FACULTAD DE CIENCIAS
UNIVERSIDAD DE CANTABRIA


# INTERFAZ DE NIVEL DE APLICACIÓN PARA EL CONTROL DISTRIBUIDO DE MOVIMIENTOS DEL ROBOT HUMANOIDE MICROBIRO

**Directores:**
**J. Javier Gutiérrez García**
**J. Carlos Palencia Gutiérrez**

**Autor: Ignacio Merino Cue**
**Marzo de 2011**

# 1. INTRODUCCIÓN

## 1.1-SISTEMAS EMPOTRADOS

Se conocen como sistemas empotrados o embebidos a los sistemas computacionales integrados en aparatos electrónicos con funciones específicas controladas por el computador que contienen, y con una fuerte interacción con su entorno. Ejemplo de éstos los encontramos en cajas registradoras, juguetes, controladores de sistemas en vehículos, aviones, electrodomésticos, etc. Su principal ventaja respecto a los computadores personales es la optimización de su tamaño y su consumo para las tareas que van a desempeñar, pues sus componentes se encuentran integrados en la placa base junto al procesador. De este modo, debido al desarrollo de núcleos cada vez más potentes y dispositivos más pequeños, logramos sistemas empotrados más eficientes en menores tamaños y pesos, como hemos podido comprobar en la telefonía móvil o los reproductores de audio digitales. Además existe la capacidad de su producción en serie, lo cual reduce al mínimo los costes.

Para la construcción de estos sistemas podemos escoger procesadores y memorias según la función que vayan a desempeñar. En contrapartida, un fallo en un componente obliga a cambiar toda la placa al completo, a diferencia de, por ejemplo, los ordenadores personales, donde encontramos más modularidad y por lo general podemos cambiar un elemento concreto.

Debido a que se busca reducir el tamaño, el consumo y el peso, carecen de las mismas prestaciones que los sistemas convencionales. Por ejemplo, no es habitual que tengan una interfaz de interacción con el usuario ni grandes memorias que gestionen ficheros, por lo que son necesarios elementos adicionales, y a veces se comunican con otros computadores formando lo que se conoce como sistemas distribuidos.

Debido a su interacción con el entorno físico cambiante, muchos de estos sistemas empotrados presentan características de tiempo real, al imponerse

requerimientos temporales sobre la ejecución de sus tareas. Por ejemplo, en muchos casos, un sistema necesita cumplimentar una tarea antes de que transcurra un determinado plazo desde la activación de la misma; en caso contrario, se considera que el funcionamiento del sistema ha sido incorrecto.

### *1.1.1-EL COMPUTADOR EN LOS SISTEMAS EMPOTRADOS*

El principal elemento de un computador es la CPU [1] (Control Processing Unit), también conocido como procesador, que controla el funcionamiento del computador y lleva a cabo el procesamiento de datos. Además de este elemento se necesitan otros, como memorias o dispositivos entrada y salida, para completar las funciones básicas que debe realizar un computador.

Centrándonos en la CPU encontramos 4 principales componentes:

- La unidad de control, que se encarga de buscar instrucciones en la memoria principal, interpretarlas y controlar su ejecución.
- La unidad de procesamiento aritmético y lógico (ALU), que procesa los datos que le manda la unidad de control y escribe los resultados en los registros.
- Los registros, que dan almacenamiento interno a la CPU para guardar las instrucciones y los resultados.
- Las interconexiones, que comunican todos estos elementos y a la CPU con el exterior y entre sí.

Internamente los computadores transfieren los datos e instrucciones de control mediante lo que se conoce como *buses*. Los hay de varios tipos pero no nos concierne especialmente; tan solo decir que los buses pueden operar en paralelo, donde la información viaja dividida paralelamente por distintas pistas de forma rápida, o en serie donde se realiza una comunicación bit a bit. Sí resulta interesante hablar sobre dispositivos de intercomunicación como lo son la UART (*Universal Asynchronous Receiver-Transmiter*) , para comunicaciones externas al computador y la SPI (*Serial Peripheral Interface*) principalmente usada para comunicaciones internas entre dispositivos integrados.

La UART se encarga de las comunicaciones por puerto serie y su principal labor es la de transformar tramas de datos que llegan en paralelo desde buses a tramas para transmitir en serie por puertos y viceversa. Hoy en día se tiene un dispositivo UART en cada puerto para gestionar las comunicaciones por puertos serie (como, por ejemplo, un puerto USB).

La SPI, por su parte, controla las transmisiones por bus serie entre dispositivos integrados. Consta de una entrada para el reloj del computador, una señal de pulsos constante que marca el ritmo de las operaciones. De este modo los datos se transmiten ocupando menor espacio en el circuito integrado y con menor consumo y coste, aunque la transmisión no sea tan rápida como podría ser si fuera paralela.

Hay otro elemento básico en un computador: las memorias primarias [2]. Éstas pueden ser de tres tipos, según su funcionalidad:

- Registros: son elementos internos de la CPU, de acceso muy rápido, pero de escasa capacidad. Almacenan datos y resultados obtenidos en los cálculos del procesador.
- Memoria principal: memoria que contiene todos aquellos datos e instrucciones que se van a ejecutar. La CPU accede a esta información mediante buses para ir ejecutando las instrucciones correspondientes. Hoy en día se usan las memorias RAM (Random Access Memory) en la mayoría de computadoras para esta función.
- Memoria caché: pequeñas memorias electrónicas que contienen información duplicada de la memoria principal, para el acceso rápido desde la unidad de control. No resulta imprescindible pero sí es conveniente su uso por la CPU con el fin de aligerar los procesos y mejorar su rendimiento.

No nos olvidamos de las memorias secundarias que cumplen la importante labor del almacenamiento masivo de datos a largo plazo como pueden ser hoy en día los discos duros o los DVD como caso de memorias ópticas.

Por otro lado, es muy común hoy en día el almacenamiento por memorias flash, que resultan especialmente rápidas en la escritura y lectura pues permiten operar al mismo tiempo sobre muchas posiciones distintas de la memoria. Debido a su pequeño tamaño se usan para dispositivos donde esto resulte importante y además

se trata de un elemento barato, resistente y silencioso. Muy común por ejemplo en las memorias USB portátiles por citar algún dispositivo o en sistemas empotrados,

## 1.2-SISTEMAS DE TIEMPO REAL

Hoy en día, y cada vez más, utilizamos aparatos electrónicos en cualquier ámbito de la vida cotidiana. En la mayoría de estos casos dichos aparatos realizan tareas complejas respondiendo a estímulos externos o a nuestros requerimientos. A estos sistemas se les conoce como sistemas de tiempo real, pues deben realizar sus tareas coordinadas con unos plazos concretos limitados por otros sistemas o por el entorno físico.

Vemos ejemplos en telefonía móvil, automoción, fabricación automatizada, controles de aviación por citar algunos. Si nos fijamos en un coche de última generación, este tendrá decenas de dispositivos que realizan automatismos en respuesta a estímulos (ABS, control de tracción, luces adaptativas…) y dichas tareas han de realizarse en momentos concretos y habitualmente terminar para que empiecen otras de forma coordinada.

Es habitual que estos dispositivos sean a su vez sistemas empotrados que funcionan en conjunto coordinados por un computador central, formando lo que se conoce como un sistema distribuido. El reto del programador reside en que todos funcionen y hagan su tarea cuando les toque cumpliendo sus plazos temporales. En ocasiones todas las tareas se realizan con un mismo procesador, por lo que se debe asignar el control del mismo a cada tarea según las necesidades. Para estos casos se usan distintas técnicas de planificación de tareas, como asignar prioridades.

Por ejemplo, en el control de acceso a una estación de metro. La puerta no debe abrirse hasta que se termine la comprobación de que el billete es correcto. Posteriormente debe mantenerse abierta hasta que el sensor detecte que la persona atravesó la puerta, y cerrarse en ese instante. Es importante en este tipo de sistemas, la estabilidad y la seguridad de los procesos, pues de ello depende el buen funcionamiento del mismo y en ocasiones como puede suceder en vehículos, la seguridad de los usuarios.

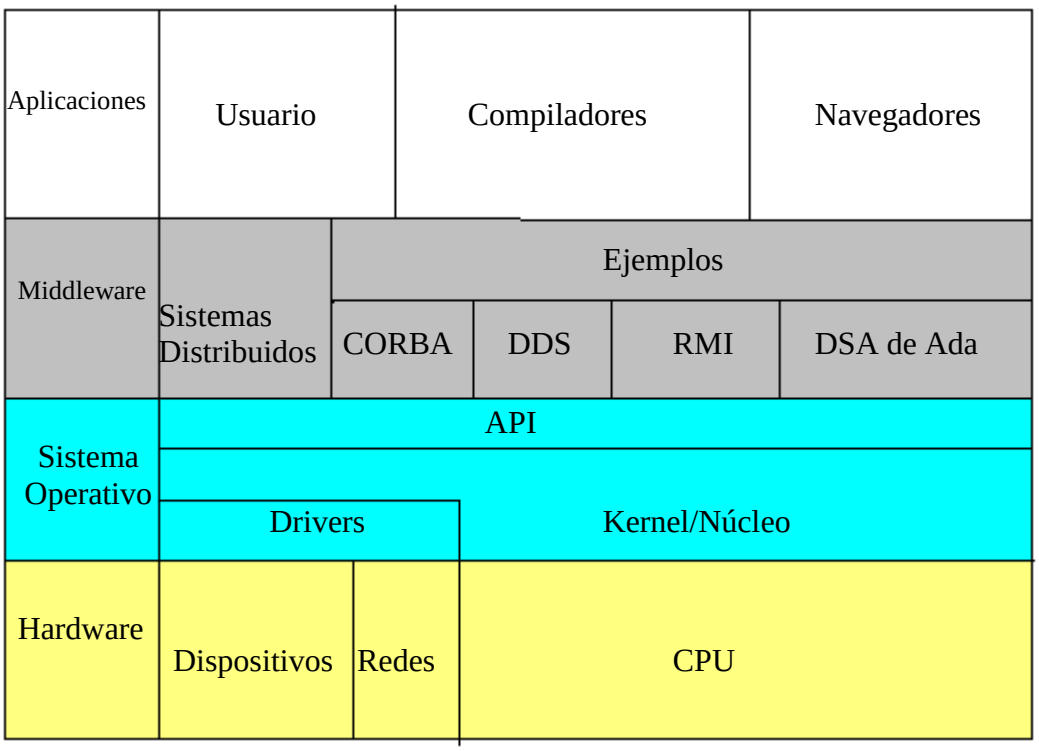
## 1.3-SOFTWARE DEL SISTEMA

Toda computadora necesariamente con el fin de ofrecer sus servicios, contiene una serie de equipamiento lógico a través del cual el usuario interactúa con la máquina, tanto para proporcionar información, como para obtenerla. Todo este soporte es conocido como software o "parte blanda" de la computadora. Debemos diferenciar a criterio de su funcionalidad distintos tipos: software de desarrollo, para que programadores generen distintas aplicaciones; software de aplicación, para que el usuario lleve a cabo distintas tareas; o software del sistema, que se encarga de desvincular a los usuarios de los detalles referentes al tipo de computadora en la que esté trabajando y lo aísla del uso de procedimientos internos como la gestión de las memorias, recursos, etc.

En la mayoría de los computadores, la gestión de los elementos físicos que componen el mismo (hardware), como pudieran ser memorias, dispositivos de entrada y salida, o tarjetas de sonido están gestionados por el sistema operativo. Lo hace a través de los drivers, pequeños programas informáticos asociados a cada elemento periférico que abstrae al sistema operativo del hardware y le ofrece una interfaz mediante la cual poder usarlo.

Para algunas aplicaciones que funcionen de forma remota accediendo a recursos de otras computadoras, se sitúa por encima del sistema operativo una capa conocida como *middleware* de distribución cuya función es abstraer de las complejidades de las redes de comunicación y del uso del sistema operativo ofreciendo una API (*Application Programming Interface*), es decir, una serie de funciones de alto nivel para manejar las utilidades que se ofrecen. Algunos middlewares de distribución típicos son CORBA (*Common Object Request Broker Architecture* — arquitectura común de intermediarios en peticiones a objetos) [7], del que hablaremos en el siguiente apartado, RMI (*Java Remote Method Invocation*), para objetos distribuidos Java, DSA (*Distributed System Annex*) para objetos Ada, por citar algunos.

En ocasiones, todas estas funciones que hemos descrito las realiza un único elemento de software. Esto ocurre frecuentemente en sistemas empotrados o de poca complejidad tomando el nombre de *firmware.*



Software del sistema

### *1.3.1-SISTEMAS DISTRIBUIDOS*

Llamamos sistemas distribuidos a aquellos cuyos componentes hardware y software funcionan sobre distintos procesadores que se comunican y coordinan sus acciones transmitiéndose, mediante un protocolo, una serie de información. Hoy en día su aplicación es infinita encontrándolos en cualquier actividad diaria. Es muy habitual en nuestro entorno encontrar aparatos que realicen una tarea controlados a partir de dispositivos que ofrezcan mayor facilidad de uso al usuario. Podemos citar por ejemplo el caso doméstico del manejo de una impresora conectada a la red manejándose desde nuestro PC, los controles de un coche, o la recepción de datos desde un sensor presente en un satélite en órbita.

El modelo más común de sistemas distribuidos es el de Cliente-Servidor en el que una máquina cliente solicita un servicio que proporciona otra máquina llamada servidor. Este servicio puede ser la ejecución de un programa o el acceso a información, generalmente. Estas solicitudes y respuestas van reguladas por un protocolo que fija una serie de reglas y formatos entre ambas máquinas. En ellas situamos módulos de software que dan el formato y definen los mensajes que se pretenden intercambiar, del mismo modo que interpretan y descifran la información

que se recibe. Estos módulos funcionan como una capa por encima del sistema operativo, por lo que la comunicación se realiza de forma homogénea entre componentes de distinta naturaleza.

## 1.4-CORBA

Se conoce como CORBA [7] a un estándar definido por OMG (*Object Management Group*) para el desarrollo de aplicaciones distribuidas, que proporciona facilidad para utilizar remotamente métodos pertenecientes a objetos creados en otra máquina.

La base de la comunicación Cliente-Servidor está soportada internamente por un elemento de software conocido por ORB (*Object Request Broker*) que manda al programa servidor (el que tiene acceso al objeto que se pretende usar) los requerimientos que hace el cliente. El ORB recoge la respuesta proporcionada y se la devuelve al demandante de la misma. De esta forma, la comunicación nunca se establece explícitamente entre el Cliente y el Servidor, por lo que no es necesario que ambos estén escritos en el mismo lenguaje. Esta característica convierte a CORBA en una tecnología adecuada para el desarrollo de aplicaciones distribuidas y apta para sistemas totalmente heterogéneos.

### *1.4.1-STUBS Y SKELETONS*

Dentro de CORBA, la idea, y por otra parte gran fuerza de esta tecnología es poder utilizar objetos remotos como si fueran locales. Por un lado, tendremos un programa funcionando como servidor (donde reside el objeto en cuestión) comunicándose a través de su ORB con un programa que funciona como cliente. Puesto que en el programa del cliente no tenemos el objeto, necesitamos un "doble" que implemente la interfaz del objeto, sobre el que programar las llamadas y esperar los retornos. Este elemento del cliente es el conocido como *Stub* (del inglés "*stunt doublé"*), el cual envuelve el ORB, funcionando como interfaz para el programa. Él solo se encarga de recoger la petición para el objeto, transformarla y enviarla mediante el ORB (operación considerada como *marshalling*), escuchar las respuestas y retornarla como si fuera el resultado de un método local. Esta misma

labor en el lado del servidor la realiza el *Skeleton* que a grandes rasgos podemos considerarlo un *stub* del programa servidor. Por suerte para el desarrollador, ambos elementos son autogenerados, como veremos a continuación.

### *1.4.2-EL LENGUAJE IDL*

Todos los componentes de la programación y uso de CORBA son objetos con una interfaz e identidad concretas y con la posibilidad de estar implementados en lenguajes de programación distintos. Se comunican entre sí para obtener los correspondientes servicios a través de un "Bus software" (el ORB). Para toda esta programación, CORBA ofrece un lenguaje de definición de interfaces conocido como IDL. En él se definen, de forma orientada a objetos, los métodos y campos que ofrece cada componente y que podrán ser usados por otro componente a través del ORB. Estas interfaces se mapean, o dicho de otro modo, se transforman mediante un compilador IDL al lenguaje de programación escogido (OMG ofrece mapeos de IDL a los lenguajes de programación más comunes) y se generan así todos los ficheros necesarios para la comunicación con el ORB (skeletons, stubs…) y ofrecer la interfaz definida, además de las utilidades que cada objeto CORBA ofrece.

### *1.4.3-ADAPTADOR DE OBJETOS*

El adaptador de objetos es un elemento CORBA en el que registraremos las implementaciones de los objetos que se utilizarán remotamente y que les proporciona la funcionalidad de un objeto CORBA para ser utilizado como tal. Como su nombre indica, se trata de un componente que adapta nuestro objeto a las necesidades que le requerirá el ORB, le permite acceder a los servicios que ofrece (como la generación de referencias para el objeto), y hace las correspondencias entre las llamadas que llegan al servidor y las implementaciones. Las referencias del objeto son identificadores que contienen información acerca de la máquina, la dirección dentro de la misma, el nombre y distintos datos, de modo que cualquier máquina pueda encontrarlo a partir de esta referencia, (hablaremos en posteriores apartados de distintos ejemplos). El adaptador más utilizado hoy en día es el POA (*Portable Object Adapter*). Varios objetos pueden estar registrados en un mismo

POA y en un servidor puede haber varios POA, gestionados por un gestor de POA o *POAManager* que gestiona el flujo de peticiones hacia los adaptadores en base a las políticas y prioridades que se pueden definir

### *1.4.4-EL IOR Y EL SERVIDOR DE NOMBRES*

Para que el cliente localice el objeto en cuestión, debemos proporcionarle un identificador con el cual pueda localizarlo de entre todas las máquinas del mundo, a través de un puerto de entrada a la computadora concreto y en una ubicación determinada. Esta información viene proporcionada por lo que se conoce como IOR. Se trata de una larga cadena de números que contiene información de la localización de la computadora contenedora a través de la red, el nombre del objeto en cuestión, el puerto por el que comunicarnos y la situación en el sistema de archivos local donde se almacenó. Una vez que tenemos localizada la instancia, el ORB del servidor y el del cliente se comunican preferentemente por el protocolo IIOP (*Internet Inter-Orb Protocol)* para intercambiar la información. Se trata de un protocolo que usa la capa de transporte TCP/IP y que da soporte a la especificación genérica que asienta la comunicación entre dos ORB: la GIOP (*General Inter-ORB Protocol*).

Para hacer aplicaciones dinámicas necesitamos ser capaces de encontrar el IOR en cada ejecución sin necesidad de que el usuario intervenga. Hay que tener en cuenta que el IOR cambia cada vez que se crea el objeto, lo que hace necesaria una forma de obtener el identificador de forma automática. Para esto, se suele utilizar lo que se conoce como Servidor de Nombres. Se trata de un programa en el que el servidor puede registrar el objeto bajo un nombre o identificador. A este programa, localizado en cualquier máquina, no necesariamente la del servidor, podemos acceder desde el cliente sabiendo su dirección y puerto en el que escucha, y obtener la instancia a través del nombre con el que se registró. De este modo obtendremos un objeto CORBA, que podremos convertir al objeto en cuestión mediante la operación conocida como *casting* o *narrowing*. El servidor de nombres es una utilidad nativa en los entornos de desarrollo de la mayoría de los lenguajes: por ejemplo en Java, dentro del JDK (*Java Development Kit* o kit de desarrollo Java) existe bajo el nombre de tnameserver.exe. Además, los paquetes CORBA más habituales, como Orbacus, Orbix, PolyORB, poseen los suyos propios.

### *1.4.4-RESUMEN*

La idea por tanto es que el ORB del cliente se comunique mediante el protocolo IIOP con el servidor de nombres para obtener la referencia del objeto registrado. Con esta referencia establecemos las sucesivas comunicaciones IIOP con el ORB donde se localiza el adaptador de objetos (POA) que contiene el objeto sobre el que invocamos los métodos requeridos y devolvemos al ORB del cliente sus resultados. El *stub* del cliente retorna el resultado al hilo del cliente que lo ejecutó, y el programa prosigue como si nada de esto hubiera sucedido de forma remota.

## 1.5- REDES INALAMBRICAS

### *1.5.1-INTRODUCCION*

Las redes inalámbricas, también conocidas por *wíreless*, comunican una serie de nodos sin utilizar cables. Se utilizan ondas electromagnéticas en todo tipo de rangos de frecuencia, según su aplicación. Entre sus ventajas, aparte de la obvia libertad de movimientos de los nodos, están la reducción de los costes al no necesitar cables y poder interconectar muchos nodos con el mismo servidor de señal. Por otro lado, presentan las desventajas de un menor alcance de las señales, la pérdida de tasa de velocidad con respecto al cable y la menor seguridad, ya que es más fácil captar las señales por nodos ajenos a la red.

Encontramos distintas clasificaciones según la cobertura de las redes. Por citar algunas populares, encontramos las redes WPAN, redes de ámbito personal y doméstico como la domótica o teléfonos inalámbricos, donde utilizamos tecnologías como el *Bluethooth o ZigBee*. En este trabajo utilizaremos *ZigBee* por lo que extenderemos la información acerca de esta tecnología en el siguiente apartado.

También encontramos las redes WLAN en el ámbito de edificios o áreas locales no muy grandes a la que pertenece la tecnología WiFi. Por otro lado, existen las redes del tipo WMAN (de ámbito metropolitano) de las cuales existen muy pocas en España, y las WAN, que se asocian a las tecnologías de los móviles 3G.

### *1.5.2-ZIGBEE*

ZigBee [3] es el nombre de la especificación de una serie de protocolos que cumplen el estándar 804.15.4 de redes de área personal WPAN. Se creó para cubrir la necesidad de redes baratas de baja tasa de envío de datos y bajo consumo pero de modo seguro y con alta fiabilidad. Se trata de una tecnología eficiente con un consumo mínimo de sus receptores y emisores y mucho más barata de fabricar que sus competidoras (por la sencillez de sus circuitos), lo cual la convierte en la red de comunicación ideal para domótica y otras aplicaciones domésticas que tan en auge están actualmente. Su única objeción frente a otras tecnologías como el Bluetooth es su menor velocidad de transferencia, aunque esto no es un impedimento para el desarrollo de las aplicaciones domésticas o de juguetería que ya comentamos.

Su alcance va desde los 10 a los 75m y puede usar frecuencias en los rangos de 2.4GHz (mundial) 868 MHz(Europa) y 915MHz(USA). Acepta hasta 255 nodos, estando la mayor parte del tiempo dormidos, ahorrando así gran cantidad de energía en comparación con otras redes inalámbricas.

Se tienen tres tipos de dispositivos según su papel dentro de la red: un Coordinador, un *Router* y un *End Device*. El primero es necesario en toda red ZigBee, requiere memoria y capacidad de computar, por lo que es el dispositivo más caro. Un *Router*, interconecta dispositivos y ofrece por otra parte un nivel de aplicación para ejecución de código de usuario. Por último, un *End Device* apenas necesita memoria, simplemente puede intercomunicarse con su dispositivo padre, por lo que puede estar dormido la mayor parte del tiempo, ahorrando batería. Se trata de un dispositivo de precio muy reducido, con funcionalidad limitada, pero a la vez práctico para la construcción de redes.

## 1.6-OBJETIVOS DE ESTE TRABAJO

En este trabajo se pretende desarrollar e implementar una API con una serie de funciones de alto nivel para el manejo simplificado de un robot bípedo construido por la Universidad de Valencia, conocido como Microbiro. Se trata de un sistema empotrado cuyo *firmware* ya está programado por sus desarrolladores, incluyendo las funciones que controlan sus movimientos, recepción y envío de señales. El objetivo es establecer comunicaciones desde un PC mediante una conexión inalámbrica ZigBee y encapsularlo con una capa de alto nivel, ofreciendo así una interfaz para que otros programadores puedan desarrollar aplicaciones sin prestar atención a las comunicaciones.

La idea, por otro lado, es poder desarrollar programas informáticos través de la API en varios lenguajes de programación. Desde Java, donde inicialmente se va a trabajar, hasta Ada, con la que futuros desarrolladores puedan beneficiarse de las posibilidades que ofrece en el ámbito del Tiempo Real. Con este fin se introducirá la tecnología CORBA, que es capaz de soportar como vimos anteriormente distintos lenguajes y que, además, nos dará la posibilidad de una ejecución y control remotos del robot. Se pretende también poder manejar simultáneamente dos terminales Microbiro, mediante el uso de un mismo módulo XBee a través del mismo programa Cliente y así comprobar la ligereza de las comunicaciones y la poca carga de código que supone el uso de CORBA.

A partir de nuestra API se podrá generar un sistema distribuido de 4 máquinas:

- Un robot Microbiro.
- Una máquina funcionando como servidor, emisor de instrucciones y coordinador de una red WPAN mediante ZigBee.
- Otra máquina haciendo la función de servidor de nombres del entorno CORBA.
- Un computador que ejecutará el programa (en Java o Ada) con las instrucciones que se requieran del robot, pudiendo estar situado en cualquier lugar del mundo conectado a la red.

Todo ello con la mayor portabilidad y heterogeneidad posible, y por supuesto con una comunicación estable.

## 1.7 ESTRUCTURA DE ESTA MEMORIA

En la presente memoria explicaremos de forma concisa los aspectos más importantes sobre el desarrollo del proyecto en el que hemos trabajado. Primeramente, en el capítulo 2 introduciremos la tecnología que usamos: por un lado el uso de los módulos XBee de intercomunicación mediante la red ZigBee, y por otro lado el propio robot Microbiro, entrando en aspectos de su estructura y *software* interno que son necesarios comprender para el desarrollo de nuestros objetivos.

En el capítulo 3 entraremos a describir una serie de modificaciones que se realizaron en el *firmware* del robot: introdujimos un nuevo movimiento de saludo y una serie de funciones para obtener datos de los sensores. Ya en el capítulo 4 entramos con aspectos de programación y comentaremos la definición de nuestra API para el control del robot.

En el capítulo 5 se tratará la definición de la API mediante CORBA para el control distribuido, donde comentaremos el procedimiento para obtenerla a partir de la definición de una interfaz IDL.

Finalmente describiremos unos casos de uso para controlar robots de forma distribuida y unas conclusiones de todo el trabajo desarrollado, para terminar con la bibliografía y un anexo que contiene una pequeña guía de usuario de los programas desarrollados.

# 2-TECNOLOGIA DEL ROBOT MICROBIRO

## 2.1-ZIGBEE

En este apartado presentaremos los conceptos clave para comprender la tecnología de las redes ZigBee, así como su gestión mediante módulos XBee de la serie 2.

### *2.1.1-MODULOS XBEE*

Ya tratamos con anterioridad lo que se conoce por ZigBee, y en base a esto, una red ZigBee es aquella que sigue esta especificación. En este proyecto trabajaremos con una red de este tipo y la gestionaremos mediante unos elementos *hardware* conocidos como módulos XBee, funcionando como dispositivos de la red. Existen dos tipos de módulos [4]: los de la serie 1 y los de la serie 2 (también conocidos como 2.5), que pese a tener la misma salida de pines, no son compatibles entre sí, pues trabajan con protocolos diferentes. La mayor diferencia reside a nivel de *firmware* interno, siendo la versión 802.15.4 la correspondiente a los módulos de la serie 1 y las versiones ZB o ZNet2.5 las de los módulos serie 2.

Existen muchísimos módulos con distintas características en función de precio, alcance, consumo y uso. En nuestro caso utilizaremos un módulo XBee serie 2 con adaptador USB para el control desde un computador. Toda la configuración del módulo, así como la instalación del *firmware* se realiza cómodamente a través de un software que distribuye gratuitamente DIGI (empresa fabricante de XBee) llamado X-CTU. Existen dos modos de configuración para los *firmware* en la serie 2: modo AT o modo API. La diferencia más importante es que, en el primer caso, la dirección de destino de la información es un campo de configuración del módulo, mientras que en el modo API (el que nosotros usaremos) la dirección va incluida en el paquete de envío. Además debemos escoger el firmware adecuado a nuestro tipo de dispositivo, es decir, Cordinador, *End Device* o *Router*

La configuración del dispositivo se puede realizar vía X-CTU o mediante el envió de comandos AT desde el computador al módulo. Se trata de una trama que contiene un código identificador del parámetro modificar y su nuevo valor, o bien, simplemente el identificador, en cuyo caso, el módulo retorna el valor que contiene. Hay una larga lista de parámetros configurables con valores por defecto; por citar algunos, está el identificador de la red PANID (*Personal Area Network IDentificator*), el *Operating Channel* (CH) , que da el canal operativo para la comunicación, El *Node Join Time* (NJ), que mide en segundo el tiempo durante el cual pueden conectarse otros dispositivos……

### *2.1.2.API XBEE*

El control e intercambio de información con el robot se realiza a través del módulo XBee. Para ello, el computador debe enviar a través de ese módulo XBee una trama de bytes en un formato apropiado para que el *firmware* sea capaz de interpretar y efectuar las operaciones indicadas. Es, por tanto, muy importante el manejo de la comunicación de envío y recepción de datos.

DIGI ofrece públicamente una API [5] (en lenguaje Java) para manejar las redes y los adaptadores. Esta API se engloba básicamente en 2 grandes grupos de clases: paquetes de envío y paquetes de recepción, que pueden ser enviados de modo síncrono, con recepción de confirmación, o asíncrono;. Éstos son heredados de dos objetos primitivos: *XBeeResponse y XbeeRequest.* Además se puede configurar el adaptador mediante envíos de comandos AT, pudiendo así establecer el PANID, *Operating Channel* y otros parámetros necesarios para establecer comunicación con una red. Como vemos tenemos total manejo de la comunicación a través de esta API.

Para usar esta API, primero se debe crear un objeto de la clase *XBee*, a través del cual tendremos acceso a todas las utilidades de comunicación. En nuestro caso estamos usando chip y adaptador de la serie 2 por lo que utilizaremos objetos tipo *ZNetTxRequest* para los paquetes de envío, en cuya creación debemos proporcionarle:

- Un *cluster* identificativo, consisteste en unos códigos hexagesimales que el receptor utilizará para clasificar la naturaleza del mensaje.

- Los datos del envío, empaquetados en un *array* de enteros de un byte de tamaño.
- La dirección del destinatario (de la que daremos más detalles posteriormente).

Para el envío usaremos generalmente la función *sendSynchronous()*, indicando el tiempo máximo de espera para la respuesta de confirmación, aunque en algunos casos recurriremos al método *sendAsynchronous()*, como veremos luego.

Por otro lado, usaremos el tipo *ZNetExplicitRxResponse* para la recepción de paquetes enviados desde el robot. Se dará la situación que tras una petición concreta al robot esperaremos una respuesta. Esta se obtiene a través del método *getLastResponse()* que nos retorna un objeto *XbeeResponse* con el último paquete recibido en el dispositivo. Posteriormente tendremos que hacer un casting al tipo deseado después de asegurarnos que el paquete es compatible.

El direccionamiento con el que trabajamos es de 64 bits, óptimo para dispositivos XBee de la serie 2. La dirección de un receptor se definirá a partir de su MAC (Media Access Control), que consiste en un número propio único de los elementos *hardware* de comunicaciones. Utilizaremos la dirección como un *array* de enteros de dimensión 8, en el que cada elemento representa la codificación decimal de cada par de digitos hexadecimales de la MAC.

Ejemplo:

MAC: 0x00:12:4B:00:00:81:68:36

Dirección para ZigBee: (0, 18, 75, 0, 0, 129, 104, 54)

## 2.2-MICROBIRO

Microbiro es un robot articulado bípedo de 20 grados de libertad (GDL), desarrollado por la Universidad de Valencia. Tan alto número de GDL dota al autómata de gran movilidad y capacidad de realizar movimientos complejos, debido además a su bajo peso, tan solo 550gr.

| | GDL |
|---|---|
| Pie | 2 |
| Rodilla | 1 |
| Cadera | 3 |
| Torso | 1 |
| Brazos | 3 |
| Cabeza | 1 |

Tabla1.Grados de libertad de cada elemento del Robot Microbiro

Como vemos en la tabla 1, la mayor parte de la movilidad reside en los miembros inferiores. Esto proporciona al robot gran capacidad en el desarrollo de movimientos de paso, donde reside la gran dificultad para los robots bípedos.

Para controlar tal cantidad de posibles movimientos y proporcionar el cómputo necesario, Microbiro cuenta con un procesador ARM7TDMI de 32 bits, integrado en la placa del propio robot. Con el procesador coexisten varios periféricos como, conversores A/D, UART´s, SPI, etc., que proporcionan mayores prestaciones en las comunicaciones y el control de los distintos sensores y servomotores, también conocidos como servos.

Además de estos elementos, la placa de un robot Microbiro consta de un chip XBee de la serie II para las comunicaciones inalámbricas. Se trata de un módulo integrado capaz de funcionar como emisor y receptor con una interfaz cómoda proporcionada por el fabricante. Sin embargo, también existe la posibilidad de establecer comunicación vía USB, conveniente para depuración y desarrollo de modificaciones en el *firmware*.

### *2.2.1-ENTORNO DE DESARROLLO CRUZADO*

El procedimiento usado para trabajar con Microbiro es el conocido como desarrollo cruzado. Esto significa que todo el desarrollo del software (*firmware*) se realizará previamente en un PC y, una vez concluido, se enviará al procesador ARM7 del robot donde será finalmente ejecutado.. Toda esa programación la haremos en lenguaje C, por lo que trabajamos relativamente a alto nivel, utilizando la API que se proporciona para el manejo de los distintos dispositivos de la placa.

Una vez desarrollado el "cerebro" de nuestro robot en nuestro PC debemos colocarlo en el interior del mismo, para lo cual se compila y se manda por puerto USB a la placa. El ejecutable compilado se almacena en unas memorias *flash* de la placa y es ejecutado al encender el robot inicializando el sistema como hayamos especificado en el código implementado.

En concreto, para nuestro proyecto trabajamos con el entorno de desarrollo PN (Programmers Notepad) y el compilador "*gcc*" para el código implementado en C. El entorno PN para C suministra de por sí facilidades para efectuar la carga del programa en la memoria, previa compilación. Para ello, tenemos que instalar en nuestro PC los drivers necesarios para la comunicación por puerto USB con el controlador de la placa, y configurar el puerto de comunicación. Una vez que todo esté listo podemos comenzar el proceso de desarrollo cruzado.

### *2.2.2-FIRMWARE DE MICROBIRO*

El "cerebro" del robot consiste en un complejo programa que realiza todo el control de las utilidades del robot y de sus periféricos. En esta sección vamos a explicar brevemente cómo funcionan algunas de sus partes más importantes. Hemos de tener en cuenta que tenemos código de servicio de ejecución normal y código ISR (*Interrupt Service Routine*) de manejo de interrupciones IRQ (*Iterrupt Request*) que provienen del hardware, es decir, código que se lanza cuando ocurre un evento asíncrono, tal como la llegada de un dato a cualquier dispositivo de la placa o algún cambio de estado en algún elemento.

-*COMUNICACIONES*

Podemos comunicarnos con el robot a través del puerto USB, para lo que podemos usar el manejador de la UART (aunque no hemos trabajado con ello y, por tanto, no lo trataremos) o bien podemos hacerlo por el chip receptor cc2480 de ZigBee. Cuando un byte es recibido y llega a la SPI (que establecerá la comunicación con el procesador) se lanzará una interrupción IRQ que manejaremos a través del fichero *spiISR*. Básicamente, se irá leyendo la trama para, finalmente, generar el paquete recibido y un evento que será recogido por la función *eventHandler()*, que realizará el manejo de la misma. Esta función detecta que se trata de un evento de recepción con el comando adecuado y generará un nuevo paquete, desentramando el *cluster* y los datos del envío, entre otros parámetros. Esta trama o *frame* nuevamente estructurada según la estructura *CC2480_ZIGBEE_MSG* es enviada como argumento al método que ejecutará las instrucciones correspondientes.

-EJECUCIÓN DE MOVIMIENTOS

Los movimientos se ejecutan cuando llega un mensaje con un *cluster* concreto definido para tal fin (corresponde con el valor hexagesimal 0xC07E). Cuando el *firmware* detecta que un mensaje de este tipo ha llegado, ejecuta la función *setInstruccion()*, pasándole como parámetro el valor que contiene el mensaje. Cada movimiento a ejecutar tiene un valor que coloca en cola a la acción en cuestión.

Todos los movimientos se controlan mediante una variable llamada *transitionMatrix*. Se trata de una ristra de valores en la que cada fila representa a cada operación concreta. De este modo por ejemplo, la operación “andar” viene descrita en la fila 3, y en ella viene la secuencia de operaciones a realizar: ponerse en posición de inicio, esperar 2 segundos, repetir 5 veces paso hacia adelante, parar, y volver a posición de inicio, etc.

El control de todo esto lo realiza una función que, según la fila que tengamos que ejecutar, va leyendo el siguiente elemento de la matriz y haciendo las operaciones correspondientes. Por medio de una serie de *flags* (también conocidas

como variables de estado) que el lazo de control va variando, el proceso va entrando en el código que le corresponde en la función UbiroControl(). Así, por ejemplo, si se ordenó andar al robot y el *flag FRONT* esta activado, se ejecuta *WalkFrontControl()* que maneja los movimientos que deben realizar los servos.

Cada unos de los movimientos disponibles tiene sus controles descritos en un fichero en lenguaje C con todas las posibilidades. *Walk.c* contiene las opciones de andar hacia adelante o hacia atrás, girar, pasos completos, primer paso, último paso etc. Cuando se selecciona qué movimiento hacer se escogerá la ristra de valores concretos que queremos que se pase a los motores, en cada iteración temporal. De este modo con cada llamada a *UbiroControl()* se ejecutara *WalkFrontControl()* que tomara el siguiente valor que tendrá cada motor del robot y lo guardara en un *array* llamado *posición[]* con un valor por cada motor. Posteriormente, la función *SetPositionAllServos()* establecerá por medio del pin adecuado el valor concreto para cada uno de los servos. Una vez acabado el proceso de realizar un paso se seguirá leyendo el siguiente valor de la matriz y comenzará el proceso de nuevo, hasta que se terminen la secuencia de instrucciones para el movimiento ordenado.

Las posiciones que tendrán los motores en cada iteración de cada movimiento se almacenarán en *Tables.c* como *arrays* de valores enteros. Habrá un array por cada motor que se deba mover, y un valor entero por cada iteración necesaria para realizar el movimiento.

## -MANEJO DE SENSORES.

El firmware de Microbiro ofrece una serie de funciones sencillas para el manejo de los sensores, dentro del fichero *ADConverter.c*. La placa dispone de 3 acelerómetros para determinar su orientación espacial y un sensor de distancia, instalado en la cabeza del robot . Trataremos de explicar cómo funcionan a grandes rasgos.

En primer lugar, se ha de inicializar el uso de los sensores a través de *UbiroSensorsConfiguration()*. Posteriormente una llamada a *ReadSensors()* toma lecturas de los sensores y las almacena en variables locales que podremos leer mediante funciones que ofrece la interfaz.

El método de *ReadSensors()* es un procedimiento de bajo nivel. Para cada sensor escribimos en unos registros de memoria una serie de valores binarios que seleccionan el canal de conversión e inician el proceso. Posteriormente se testea otro registro hasta que sus últimos 4 bits son 0, indicando que el proceso de lectura ha terminado. Se desplazan a la derecha 6 bits y se cogen los últimos 10 bits del registro como resultado de la lectura del sensor, pues únicamente estos bits contienen el valor de la medida aunque usemos se esté usando un registro de 16. Por último, se colocará un valor cualquiera en el primer registro como finalización del proceso.

-PROGRAMA PRINCIPAL

El fichero *main.c* contiene el método principal que se encarga de iniciar la ejecución del programa. Básicamente se trata de iniciar todos los dispositivos necesarios, tanto de hardware como de software, UART, SPI, memoria… Después pregunta si se desea iniciar en modo cliente o teleoperación (si no se obtiene respuesta pasados unos segundos, se establece el modo cliente). El modo teleoperación sirve para comunicarse por puerto USB y acceder a otras utilidades de la placa Microbiro, como la de operar como adaptador de la red ZigBee y no como dispositivo receptor, modo en el que funcionará al iniciarse el modo cliente.

Una vez estamos en modo cliente, se inicia la red ZigBee, configurando un PANID abierto para que cualquier dispositivo pueda conectarse (valor 0xFFFF) y definiendo la placa como un *End Device* dentro de la red. Se inician finalmente la SPI y la UART y se activan las interrupciones IRQ y FIQ (interrupciones para los servos). Configuramos los sensores y los servos y los colocamos en su posición inicial. A continuación, entraremos en un lazo infinito en el que se mantiene el programa. Dentro de este lazo se van realizando llamadas a la función *cc2480_poll()*, que gestiona un *polling* para la comunicación del chip cc2480 de la red ZigBee. El *polling* consiste en testear periódicamente si se ha producido algún evento, como por ejemplo una recepción de paquete. Si se ha producido algún evento, hace una llamada al manejador de eventos *eventHandler()* que realizará las acciones oportunas. Además de esto, el lazo del programa principal envía periódicamente solicitudes de registro con otros dispositivos, esto es, envío por

ZigBee de paquetes con información sobre la dirección del robot, por si algún dispositivo a la escucha quiere comunicarse con él.

### *2.2.3-PROGRAMA DE TELEOPERACION*

Los desarrolladores de Microbiro ofrecen, junto con el robot, una herramienta para teleoperarlo desde un PC. Se trata de una compleja aplicación grafica Java que se comunica a través del Zigbee. Consta de 4 ventanas desde las que realizar diferentes operaciones:

- Conexión por puerto serie con el adaptador XBee
    - Podemos escoger el puerto y la velocidad de conexión
- Manejo del adaptador y conexión por ZigBee con el robot
    - Configurar el módulo XBee
    - Selección de la dirección del Robot
- Calibración
    - Mover cada servo
    - Fijar en el firmware las posiciones de los motores que se deseen
- Envío de instrucciones.
    - Andar, chutar, girar y todas las opciones que tiene Microbiro

El programa nos resulta especialmente interesante para el tema de la calibración de las posiciones iniciales de los motores, ya que por medio de una interfaz grafica amigable se puede ir variando la posición de cada motor. Una vez calibrado a nuestro gusto podemos guardar estas posiciones y cargarlas dentro de la placa. Se trata de una utilidad que no hemos incluido en la interfaz que se ha desarrollado para este proyecto, pues no se trata de una funcionalidad necesaria para manejar el robot, sino un procedimiento previo a su primer uso. Para ello resulta perfecto utilizar este método de calibración que los desarrolladores ofrecen.

# 3-MODIFICACIONES SOBRE EL FIRMWARE

Para completar las opciones que nos da el firmware, se introdujeron una serie de modificaciones. En primer lugar se dotó al robot de un nuevo movimiento consistente en un sencillo saludo utilizando solamente 2 servos. Por otro lado le añadimos al robot la posibilidad de que nos indicara el valor del sensor que le pidamos por medio de comunicación inalámbrica.

## 3.1. UN NUEVO MOVIMIENTO: HOLA

Lo que en principio parecía de gran dificultad, sirvió como ejercicio para comprender mejor el funcionamiento del complejo firmware con el que se trabajaba, y finalmente tan solo conllevó unas ligeras modificaciones en la *transitionMatrix (*un *array* bidimensional que ya hemos tratado con anterioridad en la sección 2.2.2), las funciones de control, y la creación de un nuevo fichero *Hola.c*

Al tratarse de un nuevo movimiento se añadió a *transitionMatrix* una nueva fila con la secuencia de operaciones que se habían de realizar. Primero un *Stop*, luego ponerse en posición de inicio, esperar 2 segundos, movimiento *Hola* y volver a posición de inicio. Por tanto un mensaje llegado con el *cluster* correspondiente a ejecución de movimientos y valor 19 en sus datos colocará a la operación de saludo en cola. Se trata del valor que utilizaremos en el código de la API que maneja el robot desde nuestro PC para hacer las peticiones de ejecución de saludo.

En *Tables.c* incluimos las posiciones que tendrán los dos motores que intervienen: hombro y muñeca derecha. El movimiento consiste en levantar el brazo suavemente y cuando está en alto mover la muñeca a ambos lados un par de veces; después el brazo vuelve a su posición inicial. Traducido a números, tenemos que el hombro se va variando desde 0 a 600 (un valor máximo) donde se queda unos instantes valores para volver a bajar a 0. La muñeca por otro lado se mantiene a 0 hasta que el hombro llega a 600, entonces varía hasta 100 (un valor intermedio)

rápidamente un par de veces y vuelve a 0. En total tenemos 93 posiciones en cada *array* de cada motor. Ambos deben tener la misma dimensión y debemos registrar su tamaño en una tercera variable en la tabla.

Para el control de todo el movimiento se creó *Hola.c*, La función *Hola()* prepara los valores de los motores introducidos para ser usados, luego la función *HolaControl()* asigna a la posición del hombro y de la muñeca el valor correspondiente con la variable externa contador que funciona como iterador y terminará cuando alcance el valor 93 que obtiene también de *Tables.c*

Tuvimos por último que añadir a las funciones *UbiroControl()* y *TeleoperationControl()* los casos en que se lea de la *transitionMatrix* que hay que realizar el movimiento de saludo.

Como vemos, se trata de un movimiento sencillo para probar el funcionamiento y, de paso, añadir una función extra al robot. En nuestro ejemplo usamos valores arbitrarios pues no necesitamos que haya coordinación entre motores, pero para reproducir movimientos de paso necesitamos un gran número de valores coordinados entre todos los motores, pues una pequeña descoordinación conlleva una pérdida de equilibrio. Se trata de un trabajo complejo digno de un gran análisis. Es por ello que en este proyecto no se ha querido trabajar el asunto por ser demasiado profundo (y fuera de los objetivos) y se ha introducido un movimiento que no exigiera equilibrio alguno.

## 3.2 OBTENCIÓN DE DATOS DE LOS SENSORES

El *firmware* recibido incluía ya una interfaz para obtener de forma sencilla los valores de los sensores en el fichero *ADConverter.c*. No obstante, su uso estaba desactivado y no había modo de obtenerlos de forma remota por ZigBee. Por ello, se introdujeron modificaciones para ofrecer esta utilidad al usuario.

En primer lugar, para activar el uso de los sensores había que activar el *Flag* o variable de estado *ADC_SUPPORT*. Además, tenemos que incluir funciones que envíen los valores leídos por medio de ZigBee. Estas funciones, una por cada

sensor, las incluimos en el fichero *cc2480_customcmd.c* en donde se encuentran las funciones de envío de datos o respuestas a peticiones externas.

Las nuevas funciones ejecutarán *readSensors()*, que leerá los valores de los sensores, y devolverá el valor del sensor correspondiente en dos valores de tipo *uint8* de 1 byte de tamaño. Esto es debido a que sólo podemos transmitir byte a byte, y el sensor retorna un valor de 10 bits. Por ello, en el primer byte metemos los 8 primeros bits y los 2 bits restantes en el segundo byte. Para el manejo de bits utilizamos las operaciones "and" y "or" usuales en Java. El resultado lo empaquetamos en un *array* de dimensión 2.

Para el envío por ZigBee usaremos las funciones que ya nos ofrece el firmware. En concreto, la función *cc2480af_send_data()* nos pide como parámetros importantes una dirección, un *cluster* y un *array* con los datos a mandar. La dirección que pondremos será 0x0000 para que cualquier dispositivo pueda recibir el mensaje, el *cluster* de envío será 0xC08F, como se definió en los dicheros ".h" del *firmware*, que nos permitirá identificar el tipo de mensaje en el receptor, y como *data* pondremos el *array* anteriormente fabricado.

Para que la modificación quede completa tenemos que interpretar el mensaje recibido, para que ejecute las funciones de envío anteriormente descritas. Ya comentamos cómo funciona la recepción de mensajes (apartado 2.3.1); la única diferencia en este caso con el mensaje de recepción de instrucciones de movimientos es el c*luster* de recepción. En este caso será 0xC08E, valor que incluiremos mediante una constante en *cc2480_config.h* y que usaremos para identificar el mensaje como petición de sensores en la función *Zigbee_MSG()* perteneciente a *cc2480_app_handler.c*. Dependiendo del dato que recibamos en el mensaje: 1, 2, 3 o 4 ejecutaremos las funciones para el acelerómetro X, Y, Z o el sensor de distancia respectivamente.

# 4- INTERFAZ LOCAL PARA MICROBIRO

En este apartado se explica la implementación en lenguaje Java de una interfaz que funciona como caparazón para facilitar a un futuro programador la utilización de robots Microbiro. La comunicación se realizará desde el PC mediante conexión inalámbrica ZigBee. Usaremos como coordinador un módulo XBee con adaptador USB correctamente configurado para comunicarse con un chip integrado en la placa del robot, con su correspondiente MAC (ver apartado 2.1.2), y en el que está configurada una red ZigBee.

## 4.1-INTRODUCCION

Se pretende ofrecer todo el abanico de operaciones que ya nos ofrece Microbiro. Como ya comentamos, el *firmware* del robot suministra operaciones para una serie de movimientos (además de las modificaciones hechas), con lo que la API vendrá conformada con todas esta opciones, a las que tenemos que añadir las propias de gestión del robot, como son funciones para conectarse, definir una dirección y realizar la comunicación.

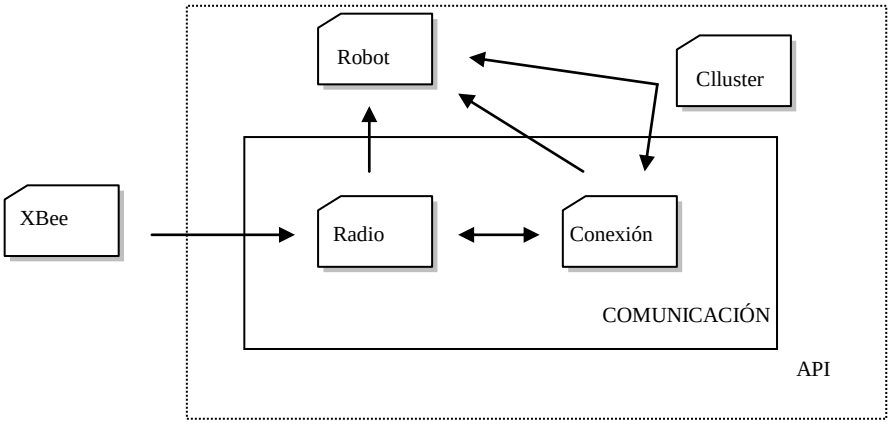
## 4.2-API DE CONTROL DE MICROBIRO POR ZIGBEE

### *4.2.1-DESCRIPCION*

Nuestra API está diseñada siguiendo la filosofía del desarrollo orientado a objetos, de modo que, en lo posible, cada elemento de programación represente un elemento físico. De esta forma, el diseño y la estructura del programa resultan más intuitivos para el programador.

La clase sobre la que ofreceremos las funciones de manejo de Microbiro será la clase *Robot*. Esta clase necesitará el apoyo de la clase *Conexión* que se encarga de gestionar las labores de la red inalámbrica. A su vez, se requiere el uso

de la clase *Radio* para manejar el adaptador XBee y de una pequeña clase llamada *Cluster* que nos dará soporte a los *cluster* de los paquetes de envío y recepción. A continuación describiremos sus métodos y su significado traducido al entorno real del problema.




### *4.2.2-CLASES*

-CLUSTER

Un cluster es un elemento de los mensajes de envío y recepción mediante ZigBee, de modo que el receptor sea capaz de identificar la naturaleza de la información que le llega. Esta clase surge de la necesidad de manejar objetos de *cluster* distintos para comparar los mensajes recibidos y proceder de modo correcto en función de lo que nos llega.

La clase *Cluster* tiene dos parámetros enteros privados, *msb* y *lsb* a los que se les da valor mediante un constructor. Para leer estos valores disponemos de los correspondientes métodos *getMsb()* y *getLsb()*, que retornan el entero de la variable en cuestión.

-RADIO

Se trata de la clase que viene a representar el adaptador emisor y receptor de las señales inalámbricas. A través de ella podremos configurar los parámetros del adaptador y acceder de forma estática (no sobre el objeto creado, sino de la clase) a ciertos valores necesarios para el envío de operaciones. Consta de una clase enumerada con los valores de teleoperación, una serie de variables privadas con los valores de configuración, un objeto publico *xBee* que representa, en esencia, la radio, proporcionando sus utilidades de envío y recepción. También tiene un constructor, una función para "encender" la radio, y funciones para cambiar los parámetros de configuración.

El constructor *Radio*() se encarga de crear un objeto *XBee* a partir del cual podremos iniciar la comunicación, como ya introdujimos anteriormente. Para ello debemos llamar al método *openRadio()* que necesita como parámetros el puerto y la velocidad a través de los cuales se comunica el PC con el adaptador XBee que vamos a utilizar. Internamente este método ejecutará el método *open()* del objeto *xBee* creado en el constructor, lo que abrirá la comunicación PC-módulo XBee.

Para que nuestro adaptador XBee pueda comunicarse con el chip cc2480 integrado en la placa es importante que esté configurado con los parámetros adecuados. Por ello necesitamos funciones de configuración que podamos utilizar para comunicarnos con distintos receptores y poder así modificar lo que sea necesario (por ejemplo, el PANID o el *Node Join Time* de los que hablamos en el apartado 2.1 de esta memoria). Se trata de valores presentes en la versión de *firmware* que usa el módulo XBee y que dotan al mismo de distintas capacidades. El más importante, y sobre el que ya hemos hablado, es el PANID, que representa el identificador de la red a la que el dispositivo va a conectarse, y que, como veremos luego, no necesita en nuestro caso de un valor específico, pues la red en el robot está configurada de modo que cualquier elemento con cualquier PANID configurado pueda acceder a ella. El resto de parámetros no los tocaremos, pues no tienen utilidad concreta en nuestro programa, simplemente llevarán el valor por defecto que no restrinja las posibilidades de comunicación

Todas las funciones de configuración, en esencia, realizan operaciones similares. Se trata de enviar a través del X*Bee* (elemento que representa nuestra

radio) lo que se conoce como comandos AT. Estos comandos los cuales se crearán por medio de la API que ofrece XBee (apartado 2.1), dando el identificador del parámetro y el valor que pretendemos dar. Posteriormente se envía al adaptador a través de la función *sendATComand()*.

-CONEXIÓN

En este caso, la clase no va a representar a un elemento físico real, sino a lo que es una conexión en sí misma. Como toda conexión en la vida real, necesitará un transmisor (que en este caso vendrá representado por un objeto de la clase *Radio* antes descrita) y tendrá una serie de campos que luego analizaremos. Por supuesto, nos ofrece el método *envía()* mediante el cual hacer nuestras peticiones al robot y con el que obtendremos las respuestas requeridas. Tales respuestas se almacenarán en una variable local que podremos leer externamente por el método *getRespuesta()*, y dispondremos de algunas variables de estado para conocer si se ha establecido conexión y si se ha configurado la misma.

El constructor *Conexion()* necesita el puerto y el *BaudRate* (velocidad del puerto) a través de los cuales nos comunicaremos con el módulo XBee. Así, por tanto, se crea un objeto *Radio* y asignamos el puerto y su velocidad a las variables privadas locales correspondientes, pues las necesitaremos en el método *conectar()*. Este método conecta la radio, ejecutando *openRadio()*, y la configura a través del método local *setParametros()*, que llama sucesivamente a las funciones de configuración del módulo XBee con los valores necesarios para el robot.

El método más importante quizás, de la API en conjunto, es el método *envía()*. Necesita recibir una dirección del receptor, un *cluster* identificador del paquete y los datos a envíar. En nuestro caso necesitamos hacer diferenciación de dos tipos de envío: un envío asíncrono de datos, del que no esperamos respuesta por parte del receptor (como, por ejemplo, un envío de instrucciones de movimientos) o un envío síncrono de petición en el que queremos una contestación (por ejemplo, pedir datos de un sensor). A través del *cluster* del envío detectamos en el código qué modo vamos a seguir para enviar los datos, para lo cual, usaremos los métodos que nos proporciona el objeto *xBee* del objeto *Radio* asociado a nuestra conexión. Haremos un envío asíncrono cuando esperemos que el robot nos dé una respuesta

(la confirmación de recepción será la respuesta del robot), tratando ésta aparte y discriminando las que no correspondan a nuestra acción requerida. Cuando el programa localice entre los paquetes recibidos por el módulo XBee una respuesta que cumpla las características que buscamos, registraremos el valor que trae consigo el paquete en una variable privada del objeto que luego podrá leerse mediante su correspondiente método. Si no se recibe respuesta en un plazo de 2 segundos, el método retornara un valor "false".

En cambio si queremos hacer un envío esperando contestación, haremos un envío síncrono, mediante el cual obtenemos una confirmación, con lo que podemos actuar en consecuencia repitiendo el envío si fuera necesario y retornando el éxito o fracaso del envío.

-ROBOT

La clase *Robot* es la principal de la API, la única que un usuario debe manejar directamente y la que ofrece todos los métodos de control del robot. El concepto es sencillo: cada objeto representa a una unidad Microbiro y, por tanto, cada objeto tiene que tener asociada una conexión para establecer comunicación y una dirección que le diferencia de otros robots. Lógicamente existirán, aparte de los métodos de control de movimientos y sensores, métodos para conectar, configurar puerto y dirección, verificar si hay comunicación y envío de datos, etc.

El constructor de la clase *Robot* necesita que le pasen una conexión a través de la cual realizará sus operaciones y simplemente asocia esta conexión al campo del objeto creado para que el resto de funciones operen con ella. A posteriori podremos cambiar el puerto de la conexión mediante la función *setPort()*. Así, por ejemplo, el método *conectar()* del objeto *Robot* inicia la conexión, quedando ésta abierta para que cualquiera pueda usarla; es decir, si hay más robots utilizando el mismo modulo XBee para la comunicación, deberán usar el mismo objeto *Conexión* pues usan también el mismo objeto *Radio*. Si uno de los robot abre la conexión, ésta estará conectada para que cualquiera pueda utilizarla y no tendrán que iniciarla individualmente cada uno.

El envío de órdenes se realiza con dos *cluster* distintos, dependiendo del tipo de petición: sensores o comando. La primera la usamos para el envío de

requerimiento de valores de sensores mediante la función *getSensor*(), a la que se le pasa el sensor como parámetro. 1, 2 o 3 para los acelerómetros x, y, o z respectivamente, y el valor 4 para el sensor de la distancia. Ya vimos en el apartado de modificaciones cómo desentrama el robot estos datos para hacer la lectura adecuada y posteriormente devolver al PC la respuesta.

Por otro lado el envío de comandos lleva asociada una función por cada orden. Así, por ejemplo, disponemos de la funcion *giraDerecha(), chuta(), saluda()*… las cuales sólo se diferencian en el dato que enviamos al robot y que obtenemos de la clase enumerada *TelInstructionsEnum*, presente como estática (elemento de clase y no de objeto) en la clase *Radio*.

Por razones de calibración y ajuste de movimientos, muchas de estas funciones, aunque están presentes en la API que ofrecemos, no puede ofrecerlas aún Microbiro, bien por que faltan de implementación en su *firmware*, bien porque el robot no es capaz de realizas de forma controlada aún. No obstante se creyó conveniente incluirlas para futuras mejoras.

### *4.2.3- METODOS DE LA API*

A continuación indicaremos los métodos que ofrece la API implementada:

- Robot(Conexión)
  - Constructor del robot.
  - Necesita un objeto Conexión a través del cual comunicarse
- hola()
  - Chequea el estado de la conexión
- conectar()
  - Abre la conexión PC-Microbiro
- setAddress(int[] address)
  - Cambia la dirección del robot para las comunicaciones
  - Por defecto está establecida en (0, 18, 75, 0, 0, 129, 104, 54)
- setPort(String port)
  - Cambia el puerto de comunicación

- enviar(int msb,int lsb, int[] data)
    - envía de modo sincrono unos datos con un cluster determinado por msb y lsb
- getSensor(int sensor)
    - Retorna el valor del sensor especificado
- andaAlante()
    - 5 pasos hacia adelante
- andaAtras()
    - 5 pasos hacia atrás
- pasoDerecha()
    - Un paso lateral hacia la derecha
- pasoIzquierda()
    - Un paso lateral hacia la izquierda
- giroDerecha()
    - Gira sobre sí mismo hacia la derecha una vuelta completa
- giroIzquierda
    - Gira sobre sí mismo hacia la izquierda una vuelta completa
- chuta()
    - Realiza un chut
- levanta()
    - Se levanta del suelo
- caer()
    - Se tira al suelo
- pasoAlante()
    - Un único paso hacia adelante
- pasoAtras()
    - Un único paso hacia detrás
- cuelloOn()
    - Gira el cuello escaneando el entorno
- cuelloOFF()
    - Para el giro del cuello
- agacha()
    - Se agacha
- saluda()
    - Saluda con el brazo derecho

# 5- INTERFAZ PARA CONTROL DISTRIBUIDO DE ROBOT MICROBIRO

Explicaremos a continuación la interfaz desarrollada para manejar el robot de modo remoto. Como ya introdujimos, utilizaremos objetos distribuidos CORBA como mecanismo para comunicarnos remotamente con el servidor. A partir de esta interfaz cumpliremos los objetivos planteados en cuando al manejo de Microbiro desde varias plataformas (Windows y Linux) y lenguajes (Java y Ada).

## 5.1- DEFINICIÓN Y COMPILADO DE LA INTERFACE

Un sistema CORBA se genera a través de la definición de una interfaz en IDL. Nuestra interfaz local está programada en Java, con lo cual recurriremos a un mapeo de IDL a este lenguaje, utilizando el compilador *Orbacus* que se ofrece gratuitamente a través de la red.

Se definió la siguiente interfaz:

```
module Ubir{
interfaz Ubiro{

    typedef long address[8];
    boolean conectar();
    void setAddress(in address a);
    void setPort(in string port);
    long getSensor(in long numeroSensor);
    boolean enviar(in long msb,in long lsb, in long data1, in long data2);
    boolean andar();
    boolean para();
    boolean andaAtras();
    boolean pasoDerecha();
    boolean pasoIzquierda();
```

```
        boolean chuta();
        boolean saluda();
        boolean giroDerecha();
        boolean giroIzquierda();
        boolean levanta();
        boolean caer();
        boolean agacha();
        };
};
```

Para compilar, ejecutaremos en la consola el compilador *jidl.exe*, pasándole la interfaz como argumento. Al compilarla, se autogeneran las clases en lenguaje Java necesarias para generar el sistema CORBA y que acompañarán a los programas servidor y cliente, pues necesitan usar sus servicios, como ya veremos. Estas clases son: *Ubiro*, *UbiroHolder*, *UbiroPOA*, *_UbiroStub*, *UbiroHelper* y *UbiroOperations*

Para poder ofrecer finalmente una interfaz necesitamos implementar, por un lado, la clase del objeto que contenga el código que queremos ejecutar, a la que llamaremos Servant, y un programa Servidor que se encargue de:

- Crear dicho objeto.
- Registrarlo en el POA.
- Activar el ORB para que se mantenga a la espera.
- Ofrecer al programa Cliente una solución para localizar el objeto.

En nuestro caso, esta última cuestión la resolvemos utilizando un servidor de nombres, pero existen otros mecanismos como guardar la IOR del objeto en un fichero legible desde el cliente o escribirlo en pantalla para que lo resuelva otro programa.

## 5.2-SIRVIENTE Y SERVIDOR

A continuación, vamos a describir la implementación que debemos hacer para un programa que active el servidor de un objeto. Primeramente hemos de establecer el *Servant*. Ésta será la clase de la cual haremos objetos, por lo que debe

extender a *UbiroPoa*, que a su vez implementa *UbiroOperations* (o lo que es lo mismo, tiene que proporcionar código a los métodos definidos en la interfaz).

El código del *Servant* por tanto es muy sencillo, ya que disponemos de la API ya creada y sus método enumerados en el punto 4.2.3. Simplemente hemos de crear un objeto *Robot* y ejecutar sus instrucciones en los métodos de la interfaz que definimos.

El Servidor, por el contrario, es algo más complejo. Concretamente, en nuestro caso utilizaremos un servidor de nombres para registrar los objetos y le definiremos de modo que podamos simultanear el control de dos robots, creando por separado dos instancias del *Servant* con la misma conexión y registrándolas por separado. De este modo podremos obtener, desde un cliente, el control de dos Microbiro.

Para que el servidor haga todo esto tenemos que:

- Iniciar el ORB. Necesitamos pasarle como argumentos el host y el puerto donde se encuentra el Servidor de Nombres.
- Buscar en el ORB la referencia al *RootPOA* (tratado en la introducción 1.4.3) y convertirla (operación de *narrow*) a un objeto *POA.*
- Crear los objetos que queremos registrar, en nuestro caso primero creamos un objeto *Conexión* y se lo pasamos a 2 objetos distintos del *Servant*.
- Activamos estos objetos en el *POA* y obtenemos sus referencias IOR (introducido en el apartado 1.4.4).
- Buscamos el servidor de nombres. El ORB los buscará en la dirección IP y puerto especificados al inicializarlo. Posteriormente a obtener su referencia, la convertiremos a un objeto *CosNaming.NamingContext*. Se trata de un tipo propio de la librería CORBA con las utilidades y propiedades de un servidor de nombres.
- Creamos los identificadores (*CosNaming.NameComponent*) mediante los cuales localizaremos los objetos en el servidor de nombres.
- Registramos en el servidor de nombres conjuntamente objetos e identificadores.

- Finalmente activamos el *RootPOA* y inicializamos el ORB.

## 5.3-CLIENTE

Se trata de la clase que dará uso a los métodos que ofrece la interfaz, obteniendo una referencia al objeto registrado en el *POA* del Servidor para usarlo como un objeto local. Esta clase debe tener acceso a todo el código que autogenera el compilador IDL distribuido en las clases correspondientes y deberá obtener la referencia al objeto. En nuestro caso, esto se hará a través del servidor de nombres al que el Cliente accederá a través de la dirección IP y el puerto en el que se mantiene a la escucha. Veamos qué pasos se deben seguir para obtener un objeto que pueda ser utilizado como local:

- Se inicia el ORB, que dará soporte al programa donde funciona el Cliente.
- Se obtiene el Servidor de nombres.
    - El ORB buscará el objeto que le pidamos en la dirección IP y puerto que hemos indicado en la ejecución.
- Construimos las estructuras de identificadores que buscaremos en el servidor de nombres.
    - En Java se trata de un objeto *NameComponent*, pero en cada lenguaje hay distintos mecanismos.
- Buscamos las referencias y obtenemos los objetos.
- Convertimos mediante la función *narrow* dichos objetos al tipo deseado. Un objeto *Robot* en nuestro caso
- Ahora podemos usarlo como un elemento del programa local.

# 6-EJEMPLOS DE USO

## 6.1 CONTROL DISTRIBUIDO DE UN ROBOT

Ilustraremos todo el proceso descrito anteriormente mediante un ejemplo de uso. En este ejemplo utilizaremos un computador que funcionará como servidor conectado por USB a un adapatador XBee de la serie2, no muy alejado de una unidad Microbiro. Además, utilizaremos otro computador con sistema operativo Windows o Linux que podrá encontrarse en cualquier parte del mundo y que hará las veces de Cliente


### *6.1.1 SERVIDOR*

El objetivo del programa correspondiente a este ejemplo es el de operar con el robot desde una máquina remota distinta a en la que estamos ejecutándo. Para todos los casos que plantearemos a continuación el servidor será un PC con los siguientes componentes:

- Un adaptador USB para un dispositivo XBee serie2.

    - Este adaptador tendrá que tener instalado la versión ZB *Coordinator* API Mode como firmware del dispositivo (explicado en 2.1.1).
- Driver FTDI para el control del dispositivo XBee.
- JRE (Java Runtime Enviroment), es decir, el entorno de desarrollo Java con los compiladores y utilidades para la ejecución del programa.
    - El JRE (no necesariamente la última versión) contiene el servidor de nombres.
- Librería rxtxSerial.dll alojada en una carpeta del PATH (por ejemplo, WINDOWS/System32) para trabajar con el puerto serie USB.
- Programa servidor contenido en el ejecutable *Servidor.jar*. Un ejecutable con el programa Servidor que utiliza la API creada en el presente trabajo.

Para iniciar el funcionamiento del servidor, una vez instalado el adaptador y reconocido por el PC, debemos iniciar el servidor de nombres. Usaremos el que nos proporciona el entorno JRE de Java que se ejecuta desde consola, con la siguiente línea:

```
tnameserv –ORBInitialPort 2000
```

Por defecto, tnameserv funciona con el puerto 900. Dado que para probar un puerto distinto usararemos el puerto 2000, se lo especificamos con la etiqueta –ORBInitialPort.

Si el servidor de nombres inició correctamente, abrimos otra consola para iniciar el programa java del siguiente modo:

```
java –jar Servidor.jar –ORBInitialPort 2000
```

El fichero *Servidor.jar* es un ejecutable, por tanto, no necesita ser compilado. Necesitamos indicarle el puerto, aunque estemos funcionando en la misma máquina; sin embargo, no es necesario indicarle la dirección IP, puesto que por defecto la dirección es local.

En estos ejemplos correremos tanto el servidor de nombres como el servidor del programa en la misma máquina, pero sería perfectamente factible hacerlo en máquinas distintas. Simplemente deberíamos indicar en la ejecución de *Servidor.jar* la IP donde debe encontrar el servidor de nombres a través de la etiqueta:

```
–ORBInitialHost host.
```

### *6.1.2 CLIENTE JAVA*

El Cliente obtiene el objeto robot como, ya se describió con anterioridad, y lo usa como un objeto local, pudiendo así ejecutar las funciones que ofrecemos a través de la interfaz. Concretamente pide al usuario que elija a través del teclado una opción de las que ofrece el programa:

1. Conectar
2. Andar
3. Chutar
4. Saludar
5. Cambiar dirección
6. Datos de los sensores
7. Salir
8. Menú

Antes de poder realizar cualquier operación, será necesario ejecutar la opción 1 *conectar*. Ya comentamos que esta función iniciará la conexión del adaptador con el robot y enviará un mensaje de saludo al robot. Si la conexión no se establece en el primer intento repetirá la operación de nuevo, pues sobre todo en el primer uso el primer intento de comunicación lo usa para reconocer la ruta (no conecta directamente).

Una vez conectado podemos ordenar cualquier acción. Si en algún momento se pierde la conexión, el programa nos lo indicara mediante un mensaje de texto y podremos volver a conectarnos. Si queremos usar otro robot, tendremos que pulsar la opción 5 y pasarle número a número la *address* del nuevo robot que queremos utilizar. La opción 6 nos proporcionará las coordenadas de los 3 sensores de los acelerómetros X, Y, Z y el valor de la distancia. Finalmente para abandonar el programa pulsaremos la opción 7.

-EJECUCIÓN

El programa Cliente estará alojado en otra computadora, por lo que necesitará tener acceso por la red al servidor de nombres. Deberemos

proporcionarle la dirección IP y el puerto a través del cual permanece a la escucha y así establecer la comunicación remotamente. El PC en cuestión necesitará simplemente:

- Entorno Java JRE, versión 6 recomendada
- Librería *rxtxSerial* alojada en una carpeta del PATH
- Ejecutable *Cliente.jar*

Para la ejecución abriremos la consola de Windows, nos situamos en la carpeta que contiene el fichero *Cliente.jar* y escribimos:

*java –jar Cliente.jar –ORBInitialPort 2000 –ORBInitialHost host*

Si la dirección IP y el puerto son los correspondientes a los del servidor de nombres y la configuración de la red permite la comunicación, se iniciará el programa pidiendo al usuario que escoja la acción a realizar.

Disponemos de las versiones para Windows y para Linux, que son semejantes, pues el estándar de CORBA debería conseguir que nuestro programa fuera multiplataforma, como también debe suceder con la programación Java. No obstante, la versión Linux requiere una mínima modificación respecto a la otra versión: el servidor de nombres se llama *NameService* frente a *NamingService* en el caso del sistema operativo Windows.

### *6.1.3. CLIENTE ADA*

Uno de las grandes fuertes del uso de CORBA es la posibilidad de implementar el Cliente en un lenguaje de programación distinto al utilizado para el Servidor. Con objeto de demostrar esta capacidad se desarrolló un sencillo programa Cliente escrito en ADA [8] que interoperará remotamente con el Servidor Java expuesto en el punto 6.2.1.

El programa, en esencia, sigue el mismo procedimiento que un Cliente en Java como el que vimos antes; es decir, obtiene el ORB, luego busca el servidor de nombres y en él busca una referencia al objeto Robot que queremos, lo convierte a un objeto local y lo usa como si fuera parte del mismo. No obstante, alguna de las operaciones y caminos para lograr esto difieren un poco respecto a los correspondientes en el caso Java, contradiciendo en parte la idea de estándar que se pretende para la tecnología CORBA. No entraremos a analizar estos detalles por

tratarse de puro código de programación, lo cual no es objeto de análisis en esta memoria, simplemente comentar que se trata de diferencias en los nombres de algunos métodos y elementos que dificultaron la, a priori, sencilla tarea de transcribir el Cliente Java al nuevo Cliente Ada.

El programa, una vez obtenida la referencia del robot, le pide conectarse y espera 5 segundos a que se produzca el proceso, para después ordenar la acción de andar hacia adelante al robot. Como puede verse no se trata de un código sofisticado, simplemente es una prueba para comprobar el funcionamiento de PolyORB, versión CORBA para Ada que utilizamos para el desarrollo.

Para la ejecución, una vez compilado [9] y obtenido el ejecutable "*cliente*" ejecutaremos en la consola:

*cliente -ORBInitRef NameService=corbaloc::"Host":"Puerto"/NameService*

De este modo, estamos indicando al cliente que obtenga la referencia inicial *NameService* (Servidor de nombres) a partir de un *corbaloc* obtenido de la dirección IP y el puerto indicados, por protocolo IIOP (indicado mediante :: por defecto) y con el objetivo de encontrar *NameService* (escribiendo "/NameService" ).

## 6.2 CONTROL DISTRIBUIDO DE DOS ROBOTS

### *6.2.1 SERVIDOR*

Para este ejemplo el programa Servidor hará labores similares a las del ejemplo anterior. El único extra será el de registrar otro objeto más para poder usar dos robots por separado. Existe un problema en el manejo de un módulo XBee, y es que cuando un elemento toma uso del adaptador, bloquea el acceso de otras instancias. Por este motivo, para poder manejar simultáneamente dos robots, el Servidor creará un único objeto de la clase *Conexión* para ambos objetos *Robot* que crearemos y registraremos por separado. Por lo demás, los requerimientos para el PC y su ejecución son similares al ejemplo ya visto.

### *6.3.2.CLIENTE*

En este caso el programa cliente tendrá que localizar en el servidor de nombres dos objetos con sus identificadores correspondientes. Posteriormente se los asociará a dos robots distintos a los que se le asignará las direcciones correspondientes mediante la operación *setAddress*() que proporciona nuestra API. Esto es necesario, en el caso del segundo robot, pues la dirección asociada por defecto a los objetos es la que corresponde al Robot 1. Tras configurar ambos Microbiros, continuará el resto del operaciones del programa.

El ejemplo en particular es un programa para poder realizar operaciones simultáneas con dos robots. El programa tiene dos fases diferenciadas: elección del destinatario de la operación y elección de la operación.

1. Escoge robot:
    1) Pulsando 1, escogeremos operar sobre el Robot 1 (el que así hayamos definido internamente).
    2) Pulsando 2, escogemos enviar las instrucciones al Robot 2.
    3) Pulsando 3, operaremos simultáneamente sobre ambos robots.
2. Escoger Operación:
    1) Conectar
    2) Andar
    3) Chutar
    4) Saludar
    5) Cambiar dirección
    6) Datos de sensores
    7) Salir
    8) Menú

Tras enviar una operación el programa vuelve al punto 1 donde volver a escoger destinatario de la orden

Por lo demás los requerimientos y la ejecución de este programa son similares a los ya comentados en el anterior ejemplo. Hemos de considerar que como la Conexión será común para ambas instancias (por usar el mismo módulo

XBee) cualquiera de los robots podrá llamar a la operación *Conectar*, y una vez establecida la conexión no hará falta que el otro robot la inicie para su propio uso.

Como ocurría en el caso anterior, esta versión de Cliente esta disponible tanto para el uso en Windows como en Linux.

# 7.CONCLUSIONES

A tenor de los buenos resultados obtenidos podemos considerar a CORBA como una tecnología adecuada para el control distribuido de sistemas, con múltiples aplicaciones en diversos campos como, por ejemplo, la robótica o, el control de las condiciones de un laboratorio, por citar algunas aplicaciones. Podemos definir una aplicación CORBA sencilla de hacer pero especialmente sencilla de utilizar y programar con ella. La verdadera fuerza en su aplicación es que futuros programadores podrán usar la API distribuida como una API local sin necesitar conocimientos especiales sobre comunicaciones ni el *Middleware* ORB.

Por otro lado el uso de ZigBee para las comunicaciones inalámbricas ha sido altamente satisfactorio. Su funcionamiento ha resultado ser muy estable y el consumo de baterías realmente bajo. Su manejo gracias a la API que DIGI ofrece para el manejo de sus módulos, es bastante asequible y cómodo, lo que permite utilizarlo para desarrollar aplicaciones complejas.

Hemos conseguido finalmente una API práctica e intuitiva apta para cumplir todos los objetivos planteados. Funciona correctamente tanto para el manejo de un robot como para dos, y lo verdaderamente interesante, hemos logrado que funcione para aplicaciones multiplataformas probando su uso en sistemas Windows y Linux a través de Java y Ada. Ésto convierte a nuestro trabajo en apto para un uso heterogéneo, lo cual, hoy en día, es uno de los principales objetivos de cualquier aplicación.

# 8.BIBLIOGRAFÍA

[1] William Stallings. "Organización y Arquitectura de Computadores" 5ªEdición Editorial Pearson Pentice Hall

[2] http://es.wikipedia.org/wiki/Memoria_(inform%C3%A1tica)

[3] www.zigbee.es

[4] http://www.xbee.cl/diferencias.html

[5] http://code.google.com/p/xbee-api/

[6] http://web.progress.com/en/orbacus/orbacus_index.html

[7] Object Management Group. "CORBA Core Specification". OMG Document, v3.0formal/02-06-01, July 2003.

[8] S. Tucker Taft, Robert A. Duff, Randall L. Brukardt, Erhard Ploedereder, and PascalLeroy (Eds.). "Ada 2005 Reference Manual. Language and Standard Libraries.International Standard ISO/IEC 8652:1995(E) with Technical Corrigendum 1and Amendment 1". LNCS 4348, Springer, 2006.

[9] Ada-Core Technologies, The GNAT Pro Company, http://www.adacore.com/

# GUÍA DE USUARIO PARA TRABAJAR CON MICROBIRO

# ÍNDICE

# 1-GUÍA DE USUARIO

En esta guía describimos algunos detalles importantes para trabajar con Microbiro y desarrollar software a partir de la API que se ha realizado para el control inalámbrico por ZigBee y el manejo remoto a partir de objetos distribuidos CORBA

En el apartado 2 describiremos las utilidades y programas necesarios para componer el entorno de desarrollo necesario para trabajar con Microbiro y con la API.

En el punto 3 explicaremos cómo hacer modificaciones en el *firmware* de Microbiro trabajando en desarrollo cruzado y como utilizar un programa de comunicaciones por puerto serie para realizar el "debug"

Por último, en los sucesivos puntos 4 y 5 detallaremos el modo de trabajar con la API y con la interfaz definida por CORBA respectivamente

# 2-ENTORNO DE DESARROLLO DE MICROBIRO

Describiremos en este apartado todo lo que necesitamos para trabajar y funcionar con Microbiro, así como la manera en la que podemos descargar de forma gratuita todas las herramientas necesarias.

## *2.1. ECLIPSE*

En primer lugar necesitamos un entorno de desarrollo para programar en Java. Hemos utilizado y recomendamos el uso de la herramienta *Eclipse IDE -Java EE Developers* que puede descargarse desde

http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/heliossr2

## *2.2. PROGRAMMERS NOTEPAD EDITOR*

Para trabajar con el *firmware* del robot y cargarlo en la placa del mismo, recomendamos el uso de *Programmers Notepad Editor*, que puede descargarse desde http://www.pnotepad.org/download/

## *2.3. X-CTU*

La configuración del módulo XBee, así como la descarga y carga del *firmware* del mismo se hace desde un software que distribuye la compañía DIGI llamado X-CTU. Podemos obtenerlo en

http://www.digi.com/support/productdetl.jsp?pid=3352&osvid=57&s=316&tp=5&tp2=0

## *2.4. DRIVER FTDI*

Para trabajar desde nuestro PC con el módulo XBee necesitamos un driver FTDI que podremos descargar de http://www.ftdichip.com/Drivers/D2XX.htm

## *2.5. PAQUETE DOCUMENTACIÓN*

Se necesitará nuestro paquete de ficheros que contendrá el *firmware* del Robot, los ejecutables de los programas ejemplo, un banco de trabajo para Eclipse con los proyectos para los distintos programas, la API desarrollada y el programa de teleoperación que trae Microbiro con el que realizaremos la calibración de sus servos. Dentro de estos ficheros encontraremos también todas las librerías necesarias.

## 2.6. API XBEE

Puede ser necesaria la descarga, si se parte de cero a la hora de desarrollar alguna aplicación con nuestra API, la API de XBee que DIGI distribuye desde el siguiente enlace http://code.google.com/p/xbee-api/downloads/list

## 2.7. HERCULES

Por último conviene contar con un programa para controlar las comunicaciones de envío y recepción de los puertos USB. Recurrimos a la herramienta Hercules que permite recibir así como enviar comandos directamente por un puerto serie. Nos resulta especialmente útil a la hora del "debug" del *firmware* de Microbiro y su uso es bastante sencillo. Se puede obtener desde aquí

http://www.hw-group.com/products/hercules/index_en.html

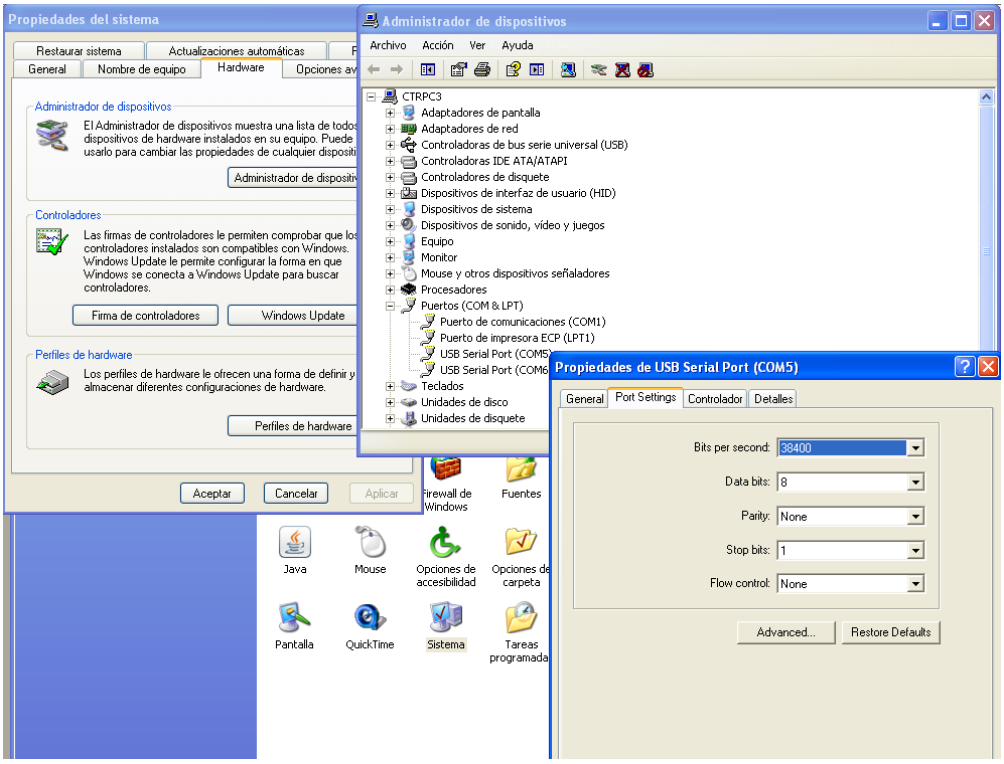
# 3-CÓMO DESARROLLAR SOFTWARE PARA MICROBIRO

Explicaremos en este apartado cómo trabajar con PN Editor para el desarrollo cruzado y trabajar con el *firmware* del robot.

## *3.1-Desarrollo cruzado con PN Editor*

Lo primero de todo es abrir el proyecto. Para ello pulsamos "File/Open Project" y seleccionamos el fichero de tipo ".pnproj" que todo proyecto PN debe traer consigo. Una vez hecho esto, tendremos en la ventana *Projects* la estructura de carpetas y los ficheros de nuestro proyecto en cuestión. Haciendo doble *click* abriremos el editor de cada fichero con el que podremos modificar y guardar.

Ahora debemos conectar la placa por USB al PC. Si tenemos correctamente instalado el driver, todo se hará automático. En "Panel de control/Sistema/Hardware/Administrador de dispositivos" se puede ver el puerto COM por el que se ha conectado, cambiarlo y definir el *Baud Rate* y el *Alive Interval*. Algo que seguramente será necesario cambiar más adelante (ver figura)



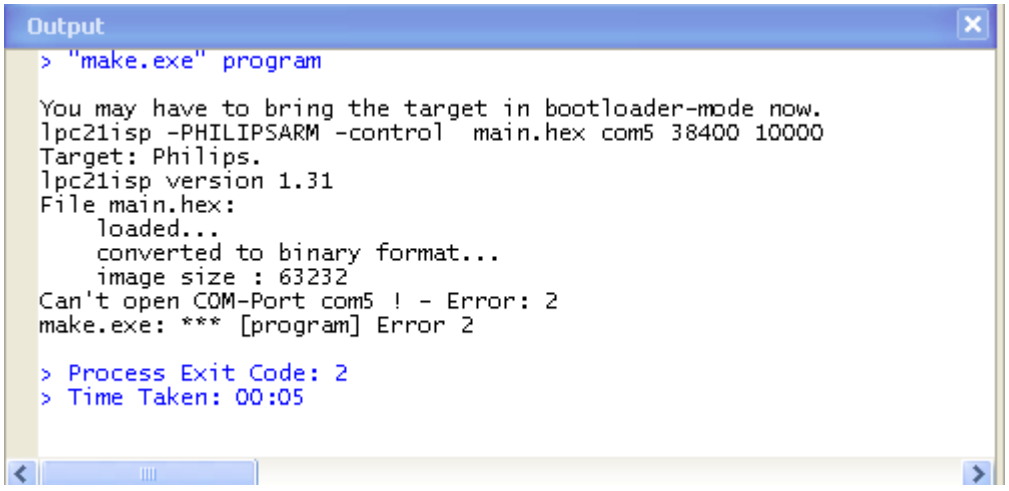
Concretamente el *Baud Rate* para cargar el *firmware* en la placa de Microbiro debe ser de 38400 y un *Alive interval* de 10000.

Las utilidades para el compilado o la carga del *firmware* a través de PNEditor vendrán de la mano del proyecto y aparecerán en la pestaña superior "Tools". Tenemos dos opciones: "Make All" compilará el programa, para ello es necesario estar editando el fichero "main.c" y tenerlo en *focus*. La otra opción importante es "Make Program" que generará los archivos binarios (si no hemos compilado antes) y cargará el programa en la placa.

## CARGA DEL PROGRAMA

Para la carga del programa es importante tener configurado el puerto como vimos anteriormente, y además tener colocado correctamente el jumper de configuración del robot. Si todo está listo, pulsaremos "Make Program" y comenzará el proceso de carga.

Es posible que la primera vez no acertemos con el puerto por el que se realiza el proceso, en tal caso veremos el siguiente mensaje de error:

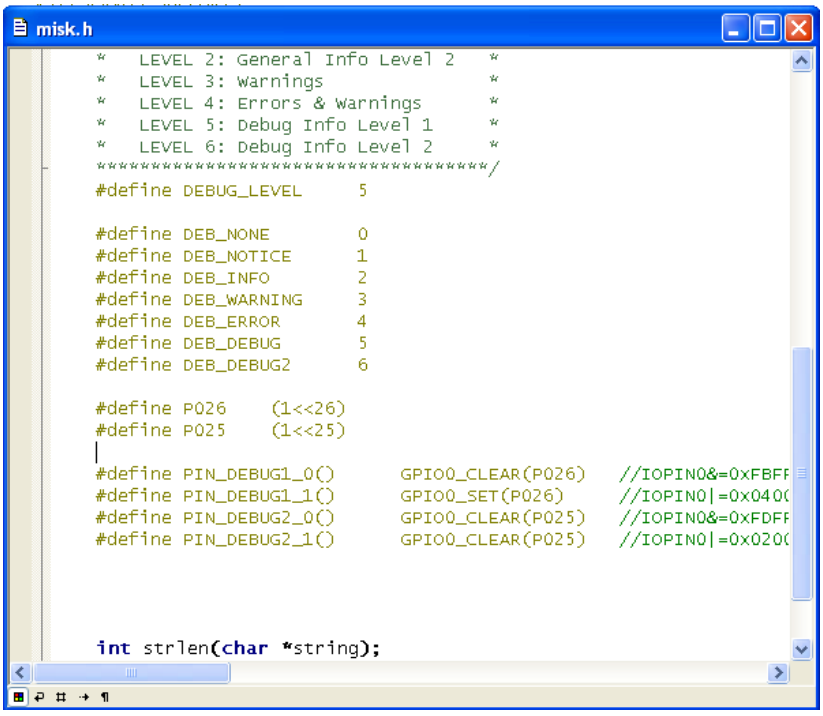



En él, vemos que PNEditor está intentando conectarse por el puerto COM5 a 38400 de *Baud Rate* y 10000 de *Alive Interval*. Si el robot está encendido, puede ser que no esté asociado al puerto COM5 y deberemos buscar en "Administrador de Dispositivos" la conexión USB de nuestro Robot y asignarle el puerto COM5 para que la carga se realice.

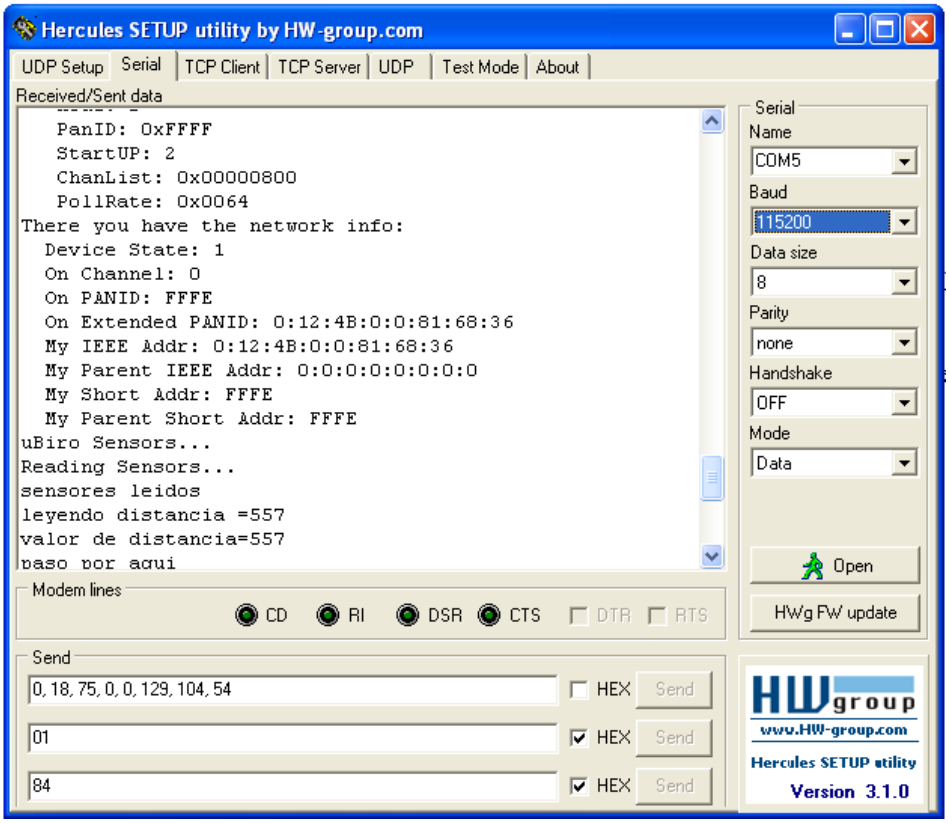
Si la carga se ha realizado con éxito, simplemente debemos retirar el jumper de configuración y resetear Microbiro (tal vez hagan falta un par de intentos).

## *3.2-USO DE HERCULES PARA EL DEBUG*

Una vez instalado correctamente el *firmware* podemos utilizar un programa de comunicaciones por puerto serie para ver cómo se comporta Microbiro ante distintos procesos. La UART de la placa del robot envía por el puerto USB todo aquello que le vayamos indicando durante nuestra programación a través de la función *doc()* . Este comando admite una serie de prioridades como primer parámetro definidas en el fichero "misk.c", y posteriormente un mensaje de texto:



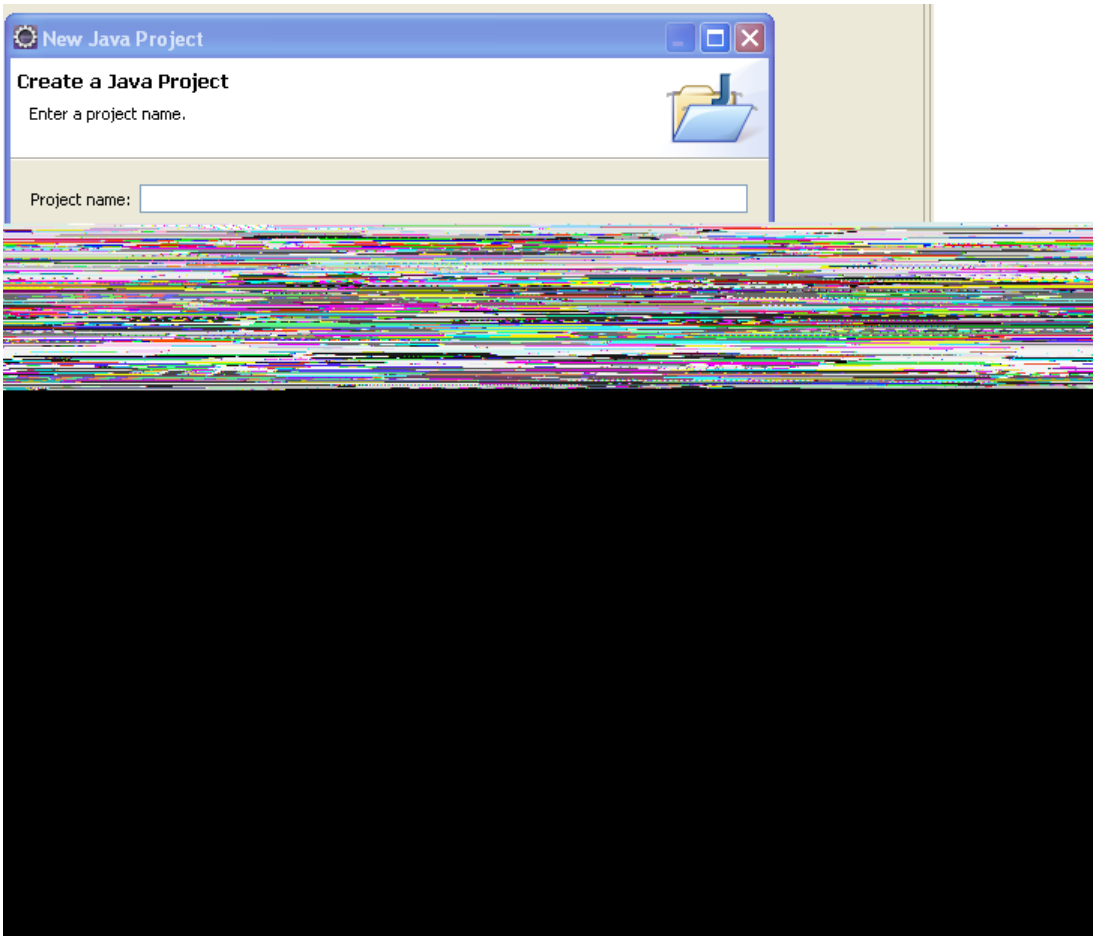

Podremos por tanto leer esta serie de mensajes de "debug" que el código del *firmware* vaya produciendo a través de un programa de comunicación por puerto serie. En nuestro caso utilizamos Hercules, realmente sencillo y práctico. Debemos configurarlo con el puerto correspondiente y a la velocidad de 115200, pues a esa velocidad se abre en el firmware la UART:




A través de Hercules también podremos enviar comandos, o resetear la placa activando y desactivando la casilla DTR.

# 4.-CÓMO DESARROLLAR SOFTWARE DE COMUNICACIÓN ZIGBEE CON MICROBIRO

Utilizaremos Eclipse para trabajar con nuestra API de comunicación. Partiremos del proyecto ya creado y contenido en la carpeta "API" que viene en la documentación que se ofrece. Para cargar el proyecto en Eclipse simplemente iniciamos el programa, creamos una carpeta de trabajo (*workspace*) y cuando ya estemos en la ventana principal, pulsamos: "File/new/ Project/Java Project" y accedemos a la siguiente ventana:



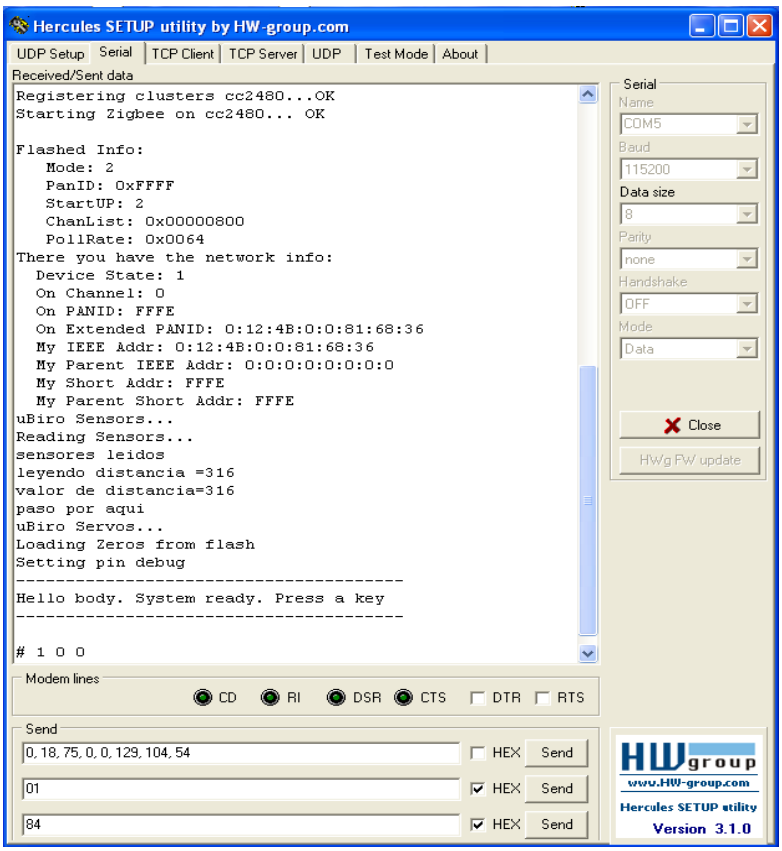

Para trabajar a partir de un proyecto ya creado, seleccionamos "Create Project from existing source", y seleccionamos la carpeta contenedora, en nuestro caso será la carpeta API.

Hecho todo esto, disponemos en principio de todas las clases necesarias para trabajar con la API y sus librerías y poder trabajar con ellas para desarrollar software que trabaje con ZigBee para comunicarse con Microbiro.

Para trabajar con la API, primeramente se debe crear un objeto de la clase Conexión especificándole al constructor el puerto por el que está conectado el módulos XBee y el *Baud Rate* del mismo. El puerto debe darse como un *String* con la palabra "COM" seguido del número del puerto en cuestión, es decir, para indicar el puerto 5, daremos el argumento "COM5". El *Baud Rate* será un entero con el valor con que tenemos definido en la configuración del puerto.

Tras crear el objeto de Conexión podremos obtener la instancia de la clase Robot con la que podremos ejecutar todos los métodos necesarios para manejar Microbiro. Robot necesita el objeto de Conexión como argumento en el constructor y nada más. Ahora podemos definir la dirección del robot a través de setAddress() si no lo hemos definido directamente modificando la clase y poniendo el valor por defecto que vamos a usar. Para poder ejecutar los movimientos o tomar los datos de los sensores, debemos ejecutar antes *conectar()* . El proceso de conexión dura unos segundos y no siempre conecta en el primer intento pues hace un reconocimiento de la ruta; sin embargo, no debería haber problemas en la segunda tentativa. En tal caso, se debe revisar que la *address* sea la correcta, el robot esté encendido y su *firmware* haya cargado correctamente y se encuentre en la posición inicial (con Hercules lo sabremos fácilmente si en la carga del programa, el robot ha enviado la frase "Hello Body, System ready."). Si el problema persiste, deberemos resetear el robot o incluso recargar sus baterías si no hay manera de conectarnos.

En la siguiente imagen vemos la secuencia de datos que Microbiro envía a través de la comunicación USB cuando arranca correctamente. Podemos ver además la MAC del mismo en "My IEEE Addr: 0:12:4B:0:0:81:68:36" a partir de la cual podemos obtener la *address* como vimos en la memoria.

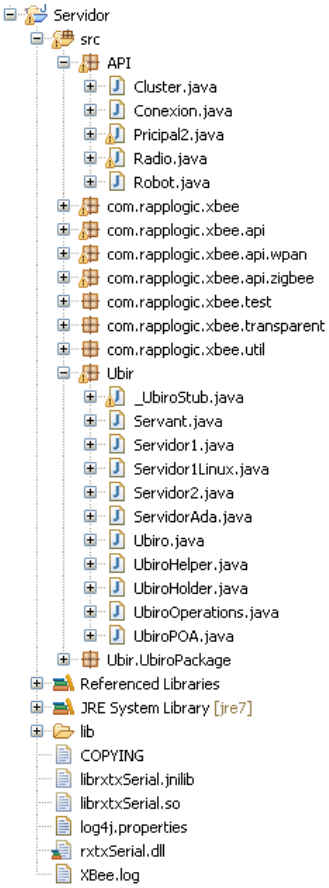



Ahora, si pretendemos trabajar con varios robots, es tan fácil como crear nuevos objetos de la clase Robot y definir su *address* correctamente. Si utilizamos el mismo módulo XBee para manejar todos, únicamente necesitamos un objeto Conexión y lo usaremos como parámetro para la creación de todos los robots. Si esto es así, solamente un robot deberá ejecutar *conectar()*, este robot se encargará, por así decirlo, de abrir la conexión, que como va a ser única para todos, estará ya disponible para ser usada por cualquier objeto. En resumen, necesitamos tantos objetos de Conexión como módulos XBee utilicemos, y debemos ejecutar el método *conectar()* para cada módulo distinto desde un robot cualquiera de los que lo utilizan.

# 5-CÓMO DESARROLLAR PROGRAMAS DISTRIBUIDOS

Para objetos distribuidos con Microbiro utilizaremos la tecnología CORBA de la que ya hablamos ampliamente en la memoria. Contamos en la documentación que ofrecemos con proyectos ya creados para Eclipse para el programa Servidor y para Clientes de uno y dos robots.

## *5.1-Programa Servidor*




Como ya hicimos en el apartado 4 debemos crear para Eclipse un nuevo proyecto a partir de la carpeta "Servidor" que trae la documentación. Siguiendo los pasos que antes retratamos tendremos preparado todo para trabajar con ello.

Dentro del proyecto Servidor encontramos en su directorio "src" distintos paquetes de Clases. Por ejemplo, contamos con la API desarrollada, necesaria para que el Servant cree los objetos que luego se van a registrar. El paquete importante, por otro lado, es "Ubir" en el que encontramos todas las clases necesarias para que funcione un servidor CORBA. Aparte de las clases autogeneradas por el compilador de IDL, contamos con el Servant, y una serie de programas Servidor para funcionar con 1 robot (Servidor1), dos robots (Servidor2), un robot para clientes programados en Ada95 (ServidorAda) y un servidor para clientes Java desde sistemas operativos Linux (ServidorLinux).

Como ya vimos en la memoria, el Servant se encarga de implementar a partir de nuestra API todos los métodos que la interface de IDL define. El servidor por otro lado crea un objeto de Conexión y se lo pasa como argumento para el constructor del objeto de Servant que luego registraremos en el POA y cuya referencia obtendremos del servidor de nombres

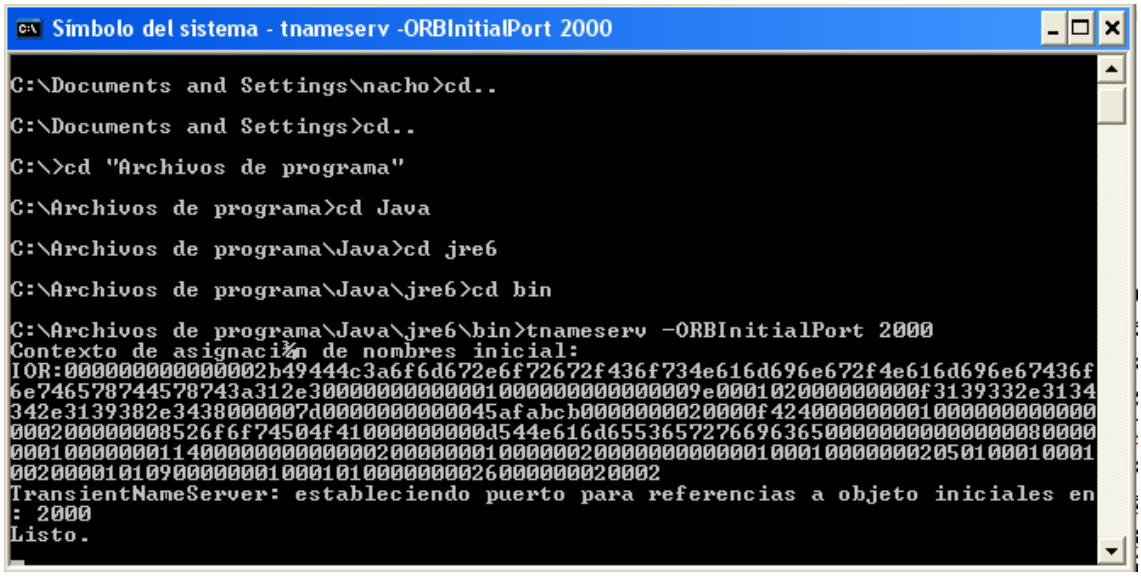
## Ejecución

Utilizaremos “tnameserv.exe” como servidor de nombres. Se trata de un servicio que JRE trae consigo desde las versiones más primitivas y que deberemos ejecutar antes de iniciar el Servidor. Para ello desde la misma consola de Windows (Inicio/Programas/Accesorios/Símbolo de Sistema) nos movemos hasta la carpeta “bin” de la versión de JRE instalada (en mi caso JRE6). Allí ejecutamos la siguiente linea:

tnameserv –ORBInitialPort 2000

A partir de la cual especificamos mediante la etiqueta “-ORBInitialPort” el puerto a través del cual se accede desde fuera al servidor de nombres que estamos ejecutando, 2000 en nuestro caso.

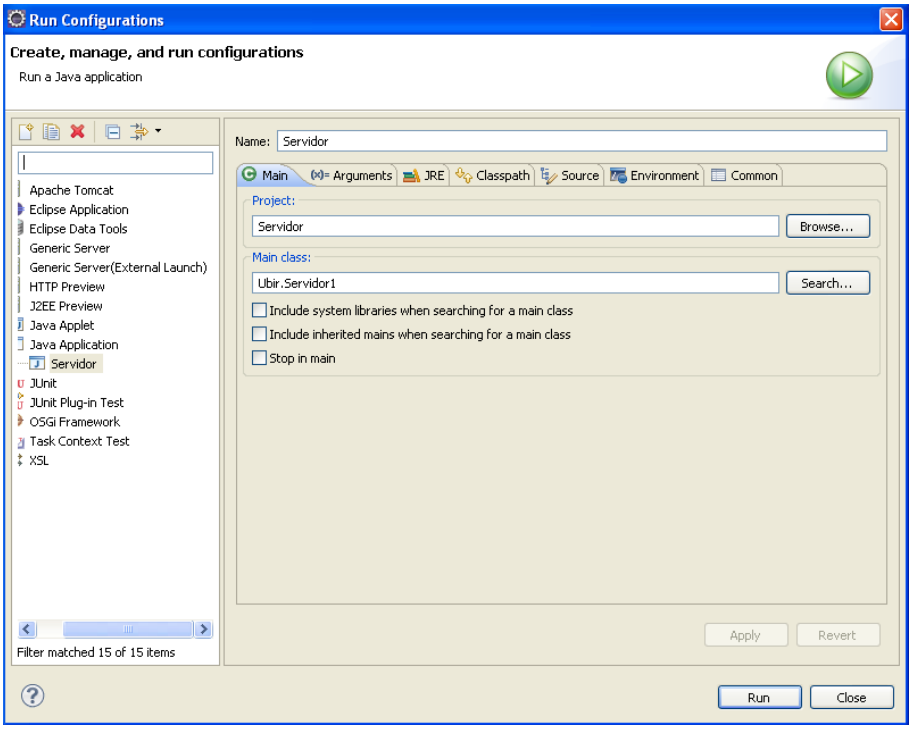
Vemos en la figura un ejemplo de la ejecución donde tnameserv, al iniciar correctamente, nos indica su IOR propio, y el puerto utilizado (si no se especifica tomará el puerto 900)



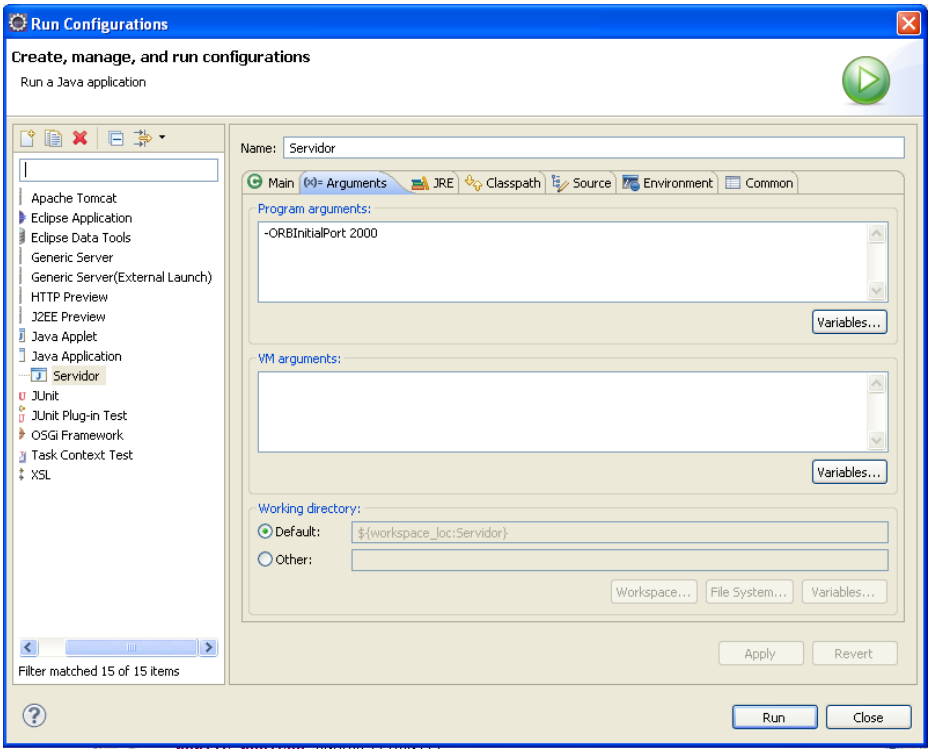

Una vez iniciado tnameserv, ya podemos ejecutar el programa servidor que deseemos según nuestras necesidades (uno o dos robots, Linux o Ada95). Tendremos dos opciones: o bien ejecutarlo desde Eclipse, o bien utilizando el ejecutable que ofrecemos en la documentación

*Desde Eclipse*

Desde Eclipse deberemos pulsar sobre la clase Servidor que vayamos a utilizar con el botón derecho y seleccionar "Run as/Run Configuration" y aparecerá la siguiente ventana:




Si necesitamos pasarle opciones de ejecución, lo haremos a través de la pestaña "Arguments" del siguiente modo:



Aquí en nuestro caso especificamos la etiqueta "-ORBInitialPort 2000" para que encuentre el servidor de nombres correctamente. Si además estuviéramos haciendo la ejecución desde otro ordenador, deberíamos especificar el Host donde se encuentra tnameserv mediante la etiqueta "–ORBInitialHost" y especificar a continuación la IP de dicha computadora.

Tras esto, pulsamos "Apply" para guardar cambios y "Run" para iniciar la ejecución.

*Desde Ejecutable*

En la documentación se pueden encontrar una serie de ficheros .jar . Se trata de ficheros ejecutables para los distintos tipos de Servidor con los que contamos y que podemos ejecutar directamente desde la consola de Windows.

Todas las librerías necesarias están contenidas dentro del fichero, pero en ocasiones es necesario colocar la librería "rtxSerial.dll" dentro de una carpeta que pertenezca al PATH de Windows, por ejemplo, en "Windows/System32".

Para ejecutar, deberemos abrir la consola y situarnos en la carpeta que contiene el fichero .jar. Tomaremos como ejemplo Servidor.jar para lo cual ejecutaremos la siguiente línea:

java –jar Servidor.jar –ORBInitialPort 2000 –ORBInitialHost localhost

No requiere compilación pues ya disponemos del ejecutable, por lo que simplemente indicamos que vamos a ejecutar un fichero .jar mediante la etiqueta "-jar". Debemos indicar al programa el puerto y el host donde se encuentra el servidor de nombres, que para este ejemplo será a través del 2000 en el mismo ordenador donde estamos ejecutando (lo indicamos a través de la etiqueta "localhost"). En caso de ser en otro ordenador deberíamos especificar la dirección IP del mismo.

## *5.2-Programa Cliente*

Una vez ejecutado el Servidor, con el objeto registrado en el servidor de nombres y el POA activado, podemos trabajar con dicho objeto a través de un programa Cliente. Del mismo modo que hicimos antes con la carpeta Servidor de la

documentación, podemos trabajar con Eclipse desde proyectos ya creados de Clientes. Seguimos los pasos antes descritos y tendremos dentro del Eclipse la jerarquía de carpetas y clases que necesitamos para varios casos y ejemplos ya implementados.

A partir de estos ejemplos se puede empezar a crear software distribuido. Cualquier ejemplo de Cliente realiza la tarea de encontrar las referencias a los objetos que el programa Servidor correspondiente registra y los utiliza para trabajar como si fueran objetos locales. A partir de ellos podemos desarrollar cualquier programa utilizando el robot Microbiro de forma remota.

Si por ejemplo vamos a trabajar con un robot Microbiro desde un Cliente Java en Linux debemos ejecutar "ServidorLinux" de entre las opciones de servidores que tenemos. El Cliente deberá buscar el objeto remoto a partir del mismo identificador con el que el servidor lo registró, es decir, en este caso el "id" y el "kind" del objeto *NameComponent*, que sirve como identificador, son "Robot" y "1" respectivamente. El programador deberá prestar atención a esto según qué servidor utilice.

La ejecución es muy parecida a la que vimos para el caso del programa Servidor, se procede del mismo modo tanto para ejecutar desde Eclipse o desde consola a través de los ficheros .jar que ofrecemos