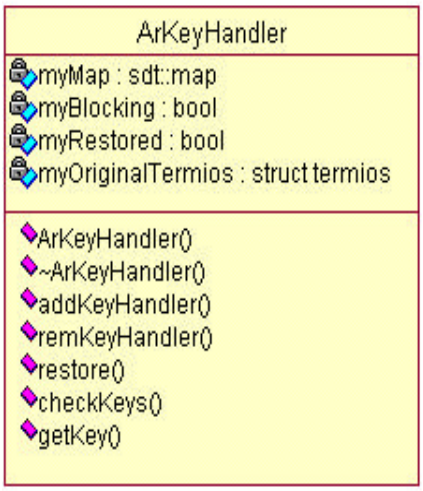

Figura 12.10: Diagrama UML de ArKeyHandler

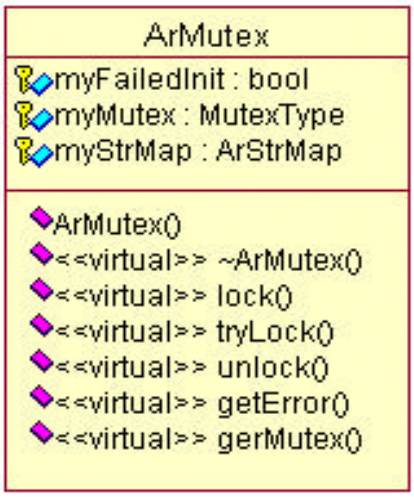

Figura 12.11: Diagrama UML de ArMutex

necesario crear punteros a funciones miembro de la clase Comando que manejan las entradas a través del teclado. Su diagrama UML se puede observar en la figura

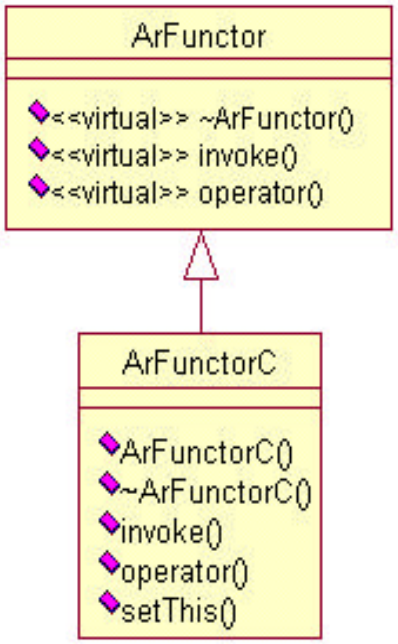

Figura 12.12: Diagrama UML de ArFunctor y ArFunctorC

12.12

La clase **Comando** ha sido desarrollada con el único objetivo de interceptar y manejar las entradas a través del teclado. Por ello observamos en ella instancias a las clases ArKeyHandler, ArFunctor e InfoCliente. Cada vez que el usuario de la aplicación oprima las teclas 'up', 'down', 'right', 'left', 'space' o 's' del teclado, el estado del objeto de la clase InfoCliente instanciado en Comando cambiará. De este modo se enviará al servidor la orden que el usuario de la aplicación cliente desea ejecutar en el robot.

La clase **InfoCliente** es la clase principal de la aplicación. A través de un objeto de esta clase se recibirá información del servidor. Esta información, junto con la proveniente de las entradas a través del teclado será procesada y con ella se realizarán mínimas tareas de control. Después se enviarán al servidor las velocidades pertinentes a establecer en el robot a través de sockets.

En la figura 12.8 se puede observar el diagrama de flujo de la aplicación. En la sección de declaraciones se crea un objeto de la clase InfoCliente y otro de la clase Comando. En el constructor de la clase Comando están implementadas todas las instrucciones para la intercepción y el manejo de las entradas a través del teclado. Por lo tanto, con la simple declaración de un objeto de esta clase en el programa

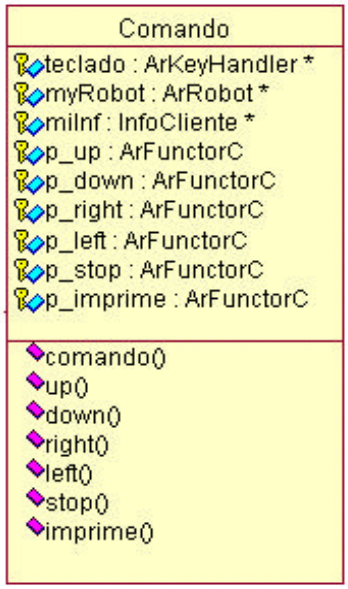

Figura 12.13: Diagrama UML de Comando

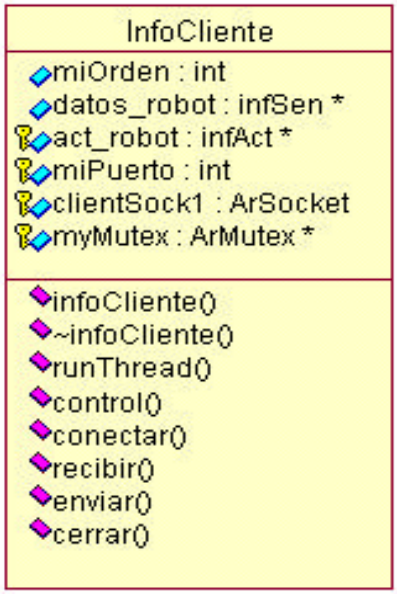

Figura 12.14: Diagrama UML de InfoCliente

principal se podrán manejar las entradas a través del teclado que introduzca el usuario.

Después de la sección de declaraciones se lanza un thread que se encargará de recibir y enviar información al cliente. En dicho thread se invocan funciones para la conexión con el puerto abierto en el servidor, la recepción de los datos sensoriales, el control conjunto de esta información y los comandos introducidos a través del teclado y el envío de las velocidades a establecer en el robot. Por último se comprueba el estado del thread a través de la variable *myRunning* y si procede se detiene el subproceso y se vuelve a lanzar.

12.3.2. Manejo de la entrada a través del teclado

Uno de los principales cometidos de la aplicación es controlar la entrada a través del teclado para hacer llegar al robot las órdenes de teleoperación dadas por el usuario.

Para ello ha sido construida la clase **Comando**. Para obtener el control del teclado se utilizan las funciones *Aria::getKeyHandler()* y *Aria::getKeyHandler()* y un puntero a un objeto de la clase ArKeyHandler. Estas funciones son llamadas en el constructor de la clase del modo siguiente:

```
ArKeyHandler * teclado;
ArRobot *myRobot;

if ((teclado = Aria::getKeyHandler()) == NULL){

    teclado = new ArKeyHandler;
    Aria::getKeyHandler();
    myRobot->attachKeyHandler(teclado);

}
```

Con esta simple secuencia de instrucciones hemos conseguido el control del teclado. Ahora deberemos definir qué teclas queremos interceptar y qué funciones queremos ejecutar cuando éstas sean pulsadas.

A continuación se muestra qué teclas serán interceptadas y qué ocurrirá cuando sean pulsadas:

- 'Up'. El robot aumentara su velocidad.
- 'Down'. El robot disminuirá su velocidad.

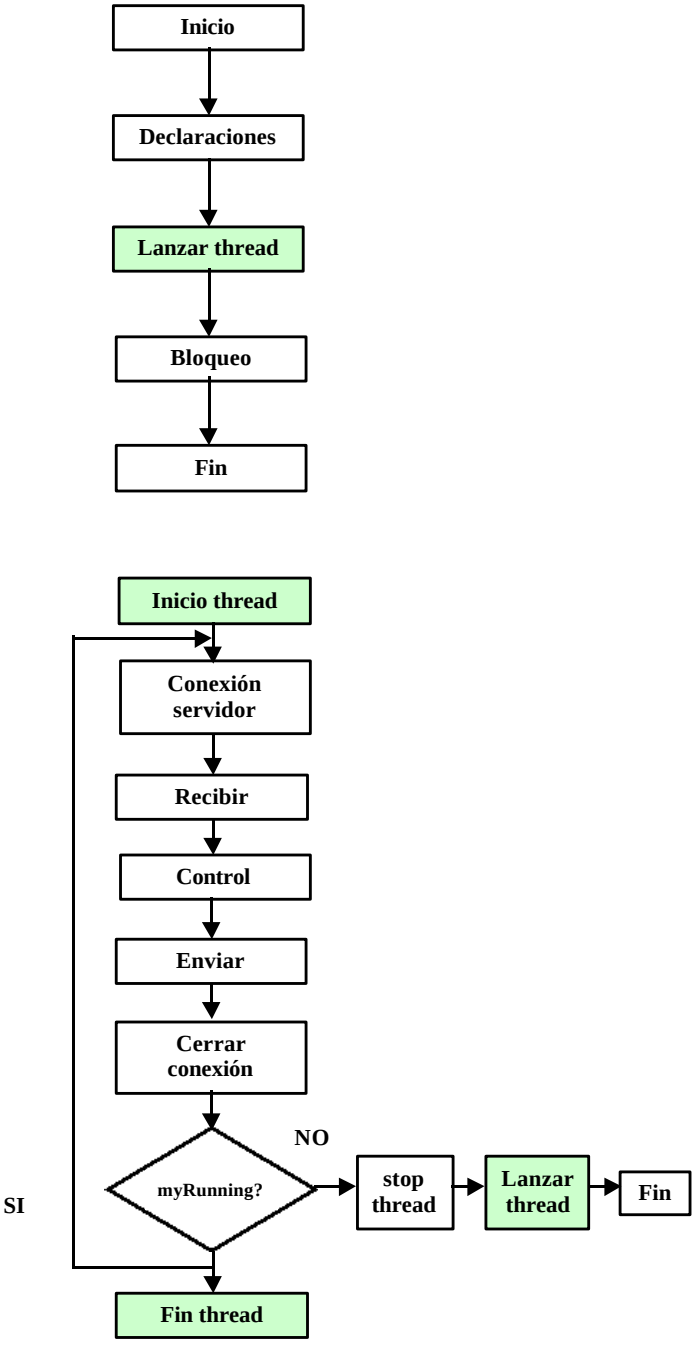

Figura 12.15: Diagrama de flujo

- 'Right'.El robot girará hacia la derecha.
- 'Left'. El robot girará hacia la izquierda.
- 'Espacio'. El robot se detendrá.
- 's'. Aparecerán en pantalla las diversas lecturas sensoriales.

Se aclara que en el cuerpo de las funciones anteriores no se transmiten al servidor las órdenes del usuario, sino que tan sólo se cambia el valor de la variable *miOrden* de la clase InfoCliente. Posteriormente se analiza el valor de esta variable y se envían al servidor las órdenes oportunas.

Para ello el primer paso es definir las funciones que serán ejecutadas y mediante la clase ArFunctorC punteros a estas funciones. Después, mediante el método *ArKeyHandler::addKeyHandler()* se asocia la tecla deseada al puntero a la función que queramos que sea ejecutada.

Por ejemplo, para que al ser pulsada la tecla 'up' sea invocada la función *Comando::up()* se escriben las siguientes instrucciones:

```
//Declaración de un puntero a una función de la clase Comando
ArFunctorC<Comando> p_up;
//Asociación del puntero a la función miembro up
p_up(this,&comando::up);
//Asociación de la tecla UP al puntero p_up
teclado->addKeyHandler(ArKeyHandler::UP, &p_up)
```

Con todo lo anteriormente enunciado queda descrito el proceso de control de las entradas a través del teclado.

12.3.3. Recepción, control y transmisión de información

Una vez que se ha resuelto el problema del control del teclado, queda afrontar cómo se recibe, se procesa y se envía información al servidor.

Al igual que en la aplicación servidor y por los mismos motivos el proceso de recepción de los datos sensoriales y envío de comandos tendrá lugar en un thread o subproceso del programa principal (ver sección 12.2.4).

En el capítulo 11 vimos las diversas funciones disponibles para el uso de sockets en una aplicación. Todas esas funciones están presentes en la clase envolvente ArSocket, instanciada en InfoCliente.

El primer paso es conectarnos a la máquina donde corre la aplicación servidor (LIN) y al puerto abierto para las comunicaciones (puerto 7777). Para ello haremos una llamada a la función *connect* con los argumentos adecuados.

Después de ello, con la llamada a la función *read* se recibirán los datos sensoriales. La estructura concreta de datos recibidos se puede consultar en la seccion 12.2.4.

A continuación se analiza cuáles son las velocidades linear y de rotación del robot, la orden de control introducida a través del teclado (por ejemplo aumentar la velocidad) y con esa información se establecen las nuevas velocidades a enviar al servidor. El límite a la velocidad linear es de 250 mm/s y para la velocidad de rotación 50 mm/s. Nunca se enviarán velocidades superiores. Todo ello se lleva a cabo en la función I*nfoCliente::control()*.

Tras esto, sólo queda enviar las velocidades, con la función *write* y examinar la variable *myRunning* que indica el estado del thread. Si fuera necesario se detiene y se lanza de nuevo.

12.3.4. Compilación de los ficheros de código fuente

La aplicación consta de un sólo fichero fuente, cliente.cpp. Para compilar el mismo y realizar el linkado con las librerías ARIA y del sistema utilizadas se ha usado el fichero /usr/local/Aria/Makefile incluido en el paquete ARIA. Este fichero contiene una serie de reglas para el compilado y linkado de ficheros fuente.

Para producir el fichero ejecutable basta con situarse en el directorio /usr/local/Aria/ y teclear:

$>make cliente

De este modo habremos creado el fichero ejecutable cliente. Para ejecutar el programa basta con teclear su nombre en una línea de comandos del siguiente modo:

$>./cliente

Capítulo 13

CORBA

13.1. Introducción

Del análisis de requisitos y de la naturaleza propia del problema se concluyó que el sistema software a desarrollar deberá ser distribuido y construido sobre una arquitectura cliente/servidor. Este hecho nos llevó a pensar en una implementación basada en sockets como método más sencillo para llegar a una solución. Sin embargo en la sección 11.13 se vió cómo una solución basada en sockets tenía limitaciones e inconvenientes. En los capítulos 13, 14 y 15 damos un paso hacia delante y abordaremos el desarrollo de una solución basada en la tecnología CORBA.

Como veremos, el desarrollo basado en CORBA aporta grandes ventajas con respecto a la solución basada en sockets. Principalmente las aplicaciones CORBA no se limitan al transporte de datos a través de la red, sino que permiten la invocación de métodos de objetos remotos implementados en otras máquinas. Además una arquitectura basada en CORBA posee una mayor flexibilidad y escalabilidad que aquellas basadas en sockets. El principal inconveniente es mayor dificultad en el diseño e implementación de las aplicaciones.

La tecnología CORBA está muy ligada a la existencia de sistemas distribuidos heterogéneos. Un sistema distribuido heterogéneo es aquel compuesto por diferentes módulos software interactuando entre sí en diversas plataformas hardware unidas entre sí por una red de área local. Las distintas plataformas hardware pueden tener arquitecturas diversas y soportar diferentes sistemas operativos.

El principal factor que conduce a la heterogeneidad en las redes de computadores es, desde mi punto de vista, que una combinación de arquitectura del computador y sistema operativo es efectiva para un determinado tipo de tareas, pero no para todas. En el caso del proyecto actual la heterogeneidad podría venir dada con la incorporación de distintas máquinas dedicadas a tareas concretas.

Por ejemplo a la transmisión de imagen captada por eventuales cámaras.

CORBA es un conjunto de especificaciones que proporciona un conjunto de abstracciones flexibles y servicios concretos necesarios para proporcionar soluciones prácticas a los problemas que surgen en entornos distribuidos heterogéneos.

CORBA no es más que una especificación normativa para la tecnología de la gestión de objetos distribuidos (DOM). La tecnología DOM proporciona una interfaz de alto nivel siutada en la cima de los servicios básicos de la programación distribuida. En los sistemas distribuidos es muy importante la definición de la interfaz y ciertos servicios como la búsqueda de módulos. CORBA es proporciona un estándar para poder definir estas interfaces entre módulos, así como algunas herramientas para facilitar la implementación de dichas interfaces en el lenguaje de programación escogido. Adicionalmente se han incluido algunos servicios estándar accesibles a todas las aplicaciones en CORBA. Además CORBA proporciona el mecanismo que permite a los distintos módulos de una aplicación comunicarse entre sí.

CORBA es independiente tanto de la plataforma como del lenguaje de la aplicación. La independencia de plataforma significa que los objetos de CORBA se pueden utilizar en cualquier plataforma que tenga una aplicación CORBA ORB. La independencia de lenguaje se refiere a que los objetos CORBA y los clientes se pueden implementar en cualquier lenguaje de programación. Así, a un objeto CORBA no le hará falta saber el lenguaje en que ha sido escrito otro objeto con el que se esté comunicando.

13.2. El grupo OMG

El Grupo de Gestión de Objetos OMG (*Object Management Group*) se formó con la misión de crear un mercado de programación basada en componentes, impulsando la introducción de objetos de programación estandarizados.

Los estatutos de la organización incluyen el establecimiento de guías para la industria y especificaciones detalladas para la gestión de objetos a fin de suministrar un marco de trabajo común para el desarrollo de un entorno de computación distribuida que abarque las principales arquitecturas de máquina y sistemas operativos.

Su propósito principal es desarrollar una arquitectura única, utilizando la tecnología de objetos, para la integración de aplicaciones distribuidas garantizando la reusabilidad de los componentes, la interoperabilidad y la portabilidad, y basada en componentes de programación disponibles comercialmente.

Mientras que las primeras propiedades de la arquitectura propuesta por el OMG están basadas en los beneficios del uso de la tecnología de objetos por la

cual ha apostado, la última corresponde a una decisión estratégica del consorcio orientada a evitar las dificultades que han tenido los productos de otros organismos de normalización como la ISO para encontrar una adecuada respuesta en el mercado. Su objetivo, más allá de generar normas, es el de asegurase de que estas normas sean utilizadas ampliamente, por lo que ninguna propuesta es adoptada como una norma a menos que describa una tecnología que ya se encuentre en el mercado o asegure una rápida disponibilidad.

La arquitectura del OMG está constituida entonces por componentes (objetos) interoperables con interfaces normalizadas, ofrecidos en el mercado por diversas empresas, que los usuarios adquieren para construir sus propias aplicaciones.

A pesar de que en su nombre se menciona a los objetos en general, el OMG no pretende cubrir todos los ámbitos de aplicación de la tecnología de objetos, los cuales incluyen los lenguajes de programación, los métodos de análisis y diseño, las interfaces gráficas de usuario, las bases de datos, y los componentes (servicios o aplicaciones encapsulados como "circuitos integrado"). Su interés está centrado en la generación de normas con la especificación de los componentes requeridos para el desarrollo de aplicaciones distribuidas (interfaces), o lo que es lo mismo, para la integración de aplicaciones.

Para lograr su propósito de construir una Arquitectura para la Gestión de Objetos (OMA, Object Management Architecture), el OMG ha establecido una base común de partida para todo el trabajo alrededor de la misma, la cual está constituida por los siguientes elementos:

- Un conjunto único de términos y definiciones par los conceptos a manejar en relación con la orientación a objetos: objeto, clases, mensaje, etc.
- Un marco de abstracción común, o modelo de objetos, que define un modelo matemático riguroso para las aplicaciones en términos de objetos y para compartir información entre aplicaciones en términos de mensajes.
- Un modelo de referencia común o arquitectura.
- Un conjunto común de interfaces, protocolos y lenguajes.

13.3. La norma CORBA

CORBA es una especificación normativa que resulta de un consenso entre los miembros del OMG, un consorcio que agrupa hoy por hoy a más de 700 empresas tanto de la industria informática como consumidores de la misma. Esta norma cubre cinco grandes ámbitos que constituyen los sistemas de objetos distribuidos:

- Un lenguaje de descripción de interfaces, llamado Lenguaje de Definición de Interfaces **IDL** (Interface Definition Language), traducciones de este lenguaje de especificación IDL a lenguajes de implementación (como pueden ser C++, Java, ADA, etc.) y una infraestructura de distribución de objetos llamada Object Request Broker (ORB) que ha dado su nombre a la propia norma: Common Object Request Broker Architecture (CORBA).

- Una descripción de servicios, conocidos con el nombre de **Servicios CORBA** (CorbaServices), que complementan el funcionamiento básico de los objetos de que dan lugar a una aplicación. Estas especificaciones cubren los servicios de nombrado, de persistencia, de eventos, de transacciones, etc. El número de servicios se amplía continuamente para añadir nuevas capacidades a los sistemas desarrollados con CORBA.

- Una descripción de servicios orientados al desarrollo de aplicaciones finales, estructurados sobre los objetos y servicios CORBA. Con el nombre de **Facilidades Comunes** (CorbaFacilities), estas especificaciones cubren servicios de alto nivel, como los interfaces de usuario, los documentos compuestos, la administración de sistemas y redes, etc. La ambición es aquí bastante amplia ya que CorbaFacilities pretende definir colecciones de objetos prefabricados para aplicaciones habituales en la empresa: creación de documentos, administración de sistemas informáticos, etc.

- Una descripción de servicios verticales denominados **Interfaces de Dominio** (CorbaDomains), que proveen funcionalidad de interés para usuarios

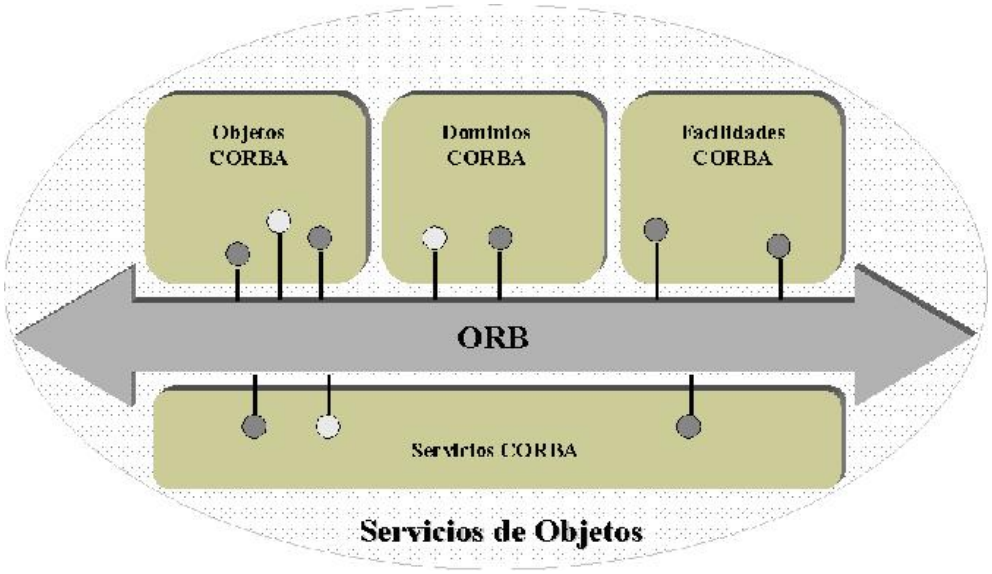

Figura 13.1: Servicio de objetos

finales en campos de aplicación particulares. Por ejemplo, existen proyectos en curso en sectores como: telecomunicaciones, finanzas, medicina, etc.

- Un protocolo genérico de intercomunicación, el Protocolo General Inter-ORB **GIOP** (General Inter-ORB Protocol), que define los mensajes y el empaquetado de los datos que se transmiten entre los objetos. Además define implementaciones, de ese protocolo genérico, sobre diferentes protocolos de transporte, lo que permite la comunicación entre los diferentes ORBs consiguiendo la interoperabilidad de elementos de diferentes vendedores. Por ejemplo el IIOP para redes con la capa de transporte TCP.

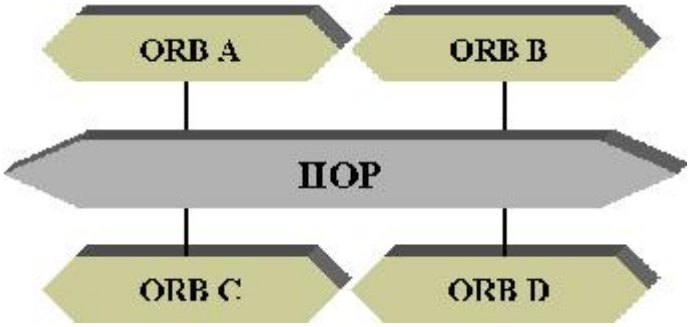

Figura 13.2: Un protocolo genérico de intercomunicación GIOP

13.4. Estructura

Object Request Broker (ORB) es el sistema intermedio (*midleware*) que establece relaciones cliente/servidor entre objetos. Mediante la utilización de un ORB, un cliente puede invocar transparentemente un método de un objeto servidor que se encuentre en la misma máquina o en otra distinta. El ORB intercepta la llamada realizada por el objeto que implementa la petición, pasa los parámetros, invoca el método y retorna los resultados. No es necesario que el cliente sepa dónde se localiza el objeto que ejecutará el método, el lenguaje en el que está programado, el sistema operativo, ni cualquier otro aspecto que no sea su interfaz. Hay que resaltar que los papeles de cliente y servidor que se dan a los objetos son simplemente para coordinar las interacciones entre ambos; estos papeles pueden cambiar ya que un objeto puede ser cliente o servidor dependiendo de la ocasión.

Su funcionamiento es el siguiente: cuando un módulo de una aplicación quiere usar un servicio proporcionado por otro módulo obtiene una referencia del objeto que provee ese servicio. Después de obtenerla, el módulo puede invocar métodos en ese objeto. La primera responsabilidad del ORB es resolver peticiones de referencias de objetos, permitiendo a los módulos de la aplicación establecer conexión entre ellos.

Otra de las responsabilidades del ORB es el marshaling y el unmarshaling. Después de que un módulo de la aplicación haya obtenido una referencia del objeto cuyos servicios quiere usar, ese módulo ya puede invocar métodos en ese objeto. Generalmente estos métodos necesitan parámetros como entrada y devuelven otros parámetros como salida. El ORB debe recibir los parámetros de entrada del módulo que llama al método y ”marshal” estos parámetros. Esto quiere decir que el ORB traduce los parámetros a un formato (on-the-wire format) que puede ser transmitido por la red hasta el objeto remoto. El ORB también ”unmarshal” los parámetros devueltos, convirtiéndolos a un formato que el módulo llamante entienda.

Todo esto se hace de forma transparente a la intervención del programador. Una aplicación cliente invocará un método remoto y recibirá los resultados como si el método fuese local.

Gracias a este proceso se consigue la independencia de plataforma, debido a que los parámetros se traducen en la transmisión a un formato independiente de la plataforma (el on-the-wire format forma parte de las especificaciones CORBA) y en recepción se convierten al formato específico de la plataforma. Un cliente ejecutándose en un sistema Macintosh podrá invocar métodos de un servidor que se ejecute en un sistema UNIX. Además de la independencia del SO usado, las diferencias de hardware (como el ordenamiento de los bytes, endianness, la longitud de las palabras, etc.) son irrelevantes puesto que el ORB hace las conversiones necesarias automáticamente.

En resumen, las responsabilidades del ORB son:

- Dada una referencia a un objeto por un cliente, el ORB localiza la correspondiente implementación del objeto (el servidor).
- Cuando el servidor está localizado, el ORB asegura que el servidor está preparado para recibir la petición.
- El ORB del lado del cliente acepta los parámetros del método que se está invocando y "marshal" los parámetros a la red.
- El ORB del lado del servidor "unmarshal" los parámetros de la red y se los entrega al servidor.
- Los parámetros de retorno, si existen, se "marshal"/"unmarshal" del mismo modo.

En la figura siguiente se muestra la estructura de un sistema distribuido basado en CORBA. En él se puede observar dos partes: una cliente y una servidora. En la parte cliente existirá un programa cliente propiamente dicho al que CORBA le añade cierta infraestructura para permitir la comunicación con el servidor a través de la red. Del otro lado, la parte servidora estará formada por el objeto que exporta su funcionalidad, integrado en el servidor (proceso en el que se ejecuta) y diversos elementos que permiten que las invocaciones realizadas por el cliente a los métodos del objeto lleguen a éste, sean procesadas y sus resultados devueltos.

- La parte cliente estará formada por:

  Los Stubs del Cliente :

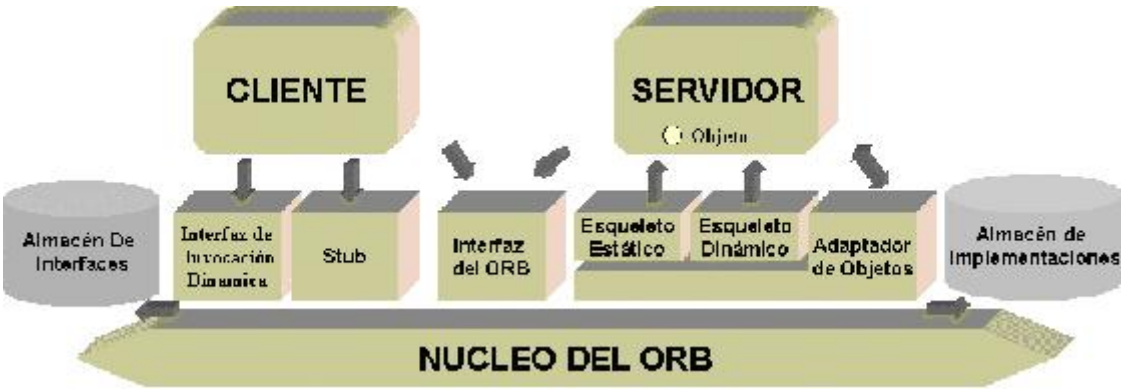

Figura 13.3: Sistema distribuido

Proporcionan una capa intermedia entre el cliente y el núcleo del ORB. Definen cómo los clientes invocan los servicios que proporcionan los objetos servidores. Desde la perspectiva del cliente, el stub actúa como una especie de proxy, dando la impresión de que la invocación se está realizando sobre un objeto local como en cualquier aplicación orientada a objetos. El stub se encarga de codificar la operación y sus parámetros, y de enviarla de forma remota.

El interfaz de invocación dinámica:

Permite descubrir en tiempo de ejecución métodos para ser invocados. CORBA define una biblioteca, que permite localizar el método, generar los parámetros, realizar la llamada remota y recoger los resultados.

El almacén de interfaces :

Permite obtener y modificar la descripción de todos los componentes que en él están registrados, los métodos que soporta y los parámetros que requiere. CORBA llama a esas descripciones firmas. El almacén de interfaces (*Interface Repository*) es una base de datos distribuida modificable en tiempo de ejecución.

El interfaz del ORB :

Consiste en unas pocas librerías de servicios locales para realizar labores auxiliares en la aplicación. Por ejemplo, CORBA proporciona APIs (*Aplication Programming Interface*, interfaces de programación de aplicaciones) para convertir referencias a objetos a cadenas. Estas llamadas pueden ser muy interesantes si se necesita comunicar o almacenar referencias a objetos.

La capacidad de soportar tanto invocaciones dinámicas como estáticas, además del almacén de interfaces, da a CORBA una gran ventaja sobre las plataformas intermedias competidoras. Las invocaciones estáticas son rápidas y fáciles de programar. Las invocaciones dinámicas proporcionan la máxima flexibilidad, pero son difíciles de programar. Esta últimas son muy usadas por herramientas que descubren servicios en tiempo de ejecución.

Por lo que respecta al lado del servidor no hay diferencia entre la invocación dinámica y estática, ya que ambas tienen el mismo mensaje semánticamente hablando. En los dos casos el ORB localiza al Adaptador de Objetos del objeto y le transmite un mensaje que contiene el nombre del servicio a invocar y sus parámetros. La implementación recibe a través del esqueleto del objeto los datos necesarios, ejecuta el servicio y retorna el resultado para que sea enviado al cliente en forma de un nuevo mensaje.

- Los elementos que componen la parte servidora son los siguientes:

El esqueleto del servidor:

Proporciona los elementos necesarios para que los clientes invoquen los servicios exportados por el objeto. Estos esqueletos realizan una función similar a los stubs del cliente tratando de hacer transparente todo el proceso de comunicación.

El esqueleto de interfaces dinámicos (DSI):

Proporciona un mecanismo de enlazado en tiempo de ejecución para servidores que necesitan manejar llamadas a métodos que no tienen esqueletos estáticos definidos.

El adaptador de objetos:

Proporciona en tiempo de ejecución un entorno para la instanciación de objetos en el servidor (proceso que los acoge), la asignación de referencias y la gestión las peticiones que les lleguen.

El almacén de implementaciones:

Proporciona en tiempo de ejecución un almacén de información acerca de las clases que el servidor soporta, los objetos instanciados y sus identificadores.

El interfaz del ORB:

Consiste en unas pocas APIs de servicios locales iguales a las de la parte cliente.

13.5. El modelo de comunicaciones de CORBA

Esta sección tiene por objeto aclarar el papel de CORBA en una red de ordenadores. Típicamente ésta consiste en sistemas que están físicamente conectados. Esta capa física proporciona el medio a través del cual tiene lugar la comunicación. Más allá de la capa física se encuentra la capa de transporte, que incluye los protocolos responsables de mover los paquetes de datos desde un punto hacia otro. CORBA es neutral respecto estos protocolos de red, es independiente del protocolo utilizado y podría funcionar con cualquiera.

El estándar CORBA especifica el Protocolo General Inter-ORB (GIOP) que, en un nivel alto, establece un estándar para la comunicación entre varios ORB´s de CORBA. GIOP es tan sólo un protocolo general, por lo que el estándar CORBA también especifica otros protocolos que detallan GIOP para usar un protocolo de transporte particular (como TCP/IP o DCE). El utilizado para redes TCP/IP se denomina Internet Inter-ORB Protocol (IIOP). Los fabricantes deben implementar este protocolo para ser considerados conformes a CORBA. Este requerimiento ayuda a asegurar la interoperabilidad entre productos CORBA de diferentes fabricantes, aunque cada uno puede tener además sus propios protocolos.

Un ORB puede soportar cualquier protocolo, pero siempre debe incluir el IIOP (se negocia el protocolo a usar al establecerse la conexión entre los ORB´s). De todas formas, los ORB´s de CORBA normalmente se comunican usando el IIOP, debido en parte a que este protocolo es el correspondiente al protocolo TCP/IP, que es el usado en Internet. En el argot CORBA/IIOP las referencias a objeto se pasan a denominar Referencias a Objetos Interoperables o IOR´s.

Resumiendo, las aplicaciones CORBA se construyen encima de los protocolos derivados de GIOP (como el IIOP). Estos protocolos están, a su vez, encima de los protocolos de transporte (TCP/IP, DCE). Las aplicaciones CORBA no están limitadas a usar sólo uno de estos protocolos, sino que se puede usar un puente para interconectar aplicaciones situadas en redes que trabajen con diferentes protocolos. Se puede ver que, más que suplantar los protocolos de transporte, la arquitectura CORBA crea otra capa (la capa del protocolo Inter-ORB) que utiliza estos protocolos como soporte. Esta es otra de las claves de la interoperabilidad entre las aplicaciones CORBA ya que no se dicta el uso de un protocolo de transporte particular.

13.6. Adaptadores de objetos

El Adaptador de Objetos juega un papel muy importante en la estructura de un sistema CORBA. Se sitúa como un pegamento entre las implementaciones de los objetos CORBA y el propio ORB, consiguiendo que las peticiones que llegan a través del ORB sean procesadas por la implementación del Objeto. Entre sus responsabilidades se encuentran:

Registrado de los objetos:

Los Adaptadores de Objetos deberán proporcionar funciones para registrar implementaciones para los objetos CORBA.

Generación de referencias a objetos:

Los Adaptadores de Objetos deberán generar referencias para los objetos que tengan registrados.

Activación de procesos servidores:

Los Activadores de Objetos deberán activar los procesos (servidores) donde los objetos puedan ser activados.

Activación de objetos:

Los Activadores de Objetos deberán activar los objetos registrados.

Multiplexación de peticiones a los objetos registrados:

Los Adaptadores de Objetos deberán asegurar que todas la peticiones sean recibidas por los objetos, aunque tengan múltiples conexiones, sin que ninguna se bloquee indefinidamente.

Gestión de las invocaciones:

Los Adaptadores de Objetos deben despachar las peticiones de objetos registrados.

En la norma CORBA han aparecido dos adaptadores de objetos: el actual adaptador POA (*Portable Object Adaptor*), y el original y ya desaparecido (en la versión CORBA2.2) BOA (*Basic Object Adaptor*).

El adaptador de objetos BOA fue el primero en ser incluido en la norma CORBA y cubría de forma mínima las funciones del Adaptador de Objetos, como el registrado de objetos, la generación de referencias, la activación de implementaciones de objetos y la gestión de las peticiones. Por este motivo no será tratado en la presente memoria.

13.6.1. Adaptador portable de objetos (POA)

- **Políticas del POA**

La especificación CORBA define un conjunto de políticas que especificarán como es la gestión que el POA va a realizar sobre sus objetos asociados. El programador podrá definir los valores para estás políticas cuando cree un nuevo POA. A continuación se comentarán las políticas y sus posibles valores:

Política de hilos:

Se usa para controlar los hilos que el servidor puede crear para procesar las invocaciones sobre los objetos que contiene. Existen dos posibilidades: la de un único hilo (SINGLE_CTRL_MODEL) o la de hilos controlados por el ORB (ORB_CTRL_MODEL). Si se utiliza la política de único-hilo todas las peticiones se procesarán secuencialmente. En el caso de utilizar multihilo, será el ORB el que los gestione. En este caso todos los objetos, incluidos el Mánager de Sirvientes, deben tener un código reentrante.

Política de vida:

Esta política se utiliza para especificar al POA si los sirvientes que se vayan a activar en él serán persistentes (PERSISTENT) o transitorios (TRANSIENT). Cuando un objeto es persistente puede sobrevivir al proceso que lo ha creado. Esto quiere decir que, si se desactiva un servidor con objetos persistentes

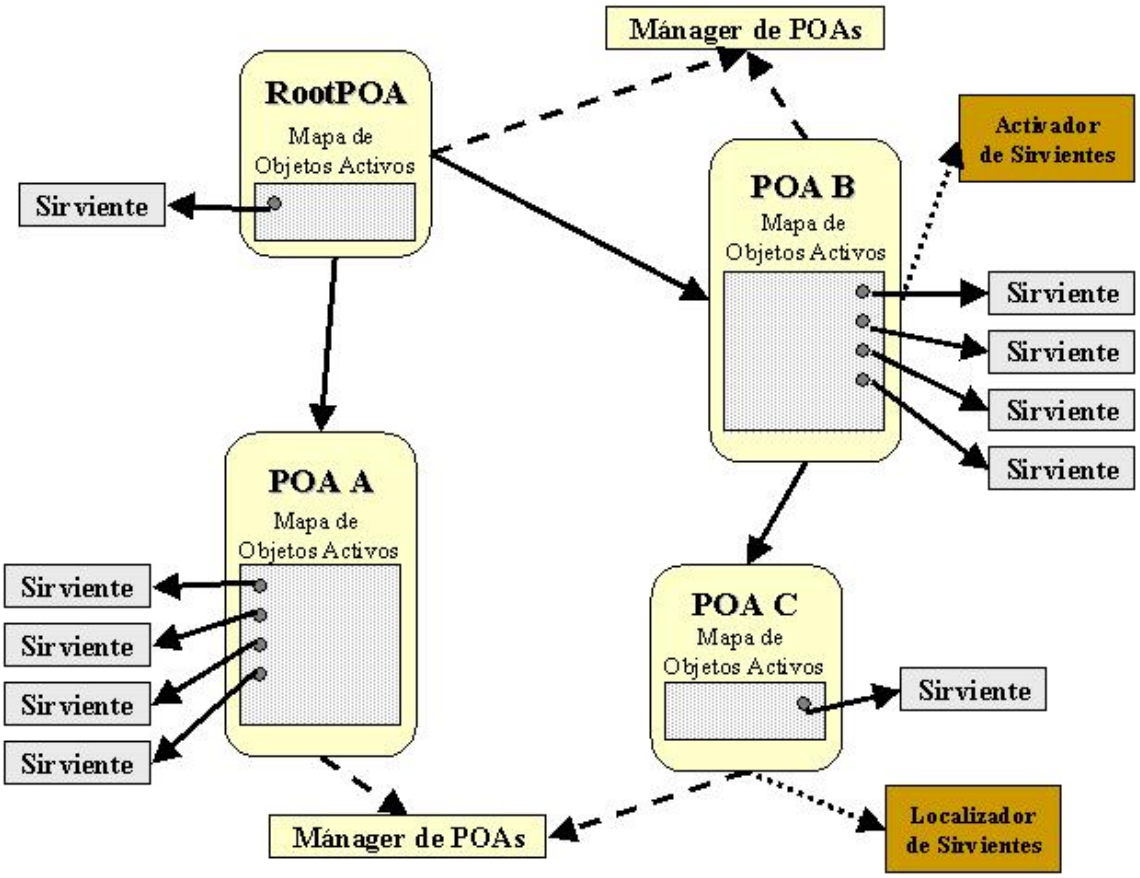

Figura 13.4: Adaptador portable de objetos

en su interior, la próxima vez que vuelva a ser activado, el POA le generará idénticas referencias, con lo que para todos sus clientes el objeto será el mismo que el que estuvieron utilizando anteriormente. Por supuesto el POA no se hará cargo de restaurar el estado de los objetos, si no que deberá ser el programador el que se encargue de almacenarlo antes de la destrucción y recuperarlo al volver a activar los objetos. Por el contrario, si los objetos son transitorios, el POA al reactivarlos les generará nuevas referencias, con lo que sus clientes no podrán reconocerlos.

Política de identificador único:

La política de Identificador Único se utiliza para indicar al POA si los sirvientes deben tener un único identificador (UNIQUE_ID) o por el contrario podrán tener varios (MULTIPLE_ID). Cuando se utiliza un único identificador el POA fuerza a que cada referencia tenga un sirviente que atienda sus peticiones.

Política de asignación de identificador:

Esta política se utiliza para indicar al POA si los identificadores de los objetos van a ser creados por la aplicación (USER_ID) o por el ORB (SYSTEM_ID).

Política de activación:

Esta política se utiliza para indicar al POA si la activación implícita está permitida (IMPLICIT_ACTIVATION) o no (NO_IMPLICIT_ACTIVATION). Si lo está, un servidor podrá crear un sirviente y activarlo en el POA mediante una única operación (_this).

Política de retención de sirvientes:

La política de Retención de Sirvientes se utiliza para indicar al POA si debe almacenar la relación sirviente-referencia en el mapa de sirvientes activos. Sus valores pueden ser retener (RETAIN) y no retener (NON_RETAIN). Si el valor elegido es no retener deberá combinarse con los valores de la política de procesamiento de peticiones USE_DEFAULT_SERVANT o USE_SERVANT_MANAGER. La utilización de NON_RETAIN como Política de Retención de Sirvientes está recomendada para servidores que tienen una gran cantidad de sirvientes, ya que el almacenado de las asociaciones requiere una gran cantidad de memoria.

Política de procesamiento de peticiones:

Esta política se utiliza para indicar al POA cómo debe procesar las peticiones que le lleguen. Existen tres posibilidades:

- Consultar solamente el Mapa de Sirvientes Activos (USE_ACTIVE _MAP_ONLY): el POA comprueba si la referencia tiene entrada en el Mapa de Sirvientes Activos; si no la tiene, retornará al cliente una excepción de objeto inexistente. Esta posibilidad no podrá ser utilizada si la política de vida es NON_RETAIN.
- Utilizar Sirviente por Defecto (USE_DEFAULT_SERVANT): en este caso si el identificador de objeto no está en el Mapa de Objetos Activos, el POA intentará despachar la petición con el Sirviente por Defecto.
- Invocar al Mánager de Sirvientes (USE_SERVANT_MANAGER): si el POA no encuentra el identificador del objeto en el Mapa de Objetos Activos utilizará el Mánager de Sirvientes para intentar encontrar un sirviente que responda a la petición. Existen dos tipos de Mánagers de Sirvientes: el Activador de Sirvientes y el Localizador de Sirvientes. Según la política de Retención de Sirvientes el POA admitirá un tipo o el otro: Activador de Sirvientes para RETAIN y Localizador de Sirvientes para NON_RETAIN.

La utilización de Sirviente por Defecto y de Localizador de Sirvientes esta desaconsejada en objetos que tienen estado. En el caso del Sirviente por Defecto, debido a que varios clientes pueden utilizar un mismo sirviente y en el caso del Localizador de Sirvientes porque existe la posibilidad que las peticiones de un mismo cliente sean tratadas por sirvientes diferentes

La utilización de Mánagers de Sirvientes permite al programador establecer criterios en la asignación de sirvientes a referencias, control del número de sirvientes activos, establecer límites en el número de sirvientes activos simultáneos, etc. Con ello puede conseguir una gestión del servidor que aproveche de forma más eficiente los recursos del sistema.

Capítulo 14

ICa

14.1. Introducción

En el capítulo anterior hemos descrito CORBA como una herramienta orientada al desarrollo de software en sistemas distribuidos heterogéneos. En el presente capítulo se intentará describir ICa, una herramienta que además de las características de CORBA, está orientada al control de sistemas complejos e incorpora prestaciones de tiempo real y de tolerancia a fallos.

ICa es un framework o entorno de desarrollo diseñado para asistir a la creación de sistemas software de medio y alto nivel en entornos industriales. Proporciona herramientas para el desarrollo de software distribuido, con prestaciones de tiempo real y tolerancia a fallos. Es el resultado del trabajo realizado en el Departamento de Automática, Ingeniería Electrónica e Informática Industrial, dentro del marco de un proyecto ESPRIT, denominado DIXIT, en colaboración con varias empresas de toda Europa.

14.2. Arquitectura de control integrado

Las siglas ICa significan Arquitectura de Control Integrado.

Entendemos por **arquitectura** un diseño software a un determinado nivel de abstracción, focalizado en los patrones de organización del sistema, que describen como se particiona la funcionalidad y como esos elementos se interconectan e interaccionan entre sí. Por un lado una arquitectura es un modelo de separación del sistema en componentes más sencillos y por otro un modelo de coordinación de los mismos.

ICa implementa una metodología de orientación a componentes: módulos de

software que interaccionan con otros a través de una interfase pública bien conocida. En el caso más general estos componentes de sitúan en un entorno distribuido y heterogéneo, formado por una o varias plataformas hardware unidas por una red de área local que les permite interactuar a través de ella. Y heterogéneo en el sentido de que las diversas plataformas lo son, tanto desde el punto de vista del hardware como de los sistemas operativos soportados.

Cuando decimos **control** nos referimos al mundo para el que ha sido desarrollado. ICa dispone de extensiones para adaptarse al control de procesos industriales, en el que aparecen restricciones de tiempo real, en el que los subsistemas deben ser tolerantes a fallos y en el que existen limitaciones de velocidad importantes. Por descontado, ICa no renuncia a entornos en los que no aparecen estas limitaciones. Otros frameworks similares, como COM u otras implementaciones CORBA, han sido desarrollados para aplicaciones principalmente ofimáticas y se aplican de una manera forzada en entornos que se encuentran más allá de sus especificaciones de diseño. ICa ataca el problema desde el punto de vista contrario: se dispone de la tecnología para atacar las restricciones complejas, por lo que podemos cubrir los sistemas que no requieren algunas de estas prestaciones simplemente renunciando a ellas.

Por último con el término **integrado** se hace referencia a una de las características principales de ICa: su capacidad para integrar tecnologías, plataformas y lenguajes diversos. A la hora de integrar componentes para formar un sistema existen técnicas diversas y los desarrolladores deben conocer y controlar todas ellas para utilizar la más adecuada en cada caso. Distinguimos cuatro tipos básicos de mecanismos de integración:

- Integración *en-thread*: se implementa mediante invocación a funciones. La integración ocurre dentro del mismo proceso, compartiendo el espacio de direcciones y sin que exista concurrencia en la ejecución.
- Integración *en-proceso*: la comunicación ocurre entre subprocesos dentro del mismo proceso, de forma concurrente y compartiendo el espacio de direcciones.
- Integración *en-máquinas*: Los elementos a integrar se sitúan en la misma máquina, coordinados por el mismo sistema operativo, pero en espacios de direcciones diferentes.
- Integración *en-red*: Una red de ordenadores es el elemento integrador, ya que los distintos componentes están situados en varias máquinas.

Los sistemas de control se diseñan generalmente por capas, formando lo que se conoce como la pirámide de control. En las capas inferiores se intercambia información a altas velocidades y con un nivel de abstracción bajo, mientras que,

a medida que vamos ascendiendo por la misma, los flujos de datos disminuyen pero aumenta el nivel de abstracción de los mismos. En las capas superiores tenemos control inteligente, en el que la información que se intercambia es abstracta y desarrollada con metodologías diversas, proporcionando principalmente soporte a la decisión (DSS). La integración de estas metodologías de control, junto con la integración de diversas plataformas y sistemas operativos son los objetivos de ICa. La siguiente figura muestra la pirámide de control en un proceso.

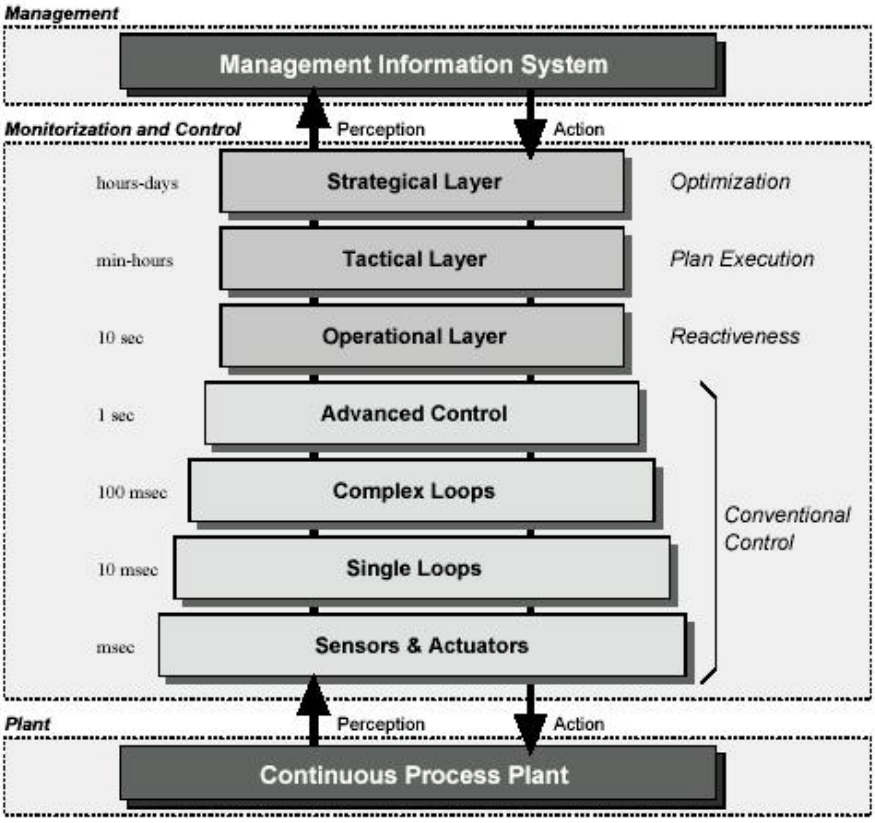

Figura 14.1: Pirámide de Control

14.3. Características

ICa ha sido diseñado para el desarrollo de software distribuido de alto nivel. En concreto se puede definir ICa de lo siguientes modos:

En primer lugar, ICa es una metodología de desarrollo, que asiste al diseñador a la hora de decidir como implementar la arquitectura del sistema. Ayuda a

realizar, de forma sencilla, sistemas distribuidos o no, proporcionando las líneas maestras para el diseño de los mismos.

En segundo lugar ICa es un entorno de desarrollo de software. Proporciona librerías para los distintos lenguajes en los que es posible desarrollar los componentes, están disponibles compiladores para asistir en la creación de los distintos componentes y herramientas para mantener las comunidades de componentes que se desarrollan.

Por último ICa es un marco en el que desarrollar nuevas herramientas para la construcción de nuevos sistemas, en los que aparezcan restricciones y requerimientos mucho más allá de lo que nosotros hayamos podido imaginar. Desde el principio ha sido desarrollado con una mentalidad modular y abierta, y orientado de tal forma que los nuevos trabajos que se desarrollen sobre ICa proporcionen la mejora del mismo.

ICa está derivado en gran medida de los requerimientos de CORBA, de donde toma una parte muy importante de sus especificaciones funcionales. De hecho, ICa es compatible a nivel del lenguaje de descripción de interfases, con CORBA 2.0, así como la mayor parte de la interfase de programación desde C++. Por otro lado se han añadido muchas prestaciones a CORBA, tanto para el desarrollo de proyectos en control de procesos (tiempo real, tolerancia a fallos, flujos de datos, etc.) como otros tipos de sistemas. También se ha simplificado la especificación de CORBA, con el fin de hacerlo más previsible y funcional en entornos industriales. Por dar una definición, ICa es una implementación particular de CORBA, centrada en la integración en entornos heterogéneos y con tecnologías heterogéneas, compuesto por una metodología de desarrollo, un conjunto de componentes genéricos preconstruidos fácilmente adaptables a cada necesidad y un mecanismo integrador de los distintos componentes. La siguiente imagen muestra cualitativamente una comparativa entre CORBA e ICA.

Como en toda especificación de CORBA, el núcleo fundamental del sistema lo constituye el textbfbroker, un elemento común, presente en todos los sistemas, que actúa como un EXV de transferencia de datos entre los distintos componentes. En ICa se ha añadido una capa de abstracción adicional, denominada transporte, de manera que se pueda modificar la forma en que se comunican los distintos componentes con un mínimo de esfuerzo, incluso en tiempo de ejecución del sistema. Es posible pasar de una aplicación que utiliza memoria compartida para integrar los componentes a otra que utiliza una red de comunicaciones utilizando TCP/IP prácticamente sin modificar el código del programa.

ICa dispone de una serie de características que le ayudan a encajar en entornos industriales, así como otras que permiten flexibilizar el desarrollo de aplicaciones, como veremos a continuación.

Mediante la utilización de un mecanismo que garantiza la separación com-

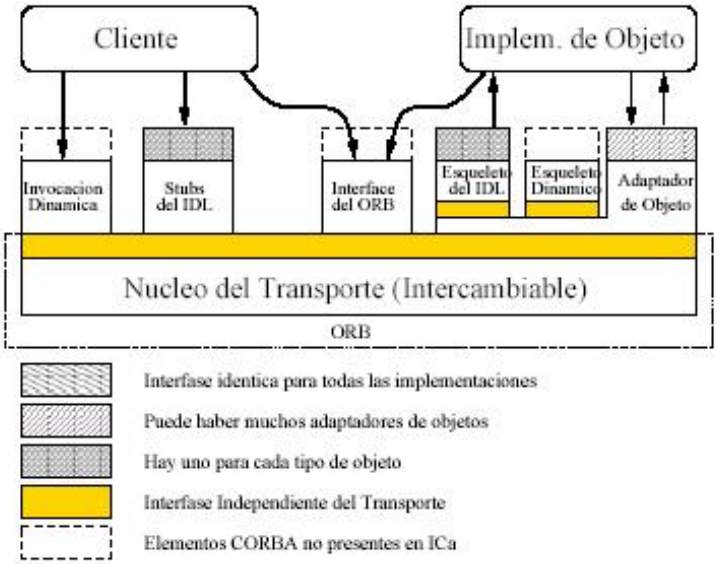

Figura 14.2: ICa y CORBA

pleta de la implementación de los clientes de los servidores, se puede garantizar que no es necesario modificar los clientes si sólo se hacen modificaciones internas en secciones privadas o protegidas en los servidores. También garantiza que podrán ser utilizados clientes con futuras versiones de los servidores, siempre que sus interfases públicas sean un superconjunto de la antigua. Esto realmente mejora las posibilidades de C++ en cuanto al mantenimiento del software y de las versiones del mismo: es posible actualizar componentes en caliente, sin recompilar los clientes que interaccionan con él y ni siquiera detenerlos, actualizando la aplicación en caliente.

ICa no es una arquitectura cerrada que proporciona un marco en el que desarrollar aplicaciones. A lo largo del tiempo se ha demostrado que este tipo de marcos de trabajo no proporcionan la suficiente flexibilidad como para cubrir un conjunto suficientemente amplio de casos de desarrollo. La forma en que ICa ataca el problema es proporcionando un entorno modular y configurable. Cada uno de los módulos que componen el sistema pueden ser eliminados y sustituidos por otros, de forma que es posible adaptar la arquitectura a cada aplicación en concreto. Por otro lado ICa proporciona también componentes genéricos preconstruidos, adaptables a cada necesidad concreta. De esta forma los equipos de desarrollo van creando componentes cada vez más específicos que proporcionan una paleta más amplia de la que disponer en desarrollos futuros. La clave principal es la ingeniería de dominios: esos desarrollos futuros particularizarán aún más los mismos y aumentará la paleta, dando una ventaja importante al grupo de desarrollo a la hora de competir en el desarrollo de aplicaciones en este dominio.

Es lo que los creadores de ICa llaman Focalización Progresiva de Dominios.

ICa presenta una capa de abstracción del transporte que permite separar completamente la implementación de los agentes del mecanismo que se utiliza para comunicarlos. A modo de ejemplo se han desarrollado tres tipos distintos de transportes, además de disponer de los mecanismos necesarios para desarrollar otros transporte en el futuro. Esto hace que los componentes puedan distribuirse utilizando redes TCP/IP, redes corporativas y potencialmente cualquier otra tecnología existente.

Otra de las características es la independencia de la plataforma: ICa presenta un API de programación y un comportamiento común a todas las plataformas en las que opera, ocultando las particularidades internas del hardware o del sistema operativo y permitiendo gestionar de forma común elementos tan particulares como los procesos de bajo peso o threads. También realiza tareas de traducción entre las representaciones internas de los datos en las distintas plataformas, de forma que puedan interoperar unas con otras. En este momento funciona en un conjunto amplio de plataformas y sistemas operativos:

- Linux sobre intelx86
- Solaris sobre spark
- OSF1 sobre ALPHA
- AIX sobre RS6000
- Win32 sobre intelx86
- Irix sobre MIPS

Para amoldarse al entorno industrial para el cual ICa ha sido desarrollado tiene extensiones que permiten desarrollar sobre él sistemas de tiempo real no críticos. Algunas de estas extensiones son:

- Llamadas con limitación de tiempo de invocación o timeouts, de forma que un cliente puede indicar que si la respuesta a la petición que ha realizado no está disponible en un determinado tiempo máximo, debe de cancelarse esa petición.
- Estimación del tiempo de ejecución requerido por un método. Un desarrollador puede estimar el tiempo requerido para la ejecución de un determinado método, de forma que pueda, en tiempo de ejecución, modificar su comportamiento, adaptando su funcionamiento a los tiempos requeridos por los servidores.

- Sistema de gestión dinámica de prioridades. Un cliente puede indicar a un servidor, en tiempo de ejecución, la prioridad de ejecución de cada uno de los métodos. ICa automáticamente traduce esto a prioridades de los distintos subprocesos que ejecutan los métodos públicos.
- Gestión de subprocesos y coordinación de los mismos. Dispone de un API capaz de enmascarar las funciones de creación, cancelación y coordinación de los distintos subprocesos, de forma independiente del sistema operativo sobre el que corre cada aplicación.

Otra de las características que hacen de ICa un entorno de programación para la industria son las extensiones para crear sistemas tolerantes a fallos. Estas extensiones están pensadas para crear aplicaciones en las que los errores en tiempo de ejecución sean particularmente graves, de forma que sea necesario introducir código de respaldo para solventar los posibles errores de ejecución. Podemos citar, por ejemplo:

- Gestión de Grupos: que permite desarrollar sistemas de desarrollo con N-versiones idénticas o N-versiones desarrolladas por equipos de trabajo diferentes. Es lo que se conoce como redundancia estática.
- Algunas de estas extensiones Para implementar redundancia estática se dispone del componente genérico llamado PermanentAgent, que gestiona, sin intervención del usuario, puntos de chequeo y bloques de recuperación.
- Para implementar redundancia dinámica están disponibles los componentes genéricos FTICaAgent y MasterFTICaAgent, permitiendo el desarrollo de grupos de agentes y sistemas de votación que los coordinen.

La arquitectura está basada en CORBA 2.1, extendiendo el estándar en aquellas partes en las que CORBA es insuficiente o ineficaz (para el desarrollo de aplicaciones de control), lo que hace de él un entorno abierto. La idea fundamental es ser compatible con CORBA siempre que ello no haga que la arquitectura final sea menos eficiente o flexible.

ICa permite el Diseño Distribuido Tardío, es decir el desarrollo de un diseño de la distribución del sistema en su totalidad con posterioridad al diseño en sí mismo de los módulos que intervienen en el sistema. Para ello utiliza una serie de transportes diferentes para las comunicaciones entre diferentes agentes y la posibilidad de incluir clases C++ que funcionen como agentes, sistemas de transporte nulo, etc. Tradicionalmente, la información arquitectónica se incorpora al sistema en las primeras etapas de desarrollo del mismo, haciendo que las decisiones erróneas en el diseño sean difícilmente solventables en etapas posteriores. El diseño distribuido tardío da la posibilidad de modificar estas restricciones arquitectónicas en cualquier etapa del desarrollo del sistema.

Otra novedad en ICa es la unificación de la definición del agente y su interfase. En contra de la filosofía de DCOM y CORBA, ICa utiliza un único fichero para la definición de la implementación y de la interfase pública del mismo. Esto asegura la congruencia de los agentes y minimiza el trabajo de los programadores. Por supuesto es posible, a partir de un fichero de ADL eliminar todos los elementos de implementación interna y generar un fichero que contenga únicamente la definición de la interfase pública para ser distribuido. Para ello, el lenguaje ICa ADL se forma por la unión de los IDLs de CORBA, para declarar la interfase, y las posibilidades de C++ para especificar la implementación de los distintos componentes.

Las interfases están abiertas a otros lenguajes: ha sido desarrollado utilizando C++ y está pensado para que los componentes también se implementen utilizando este lenguaje. Sin embargo ICa permite utilizar los agentes desde otros lenguajes. En este momento está disponible la utilización de componentes desde JAVA y también desde la metodología COM de Microsoft. En breve plazo estará disponible también la utilización para OLE automation.

La granularidad de los agentes es muy variable. Para adaptarse al amplio abanico de posibilidades que aparecen en los sistemas de control industrial, ICa permite realizar componentes sencillos que tengan desde 12 bytes de tamaño, hasta agentes de complejidad y prestaciones todo lo complejas como sea necesario, según las necesidades de cada aplicación.

El sistema goza de gran estabilidad. ICa ha sido utilizado y lo está siendo en la actualidad, en proyectos tan diversos como sistemas de control en una planta cementera de Lafarge Ciments en su planta de Contes (Francia), sistemas de supervisión y gestión de alarmas en una planta petroquímica de Repsol en Tarragona, simulación distribuida, gestión de vídeo en tiempo real en una red de área extensa, control de un sistema de laminación en una siderúrgica entre otros, en varios países diferentes. Todo ello demuestra tanto la flexibilidad para amoldarse a entornos diversos como la estabilidad de las aplicaciones desarrolladas con Ica.

Capítulo 15

Desarrollo de una solución basada en ICa

Una vez descritos los conceptos y características de la norma CORBA y de su especificación ICa, el presente capítulo está dedicado a la descripción de las aplicaciones software desarrolladas basadas en ICa.

15.1. Instalación de ICa

En primer lugar se deben instalar las librerías ICa en todas las máquinas en las que vaya a correr una aplicación ICa, es decir, en la CPU de LIN y en los terminales de la red local.

Para ello el laboratorio SCILabs (creadores de ICa) nos ha proporcionado el fichero **lib-ICa-2.01.tar.gz**. Para descomprimir este fichero, nos situaremos en el directorio raíz y teclearemos los comandos:

```
$>gzip -c lib-ICa-2.01.tar.gz
$>tar -cvf lib-ICa-2.01.tar
```

De este modo se habrá creado el directorio /usr/local/ICa que contiene las librerías, scripts y ficheros necesarios para el desarrollo y la compilación de aplicaciones ICa.

Para completar la instalación se deben añadir algunas líneas a diversos ficheros de configuración del sistema.

En el fichero /etc/hosts se de deben añadir entradas para identificar la máquina en la que correrá el servicio de nombres (aplicación icans). Para el correcto funcionamiento de las aplicaciones, el programa icans deberá correr en alguno de los terminales de la red local. Configuraremos las máquinas donde se insta-

le ICa para que icans corra en la máquina local o en uno de los servidores del laboratorio. Para ello añadiremos a /etc/hosts:

```
127.0.0.1       ICaHost   localhost.localdomain
138.100.76.245  ICaHost   c3p5.disam.upm.es
```

Por otra parte, en el fichero /dev/services se deberá configurar que puertos y protocolos usa ICa para establecer la comunicación entre las distintas máquinas de la red local. Se añadirán las siguientes líneas:

```
ICa    2001/tcp
ICa    2001/udp
```

Además se debe crear el fichero /etc/init.d/ICa con el siguiente contenido:

```
service ICa
{
flags = REUSE
socket_type = stream
wait = no
user = root
server = /usr/local/ICa/bin/Linux/icans
log_on_failure  += USERID
disable = yes
}
```

Por otra parte, para facilitar la compilación de los programas, también se añaden las siguientes variables de entorno al fichero .bashrc:

```
#ICa Configuration
# Valid machines
export MCTYPES="i386 sparc ppc cris"
# Valid systems
export OSTYPES="Linux SunOS OSF1 RTAI"

# Cross compiling
export CROSSCOMPILING="NO"
#export CROSSCOMPILING="NO"
export CROSSMACHINE="ppc"
#export CROSSMACHINE="cris"
export CROSSSYSTEM="Linux"

#export MACHINE=$(uname -m)
export COMPILINGMACHINE="i386"
export COMPILINGSYSTEM=$(uname -s)
```

```
if [[ $CROSSCOMPILING == "NO" ]]; then

export MACHINE=$COMPILINGMACHINE
export SYSTEM=$COMPILINGSYSTEM;

elif [[ $CROSSCOMPILING == "YES" ]]; then

export MACHINE=$CROSSMACHINE
export SYSTEM=$CROSSSYSTEM;

fi

#ICa path
export ICA=/usr/local/ICa
source $ICA/Configure
#User bin path
export USERBIN=$HOME/bin/$COMPILINGMACHINE
export OUTBIN=$HOME/bin/$MACHINE
export PATH=$USERBIN:$PATH
export PATH=$PATH:/sbin:/usr/sbin:.
export PATH=$PATH:/opt/java/j2re1.4.2_02/bin
```

Tras llevar a cabo todos estos pasos la instalación de ICa queda completada.

15.2. Ficheros .idl

Recordamos que la norma CORBA define un lenguaje de descripción de interfaces, llamado Lenguaje de Definición de Interfaces **IDL** (Interface Definition Language) y traducciones de este lenguaje de especificación IDL a lenguajes de implementación (como pueden ser C++, Java, ADA, etc.).

Mediante este lenguaje se podrán definir interfases de clases, métodos y tipos de datos que después serán mapeados a un lenguaje concreto. En concreto, las aplicaciones desarrolladas en el presente proyecto constan de dos ficheros idl que a continuación son descritos.

15.2.1. Fichero interfazcorba.idl

Una vez que se han instalado en las máquinas los componentes de ICa necesarios se procede a la implementación del fichero **interfazcorba.idl**.

En este fichero se describirán los interfaces de las clases y métodos que serán utilizados en las aplicaciones cliente y servidor. Se vuelve a poner de manifiesto que en el fichero idl tan sólo son descritos el interfaz de los métodos, no su implementación.

El cuerpo de los métodos descritos en el fichero idl será desarrollado en el lado del servidor. Dé este modo podrán ser invocados remotamente por objetos del cliente sin que éste conozca la implementación de los métodos. Así mismo, se podrá cambiar su implementación transparentemente, sin que esto afecte al cliente.

En el fichero **interfazcorba.idl** será declarada la clase LinRobot. Los métodos de esta clase, en líneas generales, posibilitarán la conexión y desconexión con el robot, leerán los diversos datos sensoriales del robot (velocidades, rango de los sonar, etc) y establecerán en el robot las velocidades linear y de rotación que les sean pasadas como parámetros. En concreto, los métodos declarados en interfazcorba.idl son los siguientes:

```
//Funciones de conexion y desconexion
long connect();
void disconnect();

//Funciones para la obtencion de las diferentes velocidades
//del robot
float getVelocity();
float getLeftVelocity();
float getRightVelocity();
float getRotVelocity();

//Funciones para establecer las diferentes velocidades
//del robot
void setVelocity(in float vel);
void setLeftRightVelocity(in float leftvel,in float rightvel);
void setRotVelocity(in float vel);

//Funcion que mueve el robot a una distancia dada
void  moveDistance(in float vel);

//Funciones para la obtencion de las coordenadas del robot
float getXCoordinate();
float getYCoordinate();
float getThCoordinate();

//Funciones para la obtencion del rango de los sonar
```

```
long  getSonar_Range(in long num_sonar);
short getClosestSonar();
long  getClosestRange();

//Voltios de las baterias
float getBattery();
```

15.2.2. Fichero Cosnaming.idl

El fichero **CosNaming.idl** debe ser utilizado por cualquier aplicación ICa. En él vienen declarados diversos clases y métodos necesarios para que los servicios CORBA sean capaces de localizar donde se encuentran los objetos cuyos métodos son invocados.

Este fichero ha sido proporcionado a ASLab por los ingenieros de SCILabs (creadores de ICa). Por ello no se entrará en detalles en cuanto a su estructura interna.

15.2.3. Compilación de los ficheros .idl

Una vez generados los ficheros .idl se deberá proceder a su compilación para que se generen los ficheros correspondientes escritos en lenguaje C++. Los ficheros generados a partir de los ficheros .idl serán los que realmente utilicen las aplicaciones. Para la compilación de los ficheros se deberá teclear:

$>make adl

Tras la ejecución del comando anterior se generarán los siguientes ficheros:

- interfazcorba.h
- interfazcorba.cpp
- interfazcorbaS.h
- interfazcorba.cpp
- CosNaming.h
- CosNaming.cpp
- CosNamingS.h
- CosNamingS.cpp

Estos ficheros constituyen los stubs de los clientes y los skeletons del servidor. Se recuerda que los stubs y los skeleton son capas intermedias entre el cliente, servidor y el núcleo del ORB. Definen cómo los clientes invocan los servicios que proporcionan los objetos servidores (ver sección 13.4).

15.3. Desarrollo del servidor ICa

La aplicación servidor tendrá como única misión implementar y servir los métodos descritos en el fichero interfazcorba.idl a los posibles clientes que los soliciten.

Obviamente, la arquitectura física del sistema es la misma que la vista para la solución basada en sockets (ver sección 12.1). Por lo tanto la aplicación servidor correrá sobre la CPU de LIN. El servidor se comunicará con el microcontrolador del robot a través del puerto de serie /dev/ttyS0 y con los clientes a través de la red local.

15.3.1. Arquitectura lógica de la aplicación servidor

En la presente sección se ofrecerá una descripción de la organización en clases de la aplicación servidor y del cometido de cada una de ellas. Con ello se pretende que el lector tenga una visión general del esquema de la aplicación para la posterior compresión de sus partes específicas.

En la figura 15.1 se muestra, esquemáticamente, el diagrama UML de clases de la aplicación.

La clase **LinRobot** es generada automáticamente a partir de la compilación del fichero interfazcorba.idl. En ella se declaran como virtuales todos los métodos declarados en el anterior fichero idl.

La clase **LinRobot_impl** hereda de LinRobot. En ella se implementan todos los métodos que eran virtuales en la clase LinRobot. Es la clase principal de la aplicación y sus métodos son serán invocados remotamente por los clientes.

El resto de clases, pertenecientes todas al paquete ARIA (ver capítulo 10) se usan en la implementación de los métodos de LinRobot_impl.

15.3.2. Implementación del código fuente

Tras la generación y compilado de los ficheros idl se lleva a cabo el desarrollo de la aplicación servidor. La misma constará de un sólo fichero, **servidorICa.cpp**.

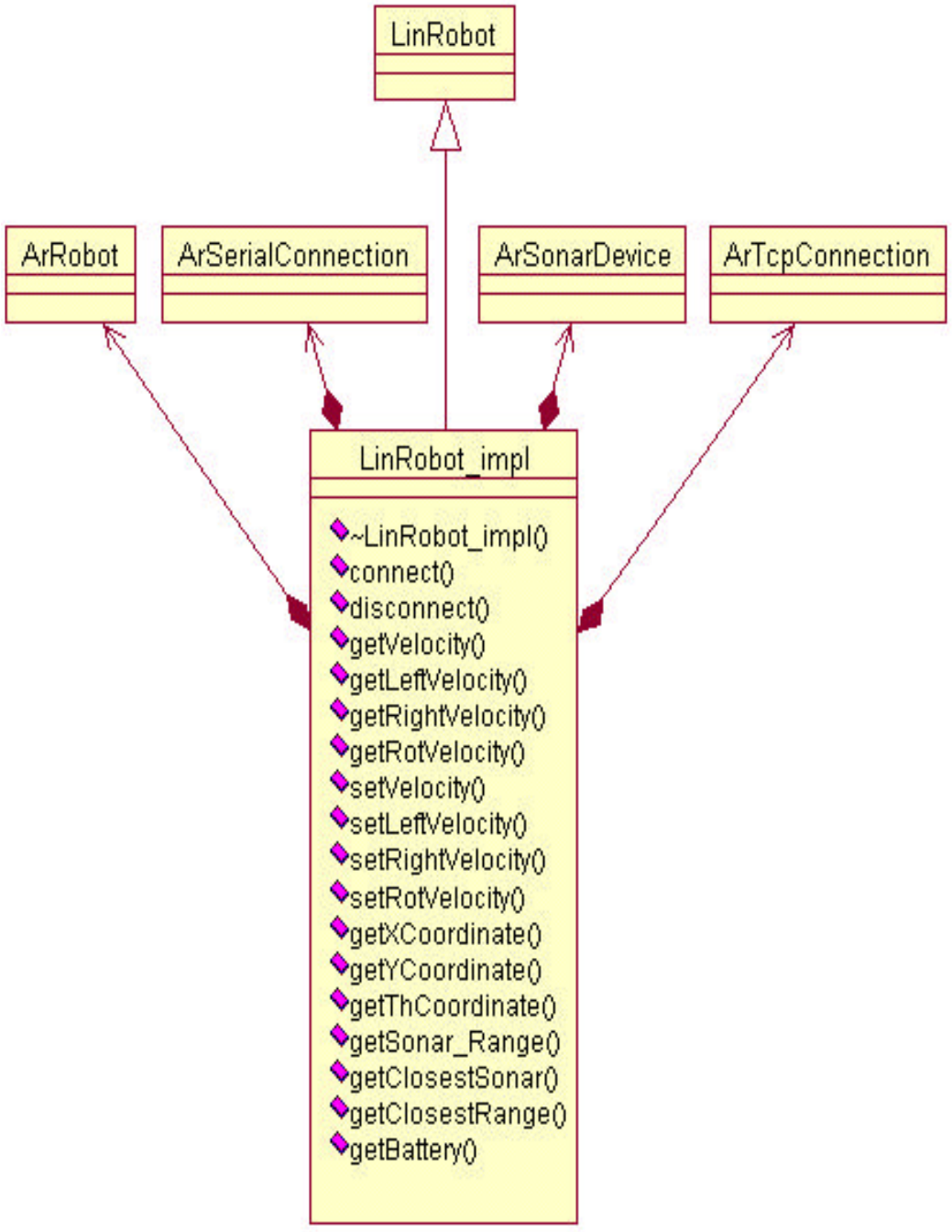

Figura 15.1: Diagrama UML del servidor ICa

En la parte inicial del fichero se incluyen todos los ficheros necesarios para el desarrollo: los ficheros skel y los ficheros ICa necesarios para configurar el adaptador de objetos POA y encarnar los objetos y servicios desarrollados:

```
#include "interfazcorbaS.h"
#include "CosNaming.h"
#include <RTPortableServer.h>
#include <RTCORBA.h>
```

Despues de esto, se declara la clase **LinRobot_impl**, que hereda de la clase **LinRobot** generada a partir del fichero interfazcorba.idl. En la misma se implementará el cuerpo de los métodos de que invocará el cliente. Como ya se ha dicho, estos métodos posibilitarán la conexión y desconexión con el robot, leerán los diversos datos sensoriales del robot (velocidades, rango de los sonar, etc.) y establecerán en el robot las velocidades linear y de rotación que les sean pasadas como parámetros.

Para su implementación en la clase LinRobot_impl se declararán varios objetos de clases de ARIA. Los detalles del código de conexión al robot, lectura de los sensores, etc. fueron explicados en el capítulo 12.

El resto del fichero es el programa principal. En él se declaran diversas variables y objetos CORBA para obtener referencias al ORB (Object Request Broker) y al POA (Portable Object Adapter), para crear un sirviente de los métodos desarrollados y para resolver las referencias a los nombres de los objetos (ver capítulo 13 y código fuente).

Tras la debida secuencia de instrucciones, los métodos definidos en el fichero interfazcorba.idl podrán ser invocados por clientes remotos.

15.3.3. Compilación y ejecución de la aplicación: Makefile

Una vez implementado el código fuente, sólo resta compilarlo y enlazarlo con las librerías utilizadas. Para ello se construye el fichero Makefile (ver apédice B.5) y se usa la herramienta make.

Make funciona a partir de un conjunto de reglas e indicaciones que el usuario escribe en el archivo Makefile. Dichas reglas indican al programa la forma en que debe tratar los archivos fuente, y cómo y dónde debe generar el archivo ejecutable.

Nuestro Makefile será configurado para que el fichero servidorICa.cpp sea enlazado con las librerías y ficheros fuente (.cpp) y de cabecera (.h) de ICa, ARIA y del sistema correspondientes. A partir de todos ellos se generará el fichero ejecutable **servidorICa**. Concretamente, nos situaremos en el directorio donde se encuentren los ficheros fuentes y el fichero Makefile y teclearemos:

```
$> make adl
$> make dep
$> make
```

Como se vio en la sección 15.2, la primera de las anteriores instrucciones, *make adl*, sirve para generar los ficheros .cpp y .h a partir de los ficheros .idl que constituyen los skeleton del servidor y stub del cliente. Con *make dep* generamos un fichero en el que quedan registradas todas las dependencias entre sí de los ficheros fuente y de cabecera de la aplicación. Con *make* generamos los ficheros objeto (.o) y el ejecutable final.

Para ejecutar la aplicación en primer lugar es necesario lanzar el programa **icans**. Este programa, ICa Name Service, es el encargado de proporcionar los servicios CORBA para la localización de objetos a partir de su interfaz. El programa icans se puede lanzar en cualquier terminal de la red local.

El paso siguiente es lanzar la aplicación servidorICa pasandole como argumento la referencia escrita por icans en el fichero NamingService.IOR. Esta referencia será un número que depende de la máquina donde se ejecuta icans. Para ello se construye el script **lanzar_servidorICa**.

15.4. Desarrollo de un cliente ICa para la teleoperación del robot

Como ejemplo de cliente ICa se desarrollará, análogamente a la solución basada en sockets, una aplicación capaz de telecontrolar el movimiento del robot. Las funciones que debe realizar dicha aplicación son las siguientes:

- Conexión y desconexión remota del robot.
- Obtención remota de la información de los sensores.
- Teleoperación del robot. El usuario de la aplicación cliente podrá teledirigir el movimiento del robot a través del teclado.

15.4.1. Arquitectura lógica de la aplicación cliente

En la presente sección se ofrecerá una descripción de la organización en clases de la aplicación cliente y del cometido de cada una de ellas. Con ello se pretende que el lector tenga una visión general del esquema de la aplicación. Las aplicaciones cliente correrán sobre cualquier terminal de la red local.

En la figura 15.2 se muestra, esquemáticamente, el diagrama UML de clases de la aplicación.

La clase **LinRobot_var** es la clase fundamental de la aplicación. Es generada a través de la compilación del fichero interfazcorba.idl. A través de sus métodos invocaremos los servicios del servidor remoto. Es decir, los métodos de LinRobot_var, en realidad se ejecutarán en la CPU de LIN, máquina donde corre el servidor.

La clase **InfoSensorial** posee atributos destinados a almacenar la información proveniente de los sensores del robot. Para ello instancia a LinRobot_var, a través de cuyos métodos se podrá leer dicha información sensorial.

Análogamente al cliente desarrollado mediante sockets, la clase **Comando** ha sido desarrollada con el único objetivo de interceptar y manejar las entradas a través del teclado. Por ello observamos en ella instancias a las clases ArKeyHandler, ArFunctor e InfoCliente (ver sección 12.3). Cada vez que el usuario de la aplicación oprima las teclas 'up', 'down', 'right', 'left', 'space' o 's' del teclado, se invocarán métodos de LinRobot_var que se ejecutarán sobre el robot para controlar su movimiento.

La clase **timer** permite incorporar un timer a la aplicación con el objetivo de ejecutar funciones cada cierto periodo de tiempo. En el caso concreto de esta aplicación, cada 400 milisegundos se ejecutará un método para proteger al robot ante las colisiones.

15.4.2. Implementación del código fuente

Tras la generación y compilado de los ficheros idl se lleva a cabo el desarrollo de la aplicación cliente. La misma constará de un sólo fichero, **clienteICa.cpp**.

En la parte inicial del fichero se incluyen todos los ficheros necesarios para el desarrollo: los ficheros stub y los ficheros ICa necesarios para configurar el adaptador de objetos POA, obtener referencias a los objetos y configurar el servicio de nombres.

Después se sitúa la sección de declaración e implementación de las clases mencionadas en la sección anterior. El funcionamiento de la aplicación es el siguiente: la clase **Comando** registra y maneja entradas del usuario a través del teclado. Su implementación es prácticamente igual a la implentación de la clase Comando vista en la sección 12.3. Para el manejo de las entradas a través del teclado se sirve de objetos de las clases de ARIA ArKeyHandler y ArFunctor.

A continuación se muestra qué teclas serán interceptadas y qué ocurrirá cuando sean pulsadas:

- 'Up'. Se invocará el método *LinRobot_var::setVelocity()* para aumentar la ve-

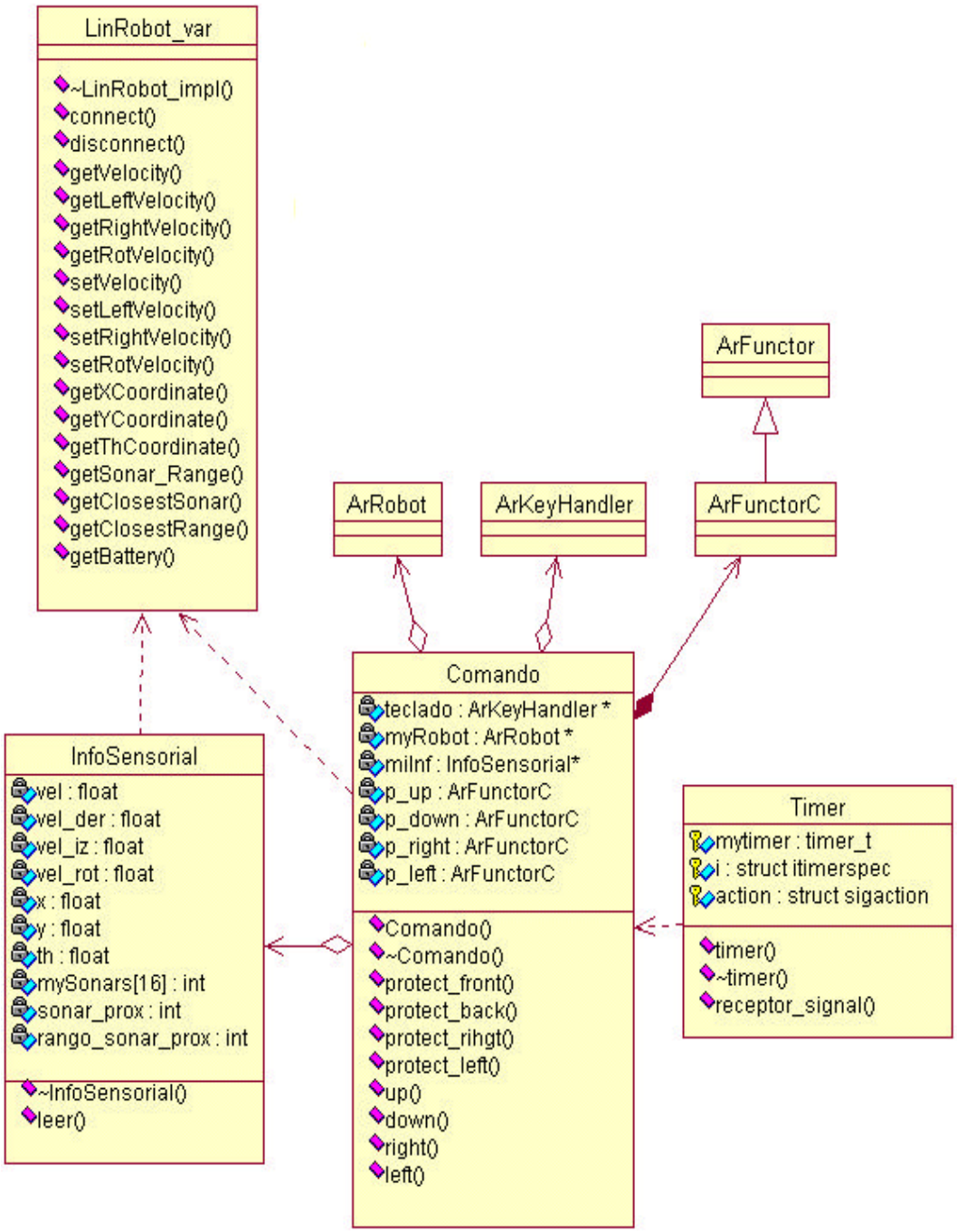

Figura 15.2: Diagrama UML del cliente ICa

locidad del robot.

- 'Down'. Se invocará el método *LinRobot_var::setVelocity()* para disminuir la velocidad del robot.
- 'Right'. Se invocará el método *LinRobot_var::setRotVelocity()* para que el robot gire hacia la derecha.
- 'Left'. Se invocará el método *LinRobot_var::setRotVelocity()* para que el robot gire hacia la izquierda.
- 'Espacio'. Se invocará el método *LinRobot_var::setVelocity()* para que el robot se detenga.
- 's'. Se invocará el método *InfoSensorial::leer()*, que a su vez invoca diversos métodos de LinRobot_var.

Antes de establecer en el robot cualquier velocidad se analiza si se pueden dar colisiones o si el valor a establecer supera el máximo establecido (250 mm/s para la velocidad lineal y 50 mm/s para la velocidad de rotación). Las funciones creadas para la protección del robot son:

```
int Comando::protect_front();
int Comando::protect_back();
int Comando::protect_right();
int Comando::protect_left();
```

Como se ha dicho antes, la clase **timer** permite la instalación de un timer en la aplicación. Al declarar un objeto de la clase timer se podrá configurar cada cuanto tiempo se generará una señal de interrupción del proceso, qué tipo de señal de interrupción se generará un cuándo comenzarán a generarse estas señales. Además se construye la función *void receptor_signal(int signal_id)*, que se ejecutará cada vez que el timer genere la señal de interrupción. La clase timer posee como atributos:

```
timer_t mytimer;
struct itimerspec i;
struct sigaction accion;
```

Todos ellos son tipos de datos definidos por el sistema en los ficheros <time.h>y <signal.h>. Los dos primeros atributos permiten configurar la frecuencia y el tiempo de inicio del timer. El tercero permite configurar qué tipo de señal se generará y qué función gestionará esa señal de interrupción. En la aplicación

cliente se generará la señal SIGALRM cada 400 milisegundos. El timer comenzará a estar operativo a los 5 s. de la declaración de un objeto de la clase timer.

La función *receptor_signal()* llamará a los métodos para proteger al robot ante colisiones implementados en la clase Comando.

La protección del robot se completa con la llamada al método *LinRobot_var :: disconnect()* en los destructores de las clases y dentro de la función *receptor_signal()* cuando la aplicación recibe las señales SIGKILL, SIGTERM O SIGHUP. De este modo, cuanto existe algún problema o se cierra la aplicación el robot se detiene y la aplicación cliente se desconecta de él remotamente.

El resto del fichero es el programa principal. En él se declaran diversas variables y objetos CORBA para obtener referencias al ORB (Object Request Broker) y al POA (Portable Object Adapter), activar el POA manager y obtener un Naming Context para resolver las referencias a los nombres de los objetos (ver capítulo 13 y código fuente).

15.4.3. Compilación y ejecución de la aplicación

Una vez implementado el código fuente, sólo resta compilarlo y enlazarlo con las librerías utilizadas. Para ello se siguen exactamente los mismos pasos que para la aplicación servidor. Se usa incluso el mismo Makefile, cambiando sólo el nombre del fichero ejecutable de servidorICa a **clienteICa** (ver sección 15.3.3).

15.5. Análisis del tiempo de respueta del servidor

Para completar el desarrollo del sistema software basado en ICa, en esta sección se efectuará un estudio de los tiempos de respuesta del servidor cuando le son solicitados diversos servicios.

Para ello se construirá una aplicación cliente que una vez conectada remotamente la robot, comenzará a lanzar threads que solicitarán simultáneamente servicios al servidor. En los mismos subprocesos se realizarán medidas del tiempo de respuesta que ofrece el servidor. Tras ello los datos se escribirán en ficheros de los que se obtendrán diversos gráficos.

En concreto, los servicios que solicita cada thread al servidor con los siguientes:

```
//Lectura remota de los datos sensoriales
void InfoSensorial::leer(){
   vel = miRobot->getVelocity();
   vel_der = miRobot->getRightVelocity();
```

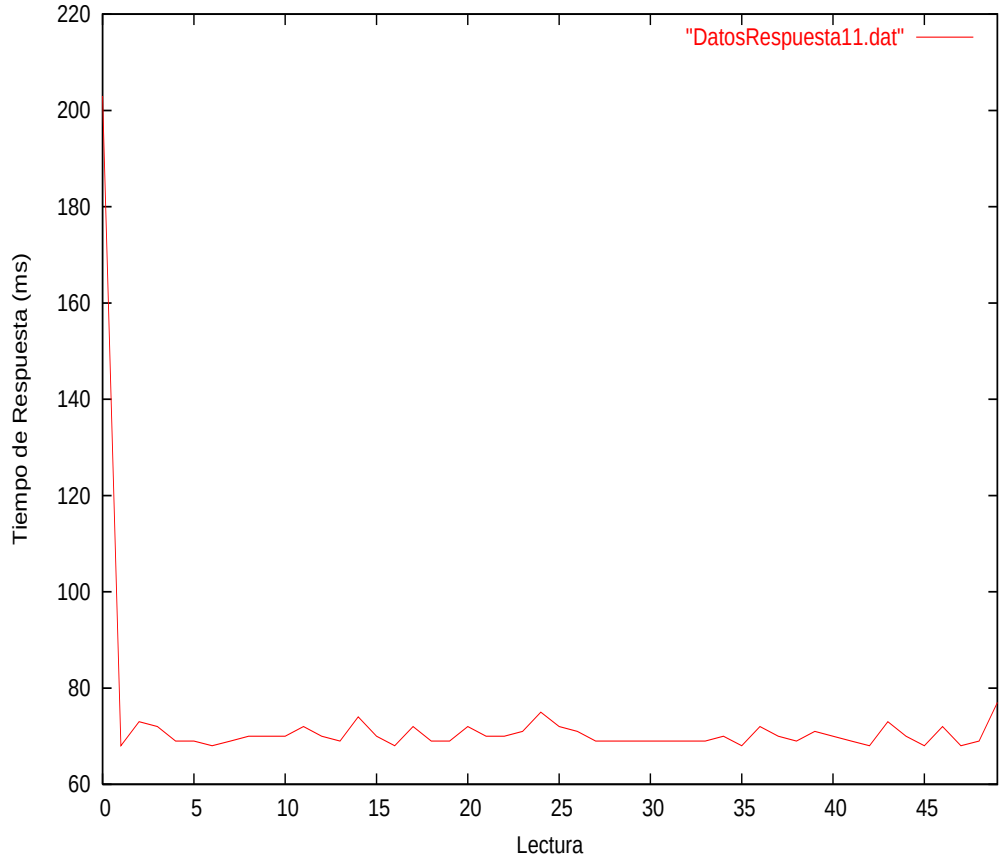

Figura 15.3: Tiempo de respuesta de un sólo cliente

```
   vel_iz = miRobot->getLeftVelocity();
   vel_rot = miRobot->getRotVelocity();

   x = miRobot->getXCoordinate();
   y = miRobot->getYCoordinate();
   th = miRobot->getThCoordinate();

   for (int j=0;j<=15;j++)
  mySonars[j]= miRobot->getSonar_Range(j);
}
```

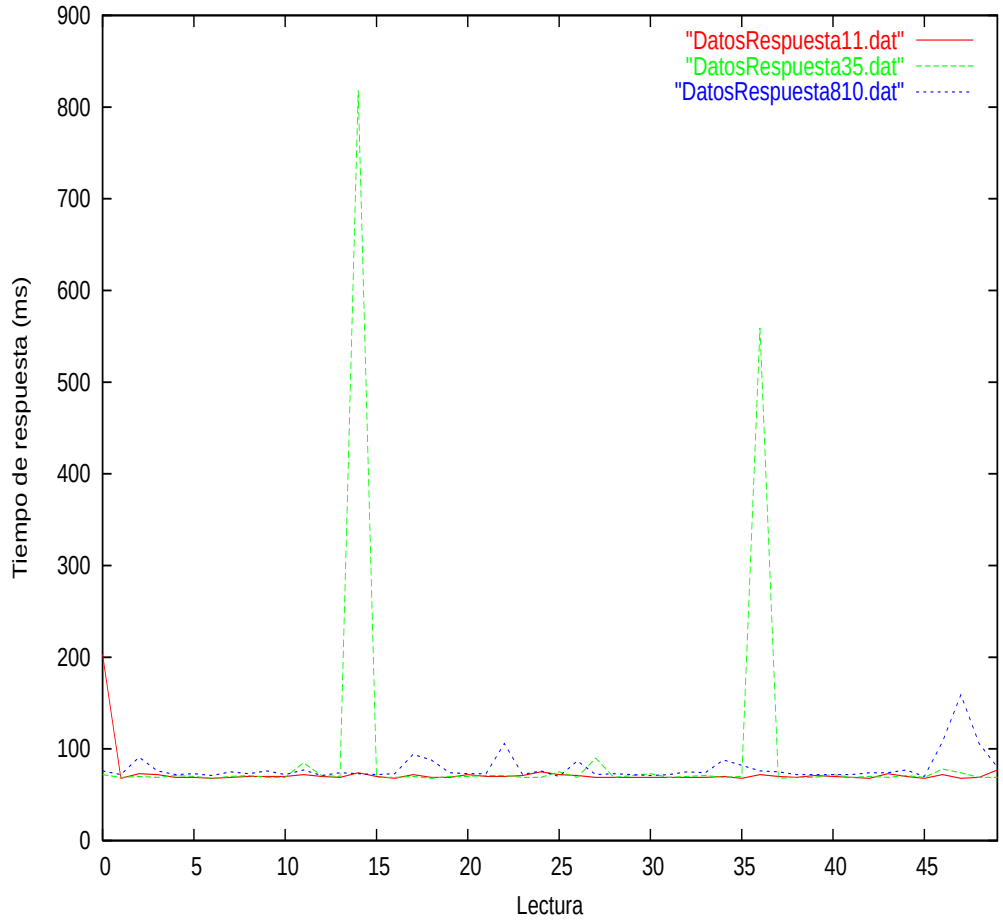

Figura 15.4: Tiempo de respuesta de tres clientes

Cada thread invocará 50 veces a la función *InfoSensorial::leer()* y medirá el tiempo de respuesta de cada lectura, así como el tiempo de respuesta transcurrido desde la primera a la última lectura.

El número de threads que solicitan servicios al servidor simultáneamente se aumenta progresivamente, desde un solo cliente hasta 10 clientes simultáneos.

La figura 15.3 representa el tiempo de respuesta en milisegundos de la serie de 50 lecturas que se realizan cuando una sola aplicación cliente requiere servicios al servidor.

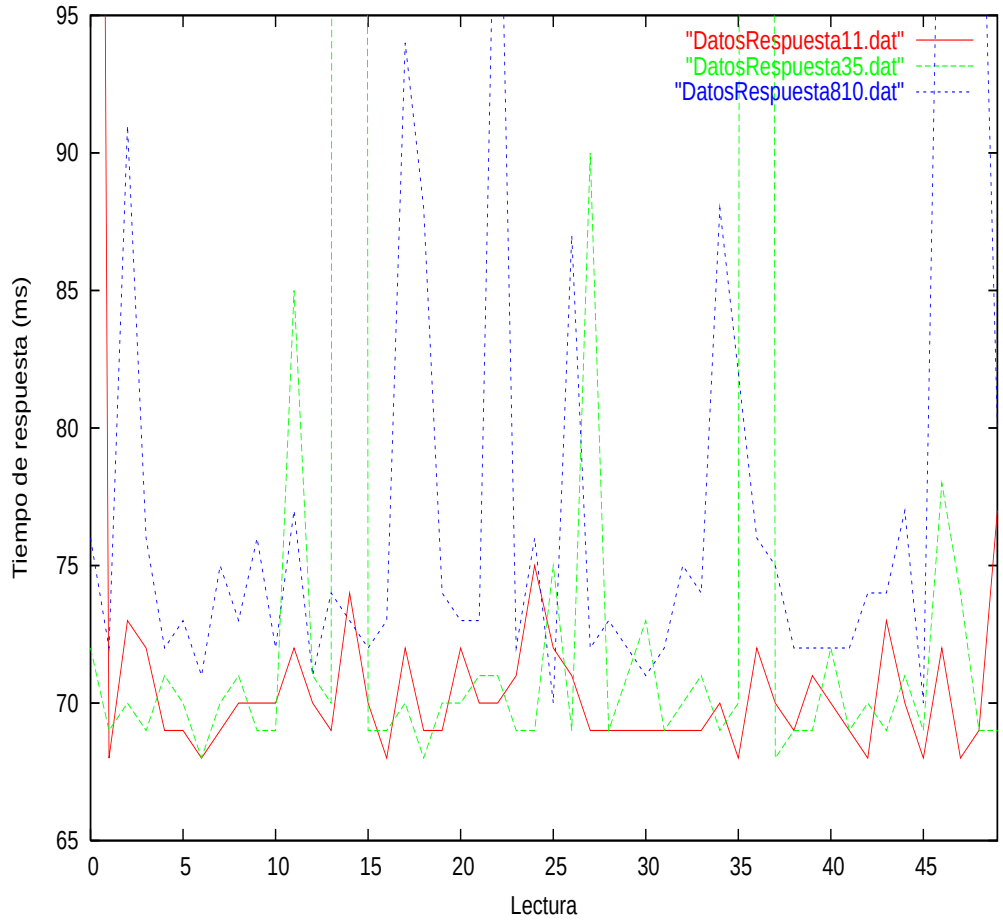

Figura 15.5: Tiempo de respuesta de tres clientes en detalle

Como se puede observar el tiempo de respuesta en las lectura es bastante regular, salvo la excepción de la primera lectura (203 milisegundos). La media de las lecturas es de **72.84 milisegundos**.

En las figuras 15.4 y 15.5 aparecen 3 gráficos que se corresponden con los tiempos de respuesta de tres clientes cuando éstos solicitan servicios al servidor simultáneamente a otros clientes. El primero de ellos solicita servicios individualmente, el segundo de ellos solicita servicios junto con otros 4 clientes simultáneos (5 clientes actuando a la vez) y el tercero de ellos solicita servicios junto a otros 9 clientes (10 clientes actuando a la vez). En concreto el segundo gráfico se corres-

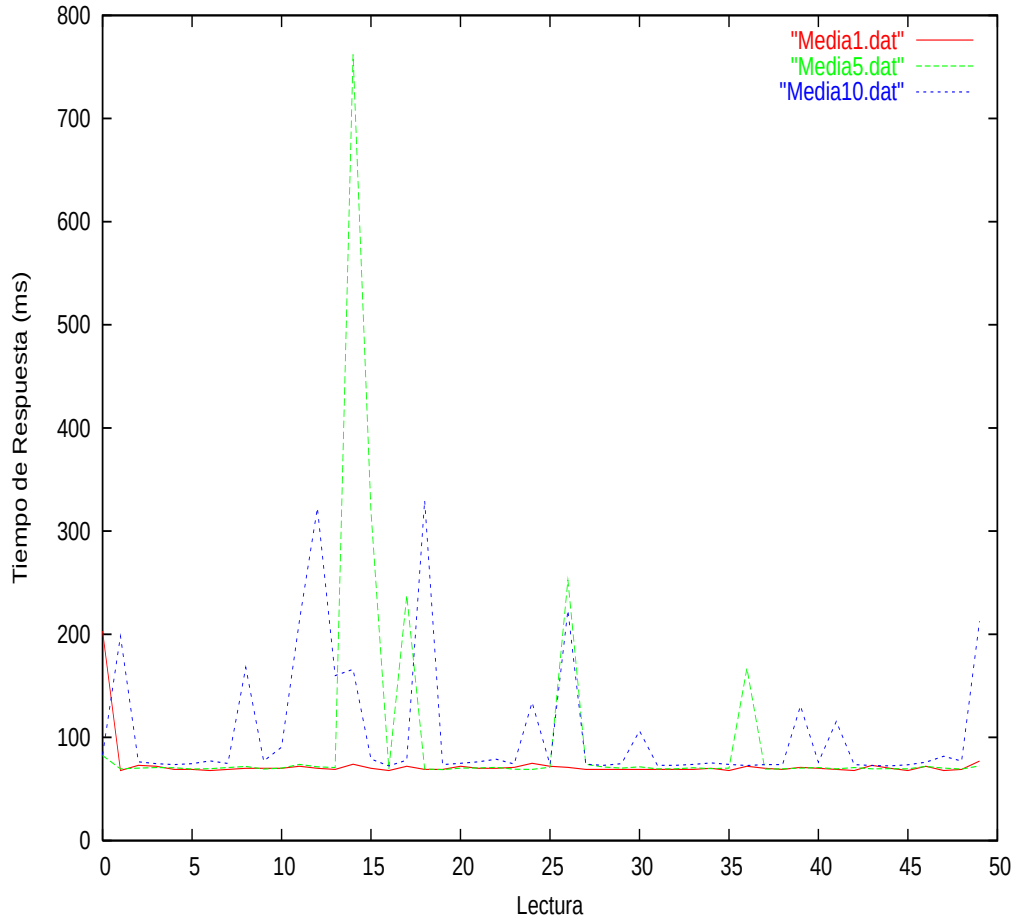

Figura 15.6: Medias de los tiempos de respuesta de tres series de clientes

ponde con el tercero de los 5 clientes que actúan simultáneamente (de ahí el subíndice 35) y el tercer gráfico se corresponde con el octavo de los 10 clientes que actúan simultáneamente (de ahí el subíndice 810).

Al contrario de lo que se pudiera pensar a priori, se observa como los tiempos de respuesta de los diversos clientes son bastante similares. Las medias de las 50 lecturas son **72.84, 95.58 y 78.92 milisegundos** respectivamente. Este hecho no nos da indicaciones concluyentes acerca del comportamiento del servidor.

Por otra parte se observa que según aumenta el número de clientes, existe un

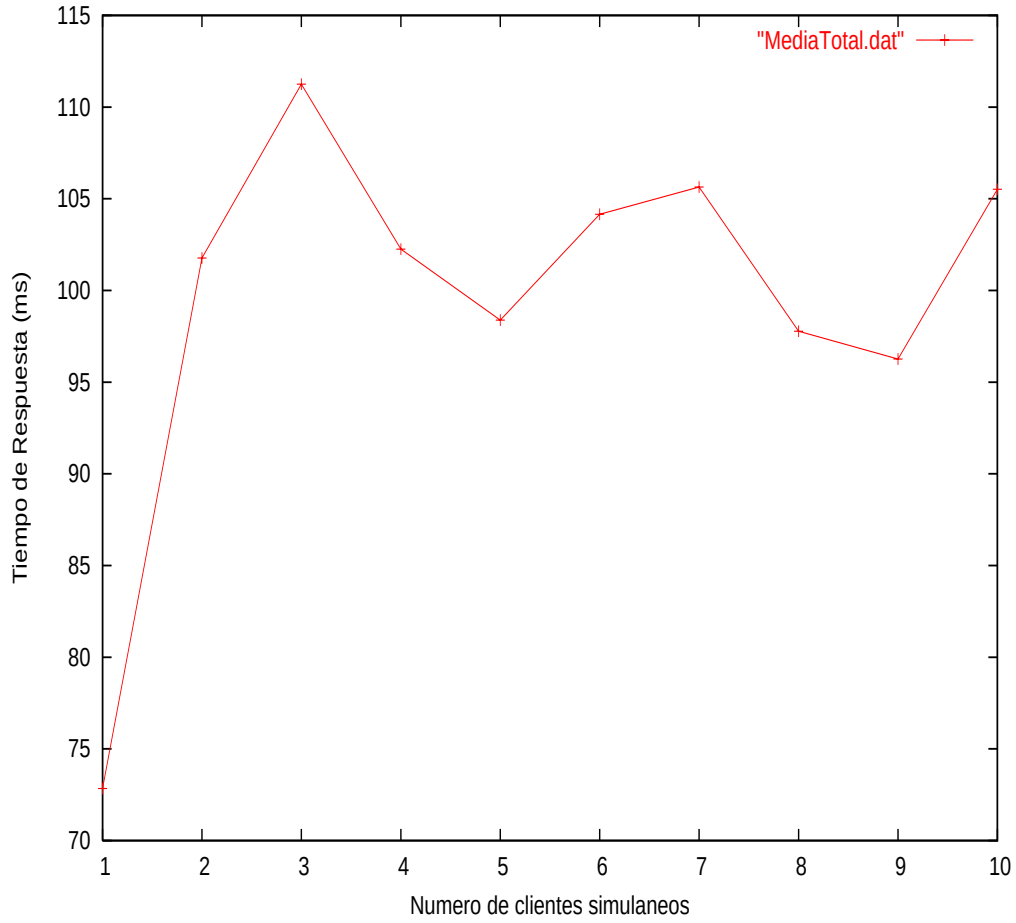

Figura 15.7: Medias de los tiempos de respuesta de todas las lecturas de 10 series de clientes

mayor número de valores dispares (picos en la gráfica). En algunos casos estos valores se disparan a tiempos de respuesta de más de 800 milisegundos. Este hecho se explica por la acumulación de peticiones de los clientes en un determinado momento, que no pueden ser atendidas inmediatamente por el servidor. Sin embargo si analizamos en valor de las varianzas de las series de lectura **356.5, 15647 y 218.85** milisegundos respectivamente este hecho que queda reflejado sólo parcialmente. El hecho de que los valores de las varianzas sean tan dispares se deben a la grande influencia de los picos del tiempo de respuesta ejercen en el análisis

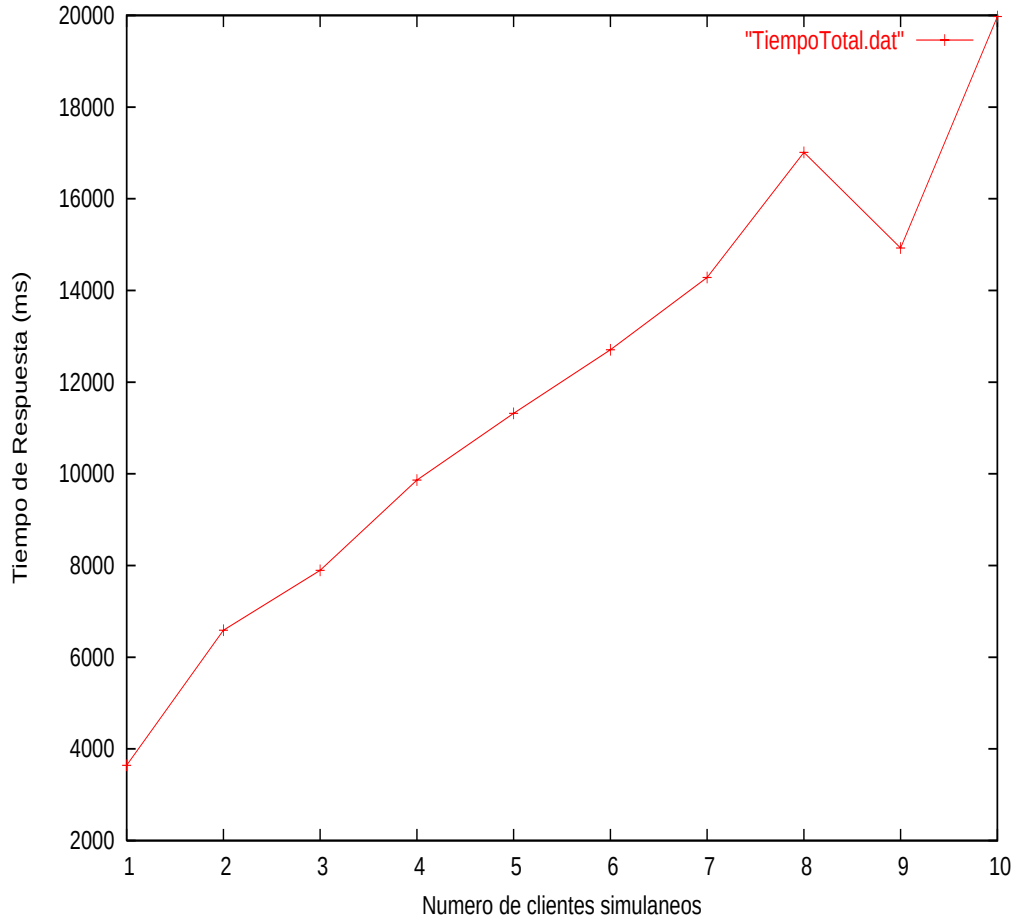

Figura 15.8: Tiempo transcurrido entre 50 lecturas de 10 series de clientes

estadístico.

Al no haber extraído conclusiones importantes, el análisis del tiempo de respuesta del servidor continuará obteniendo las medias de las lecturas de las distintas series de clientes que actúan simultáneamente. La figura 15.6 muestra la media de las lecturas de los tiempos de respuesta cuando actúan uno, cinco y diez clientes simultáneamente. Se sigue observando cualitativamente como, a media que aumenta el número de clientes, existe un mayor número de valores dispares y picos en la gráfica. Además las medias de las lecturas **72.84, 98.38 y 105.52 milisegundos** respectivamente indican como el servidor tarda más en responder

a medida que aumenta el número de clientes. Sin embargo, ese gráfico muestra sólo una tendencia cualitativa y todavía no se pueden obtener conclusiones. Las varianzas **356.5045, 11671 y 3851** también dan indicaciones de la existencia de picos en la respuesta del servidor.

La figura 15.7 muestra la media total de todas las lecturas de las series de clientes que actúan simultáneamente (desde uno a diez clientes requiriendo servicios simultáneamente). De nuevo no se pueden obtener conclusiones claras de este gráfico. Se puede ver de nuevo cómo la media en el tiempo de respuesta no está claramente relacionada con el número de clientes que solicitan servicios simultáneamente. Simplemente se puede concluir que el tiempo de respuesta es mayor cuando existe más de un cliente actuando a la vez.

Hasta el momento no se han obtenido conclusiones claras de las diferentes medidas y análisis. Por ello se ha medido también el tiempo transcurrido entre la primera y la última lectura de los clientes. La figura 15.8 muestra la media del tiempo transcurrido entre la primera y la última lectura de las 10 series de clientes (desde uno hasta diez clientes actuando simultáneamente). En efecto, se observa cómo a medida que aumenta el número de clientes aumenta el tiempo que tarda el servidor en realizar la serie de 50 lecturas. Se pasa de 3.6 s cuanto actúa un sólo cliente a casi 20 s cuanto actúan 10 clientes a la vez.

Como conclusiones a esta sección se puede decir que un servidor ICa a medida que aumenta el número de servicios solicitados ofrece tiempos de respuesta a lecturas sencillas dispersos. Es decir, no se observa en el servidor una tendencia a aumentar el tiempo de respuesta a un servicio solicitado según aumenta el número de solicitudes; el efecto que se observa es una disparidad aleatoria o no uniforme en estos tiempos de respuesta. Esto es, tan pronto se puede medir un tiempo de respuesta en torno a la media total de todas las lecturas (**99.58 milisegundos**), como se puede medir un valor muy dispar a la media anterior, independientemente del número de clientes que esté actuando.

Cuando sí se observa un incremento claro y prácticamente linear del tiempo de respuesta según aumenta el número de servicios solicitados es al medir el tiempo de respuesta entre series de solicitudes.

Parte IV

Apéndices

Apéndice A

Código fuente del sistema software basado en sockets

A.1. Fichero servidorSocket.cpp

```
/*ServidorSocket.cpp

Autor: Iván Pareja Larios

iplarios@alum.etsii.upm.es

Envía al cliente de toda la información sensorial que
proporciona la interfaz eléctrica del robot
Recepcion de las ordenes a los actuadores enviadas por
el cliente
Proteccion antiobstaculos */

#include "Aria.h"

//Clase que lanza un thread que recibe y envia
//informacion del/al  cliente mediante un socket
//Deriva de la clase ArAsyncTask que a su vez deriva de
//la clase  ArThread. Ésta  última es una clase
//envolvente (wrapper) de la libreria pthreads
class infoServidor : public ArASyncTask
{
public:
 //constructor
 infoServidor( ArRobot *robot,int port);
```

```
 //destructor
 ~infoServidor(void);
 void *runThread(void *arg);
 //lee informacin sensorial del robot
 void  leer_inf();
 //Protege ante posibles impactos
 void proteger();
 //Envia comandos al robot
 void movimiento();
 void detener();
 //Funciones para transmitir informacion al cliente:
 //Abre un puerto
 int   abrir(int puerto);
 //Acepta la conexion del cliente
 int   aceptar();
 //Envia la informacion (en concreto envia la variable
 //miembro de la clase datos_robot)
 int   enviar_inf();
 //Recibe las órdenes del cliente (variable act_robot)
 int   recibir();
 //Cierra el socket del servidor
 void  cerrar_servidor();
 //Cierra el socket del cliente
 void  cerrar_cliente();
protected:
//Estructura que describe el estado del robot
//(la información que ofrece su interfaz eléctrica)
struct inf_sen{
//Velocidades
double vel;
double vel_der;
double vel_iz;
double vel_rot;
double vel_max;
double vel_rot_max;
//Posicion respecto a la posicion inicial
double x;
double y;
double th;
//Sonars
int mySonars[16];
int sonar_prox;
int rango_sonar_prox;
```

```
//Resto de informacion
int motor_pac;
int sonar_pac;
bool sonars_en;
bool motors_en;
double volt;
double radio;
double diagonal;
int stall;
bool stall_der;
bool stall_iz;
int flags;
double compass;
unsigned char valor_analogico;
unsigned char estado_in_dig;
unsigned char estado_out_dig;
};
//Estructura
struct inf_act{
double vel;
double vel_der;
double vel_iz;
double vel_rot;
};
inf_sen *datos_robot;
inf_act *act_robot;
//Puntero a un objeto de la clase ArRobot
//Esta clase incorpora como fuciones miembro todas
//aquellas necesarias para la comunicación con el robot.
ArRobot *myRobot;
// Sockets cliente y servidor
    ArSocket serverSock, clientSock;
//Puerto que se usará (sera un parámetro que se pasa
//al constructor de la clase)
int miPuerto;
};

//Sección de implementación de la clase declaradas
//Constructor
infoServidor :: infoServidor(ArRobot *robot,int port):
myRobot(robot),miPuerto(port){
datos_robot=new(inf_sen);
act_robot=new(inf_act);
```

```
}

//Destructor
infoServidor::~infoServidor(void){
delete datos_robot;
delete act_robot;
cerrar_servidor();
}

//Implementacion de las rutinas  en el thread
void * infoServidor::runThread(void *arg){
while (1){
//Acepta la conexion del cliente. Si existe algun error
//de conexion el robot se detiene
//y se vuelve a invocar el thread.
if (aceptar()==(-1)){
detener();
create();
break;
};
//Lee la informacion sensorial
leer_inf();
//Envio de la variable datos_robot
enviar_inf();
//Recibe la variable act_robot con las velocidades
//deseadas para el robot.
//Si existe algun error de conexion el robot se detiene
//y se vuelve a invocar el thread.
if (recibir()==(-1)){
detener();
create();
break;
};
//Proteccion ante posibles impactos
proteger();
//Ejecución de las órdenes del servidor
movimiento();
//Retardo software que permite a los motores alcanzar
//la velocidad deseada
ArUtil::sleep(10);
//Cierre de la conexion con el cliente
cerrar_cliente();
//Comprobacion de la variable que indica si el proceso se
```

```
//debe detener.
//Si es necesario se interrumpe y se relanza el thread
if (!myRunning){
create();
break;
}
}
return (NULL);
}


//Pasa al robot las velocidades deseadas por elcliente
void infoServidor::movimiento(){
myRobot->lock();
myRobot->setVel(act_robot->vel);
myRobot->setRotVel(act_robot->vel_rot);
myRobot->unlock();
}

//Detencion del robot
void infoServidor ::detener(){
myRobot->lock();
myRobot->setVel(0);
myRobot->setRotVel(0);
myRobot->unlock();
}

//Lectura de toda la informacion sensorial
void infoServidor :: leer_inf(){
//Velocidades
datos_robot->vel=myRobot->getVel();
datos_robot->vel_der=myRobot->getLeftVel();
datos_robot->vel_iz=myRobot->getRightVel();
datos_robot->vel_rot=myRobot->getRotVel();
datos_robot->vel_max=myRobot->getMaxTransVel();
datos_robot->vel_rot_max=myRobot->getMaxRotVel();
//Posicion
datos_robot->x=myRobot->getX();
datos_robot->y=myRobot->getY();
datos_robot->th=myRobot->getTh();
//Sonars
datos_robot->sonar_prox=
myRobot->getClosestSonarNumber(0,360);
```

```
datos_robot->rango_sonar_prox=
myRobot->getClosestSonarRange(0,360);
for (int j=0;j<=15;j++)
datos_robot->mySonars[j]= myRobot->getSonarRange(j);
//Resto de informacion
datos_robot->motor_pac =myRobot->getMotorPacCount();
datos_robot->sonar_pac=myRobot->getSonarPacCount();
datos_robot->sonars_en=myRobot->areSonarsEnabled();
datos_robot->motors_en=myRobot->areMotorsEnabled();
datos_robot->volt=myRobot->getBatteryVoltage ();
datos_robot->radio=myRobot->getRobotRadius();
datos_robot->diagonal=myRobot->getRobotDiagonal();
datos_robot->stall=myRobot->getStallValue();
datos_robot->stall=myRobot->isLeftMotorStalled();
datos_robot->stall=myRobot->isRightMotorStalled();
datos_robot->flags=myRobot->getFlags();
datos_robot->compass=myRobot->getCompass();
datos_robot->valor_analogico=myRobot->getAnalog();
datos_robot->estado_in_dig=myRobot->getDigIn();
}

//Funcion para proteger al robot de eventuales impactos
void infoServidor::proteger(void){
if (act_robot->vel>0)
for (int i=1;i<7;i++)
if (datos_robot->mySonars[i]<400)
act_robot->vel=0;
if (act_robot->vel<0)
for (int i=9;i<15;i++)
if (datos_robot->mySonars[i]<400)
act_robot->vel=0;
if ((act_robot->vel_rot>0)&&((datos_robot->mySonars[0]<300)
||(datos_robot->mySonars[1]<300)||
(datos_robot->mySonars[8]<300)
||(datos_robot->mySonars[9]<300)))
act_robot->vel_rot=0;
if ((act_robot->vel_rot < 0)&&((datos_robot->mySonars[6]<300)
||(datos_robot->mySonars[1]<300)
||(datos_robot->mySonars[14]<300)
||(datos_robot->mySonars[15]<300)))
act_robot->vel_rot=0;
}
```

```
//Abrir el puerto al que se conectará el cliente
int infoServidor :: abrir(int puerto){
   if (serverSock.open(puerto, ArSocket::TCP))
     printf("\nAbierto el puerto %i del servidor",miPuerto);
   else{
    printf("\nNo abierto puerto servidor: %s\n",
serverSock.getErrorStr().c_str());
     return(-1);
   }

}

//Aceptar la conexion del cliente
int infoServidor::aceptar(){
   if (serverSock.accept(&clientSock)) ;
   else{
    printf("\nError aceptando la infomacion del cliente");
return(-1);
  }
}

//Enviar la informacion sensorial
int infoServidor ::enviar_inf(){
  if (clientSock.write(datos_robot, sizeof(inf_sen))
  == sizeof(inf_sen));
  else
  {
    printf("\nError enviando la informacin al cliente");
    return(-1);
  }
}

//Recibir informacion
int   infoServidor::recibir(){
//para ver cuantos bytes hemos leido
int b_leidos;
b_leidos=clientSock.read(act_robot, sizeof(inf_act));
if (b_leidos > 0);
else{
printf("\nNo hemos recibido ordenes del cliente");
return(-1);
}
}
```

```
//Cerrar la conexion con el cliente
void infoServidor :: cerrar_cliente(){
 clientSock.close();
}

//Cerrar el puerto del servidor
  void infoServidor :: cerrar_servidor(){
  serverSock.close();
  printf("\nSocket servidor cerrado,puerto:%i",miPuerto);
 }

//Comienza el programa principal
int main(){
// tcp connection (simulador)
   ArTcpConnection tcpConn;
   //para la conexión por el puerto de  serie al robot
  ArSerialConnection serConn;
  // robot
   ArRobot robot(NULL, false,true, true);
  // sonar device, se debe aadir al robot
  ArSonarDevice sonar;
   //Variable de la libreria estanadar de c
   std::string myHost;
   //para saber a que puerto estamos conectados
   int puerto;

   // Se inicia Aria
   Aria::init();
// Se añaden los sonars al robot
   robot.addRangeDevice(&sonar);
   // Vemos si nos podemos conectar al simulador
  tcpConn.setPort();

if (tcpConn.openSimple()) {
     // Nos hemos conectado al simulador
     printf("\nConectandonos al simulador a traves de TCP");
     robot.setDeviceConnection(&tcpConn);
puerto=tcpConn.getPort();
myHost=tcpConn.getHost();
printf("\nEstamos conectados al puerto %i ",puerto);
printf("\nNombre del host: %s",myHost.c_str());
}
```

```
else {
    //Si no nos hemos conectado al simulador, conexión
//al robot a traves del puerto de serie
    serConn.setPort();
    printf("\nNo nos hemos conectado al simulador");
    printf("\nIntentamos conectarnos alrobot a traves
del puerto de serie");
     robot.setDeviceConnection(&serConn);
   }

  // Intentamos conectarnos al robot.
   if (!robot.blockingConnect()){
    printf("\nNo nos pudimos conectar ... saliendo");
   Aria::shutdown();
   return 1;
}

   // Hacemos correr al robot en su propio thread
   robot.runAsync(true);
   //Activamos motores y sonar
   robot.comInt(ArCommands::ENABLE, 1);
   robot.comInt (ArCommands::SONAR, 1);

//Creamos el objeto que permitira el intercambio de
//información con el cliente
   infoServidor inf(&robot,7777);
   inf.abrir(7777);
  inf.create();

   //Ahora simplemente esperamos a que se interrumpa
//la conexion para abandonar el programa
   robot.waitForRunExit();
Aria::shutdown();
   return 0;
}
```

A.2. Fichero clienteSocket.cpp

```
/*clienteSocket.cpp

Autor: Iván Pareja Larios

iplarios@alum.etsii.upm.es

Recibe toda la información sensorial que proporciona el
interfaz eléctrico del robot y envia orden de movimiento
mediante un socket lanzando un thread.
El programa principal (main) se limita a interceptar las
teclas up,down, right, left, space y 's' del teclado.
Cuando éstas son pulsadas se modifica el comando a
transmitir al servidor o bien muestra por pantalla la
informacion sensorial*/

//Incluimos todas las clases de Aria
#include "Aria.h"

/*Clase que lanza un thread que recibe y envia informacion
del/al cliente mediante un socket
Deriva de la clase ArAsyncTask que a su vez deriva de
la clase ArThread. Ésta última es una clase envolvente
(wrapper) de la libreria pthreads*/
class infoCliente : public ArASyncTask
{
public:
//Constructor
 infoCliente(ArMutex *mutex, int puerto);
 //Destructor
 ~infoCliente();
 //Función en la que definirán las tareas que se
 //realizan en el thread
 void * runThread(void *arg);
 //Función que decide los comandos a enviar al servidor
 //Dependiendo de qué tecla ha sido pulsada modificará
 //la velocidad deseada para el robot
 void  control();
 //Conexión con el puerto abierto por el servidor
 int   conectar(int port);
 //Recepción de una estructura con la información sensorial
 int   recibir();
```

```
 //Envia las velocidades deseadas para el robot
 int   enviar ();
 //Función que cierra el socket al servidor
 void  cerrar();
 //entero que sirve para realizar el control de velocidad.
 //Cambiará en función de qué tecla sea pulsada
 int miOrden;
 //Estructura que describe el estado del robot
 //(la información que ofrece su interfaz eléctrica)
struct inf_sen{
//Velocidades
double vel;
double vel_der;
double vel_iz;
double vel_rot;
double vel_max;
double vel_rot_max;
//Posicion respecto a la posicion inicial
double x;
double y;
double th;
//Sonars
int mySonars[16];
int sonar_prox;
int rango_sonar_prox;
//Resto de informacion
int motor_pac;
int sonar_pac;
bool sonars_en;
bool motors_en;
double volt;
double radio;
double diagonal;
int stall;
bool stall_der;
bool stall_iz;
int flags;
double compass;
unsigned char valor_analogico;
unsigned char estado_in_dig;
unsigned char estado_out_dig;
};
inf_sen *datos_robot;
```

```
protected:
//Estructura que será enviada al servidor con las
//velocidades deseadas para el robot
struct inf_act{
double vel;
double vel_der;
double vel_iz;
double vel_rot;
};
inf_act *act_robot;

//Puerto que se usará
int miPuerto;
// Socket cliente
    ArSocket clientSock1;
//Objeto de la clase ArMutex, wrapper de los servicios
//mutex del sistema operativo. Garantiza la exclusividad
//mutua
ArMutex *myMutex;
};

//Clase dedicada a la intercepcion del teclado
class comando {
public:
 //Constructor
 comando(ArRobot *robot, infoCliente *inf);
 //Funciones que seran invocadas al pulsar las teclas:
 //up
 void up();
 //down
 void down();
 //right
 void right();
 //left
 void left();
 //space
 void espacio();
 //s
 void imprime();
protected:
//Objetos que permitira el manejo de las teclas
ArKeyHandler *teclado;
ArRobot *myRobot;
```

```
//Objeto de la clase infoCliente para poder imprimir por
//pantalla los datos sensoriales pertinentes
infoCliente *miInf;

//Declaracion de punteros a los métodos up,down,right,left
ArFunctorC<comando> p_up;
ArFunctorC<comando> p_down;
ArFunctorC<comando> p_right;
ArFunctorC<comando> p_left;
ArFunctorC<comando> p_espacio;
ArFunctorC<comando> p_imprime;
};


//Sección de implementación de las clases
//Constructor
infoCliente::infoCliente (ArMutex *mutex, int puerto):
myMutex(mutex)
{
miPuerto=puerto;
miOrden=0;
datos_robot=new(inf_sen);
act_robot=new(inf_act);
}

//Destructor
infoCliente:: ~infoCliente(){
cerrar();
delete(datos_robot);
delete(act_robot);
}

//Implementacion rutinas que se realizan en el thread
void * infoCliente::runThread(void *arg){
while (1){
//Conexión con el servidor
conectar(miPuerto);
//Recepcion de datos (variable datos_robot)
recibir();
//Control del robot. En funcion de las teclas
//pulsadas se determina la velocidad deseada
control();
//Envio de la variable act_robot con las velocidades
```

```
//deseadas para el robot
enviar();
//Cerramos la conexion
cerrar();
//Comprobacion de la variable que indica si el
//proceso se debe detener. Si es necesario se
//interrumpe el thread y se vuelve a lanzar
if (!myRunning){
create();
break;
}
}
return (NULL);
}

//Control del robot. En función de las teclas pulsadas se
//determina la velocidad deseada
void infoCliente::control(){
act_robot->vel=datos_robot->vel;
act_robot->vel_der=datos_robot->vel_der;
act_robot->vel_iz=datos_robot->vel_iz;
act_robot->vel_rot=datos_robot->vel_rot;
switch (miOrden){
case 1:
//Hacia delante;
act_robot->vel_rot=0;
if (datos_robot->vel<250)
act_robot->vel+=20;
else
act_robot->vel=250;
break;
case 2:
//Hacia atras
act_robot->vel_rot=0;
if (datos_robot->vel>(-250))
act_robot->vel-=20;
else
act_robot->vel=(-250);
break;
case 3:
//Hacia la derecha
act_robot->vel=0;
if (datos_robot->vel_rot>(-50))
```

```
act_robot->vel_rot-=10;
else
act_robot->vel_rot=(-50);
break;
case 4:
//Hacia la izquierda
act_robot->vel=0;
if (datos_robot->vel_rot<50)
act_robot->vel_rot+=10;
else
act_robot->vel_rot=50;
break;
case 5:
//Parar
act_robot->vel=0;
act_robot->vel_rot=0;
break;
case 6:
//Empezar
act_robot->vel=100;
break;
}
}

//Conexión con el puerto abierto en el servidor
int infoCliente ::conectar(int port){
if (clientSock1.connect("localhost", port, ArSocket::TCP))
;
else{
myMutex->lock();
printf("\nNo nos hemos conectado al servidor: %s",
clientSock1.getErrorStr().c_str());
myMutex->unlock();
return(-1);
}
}

//Recepción de la información
int infoCliente :: recibir(){
    int b_recibidos;
   b_recibidos=clientSock1.read(datos_robot,sizeof(inf_sen));
if (b_recibidos > 0);
   else {
```

```
      myMutex->lock();
  printf("\nError esperando la informacion del servidor");
  myMutex->unlock();
      return(-1);
   }
}

//Enviar información
int infoCliente ::enviar (){
if (clientSock1.write(act_robot, sizeof(inf_act))
== sizeof(inf_act));
else{
printf("\nError Enviando orden al servidor\n");
return(-1);
}
}

//Cerrar conexión
void infoCliente :: cerrar(){
clientSock1.close();
}

//Constructor
comando::comando (ArRobot *robot, infoCliente *inf):
//Punteros a las funciones up,down,right,left,
//espacio,imprime
p_up(this,&comando::up),
p_down(this,&comando::down),
p_right(this,&comando::right),
p_left(this,&comando::left),
p_espacio(this,&comando::espacio),
p_imprime(this,&comando::imprime)
{
myRobot=robot;
miInf=inf;
//Instrucciones para la intercepción y manejo del teclado
if ((teclado = Aria::getKeyHandler()) == NULL){
     teclado = new ArKeyHandler;
     Aria::setKeyHandler(teclado);
     myRobot->attachKeyHandler(teclado);
   }

if (!teclado->addKeyHandler(ArKeyHandler::UP,&p_up))
```

```
   printf("\nError");
if (!teclado->addKeyHandler(ArKeyHandler::DOWN,&p_down))
   printf("\nError");
if(!teclado->addKeyHandler(ArKeyHandler::RIGHT,&p_right))
  printf("\nError");
if(!teclado->addKeyHandler(ArKeyHandler::LEFT,&p_left))
   printf("\nError");
if(!teclado->addKeyHandler(ArKeyHandler::SPACE,&p_espacio))
   printf("\nError");
if (!teclado->addKeyHandler('s', &p_imprime))
  printf("\nError");
}

//Funciones que son invocadas cuando se pulsan las teclas
void comando::up(){
miInf->miOrden=1;
}

void comando::down(){
miInf->miOrden=2;
}

void comando::right(){
miInf->miOrden=3;
}

void comando::left(){
miInf->miOrden=4;
}

void comando::espacio(){
miInf->miOrden=5;
}

void comando::imprime(){
printf("\n\n\n");
for (int i=0;i<8; i++)
  printf("Sonar %d: %4d\t",
i+1,miInf->datos_robot->mySonars[i]);
printf("\n");
for (int i=8;i<16; i++)
  printf("Sonar %d:%4d\t",
i+1,miInf->datos_robot->mySonars[i]);
```

```
printf("\n");
printf("\nLa posicion es x: %5f, y: %5f, th: %5f",
miInf->datos_robot->x, miInf->datos_robot->y,
miInf->datos_robot->th);
printf("\nVelocidad de translacion: %5f\t
Velocidad de rotacion: %5f",
miInf->datos_robot->vel,miInf->datos_robot->vel_rot);
}

//Programa principal
int main(){
// Mutex que garantiza exclusividad mutua
    ArMutex mutex;
ArRobot robot;
//Iniciamos Aria
Aria::init();
//Objeto para la comunicacion con el servidor.Puerto 7777
infoCliente inf(&mutex,7777);
//Objeto para el manejo del teclado
comando comand(&robot,&inf);
//Se lanza el thread para la comunicació con el cliente
inf.create();

robot.run(false);
Aria::uninit();
return 0;
}
```

Apéndice B

Código fuente del sistema software basado en ICa

B.1. Fichero interfazcorba.idl

```
module interfaz{

interface LinRobot{

  //Funciones de conexion y desconexion
  long connect();
  void disconnect();

  //Funciones para la obtencion de las diferentes
  //velocidades del robot
  float getVelocity();
  float getLeftVelocity();
  float getRightVelocity();
  float getRotVelocity();

  //Funciones para establecer las diferentes
  //velocidades del robot
  void  setVelocity(in float vel);
  void  setLeftRightVelocity(in float leftvel,
in float rightvel);
  void  setRotVelocity(in float vel);

  //Funcion que mueve el robot a una distancia dada
```

```
  void  moveDistance(in float vel);


  //Funciones para la obtencion de las coordenadas
  //del robot a partir de su posicion inicial
  float getXCoordinate();
  float getYCoordinate();
  float getThCoordinate();

  //Funciones para obtener el rango de los sonar
  long  getSonar_Range(in long num_sonar);
  short getClosestSonar();
  long  getClosestRange();

  //Voltios de las baterias
  float getBattery();

};

};
```

B.2. Fichero servidorICa.cpp

```
/*********************   servidorICa.cpp  ******************

Autor: Iván Pareja Larios

iplarios@alum.etsii.upm.es

Pone a disposción de los clientes los metodos declarados
en el fichero interfazcorba.idl.
Estos métodos permiten la conexión y desconexión remota
del robot, la lectura del rango de los sonar, de las
coordenadas del robot respecto a su posición inicial y del
estado de las baterías. Permiten también establecer en el
robot remotamente su velocidad lineal y de rotación. */

#include "interfazcorbaS.h"
#include "CosNaming.h"
#include <RTPortableServer.h>
#include <RTCORBA.h>
#include <iostream.h>
#include "Aria.h"

//Seccion de implentacion del fichero idl

class LinRobot_impl: public virtual POA_interfaz::LinRobot{
public:
//Objetos ARIA para la comunicación con el robot
//Objeto principal para interactuar con el robot
    ArRobot myRobot;
// tcp connection (simulador)
    ArTcpConnection tcpConn;
    //para la conexion por el puerto de  serie al robot
   ArSerialConnection serConn;
    // sonar device
  ArSonarDevice sonar;

//Destructor
    ~LinRobot_impl()ICA_THROW_DECL (CORBA::SystemException);

//Funciones de conexion y desconexion
    long connect() ICA_THROW_DECL (CORBA::SystemException);
void disconnect()ICA_THROW_DECL (CORBA::SystemException)
```

```
{ myRobot.stopRunning();
      myRobot.disconnect();
};

    //Funciones para la obtencion de las diferentes
//velocidades del robot
    float getVelocity()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getVel());};

    float getLeftVelocity()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getLeftVel());};

    float getRightVelocity()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getRightVel());};

    float getRotVelocity()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getRotVel());};
    //Funciones para establecer las diferentes
//velocidades del robot
    void setVelocity(float vel)
ICA_THROW_DECL ( CORBA::SystemException ){
       myRobot.lock();
       myRobot.setVel(vel);
       myRobot.unlock();
       };

    void setLeftRightVelocity(float leftvel,float rightvel)
ICA_THROW_DECL ( CORBA::SystemException ){
      myRobot.lock();
      myRobot.setVel2(leftvel,rightvel);
      myRobot.unlock();
      };

    void setRotVelocity(float vel)
ICA_THROW_DECL ( CORBA::SystemException ){
      myRobot.lock();
      myRobot.setRotVel(vel);
      myRobot.unlock();
      };
```

```
    //Funcion que mueve el robot a una distancia dada
    void  moveDistance(float vel)
ICA_THROW_DECL ( CORBA::SystemException ){
       myRobot.lock();
       myRobot.move(vel);
       myRobot.unlock();
       };

//Funciones para la obtener las coordenadas del robot
float getXCoordinate()ICA_THROW_DECL
( CORBA::SystemException ){
      return(myRobot.getX());};

float getYCoordinate()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getY());};

float getThCoordinate()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getTh());};

//Funciones para la obtencion del rango de los sonar
long getSonar_Range(long num_sonar)
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getSonarRange(num_sonar));};

short getClosestSonar()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getClosestSonarNumber(0,6.2918));};

long getClosestRange()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getClosestSonarRange(0,6.2918));};

//Voltios de las baterias
float getBattery()
ICA_THROW_DECL ( CORBA::SystemException ){
      return(myRobot.getBatteryVoltage());};
};

//implementacion de la funcion connect()
long LinRobot_impl::connect()
```

```
ICA_THROW_DECL ( CORBA::SystemException ){

// Se anaden los sonars al robot
  myRobot.addRangeDevice(&sonar);
// Vemos si nos podemos conectar al simulador
  tcpConn.setPort();

if (tcpConn.openSimple()) {
    // Nos hemos conectado al simulador.
    //aplicamos robot device connection al simulador
myRobot.setDeviceConnection(&tcpConn);
}
else {
    //Si no nos hemos conectado al simulador nos
//conectamos al robot a traves del puerto de serie
    serConn.setPort();
myRobot.setDeviceConnection(&serConn);
   }

  // Intentamos conectarnos al robot.
//Si falla la conexion se sale
  if (!myRobot.blockingConnect()){
   Aria::shutdown();
   return 1;
}

   // Hacemos correr al robot en su propio thread
   myRobot.runAsync(true);
   //Activamos motores y sonar
   myRobot.comInt(ArCommands::ENABLE, 1);
   myRobot.comInt (ArCommands::SONAR, 1);
}

//Destructor
LinRobot_impl::~LinRobot_impl()
ICA_THROW_DECL ( CORBA::SystemException ){
disconnect();
};

////////////////////////////////////////////////////
//Comienza el programa principal
int main(int arg, char **argv){
Aria::init();
```

```
//Declaracion de objetos CORBA
    RTCORBA::RTORB_var              rtorb;
    CORBA::Object_ptr               nameService;
    CosNaming::NamingContext_var    namingContext;
    CosNaming::Name                 name;
    RTPortableServer::POA_var       rtpoa;
    PortableServer::POAManager_var  mgr;
    interfaz::LinRobot_var          phv;

    //Servante de la clase LinRobot_impl y sus metodos
    LinRobot_impl * robot_servant;

    //Obtención de una referencia al RTORB
cout << "[ servidor app    | main        ]
- Obtaining a RTORB reference...";
    CORBA::ORB_var orb = CORBA::ORB_init(arg, argv);
    CORBA::Object_var obj =
orb->resolve_initial_references( "RTORB" );
    rtorb = RTCORBA::RTORB::_narrow( obj );
    cout << "done." << endl;

    if( CALL_IS_NIL( rtorb.in() ) )
cerr  << "[ crbm app | main ] - ERROR:
Couldn’t get a reference to RTORB"<< endl;

//Obtención de una referencia al Root POA
cout << "[ servidor app | main ] -
Obtaining a Root POA reference...";
CORBA::Object_var obj2 =
rtorb->resolve_initial_references( "RootPOA" );
rtpoa = RTPortableServer::POA::_narrow( obj2 );
cout << "done." << endl;

if( CALL_IS_NIL( rtpoa.in() ) )
cerr  << "[ servidor app | main ] - ERROR:
Couldn’t get a reference to RTPOA"<< endl;

    //Activación del POA Manager
cout << "[ servidor app | main]-Activating POA Manager";
mgr = rtpoa->the_POAManager();
mgr -> activate();
cout << "done." << endl;
```

```
//Espacio en memoria para el servante
    robot_servant = new LinRobot_impl();

//Handler para el servante
    cout << "[servidor | main] - Creating servant handle...";
CORBA::Object_var ph_obj = robot_servant->_this();
phv = interfaz::LinRobot::_narrow( ph_obj );
cout << "done." << endl;

//Configuración del servicio de nombres
cout << "[servidor | main] - Resolving name service";
nameService =
rtorb->resolve_initial_references("NameService");
namingContext =
CosNaming::NamingContext::_narrow( nameService );
cout << "done." << endl;

    name.length(1);
name[0].id=CORBA::string_dup("SERVIDOR_ROBOT_SERVANT");
name[0].kind = CORBA::string_dup( "" );

//Manejo de excepciones
try{
cout << "[ crbm app | main ] - Rebinding name...";
namingContext->rebind( name, phv.in() );
cout << "done." << endl;
    } catch( ... ){
    cerr <<"[crbm | main]-ERROR:cannot rebind servant" << endl;
    }

    //Inicialización del ORB. Todo listo.
cout << "[crbm | main] - ICa ORB initialized." << endl;
    cout << "[crbm | main] - Now running ORB..." << endl;

try{
rtorb->run();
} catch ( ... ) {
cout << "[ crbm | main] -
 ERROR: cannot run ORB." << endl;
}

   return 0;}
```

B.3. Fichero clienteICa.cpp

```
/********************* clienteICa.cp **********************

Autor: Iván Pareja Larios

iplarios@alum.etsii.upm.es

Este cliente teleopera el robot
Incorpora funciones de proteccion ante impactos
Timer que se ejecuta cada 400 ms, analiza la velocidad y
los posibles obstaculos y eventualmente detiene el robot
Conexion remota
Desconexion al acabar el programa*/



#include "interfazcorba.h"
#include "CosNaming.h"
#include <RTPortableServer.h>
#include <unistd.h>
#include <iostream.h>
#include <Aria.h>

//Objeto que permitira la invocacion de metodos remotos
interfaz::LinRobot_var miRobot;

ArMutex mutex;

//Clase para almacenar la informacion sensorial del robot
class InfoSensorial{

public:
//Velocidades
float vel;
float vel_der;
float vel_iz;
float vel_rot;
//Posicion respecto a la posicion inicial
float x;
float y;
float th;
//Sonars
```

```
int mySonars[16];
int sonar_prox;
int rango_sonar_prox;

    ~InfoSensorial(){miRobot->disconnect();};
    void leer();
};

//Clase dedicada a la intercepcion del teclado
class comando{

public:
//Constructores
comando(ArRobot *robot,InfoSensorial *inf);
    ~comando();
//Funciones para progeter el robot ante colisiones
int protect_front();
int protect_back();
int protect_right();
int protect_left();
//Funciones que seran invocadas al pulsar las teclas:
//up
void up();
//down
void down();
//right
void right();
//left
void left();
//space
void espacio();
//s
void imprime();

protected:
//Objetos que permitira el manejo de las teclas
ArKeyHandler *teclado;
ArRobot *myRobot;
//Objeto de la clase infoCliente para poder imprimir
//por pantalla los datos sensoriales pertinentes
InfoSensorial *miInf;
//Declaracion de punteros a los métodos up, down, right,
//left, espacio e imprime
```

```
ArFunctorC<comando> p_up;
ArFunctorC<comando> p_down;
ArFunctorC<comando> p_right;
ArFunctorC<comando> p_left;
ArFunctorC<comando> p_espacio;
ArFunctorC<comando> p_imprime;
};

//Funcion para el manejo de señales
void receptor_signal(int signal_id){
comando comandoObj;
    switch (signal_id){
case SIGUSR1:
cout << "Capturada signal SIGUSR1" << endl;
        break;
//Señal generada por el timer
case SIGALRM:
comandoObj.protect_front();
comandoObj.protect_back();
comandoObj.protect_right();
comandoObj.protect_left();
break;
//Señal kill
case SIGKILL:
cout << "Capturada signal SIGKILL" << endl;
miRobot->disconnect();
break;
//Señal term
case SIGTERM:
cout << "Capturada signal SIGTERM" << endl;
miRobot->disconnect();
break;
//Señal hup (generada cuando se cierra la consola
//desde la que fue lanzada la aplicación)
case SIGHUP:
cout << "Capturada signal SIGTERM" << endl;
miRobot->disconnect();
break;
default:
cout << "No capturada signal " << endl;
}
}
```

```
//Clase que permite la instalacion de un timer
class timer{

protected:
//objetos para instalar el timer y establecer su
//frecuencia
timer_t mytimer;
struct itimerspec i;
//Estructura para el manejo de las signal generadas
//por el timer (u otras)
struct sigaction accion;

public:
//Constructor: toma como parametros la frecuencia
//del timer
timer(long,long);
~timer(){miRobot->disconnect();
};

//Implementacion de la función leer() de la clase
//InfoSensorial.Almacena en variables los datos sensoriales
//que proporciona el robot
void InfoSensorial::leer(){
mutex.lock();
vel = miRobot->getVelocity();
vel_der = miRobot->getRightVelocity();
vel_iz = miRobot->getLeftVelocity();
vel_rot = miRobot->getRotVelocity();
x = miRobot->getXCoordinate();
y = miRobot->getYCoordinate();
th = miRobot->getThCoordinate();
for (int j=0;j<=15;j++)
mySonars[j]= miRobot->getSonar_Range(j);
mutex.unlock();
}

//Implementación de la clase comando
//Constructor
comando::comando (ArRobot *robot, InfoSensorial *inf):
//Punteros a las funciones up, down, right, left,
//espacio,imprime
p_up(this,&comando::up),
p_down(this,&comando::down),
```

```
p_right(this,&comando::right),
p_left(this,&comando::left),
p_espacio(this,&comando::espacio),
p_imprime(this,&comando::imprime){
myRobot=robot;
miInf=inf;
//Instrucciones para hacernos con el control del
//teclado
if ((teclado = Aria::getKeyHandler()) == NULL){
teclado = new ArKeyHandler;
     Aria::setKeyHandler(teclado);
     myRobot->attachKeyHandler(teclado);
   }

    //Instrucciones para asignar punteros a funciones a las
//teclas pulsadas que se quieren controlar
if (!teclado->addKeyHandler(ArKeyHandler::UP, &p_up))
   printf("\nAqui hay un fallo");
if (!teclado->addKeyHandler(ArKeyHandler::DOWN,&p_down))
   printf("\nAqui hay un fallo");
if (!teclado->addKeyHandler(ArKeyHandler::RIGHT,&p_right))
  printf("\nAqui hay un fallo");
if (!teclado->addKeyHandler(ArKeyHandler::LEFT,&p_left))
   printf("\nAqui hay un fallo");
if (!teclado->addKeyHandler(ArKeyHandler::SPACE,&p_espacio))
   printf("\nAqui hay un fallo");
if (!teclado->addKeyHandler('s', &p_imprime))
printf("\nAqui hay un fallo");
}

comando::~comando(){
miRobot->disconnect();
}

//Protege ante impacto frontal
int comando::protect_front(){
mutex.lock();
if (( miRobot->getVelocity())>=0){
for (int i=1;i<7;i++) {
if (miRobot->getSonar_Range(i)<400){
miRobot->setVelocity(0);
               mutex.unlock();
               return (-1);
```

```
                }
          }
mutex.unlock();
return 1;
}
    else{
mutex.unlock();
        return 1;
}
}

//Protege ante impacto trasero
int comando::protect_back(){
mutex.lock();
if (( miRobot->getVelocity())<=0){
for (int i=9;i<15;i++) {
if (miRobot->getSonar_Range(i)<400){
miRobot->setVelocity(0);
                mutex.unlock();
return (-1);
            }
}
mutex.unlock();
return 1;
}
else{
mutex.unlock();
return 1;
    }
}

//Protege ante impacto al girar a la derecha
int comando::protect_right() {
mutex.lock();
    if (miRobot->getRotVelocity()<0){
if ((miRobot->getSonar_Range(6)<300)
||(miRobot->getSonar_Range(1)<300)
||(miRobot->getSonar_Range(14)<300)
||(miRobot->getSonar_Range(15)<300)){
miRobot->setRotVelocity(0);
mutex.unlock();
return (-1);
        }
```

```
        mutex.unlock();
        return 1;
     }
     else{
mutex.unlock();
        return 1;
}
}

//Protege ante impacto al girar a la izquierda
int comando::protect_left() {
mutex.lock();
    if (miRobot->getRotVelocity()>0){
        if ((miRobot->getSonar_Range(0)<300)
||(miRobot->getSonar_Range(1)<300)
||(miRobot->getSonar_Range(8)<300)
||(miRobot->getSonar_Range(9)<300)){
miRobot->setRotVelocity(0);
mutex.unlock();
return (-1);
        }
        mutex.unlock();
        return 1;
}
    else{
        mutex.unlock();
        return 1;
    }
}

//Funciones llamadas cunando son pulsadas las teclas
//Aumenta, si procede, la velocidad linear
void comando::up(){
mutex.lock();
    miInf->vel = miRobot->getVelocity();
    mutex.unlock();
if ((miInf->vel)<250)
miInf->vel+=20;
else
miInf->vel=250;
mutex.lock();
miRobot->setRotVelocity(0);
miRobot->setVelocity(miInf->vel);
```

```
    mutex.unlock();
}

//Disminuye, si procede, la velocidad linear
void comando::down(){
mutex.lock();
    miInf->vel = miRobot->getVelocity();
    mutex.unlock();
    if ((miInf->vel)>(-250))
miInf->vel-=20;
else
miInf->vel=(-250);
mutex.lock();
miRobot->setRotVelocity(0);
miRobot->setVelocity(miInf->vel);
    mutex.unlock();
}

//Varía, si procede, la velocidad de rotación
void comando::right(){
mutex.lock();
    miInf->vel_rot=miRobot->getRotVelocity();
    mutex.unlock();
    if ( miInf->vel_rot>(-50))
miInf->vel_rot-=10;
else
miInf->vel_rot=(-50);
mutex.lock();
miRobot->setRotVelocity(miInf->vel_rot);
miRobot->setVelocity(0);
    mutex.unlock();
}

//Varía, si procede, la velocidad de rotación
void comando::left(){
mutex.lock();
    miInf->vel_rot=miRobot->getRotVelocity();
    mutex.unlock();
if ( miInf->vel_rot<50)
miInf->vel_rot+=10;
else
miInf->vel_rot=50;
mutex.lock();
```

```
    miRobot->setRotVelocity(miInf->vel_rot);
miRobot->setVelocity(0);
    mutex.unlock();
}

//Detiene el robot
void comando::espacio(){
mutex.lock();
    miRobot->setRotVelocity(0);
miRobot->setVelocity(0);
    mutex.unlock();
}

void comando::imprime(){
miInf->leer();
cout << "\n";
cout << "La velocidad es" << miInf->vel << endl;
cout << "La velocidade rotacion es" << miInf->vel_rot
<< endl;
cout << "La velocidade de la rueda derecha es"
<< miInf->vel_der << endl;
cout << "La velocidade de la rueda izquierda es"
<< miInf->vel_iz << endl;
cout << "\n";
cout << "La coordenada x es" << miInf->x << endl;
cout << "La coordenada y es" << miInf->y << endl;
cout << "La coordenada th es" << miInf->th << endl;
cout << "\n";
for(int i=0;i<16;i++)
cout << "Rango del sonar " << i << ":"
<< miInf->mySonars[i] << endl;
}

//Implementacion de la clase timer
timer::timer(long sec,long nsec){
//Configuracion manejo signals
sigemptyset(&accion.sa_mask);
accion.sa_handler= receptor_signal;
    //real time signal
accion.sa_flags=SA_SIGINFO;
//La señal generada por el timer sera SIGALRM
sigaction(SIGALRM, &accion, NULL);
//Además se manejan SIGTERM, SIGKILL y SIGHUP
```

```
sigaction(SIGTERM, &accion, NULL);
sigaction(SIGKILL, &accion, NULL);
sigaction(SIGHUP, &accion, NULL);

//Configuracion timer: frecuencia=nsec. t_inicio=tv_sec
i.it_interval.tv_sec=sec;
i.it_interval.tv_nsec=nsec;
i.it_value.tv_sec=5;
i.it_value.tv_nsec=0;

//Instalación del timer
if ((timer_create(CLOCK_REALTIME, NULL, &mytimer))<0)
perror("timer_create");
    if ((timer_settime(mytimer,0, &i, NULL))<0)
perror("setitimer");
}

////////////////////////////////////////////////////////
//Programa principal

int main(int argc, char** argv){

    //Declaracion de objetos CORBA
RTCORBA::RTORB_var              rtorb;
    CORBA::Object_ptr               nameService;
    CosNaming::NamingContext_var    namingContext;
    CosNaming::Name                 name;
    RTPortableServer::POA_var       rtpoa;
    PortableServer::POAManager_var  mgr;

    //Obtención de una referencia al RTORB
    cout << "[client | main]-Obtaining a RTORB reference";
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj =
orb->resolve_initial_references( "RTORB" );
    rtorb = RTCORBA::RTORB::_narrow( obj );
    cout << "done." << endl;

    if( CALL_IS_NIL( rtorb.in() ) )
cerr  << "[client | main] - ERROR:
Couldn't get a reference to RTORB"<< endl;

    //Obtención de una referencia al Root POA
```

```
cout <<"[client | main]-Obtaining a Root POA reference";
CORBA::Object_var obj2 =
rtorb->resolve_initial_references( "RootPOA" );
rtpoa = RTPortableServer::POA::_narrow( obj2 );
cout << "done." << endl;

if( CALL_IS_NIL( rtpoa.in() ) )
cerr  << "[ phtest | main] - ERROR:
Couldn’t get a reference to RTPOA"<< endl;

    //Activación del POA Manager
cout << "[ phtest | main] - Activating POA Manager.";
mgr = rtpoa->the_POAManager();
mgr -> activate();
cout << "done." << endl;

   //Configuración del servicio de nombres
cout << "[ phtest | main] - Resolving name service.";
nameService =
rtorb->resolve_initial_references("NameService");
namingContext=CosNaming::NamingContext::_narrow(nameService);
cout << "done." << endl;

    CosNaming::Name robot_name;

    robot_name.length(1);
    robot_name[0].id =
CORBA::string_dup( "SERVIDOR_ROBOT_SERVANT" );
    robot_name[0].kind = CORBA::string_dup( "" );

cout << "[ client | main ] - Resolving servant name.";
    CORBA::Object_var object_robot =
namingContext->resolve( robot_name );
    cout << "done." << endl;

    //Obtneción de un handler para el servante
    cout << "[client | main]-Obtaining handle for servant.";
miRobot = interfaz::LinRobot::_narrow( object_robot );
cout << "done." << endl;

//Conexion remota con el robot
    miRobot->disconnect();
    miRobot->connect();
```

```
    //Timer cada 400 ms
    timer miTimer(0,40000000);

    //Iniciamos Aria
Aria::init();
//Objeto para el control del teclado
    comando comand(&robot,&informacion);
    cout << "\nPrograma de teleoperacion de LIN" << endl;
    //Bloqueo del programa
robot.run(false);
Aria::uninit();
    return 0;
}
```

B.4. Fichero clienteICaTempo.cpp

```
/************** Robot ICa client B3Tempo ******************

Autor: Iván Pareja Larios

iplarios@alum.etsii.upm.es

Este cliente medira el tiempo de respuesta al leer los datos
sensoriales.
Se escriba el tiempo de respuesta en ficheros
DatosRespuestaij.dat donde i sera el numero de proceso
cliente y j el numero de clientes que esta requiriendo
datos simultanemamente*/

#include "interfazcorba.h"
#include "CosNaming.h"
#include <RTPortableServer.h>
#include <unistd.h>
#include <iostream.h>
#include <Aria.h>
#include <sys/timeb.h>

//Constante que indica el número de lecturas
const int K = 50;

//Objeto que permitira la invocacion de metodos remotos
interfaz::LinRobot_var miRobot;
//Contadores
int cont1=0;
int cont2=0;
int cont_thread=1;
//Para evitar concurrencias
ArMutex mutex;

//Clase para almacenar la informacion sensorial del robot
class InfoSensorial{

public:
//Velocidades
float vel;
float vel_der;
float vel_iz;
```

```
float vel_rot;
//Posicion respecto a la posicion inicial
float x;
float y;
float th;
//Sonars
int mySonars[16];
int sonar_prox;
int rango_sonar_prox;

    ~InfoSensorial(){miRobot->disconnect();};
    void leer();
};

//Implementacion de la función leer() de la clase
//InfoSensorial.Almacena en variables los datos sensoriales
//que proporciona el robot
void InfoSensorial::leer(){
mutex.lock();
vel = miRobot->getVelocity();
vel_der = miRobot->getRightVelocity();
vel_iz = miRobot->getLeftVelocity();
vel_rot = miRobot->getRotVelocity();
x = miRobot->getXCoordinate();
y = miRobot->getYCoordinate();
th = miRobot->getThCoordinate();
for (int j=0;j<=15;j++)
mySonars[j]= miRobot->getSonar_Range(j);
mutex.unlock();
}


//Clase para lanzar threads
class lanzarThread : public ArASyncTask{
public:
//Función en la que definirán las tareas que se
//realizan en el thread
void * runThread(void *arg);
};

//Implementacion lanzarThread::runThread()
//Se lanzaran desde 1 a 10 threads simultaneos que solicitan
//servicios al servidro
```

```
void * lanzarThread::runThread(void *arg){
//Contadores para lanzar los threads
int cont_interno1;
    int cont_interno_thread;
cont_interno1=cont1;
mutex.lock();
    cont_interno_thread=cont_thread;
    mutex.unlock();
//Actualizacion de contadores
if(cont_thread>=cont_interno1){
      cont_thread=1;
    }
    else
      cont_thread++;

   //Objeto para solicitar los datos al servidor
   InfoSensorial informacion;
   //Variables para medir el tiempo de respuesta
   struct timeb tp, tp_pasado, tp_total, tp_total_pasado;
   time_t *tpS=new(time_t);
   time_t *tpS_pasado=new(time_t);
   //Variables para almacenar el tiempo de respuesta
   short ms[K];
   int tempo;

   //Llamadas a funciones que devuelven el tiempo actual en
   //s y ms
   time(tpS_pasado);
   ftime(&tp_total_pasado);

   //Comienza el ciclo de K lecturas
   for(int i=0;i<K;i++){
//Llamadas a funciones que devuelven el tiempo actual
ftime(&tp_pasado);
mutex.lock();
//Lectura datos
informacion.leer();
mutex.unlock();
//Llamadas a funciones que devuelven el tiempo actual
ftime(&tp);

//Tiempo de respuesta de la lectura
if (tp.time > tp_pasado.time)
```

```
ms[i]=tp.millitm -  tp_pasado.millitm + 1000;
else
ms[i]=tp.millitm -  tp_pasado.millitm;
}
    ftime(&tp_total);
    time(tpS);
    //Tiempo de respuesta de la serie de lecturas
tempo= 1000*((*tpS)-(*tpS_pasado))+tp_total.millitm -
tp_total_pasado.millitm;

//Se crea el nombre del fichero: DatosRespuestaij.dat
char cadena[30]="DatosRespuesta";
    char *extension=".dat";
char numero1[10];
    char numero2[10];
    sprintf(numero1,"%i",cont_interno1);
    sprintf(numero2,"%i",cont_interno_thread);
    strcat(cadena,numero2);
    strcat(cadena,numero1);
    strcat(cadena,extension);

//Se registran los datos en un fichero
cout << "Escribiendo en fichero "
< cont_interno_thread << cont_interno1  << endl;
FILE *fich;

if((fich=fopen(cadena,"w+"))==NULL)
cout << "Error al abrir el fichero "
<< cont_interno_thread << cont_interno1 << endl;

for(int i=0;i<K;i++)
       fprintf(fich,"%i %i\n",i,ms[i]);

//El último registro del fichero es el tiempo de
//respuesta de la serie de lecturas
    fprintf(fich,"%i %i\n",50,tempo);

    if(fclose(fich))
       cout << "Error cerrando el fichero "
   << cont_interno_thread << cont_interno1 << endl;
     else
       cout << "Fichero " << cont_interno_thread
   << cont_interno1  <<  " cerrado con exito" << endl;
```

```
}

//Clases para manejar las señales kill, term y hup, de modo
//que cuando se cierre la terminal desde la que se lanzo la
//aplicación nos desconectemos remotamente del robot.
//Funcion para el manejo de señales
void receptor_signal(int signal_id){
comando comandoObj;
    switch (signal_id){
case SIGUSR1:
cout << "Capturada signal SIGUSR1" << endl;
break;
//Señal generada por el timer
case SIGALRM:
break;
//Señal kill
case SIGKILL:
cout << "Capturada signal SIGKILL" << endl;
miRobot->disconnect();
break;
//Señal term
case SIGTERM:
cout << "Capturada signal SIGTERM" << endl;
miRobot->disconnect();
break;
//Señal hup (generada cuando se cierra la consola
//desde la que fue lanzada la aplicación)
case SIGHUP:
cout << "Capturada signal SIGTERM" << endl;
miRobot->disconnect();
break;
default:
cout << "No capturada signal " << endl;
}
}

//Clase que permite la instalacion de un timer
class timer{

protected:
//objetos para instalar el timer y establecer su
//frecuencia
timer_t mytimer;
```

```
struct itimerspec i;
//Estructura para el manejo de las signal generadas
//por el timer (u otras)
struct sigaction accion;

public:
//Constructor: toma como parametros la frecuencia
//del timer
timer(long,long);
~timer(){miRobot->disconnect();
};

//Implementacion de la clase timer
timer::timer(long sec,long nsec){
//Configuracion manejo signals
sigemptyset(&accion.sa_mask);
accion.sa_handler= receptor_signal;
    //real time signal
accion.sa_flags=SA_SIGINFO;
//La señal generada por el timer sera SIGALRM
sigaction(SIGALRM, &accion, NULL);
//Además se manejan SIGTERM, SIGKILL y SIGHUP
sigaction(SIGTERM, &accion, NULL);
sigaction(SIGKILL, &accion, NULL);
sigaction(SIGHUP, &accion, NULL);

//Configuracion timer: frecuencia=nsec. t_inicio=tv_sec
i.it_interval.tv_sec=sec;
i.it_interval.tv_nsec=nsec;
i.it_value.tv_sec=5;
i.it_value.tv_nsec=0;

//Instalación del timer
if ((timer_create(CLOCK_REALTIME, NULL, &mytimer))<0)
perror("timer_create");
    if ((timer_settime(mytimer,0, &i, NULL))<0)
perror("setitimer");
}



//////////////////////////////////////////////////////
```

```
//Programa principal

int main(int argc, char** argv){
     //Declaracion de objetos CORBA
RTCORBA::RTORB_var              rtorb;
    CORBA::Object_ptr               nameService;
    CosNaming::NamingContext_var    namingContext;
    CosNaming::Name                 name;
    RTPortableServer::POA_var       rtpoa;
    PortableServer::POAManager_var  mgr;

    //Obtención de una referencia al RTORB
    cout << "[client | main]-Obtaining a RTORB reference";
    CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
    CORBA::Object_var obj =
orb->resolve_initial_references( "RTORB" );
    rtorb = RTCORBA::RTORB::_narrow( obj );
    cout << "done." << endl;

    if( CALL_IS_NIL( rtorb.in() ) )
cerr  << "[client | main] - ERROR:
Couldn't get a reference to RTORB"<< endl;

    //Obtención de una referencia al Root POA
cout <<"[client | main]-Obtaining a Root POA reference";
CORBA::Object_var obj2 =
rtorb->resolve_initial_references( "RootPOA" );
rtpoa = RTPortableServer::POA::_narrow( obj2 );
cout << "done." << endl;

if( CALL_IS_NIL( rtpoa.in() ) )
cerr  << "[ phtest | main] - ERROR:
Couldn't get a reference to RTPOA"<< endl;

    //Activación del POA Manager
cout << "[ phtest | main] - Activating POA Manager.";
mgr = rtpoa->the_POAManager();
mgr -> activate();
cout << "done." << endl;

   //Configuración del servicio de nombres
cout << "[ phtest | main] - Resolving name service.";
nameService =
```

```
rtorb->resolve_initial_references("NameService");
namingContext=CosNaming::
NamingContext::_narrow(nameService);
cout << "done." << endl;

    CosNaming::Name robot_name;

    robot_name.length(1);
    robot_name[0].id =
CORBA::string_dup( "SERVIDOR_ROBOT_SERVANT" );
    robot_name[0].kind = CORBA::string_dup( "" );

cout << "[ client | main ] - Resolving servant name.";
    CORBA::Object_var object_robot =
namingContext->resolve( robot_name );
    cout << "done." << endl;

    //Obtneción de un handler para el servante
    cout << "[client | main]-Obtaining handle for servant.";
miRobot = interfaz::LinRobot::_narrow( object_robot );
cout << "done." << endl;

//Conexion remota con el robot
    miRobot->disconnect();
    miRobot->connect();

//Lanzamos los threads (desde 1 a 10 threads a la vez)
for (cont1=1;cont1<11;cont1++){
cout << "Ciclo " << cont1 << endl;
for (cont2=1;cont2<=cont1;cont2++)
lanzar.create();
sleep(5*cont1);
cout << "Terminado ciclo " << cont1 << endl;
    }

    //Bloqueo del programa
while(1);
    return 0;
}
```

B.5. Fichero Makefile

```
# ICa options
MOTIF=NO
ICA=YES
DEBUG=NO

#########################################################
# Application options
# Name of the executable file
TARGET= /home/ivanesguay/bin/ServidorRobotB2

# Library and header directories----------------------

ICA_INCDIR=/usr/local/ICa/include
SYSTEM=Linux
ICA_LIBDIR=/usr/local/ICa/lib/Linux/i386

CORBA_INCLUDE=./Corba
#para incluir los ficheros y librerias de Aria
ARIA_INCLUDE=/usr/local/Aria/include
ARIA_LIB=/usr/local/Aria/lib

#########################################################
# Binutils


# Shell -----------------------------------------------
SHELL = /bin/sh

# Yacc ------------------------------------------------
YACC = yacc
YFLAGS = -l

# Ar --------------------------------------------------
AR = ar
ARFLAGS = -rvc

# Make ------------------------------------------------
MAKE = make
LESS = -M

# QT Tools --------------------------------------------
```

```
MOC      = $(QTDIR)/bin/moc
UIC      = $(QTDIR)/bin/uic

############################################################
# CC Compiler
#
CC = gcc -Wno-deprecated -w


# ICA FLAGS -------------------------------------------
CFLAGS = -I$(ICA_INCDIR) -D$(SYSTEM) -DICA_RT -DICA_MINRT
CFLAGS := -fexceptions

#ARIA FLAGS
CFLAGS :=  $(CFLAGS)  -I$(ARIA_INCLUDE)

# QT FLAGS --------------------------------------------
#CFLAGS :=  $(CFLAGS)  -I$(QTDIR)/include

# OTHER FLAGS -----------------------------------------
CFLAGS :=  $(CFLAGS)  -I/usr/X11R6/include

# CORBA IDL GENERATED FILES ---------------------------
CFLAGS :=  $(CFLAGS)  -I$(CORBA_INCLUDE)


############################################################
# CPP COMPILER
#
CPP = g++ -Wno-deprecated -w -O3

# CPP FLAGS -------------------------------------------
CPPFLAGS = $(CFLAGS)


############################################################
# LINKER
#
LD = g++

# LD FLAGS --------------------------------------------
LDFLAGS = -L$(ICA_LIBDIR)
LDFLAGS := $(LDFLAGS) -L/usr/X11R6/lib
```

```
LDFLAGS := $(LDFLAGS) -L./lib
LDFLAGS := $(LDFLAGS) -L$(ARIA_LIB)

#########################################################
# LIBRARIES
#


LIBS := $(LIBS) -lICa-1.0.1
LIBS := $(LIBS)  -lrt -lnsl -lpthread
LIBS := $(LIBS) -lXext -lXpm
LIBS := $(LIBS) -lAria


#########################################################
# MAKE RULES
#
SRCS = $(wildcard *.cpp)
OBJS = $(patsubst %.cpp, obj/%.o, $(wildcard *.cpp) )

SRCS_C = $(wildcard Corba/*.cpp)
OBJS_C = $(patsubst Corba/%.cpp,obj/Corba/%.o,$(SRCS_C))


SRCS_A = $(wildcard src/*.cpp)
OBJS_A = $(patsubst src/%.cpp, obj/src/%.o, $(SRCS_A) )

ifeq (obj/.depend,$(wildcard obj/.depend))
 # $(TARGET): $(OBJS) $(OBJS_C)  $(OBJS_A)
   $(TARGET): $(OBJS) $(OBJS_C)
  include obj/.depend
endif


#########################################################
# Compilation rules
#

%.c : %.yac
$(YACC) $(YFLAGS) $<
mv -f y.tab.c $@

obj/%.o: %.cpp
```

```
$(CPP) $(CPPFLAGS)  -o $@ -c $<

obj/Corba/%.o: Corba/%.cpp
$(CPP) $(CPPFLAGS)  -o $@ -c $<

obj/src/%.o: src/%.cpp
$(CPP) $(CPPFLAGS)  -o $@ -c $<

%.a : $(OBJS)
$(AR) $(ARFLAGS) $@ $^
ranlib $@

ifeq (include,$(wildcard include))
$(MAKE) -C include
endif


############################################################
# Linking rules
#

% :
$(LD) $(CFLAGS) $(LDFLAGS) -o $@ $^  $(LIBS)

ifeq (include,$(wildcard include))
$(MAKE) -C include
endif


############################################################
# Dependencies
#
dep :
   $(CC) $(CFLAGS) $(LESS) $(SRCS) $(SRCS_C)  > obj/.depend



############################################################
# ADL - IDL Compilation
#
adl:
cd ./Corba && $(MAKE)
```

```
#########################################################
# Cleaning object files
#
clean:
rm -f obj/*.o

clean_corba:
rm -f obj/Corba/*.o

clean_all:
rm -f obj/Corba/*.o
rm -f obj/src/*.o
rm -f obj/*.o
```

Índice de figuras

Bibliografía

[1] ActivMedia. *Aria Reference Manual*, 2003.

[2] A. CARAMAZANA. *Estándar CORBA C++/Java*.

[3] A. FRISCH. *Essential System Administration*. O'Reilly, 2002.

[4] B. GALLMEISTER. *POSIX.4*. O'Reilly, 1995.

[5] J. HEKMAN. *Linux in a Nutshell*. O'Reilly, 1997.

[6] S. HENNING, M. & VINOSKI. *Programación Avanzada en CORBA con C++*. Addison Wesley, 1999.

[7] N. JOSUTTIS. *The C++ Standard Library*. Addison Wesley, 1999.

[8] Red Hat. *Red Hat linux x86 Installation Guide*.

[9] H. SCHILDT. *C Manual de Referencia*. Mc Graw Hill, 19975.

[10] J. SIEGEL. *CORBA, Fundamentals and Programming*. Wiley, 1996.

[11] I. SOMMERVILLE. *Ingeniería de Software*. Addison Wesley, 2002.

[12] B. STROUSTRUP. *The C++ Programming Language*. Addison Wesley, 1997.