DISEÑO E IMPLEMENTACIÓN DE UNA INTERFASE PARA EL CONTROL VÍA WEB DE UN ROBOT MÓVIL Y MONITORIZACIÓN DE SU ENTORNO DE OPERACIÓN PARA VISITAS GUIADAS REMOTAS

JOSÉ MANUEL MARTÍNEZ GARCÍA

Facultad de Informática.
Universidad de Las Palmas de G.C.

Facultad de Informática. Universidad de Las Palmas de G.C.

Proyecto fin de carrera

Título: Diseño e implementación de una interfase para el control vía web de un robot móvil y monitorización de su entorno de operación para visitas guiadas remotas.
Apellidos y nombre del alumno: Martínez García, José Manuel.
Fecha : 02 de Febrero de 2010

Tutor: Hernández Sosa, José Daniel
Cotutor: Falcón Martel, Antonio
Cotutor: Cabrera Gámez, Jorge

Facultad de Informática. Universidad de Las Palmas de G.C.

Agradecimientos

Este proyecto no se podría haber llevado a cabo sin el apoyo que muchas personas me han aportado durante toda la elaboración del mismo. Es justo hacerles una mención aquí en reconocimiento a su ayuda todo este tiempo.

A mis tutores, agradezo el haberme ayudado a lograr mis objetivos y reconocer sus labores docentes no sólo en este trabajo, sino en las tantas asignaturas que me han impartido a lo largo de toda mi carrera.

En mención especial, agradezco al investigador D. Antonio Carlos Domínguez Brito el haberme "enrolado" en este proyecto. Aunque ha sido un camino largo y arduo en algunos momentos, siempre me ha ayudado a ver la luz al final del túnel. Asimismo, Le doy gracias por todos los conocimientos que me ha aportado. Sin él no hubiera sido posible dar a luz a este trabajo.

Gracias a mi madre, a mi padre, a mi hermana, a mis sobrinos, a Minerva y a todos mis amigos por la cantidad de ánimos que me han dado desde que empecé este proyecto.

De igual forma agradezco a Soraya haber participado en este proyecto, por sus sugerencias y optimismo en creer que un día como hoy conseguiríamos llegar a la meta.

Facultad de Informática. Universidad de Las Palmas de G.C.

Índice general

1. Introducción **11**
- 1.1. Introducción al problema . . . 11
- 1.2. Robots *online* y teleoperación . . . 12
- 1.3. Teleoperación en una red local (LAN) . . . 13
- 1.4. Teleoperación a través de redes de amplio alcance (WAN) . . . 14
- 1.5. Telepresencia . . . 15
- 1.6. Objetivos . . . 16
  - 1.6.1. Objetivos académicos . . . 16
  - 1.6.2. Objetivos específicos . . . 16
  - 1.6.3. Visión del sistema final . . . 17
- 1.7. Estructura del documento . . . 19

2. Estado actual del tema **21**
- 2.1. Sistemas robóticos percepto-efectores y teleoperados . . . 21
- 2.2. Middleware's para la programación de aplicaciones distribuidas . . . 23
- 2.3. Sistemas robóticos teleoperados vía Web . . . 24

3. *CoolBOT* **27**
- 3.1. Modelado de componentes . . . 28
- 3.2. Variables observables y controlables . . . 28
- 3.3. Autómata por defecto . . . 30
- 3.4. Componentes multihilo . . . 31
- 3.5. Intercomunicación entre componentes *CoolBOT* . . . 32
- 3.6. Tipos de puertos y conexiones . . . 33
- 3.7. Componentes compuestos . . . 34

4. Metodología, recursos y plan de trabajo **37**
- 4.1. Metodología . . . 37
  - 4.1.1. El proceso de desarrollo . . . 37
  - 4.1.2. Aplicando el Proceso Unificado . . . 39
- 4.2. Recursos necesarios . . . 40
  - 4.2.1. Recursos hardware . . . 40
  - 4.2.2. Recursos software . . . 40

4.3. Plan de trabajo 42

5. Análisis **43**
5.1. Modelo de dominio 43
5.2. Modelo de casos de uso 47
5.3. Actores 47
5.4. Casos de uso 48
5.4.1. Operador 49
5.4.2. Desarrollador de sistemas robóticos 49
5.4.3. Desarrollador de *CoolBOT* 51
5.5. Requisitos del sistema 54
5.6. Representación intermedia, *Marshalling* y *Demarshalling* 56
5.7. CDR: CORBA's Common data representation 59
5.7.1. Introducción 59
5.7.2. Sintaxis de transferencia CDR 59
5.7.3. Tipos primitivos 60
5.7.4. Tipos construidos 60
5.7.5. Alineamiento 61
5.8. Entorno de comunicación adaptativo: ACE 62
5.8.1. Introducción 62
5.8.2. ¿Qué proporciona ACE en comparación con las API's de C? 64
5.8.3. Manejo de memoria en ACE 65
5.8.4. *CDR Streams* en *ACE* 67
5.8.5. Ventajas de *ACE* 67
5.9. Interfaces gráficas en Java: *Swing* 68
5.9.1. Independencia de la plataforma 68
5.9.2. *Modelo Vista Controlador (MVC)* 68
5.9.3. Ventajas de usar *Swing* frente a *AWT* 69
5.10. Encapsulado de código nativo, *Swig* 69
5.10.1. Lenguajes soportados 69
5.10.2. ANSI C 70
5.10.3. ANSI C++ 71
5.10.4. Preprocesador 72
5.10.5. Opciones configurables 72

6. Diseño **73**
6.1. Operaciones de *Marshalling* de datos 73
6.2. Diseño del protocolo DC3P 74
6.2.1. Introducción al diseño de protocolos 74
6.3. Especificación del protocolo *DC3P* 74
6.3.1. Descripción general del servicio 75
6.3.2. Descripción del entorno de ejecución 76
6.3.3. Vocabulario y tipos de mensajes 78

6.3.4. Formato de los mensajes . . . 78
6.3.5. Reglas de procedimiento . . . 84
6.4. Componentes *Coolbot* con soporte de red . . . 89
6.4.1. Modelo de componente sin red . . . 89
6.4.2. Modelo de componente con red . . . 91
6.5. *Sondas de componentes* . . . 96
6.6. Construcción de *vistas* mediante *sondas* de componentes . . . 97
6.7. Interfaz de teleoperación Java . . . 103
6.8. Acceso a las *sondas* a través de *Swig* . . . 105

7. Detalles de diseño **107**
7.1. Diseño de los mensajes del protocolo DC3P . . . 107
7.2. Patrones de diseño utilizados . . . 110
7.2.1. Prototype . . . 110
7.2.2. Patrón Modelo Vista Controlador: MVC . . . 110

8. Detalles Implementación **115**
8.1. Diseño de las clases *PortPacket* y *PacketBody* . . . 115
8.1.1. CloningInterface . . . 115
8.1.2. DeepCopyingInterface . . . 116
8.1.3. PackingInterface . . . 117
8.1.4. NamingInterface . . . 118
8.1.5. DebugginInterface . . . 118
8.2. Empaquetado de datos de forma genérica y transparente al usuario . . . 118
8.3. PackingMaxLength . . . 124

9. Pruebas y resultados **127**
9.1. Componente *PlayerRobot* . . . 127
9.2. Componente *GridMap* . . . 128
9.3. Componente *NDNavigation* . . . 129
9.4. Componente *ShortTermPlanner* . . . 129
9.5. Vista PlayerRobotJavaView . . . 130
9.6. Vista SphereJavaView . . . 130
9.7. Vista GridJavaView . . . 131
9.8. Conexionado del sistema completo . . . 132
9.9. Pruebas realizadas . . . 132
9.9.1. Test1: Sistema sin uso de red . . . 133
9.9.2. Test2: Sistema teleoperado con componentes en local . . . 133

10. Conclusiones y trabajo futuro **139**
10.1. Conclusiones . . . 139
10.1.1. Conclusiones generales . . . 139
10.1.2. Conclusiones experimentales . . . 142

10.2. Trabajo futuro 143

A. Manuales de usuario **145**
A.1. Guía de creación de comandos del protocolo DC3P 145
A.1.1. Directorio 145
A.1.2. Clase UserCommand - Fichero .h 146
A.1.3. Clase USER_COMMAND - Fichero .cpp 151
A.2. Guía de creación de paquetes de datos 152
A.3. Instalación sistema de evitación 156
A.3.1. *CoolBOT* 156
A.3.2. Componente GridMap 158
A.3.3. Componente NDNavigation 160
A.3.4. Componente ShortTermPlanner 162
A.3.5. Componente PlayerRobot 163

Índice de figuras

1.1. Vista de un escenario teleoperado 12
1.2. Visión estática del sistema final 18
1.3. Visión dinámica del sistema final 18

3.1. Vista externa de componente. 28
3.2. Vista interna de componente. 28
3.3. Vista externa de componente con puerto de control y monitorización. 29
3.4. Bucle común de control. 29
3.5. Autómata por defecto. 30
3.6. Componente multihilo. 32
3.7. Puerto *MultiPacket*. 34
3.8. Conexiones MultiPacket simples ($\forall\, n,m \in \mathbb{N}; n,m \geq 1$). 34
3.9. Jerarquía de componentes. 35

4.1. Desarrollo iterativo e incremental 38
4.2. Proceso Unificado aplicado a este proyecto 40

5.1. Modelo del dominio de un componente *CoolBOT* 45
5.2. Modelo del dominio del sistema distribuido 46
5.3. Actores 48
5.4. Diagrama de caso de uso para el actor *operador* 49
5.5. Diagrama de caso de uso para el actor *desarrollador de componentes* CoolBOT . . . 52
5.6. Diagrama de caso de uso para el actor *desarrollador de CoolBOT* 53
5.7. Ejemplo de Arquitectura Von Neumann 57
5.8. Organización Little-Endian. 57
5.9. Organización Big-Endian. 57
5.10. Bytes necesarios para almacenar diferentes tipos de caracteres en la representación UTF-8. Los caracteres ASCII (C,t y d) requieren sólo un byte. Los caracteres latinos y griegos (á, ö, y Ø) requieren 2 bytes. Los caracteres asiáticos requieren 3 bytes. Los caracteres adicionales (clave de sol) requieren 4 bytes de almacenamiento. . . . 58
5.11. Ejemplo de *marshalling* de código según especificación IDL 61
5.12. Estructura y funcionalidad de ACE [ACE,2007] 63
5.13. Mensaje simple. 66
5.14. Mensaje compuesto. 66

6.1. Visión a alto nivel del servicio DC3P . . . 76
6.2. Conexiones TCP entre componentes distribuidos . . . 77
6.3. Tipos de mensajes *DC3P* . . . 79
6.4. Formato de la cabecera *DC3P* . . . 80
6.5. Formato de mensajes *Connect/Disconnect Single-Single*. . . . 81
6.6. Formato de mensajes *Connect/Disconnect Single-Multi*. . . . 81
6.7. Formato de mensajes *Connect/Disconnect Multi-Single*. . . . 82
6.8. Formato de mensajes *Connect/Disconnect Multi-Multi*. . . . 83
6.9. Formato de mensajes *RemoteConnect/RemoteDisconnect Single-Single*. . . . 84
6.18. Formato de mensajes de *Data Single* . . . 84
6.10. Formato de mensajes *RemoteConnect/RemoteDisconnect Single-Multi*. . . . 85
6.11. Formato de mensajes *RemoteConnect/RemoteDisconnect Multi-Single*. . . . 86
6.12. Formato de mensajes *RemoteConnect/RemoteDisconnect Multi-Multi*. . . . 87
6.13. Formato de mensajes *SinglePacket PortInfo Request* . . . 87
6.14. Formato de mensajes *SinglePacket PortInfo Response* . . . 88
6.15. Formato de mensajes *PortInfo Request Multi* . . . 88
6.16. Formato de mensajes *PortInfo Response Multi* . . . 89
6.17. Formato de mensajes de *Echo* . . . 89
6.19. Formato de mensajes de *Data Multi* . . . 90
6.20. Formato de mensajes *Ack/Reject* . . . 90
6.21. Modelo de diagrama de secuencia . . . 91
6.22. Secuencia de conexión . . . 92
6.23. Secuencia de desconexión . . . 92
6.24. Secuencia de conexión remota . . . 93
6.25. Secuencia de desconexión remota . . . 94
6.26. Secuencia de envío de datos . . . 94
6.27. Ejemplo de IBox y OBox . . . 95
6.28. Hilo *main*, IBox y OBox de un componente . . . 96
6.29. Hilo *INT* de un componente . . . 97
6.30. Hilo *ONT* de un componente . . . 98
6.31. Sonda Voyager II. Utilizada en 1989 para captar imágenes alrededor de la órbita de Neptuno, descubrió seis de sus ocho lunas y confirmó la existencia de los anillos casi invisibles que se encuentran alrededor de él. . . . 99
6.32. Conexionado entre un componente y su *sonda* . . . 100
6.33. Aplicación de consola para comandar y monitorizar un robot. . . . 101
6.34. Componente *Probe* aplicado a una *Vista* . . . 102
6.35. Interfaz de teleoperación . . . 103

7.1. Diseño del protocolo de comunicación D3CP . . . 109
7.2. Estructura del cuerpo los mensajes del protocolo DC3P . . . 112
7.3. Interfaces proporcionadas por la clase *PacketBody* . . . 113
7.4. Diagrama de clases del patrón *prototype* . . . 113
7.5. Diagrama de clases del patrón *MVC* . . . 114

8.1. Vista del diseño de las clases *PortPacket* y *PacketBody* 116

9.1. Componente *PlayerRobot* 128
9.2. Componente *GridMap* 128
9.3. Componente *NDNavigation* 129
9.4. Componente *ShortTermPlanner* 130
9.5. Interfaz de puertos de *PlayerRobotJavaView* 131
9.6. Vista *PlayerRobotJavaView* 132
9.7. Interfaz de puertos de *SphereJavaView* 133
9.8. Vista *SphereJavaView* 134
9.9. Interfaz de puertos de *GridJavaView* 135
9.10. Vista *GridJavaView* 135
9.11. Conexionado de componentes del sistema 136
9.12. Configuración Test1 137
9.13. Configuración Test2 137

A.1. Estructura del árbol de directorios de CoolBOT 146
A.2. Vista general de las funciones miembro públicas de un comando del protocolo DC3P 147
A.3. Interfaz de clonado 148
A.4. Interfaz de copia 148
A.5. Interfaz para el envío/recepción de datos por la red 149
A.6. Interfaz de *naming* 149
A.7. Interfaz de depuración 149
A.8. Operador de asignación 150
A.9. Assign 150
A.10. Inicialización y finalización de variables estáticas 151
A.11. Inicialización de variables estáticas 152
A.12. Fichero de implementación USER_COMMAND.cpp 153

Índice de cuadros

3.1. Variables de monitorización por defecto. 29
3.2. Variables de control por defecto. 30
3.3. Conexiones de Puertos. 33

5.1. Representación a nivel de bytes de datos construidos en CORBA. 61
5.2. Requisitos de alineamiento para tipos de datos primitivos OMG IDL. 62

6.1. Valores del campo CommandType. 80

Facultad de Informática. Universidad de Las Palmas de G.C.

Capítulo 1

Introducción

1.1. Introducción al problema

La integración del software ha sido siempre un problema poco valorado en robótica, y frecuentemente se trata de una tarea en la que es necesaria invertir mucho más esfuerzo del considerado inicialmente. La integración de sistemas software requiere demasiados recursos que sólo unos pocos grupos de investigación pueden afrontar.

La aparición de Internet, ha permitido a los investigadores acceder remotamente mediante redes de comunicación a supercomputadores, importantes bases de datos, recursos informáticos a través de *World Wide Web* (WWW), cuentas de correo electrónico, o servidores *ftp*, entre otros.

El acceso a dispositivos remotos permite la visión de un único sistema cooperativo donde cada máquina conectada a la red puede procesar información proveniente de diferentes máquinas y generar resultados aprovechables por otras terceras, constituyendo así un esquema colaborativo que permite un mejor aprovechamiento de los recursos disponibles.

El uso de las redes de ordenadores ha contribuido a ampliar el índice de campos de aplicación de la robótica y prueba de ello son los robots teleoperados, que permiten extender la capacidad de intervención humana a lugares remotos. A continuación se presentan los fundamentos de los robots *online* así como el concepto de teleoperación y las limitaciones que presenta llevarla a cabo mediante el uso de redes de área local (LAN) y redes de amplio alcance (WAN). Una vez introducido el concepto de teleoperación se detallarán en la siguiente sección los objetivos planteados en este proyecto para, finalmente describir la estructura del presente documento.

1.2. Robots *online* y teleoperación

Los robots online están limitados por la física de operación de un dispositivo mecánico, por la necesidad de comunicar a través de una conexión de bajo ancho de banda, por la imprevisibilidad del entorno del mundo real, y por la necesidad de interactuar con un humano mediante una simple interfaz. Estas limitaciones no son exclusivamente propias del mundo online. Algunos problemas similares han sido considerados en sistemas robóticos tradicionales - especialmente en un área particular de la robótica denominada teleoperación.

La teleoperación [Goldberg and Siegwart, 2001] permite la realización de un trabajo físico en un lugar remoto bajo el control de un operador (imagen). Normalmente, los sistemas teleoperados poseen los siguientes componentes:

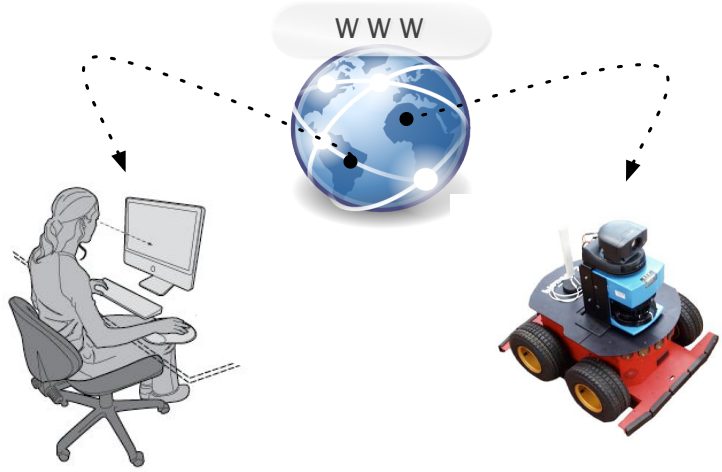

Figura 1.1: Vista de un escenario teleoperado

- El operador
- La estación operadora. En la teleoperación convencional, se tienen normalmente un número de monitores de televisión y un dispositivo mecánico el cual comandaría el sistema. En los robots online, se emplea una única estación de trabajo.
- Un robot remoto. Artilugio mecanizado cuyas acciones se dirigen a distancia. Puede tratarse de un brazo manipulador, un robot bípedo, vehículos submarinos o aéreos, etc.
- Sensores remotos. Pueden ser tan simples como el conjunto de sensores de posición del manipulador, o tan complejos como cámaras de televisión. Realizan dos roles: proporcionar información para el control local del robot y proporcionar información al operador humano distante.

- Procesador remoto. Es el encargado de combinar la información transmitida desde la estación operadora con información proveniente de los sensores remotos, para generar señales de control para el robot.

En algunos casos la comunicación entre el operador y el sistema teleoperado se realiza conectando ambos directamente vía cable, haciendo que el retraso sea imperceptible. Cuando la teleoperación se realiza a través de Internet, los retrasos en la comunicación se incrementan y el ancho de banda disponible se ve decrementado. Esto requiere encontrar alguna manera de compensar el paso de teleoperar directamente vía cable con respecto a vía Internet. Afortunadamente se han llevado a cabo un amplio número de trabajos relacionados con la teleoperación a través de comunicaciones en las que el ancho de banda disponible es limitado. Las secciones siguientes profundizan en estas ideas.

1.3. Teleoperación en una red local (LAN)

En este caso, la interfaz directa de teleoperación con el robot remoto es sustituida por una interfaz vía una estación de trabajo conectada a través de una red local. Con esa configuración se asume que la red posee un retardo imperceptible.

La interfaz es ahora un ordenador y un ratón, más que un conjunto de monitores de televisión y un controlador manual. Para implementar tal sistema, es necesario digitalizar las imágenes desde el sitio remoto, enviarlas a través de la red y mostrarlas en el monitor. Por tanto, es necesario proporcionar una interfaz a través de la cual el operador pueda comandar el manipulador remoto.

En una primera aproximación, al operador se le proporciona una visualización de la imagen en vivo a través de una ventana con un conjunto de barras deslizables en la pantalla para controlar la posición del manipulador tanto en el espacio de las articulaciones como en el Cartesiano. Dicho sistema puede trabajar sorprendentemente bien, es simple de comprender y los operadores tienen bastante flexibilidad. No obstante, a dicho sistema se le pueden incorporar diferentes mejoras.

Para darle al operador una mejor vista del lugar remoto, se podrían añadir cámaras remotas adicionales, y colocar cámaras en plataformas motorizadas. De esta manera, se podría automatizar el movimiento de estas cámaras y la selección de imágenes.

Para proporcionar información adicional al operador, podrían calibrarse las cámaras remotas y entonces, usando ese conocimiento, añadir marcas visuales en las imágenes para ayudar al operador calculando distancias y alineamientos.

Uno de los problemas de trabajar remotamente es la necesidad para el operador de mapear constantemente el movimiento final en secuencias de movimientos articulados o cartesianos. Una

acción deseada del tipo "mover un poco hacia la izquierda en esta imagen", por ejemplo, podría convertirse fácilmente en varios comandos si el hecho "izquierda en esta imagen" no sucede convenientemente a lo largo de los ejes de movimientos cartesianos o de las articulaciones. La solución es re-mapear los movimientos comandados por el operador, basados en la dirección de la vista. En la terminología robótica, esto es considerado como un marco de referencia centrado en el operador. En gráficos por ordenador, se denomina correspondencia cinemática. El trabajo requerido para mapear los movimientos centrados en el observador en comandos del manipulador puede ser minimizado distribuyendo adecuadamente las cámaras en el sitio remoto. Las cámaras fijas pueden estar alineadas con el marco base Cartesiano para el manipulador. Las cámaras pequeñas puede estar montadas y alineadas con respecto al marco de referencia del último movimiento. La mejor solución y la única que da completa libertad en la ubicación de las cámaras, es calibrar las cámaras remotas y realizar la necesaria transformación matemática.

1.4. Teleoperación a través de redes de amplio alcance (WAN)

Consideremos ahora un esquema de comunicación basado en redes WAN que introduce retardos del orden de varios segundos en las transmisiones y limita el ancho de banda disponible, cuestiones ambas que deben ser resueltas a fin de disponer de un sistema teleoperado efectivo.

El bajo ancho de banda significa que no podemos simplemente digitalizar y transmitir las imágenes desde el lugar remoto como hacíamos antes. Una técnica simple es elegir un tamaño de imagen pequeño y comprimir cada imagen (por ejemplo, usando compresión JPEG). En este caso, sacrificamos tiempo de procesamiento, codificando y decodificando cada imagen, con el fin de reducir el ancho de banda consumido.

En lo referente a los retardos o latencias de las comunicaciones, una aproximación podría ser no hacer nada y confiar en que el operador se adapte. Aunque es una solución atractivamente simple, no funcionaría correctamente. Consideremos el caso en el que el operador advierte que el manipulador remoto está a punto de golpear una pared. Aunque el operador reaccionase de manera inmediata, es posible que la colisión no pueda evitarse a tiempo, o incluso que ya haya tenido lugar. Sabiendo ese riesgo, los operadores adoptan rápidamente una estrategia "mover y esperar": realizando un pequeño, y relativamente seguro movimiento, y esperando para ver que ha sido llevado a cabo satisfactoriamente antes de realizar el siguiente movimiento. Esto es frustrantemente lento.

Una solución más adecuada para los retardos requiere atacar dos frentes: autonomía local del robot y predicción de trayectorias. Con respecto al primero, es necesario añadir "inteligencia" al controlador del robot en el lugar remoto, de manera que puedan implementarse capacidades de auto-preservación y protección del entorno. Estos mecanismos reactivos locales contribuyen a que

el operador humano disponga de un tiempo de reacción mayor, mitigando el efecto de las latencias en las comunicaciones.

Dotando al robot esclavo de inteligencia, es posible incrementar el nivel de comunicación entre el operador y el robot remoto, permitiéndole al operador asumir un rol más supervisor. Esto libera al operador de la necesidad de comandar cada detalle de la operación al lugar remoto.

El segundo frente a atacar en este tipo de comunicaciones, es suministrar al operador una interfaz más sofisticada. Por ejemplo, podría dársele al operador una simulación del robot remoto y permitirle interaccionar con él. Ésto es conocido como *monitor predictivo* y permite al operador ver el efecto de los comandos antes de ser ejecutados realmente por el robot remoto. Si el entorno remoto estuviese lo suficientemente delimitado, entonces podría considerarse incluso simular el movimiento dinámico de los objetos del lugar remoto. Proporcionando a la estación operadora un modelo del robot remoto que reaccione instantáneamente, se podría aislar enormemente al operador de los efectos de los retardos en las comunicaciones.

Incluso si el retardo en las comunicaciones no fuese un problema, existen algunos beneficios de cara a proveer al operador de una vista simulada del lugar remoto. Como ejemplo de ello cabe pensar en un sistema que alcanza puntos de vista no disponibles desde cámaras reales, por lo que no se verá afectado por la pobre visibilidad del lugar remoto, y libera al operador de la necesidad de trabajar exactamente a la misma velocidad que el lugar remoto. El operador puede experimentar con acciones *off-line*, o trabajar más rápido o más despacio que el robot remoto real.

Todas estas aproximaciones tienen un objetivo común: proporcionar información adicional en cada sitio puede reducir los efectos de las comunicaciones retardadas. La dificultad estriba en que el ancho de banda es también limitado lo que hace necesario sacrificar el envío de cierta información entre lugares remotos. Decidir qué información debe enviarse y cuándo enviarla, son quizás las partes más difíciles y críticas para controlar un robot remoto a través de una conexión de bajo ancho de banda.

1.5. Telepresencia

Telepresencia significa presencia remota y es un medio que proporciona a la persona la sensación de estar físicamente en otro lugar por medio de una escena creada por ordenador.

En los últimos años, se ha mostrado un interés creciente en la utilización de robots móviles en aplicaciones de telepresencia. Algunos ejemplos representativos son los sistemas de vigilancia activa, inspección de áreas siniestradas o de difícil acceso, los sistemas de vuelo en primera persona (FPV) y las visitas guiadas remotas.

En este último caso, el usuario realiza de forma remota un recorrido por un museo o una zona turística utilizando un robot móvil. La ruta realizada es comentada de forma audiovisual y complementada con la información capturada por los sensores del robot en tiempo real. El usuario puede adoptar una posición pasiva o bien tomar la iniciativa e interrumpir y modificar el itinerario preestablecido para obtener más información sobre algún aspecto que le resulte de interés.

1.6. Objetivos

Los objetivos generales de un Proyecto Fin de Carrera (PFC) son principalmente que el alumno ponga en práctica los conocimientos adquiridos durante la carrera, aplicándolos a un trabajo real y completo. Además aparecen los objetivos específicos del proyecto en cuestión que dirigen el desarrollo del mismo. Así pues, en esta sección se presentan los objetivos divididos en dos apartados, objetivos académicos y objetivos específicos del PFC.

1.6.1. Objetivos académicos

Como objetivos académicos en la realización de este proyecto se plantean los siguientes:

- Seleccionar y aplicar una metodología de desarrollo de software.
- Aplicar conocimientos de programación orientada a objetos.
- Aplicar conocimientos de asignaturas relacionadas con los sistemas robóticos móviles.
- Manejo de software de control de versiones.
- Manejo de herramientas de producción documental.

1.6.2. Objetivos específicos

El objetivo inicial de este proyecto era la construcción de un sistema teleoperado que permitiese la comunicación remota entre un operador y un robot móvil a través de una interfaz Web. Durante la elaboración del mismo se estimó oportuno reorientarlo con un enfoque más dirigido hacia el desarrollo de una infraestructura genérica, que tuviese por objeto facilitar y capacitar el desarrollo de aplicaciones telepresenciales, como es el caso de las destinadas a la realización de visitas guiadas remotas.

En este trabajo se ha optado por la utilización de un marco de programación de sistemas robóticos y el aprovechamiento de componentes software ya desarrollados, evitando la necesidad de tener que programarlos como parte de este proyecto. Este *framework*, se denomina *CoolBOT* y consiste en una herramienta escrita en C++ para la construcción de sistemas robóticos a partir de componentes software.

Como resultado final se obtendrá una interfaz Web de teleoperación elaborada sobre una infraestructura de red diseñada e implementada en este proyecto, que permitirá la realización de aplicaciones teleoperativas basadas en el *framework* referenciado en el párrafo anterior (*CoolBOT).* Este trabajo puede descomponerse en la siguiente serie de subobjetivos:

- Diseño de un protocolo de comunicaciones para llevar a cabo la comunicación entre componentes distribuidos. Se espera que este protocolo cumpla con unos requerimientos en términos de **eficiencia**, minimizando el trasiego de datos que viajarán por la red.
- Añadir la infraestructura de red a los componentes *CoolBOT* ya existentes con el fin de obtener un sistema robótico distribuido **portable** entre diferentes máquinas y/o sistemas operativos. Es deseable que este sistema sea además **escalable y modular** permitiendo añadir fácilmente nuevas funcionalidades futuras.
- Realizar una interfaz Web Java que permita teleoperar con el robot móvil y monitorizar su entorno de operación, abstrayendo al usuario operador lo máximo posible de los mecanismos de comunicación de bajo nivel (**transparente**).

1.6.3. Visión del sistema final

El sistema visto de manera global se corresponde con la imagen 1.2. En ésta podemos ver los elementos más característicos que integran el sistema:

- Clientes WWW : Se comunican a través de una interfaz Web por medio de la cual interaccionan con el robot.
- Sistema distribuido: Constituye el sistema robótico en sí. Permite al cliente WWW comunicarse remotamente con el robot.
- Robot teleoperado.

Aunque la imagen 1.2 nos ofrece una visión estática del sistema, es interesante ver cómo entran en juego todos los elementos del mismo anteriormente explicados para conseguir un sistema teleoperado. La figura 1.3 representa la visión dinámica de todo el sistema. En ella podemos ver

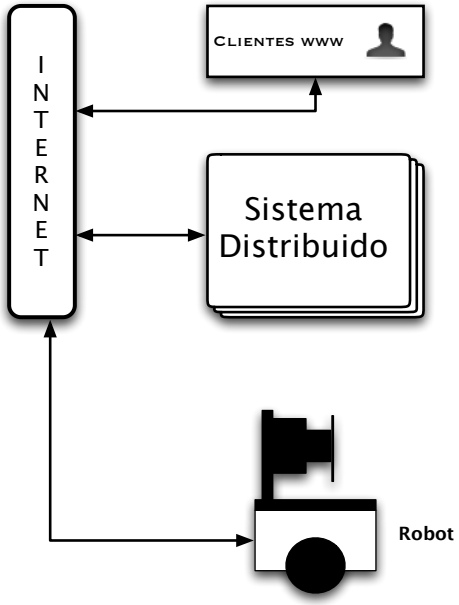

Figura 1.2: Visión estática del sistema final

cómo el cliente WWW envía una acción hacia el robot. El sistema distribuido que se encuentra entre el cliente y el robot procesa la orden del cliente y la transmite hacia el actuador remoto. De igual forma, el sistema distribuido se encarga de la lectura de sensores, que será enviada al cliente con el fin de que éste lleve a cabo el proceso de monitorización.

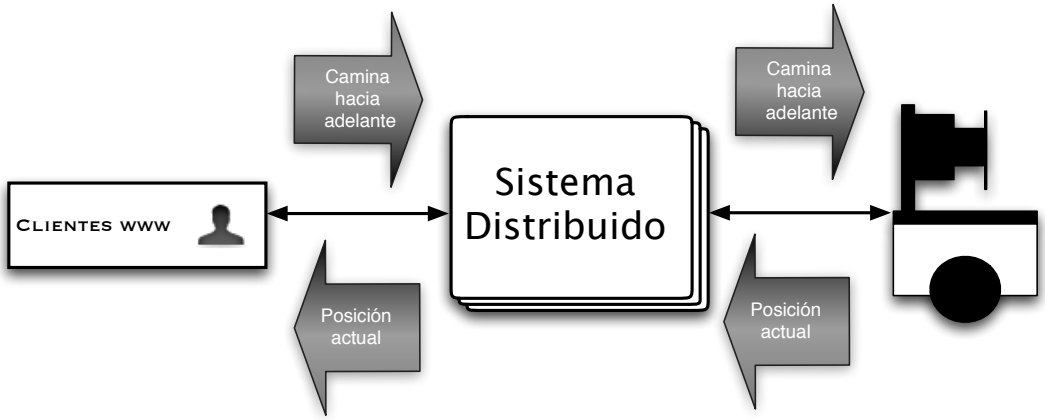

Figura 1.3: Visión dinámica del sistema final

1.7. Estructura del documento

Este documento se encuentra estructurado en los siguientos capítulos:

- Introducción: Introducción al problema.
- Estado actual del tema: Descripciones de *frameworks* existentes para la construcción de sistemas robóticos, *middlewares* para el desarrollo de aplicaciones distribuidas y ejemplos de aplicaciones basadas en telepresencia.
- *CoolBOT*: Presentación de las características principales de este *framework* utilizado como herramienta base para este trabajo.
- Metodología, recursos y plan de trabajo.
- Análisis: Se detallan los recursos y herramientas necesarios para afrontar el desarrollo de este proyecto, modelo de casos de uso y requisitos.
- Diseño: Se especifican los aspectos ligados al diseño de los principales elementos que intervienen en la elaboración de este proyecto.
- Detalles de Diseño: Aspectos muy caracterísiticos del diseño.
- Detalles de Implementación: Cuestiones reseñables de la implementación realizada.
- Pruebas y resultados: Pruebas llevadas a cabo y resultados obtenidos en las mismas.
- Conclusiones y trabajo futuro: Conclusiones extraídas de este trabajo y de las pruebas realizadas y posibles líneas futuras para la continuidad del mismo.
- Manuales de usuario: Guías para facilitar a otros usuarios el desarrollo de determinadas partes del software.

Capítulo 2

Estado actual del tema

Un sistema telerobótico controlado a través de Internet se compone de un número de dispositivos físicos tales comos robot(s), cámaras, sensores y actuadores. A veces, la integración del software encargado de manejar estos dispositivos obliga a los desarrolladores a emplear tiempo en detalles que se salen de la mera programación funcional que se desea para estos dispositivos. Además, para llevar a cabo la interacción de los elementos del sistema a través de *Word Wide Web* es necesario algún mecanismo de comunicación que proporcione métodos para la interacción entre dispositivos distribuidos y la propia interfaz. En este capítulo se presentan los sistemas robóticos percepto-efectores así como algunas herramientas o *frameworks* que facilitan el desarrollo de los mismos. En segundo lugar se describen algunos *middlewares* o capas intermedias que abstraen el nivel de red a aquellas aplicaciones que requieran métodos para la comunicación en entornos distribuidos. Finalmente se comentan algunos ejemplos de sistemas robóticos teleoperados vía Web.

2.1. Sistemas robóticos percepto-efectores y teleoperados

Los sistemas robóticos percepto-efectores son sistemas que se desenvuelven e interaccionan con un entorno mediante la acción de sensores y efectores. Una de las tareas más importantes de estos sistemas, también conocidos como sistemas autónomos inteligentes o agentes autónomos, es la adquisición del conocimiento acerca de su entorno. Para ello, es necesario llevar a cabo la obtención de una serie de medidas empleando varios sensores y extrayendo luego la información relevante de las mismas. Dentro del ámbito de este tipo de sistemas existen varias herramientas que merece la pena destacar y que pasamos a comentar a continuación:

Player / Stage

Player es un servidor que permite controlar los dispositivos de un robot y obtener información de sus sensores. Es una capa software que abstrae los detalles del hardware del robot a los desarrolladores de aplicaciones. Los algoritmos de control del robot funcionarán como clientes de Player que se conectan a través de sockets TCP/IP. Así, es posible controlar el robot enviando mensajes que sigan el protocolo de comunicación de Player o llamando a funciones de las librerías de Player que ocultan los detalles de comunicación (este último método es el más utilizado). La distribución actual de Player incluye librerías en lenguajes tan diversos como C++, Java, Python o Lisp.

Stage es un simulador de robots móviles en 2D que puede ser utilizado en conjunción con Player si deseamos hacer pruebas iniciales de nuestros algoritmos o bien no disponemos de un robot real. Para nuestro código será transparente el que estemos trabajando con un robot real o con Stage.

Advanced Robotics Interface Application (ARIA)

ARIA es una librería escrita en C++ desarrollada por la compañía ActivMedia destinada a la comunicación con sus propios robots (Pioneer, AmigoBot, etc.). Es la sucesora del sistema desarrollado en la pasada década denominado Saphira. Ha sido diseñada siguiendo el paradigma orientado a objetos, y puede ser utilizada bajo sistemas Linux o Win32. ARIA incluye una librería conocida como ArNetworking que implementa una infraestructura añadida para realizar operaciones remotas con el robot, con interfaces de usuario y con otros servicios de red. Ha sido desarrollada con la intención de permitir la ejecución de aplicaciones tanto mono-hilo como multi-hilo.

Open Robot Control Software (OROCOS)

"Orocos" es el acrónimo del proyecto *Open Robot Control Software*. El objetivo del proyecto es desarrollar un *framework* de propósito general, modular y software libre para robótica. El proyecto Orocos soporta cuatro librerías C++: herramientas para aplicaciones en tiempo real, librería para cinemáticas y dinámicas, librería para filtros bayesianos y la librería de componentes Orocos.

Orocos está dirigido a cuatro categorías diferentes de usuarios, desde desarrolladores de la infraestructura del *framework* hasta usuarios finales que serán los que programen y ejecuten sus propias tareas. Entre estos dos grupos se encuentran los desarrolladores de componentes y los desarrolladores de aplicaciones, que proporcionan funcionalidades para su utilización por parte de los usuarios finales.

Carnegie Mellon Robot Navigation Toolkit (CARMEN)

CARMEN es una colección de software *open-source* escrito en C para el control de robots móviles. Es un software modular diseñado con el fin de proporcionar primitivas de navegación incluyendo: control de sensores, evitación de obstáculos, localicación, planificación de rutas y construcción de mapas. CARMEN ha sido diseñada para operar en máquinas con sistema operativo Linux y está disponible bajo licencia GPL. Actualmente las únicas versiones disponibles son *beta*, siendo la última de ellas lanzada en octubre de 2008.

CoolBOT

En el ámbito del SIANI (Sistemas Inteligentes y Aplicaciones numéricas en Ingeniería) se ha desarrollado *CoolBOT* [Domínguez-Brito, 2003], un *framework* para la programación de sistemas robóticos orientado a componentes que implementa primitivas y mecanismos dirigidos a la resolución de algunos problemas comunes al desarrollo de aplicaciones robóticas. Este framework permite construir sistemas mediante la integración de componentes software siguiendo un modelo de autómata de puertos que incorpore controlabilidad y observabilidad. El diseño de este framework es orientado a objetos y está escrito completamente en C++ aunque al principio de su desarrollo se elaboraron algunos módulos en Java. Está disponible para los sistemas operativos Windows y GNU/Linux. En el capítulo 3 se detalla en más profundidad el diseño y funcionalidad de este *framework*.

2.2. Middleware's para la programación de aplicaciones distribuidas

Los *Middleware's* para la programación de aplicaciones distribuidas encapsulan los mecanismos de concurrencia y comunicación nativos de los sistemas operativos para crear componentes destinados a la programación de aplicaciones de red. Estos componentes abstraen las peculiaridades de cada sistema operativo y ayudan a eliminar varios aspectos tediosos, propensos a errores y no portables del desarrollo y mantenimiento de las aplicaciones de red a través de las API's de programación de bajo nivel de los sistemas operativos, tales como Sockets o POSIX threads. Algunos ejemplos de middleware's para programar aplicaciones distribudas más utilizados son:

La Máquina Virtual Java (JVM)

Proporciona una forma de ejecutar código independiente de la plataforma abstrayendo las diferencias entre sistemas operativos y arquitecturas CPU. La máquina virtual Java es responsable de interpretar código Java y traducirlo a acciones o llamadas al sistema operativo. También es responsable de encapsular detalles de la plataforma dentro de la interfaz del código portable, así que las aplicaciones están protegidas de la diversidad de sistemas operativos y arquitecturas CPU sobre las que el software Java corre.

.NET

Se trata de una plataforma de Microsoft para servicios Web XML diseñada para conectar información, dispositivos y personas de una forma común (aunque configurable). El lenguaje común en tiempo de ejecución (CLR) es el *Middleware* sobre el que los servicios .NET de Microsoft están construidos. CLR es similar a la JVM de Sun, proporciona un entorno de ejecución que maneja el código ejecutable y simplifica el desarrollo de software a través de mecanismos de manejo de memoria, integración multi-lenguaje, interoperabilidad con código y sistemas ya existentes, implantación simplificada y un sistema de seguridad.

The Adaptive Communication Environment (ACE)

ACE es un conjunto de funciones altamente portables escritas en C++ que encapsulan las funcionalidades de comunicación de los sistemas operativos nativos, tales como establecimiento de conexión, multiplexado de eventos, comunicación entre procesos, *(de)marshalling*, configuraciones estáticas y dinámicas de los componentes de una aplicación, concurrencia y sincronización. La principal diferencia entre ACE, JVM y .NET CLR es que ACE siempre ofrece una interfaz compilada, más que una interfaz de código interpretado, eliminando otro nivel de indirección y ayudando a optimizar el rendimiento de las aplicaciones. En el capítulo 5.8 se ofrece una descripción con mayor detalle de las funcionalidades que aporta este *middleware* que justifican su elección para el presente trabajo.

2.3. Sistemas robóticos teleoperados vía Web

Cuando el mundo Web es visto como una infraestructura para construir sistemas robóticos, su atractivo puede resultar triple. En primer lugar, los navegadores pueden ser una buena interfaz humana en un sistema robótico porque pueden mostrar varios media incluyendo hipertextos,

imágenes, películas, sonidos y gráficos 3D así como interactuar con ellos. En segundo lugar, el protocolo para la transferencia de hipertexto ó *Hypertext Tranfer Protocolo (HTTP)* puede ser un protocolo de comunicaciones estándar para un sistema robótico ya que los robots conectados a Internet pueden ser accesibles desde cualquier sitio de Internet a través del protocolo. En tercer lugar, puede ser posible emplear software y hardware robótico distribuido en Internet para abordar conjuntamente un objetivo común. Hoy en día es natural considerar la Web en el desarrollo de sistemas robóticos, dando lugar al concepto denominado *webtop robotics*. A continuación de describen algunos ejemplos de sistemas basados en este idea:

RobOnWeb

RobOnWeb [Goldberg and Siegwart, 2001] es un proyecto desarrollado por el Instituto Tecnológico Federal Suizo de Lausanne (EPFL). El objetivo de este proyecto consiste en desarrollar nuevos conceptos para la teleoperación de robots móviles a través de *World Wide Web*. Estos robots móviles teleoperados deben permitir interacción física real a través de Internet. En un tablero de 80x80 cm. se ubican cinco mini-robots llamados "Alice" de dos centímetros cúbicos de tamaño. Cada robot, identificado por un código de color, puede ser controlado por un usuario a través de Internet. La posición del robot es determinada mediante la imagen de una cámara colocada sobre el tablero. Además de esta cámara, una segunda permite al usuario obtener una vista lateral del tablero. Para controlar los movimientos del robot, el usuario puede seleccionar puntos de interés sobre el mapa del tablero. La nueva posición solicitada es procesada por el software de navegación, que genera segmentos de movimientos ejecutables por los robots. RobOnWeb es un proyecto que permite teleoperar vía Web dentro de un entorno multirobot. Sin embargo no permite probar el sistema con diferentes escenarios, ya que eso implicaría construir otro tablero. Además, el usuario tiene unas limitaciones en cuanto a funcionalidad, ya que el sistema no le permite definir sus propias aplicaciones o tareas.

KhepOnTheWeb

KhepOnTheWeb [KhepOnTheWeb,1997] ha sido desarrollado por el Instituto Tecnológico Federal Suizo de Lausanne (EPFL). El objetivo de este proyecto es proporcionar una infraestructura compartida por varios grupos de investigación que dedican sus esfuerzos a resolver problemas relacionados con la robótica móvil, con el fin de economizar los gastos que suponen los equipamientos requeridos por los robots Khepera. Estos son cilíndricos, de 55 mm. de diámetro. Las dimensiones del escenario remoto son 65x95 cm. Los robots están equipados con una cámara CCD de a bordo y conectados a un PC vía cable RS232. La interfaz Web accesible desde el cliente Netscape permite realizar las siguientes acciones:

- Controlar los movimientos del robot a través del envío de comandos mediante un cable RS232.

- Comandar la cámara externa situada encima del tablero.
- Seleccionar la vista de la cámara externa o de la cámara de a bordo del robot.

Capítulo 3

CoolBOT

CoolBOT [Domínguez-Brito, 2003] es un *framework* o marco de programación orientado a componentes para sistemas robóticos que se ha diseñado, desarrollado y se utiliza actualmente en el Laboratorio de Robótica e Interacción del *Instituto Universitario SIANI* y en el *Departamento de Informática y Sistemas de la ULPGC*. *CoolBOT* está inspirado en el paradigama de ingeniería del software *CBSE*, del inglés *Component Based Software Engineering* [George,2001].

El desarrollo de esta plataforma surge de la falta de paradigmas claros y estandarizados de programación para este tipo de sistemas. Esta situación va en detrimento de la reutilización de código, factor clave, cuanto más en este tipo de sistemas, donde existen muchos problemas recurrentes tales como la abstracción del hardware o la computación distribuida.

CoolBOT presenta un modelo de programación de sistemas robóticos basado en el concepto de *componente software* [George,2001], y donde los elementos que componen el software de un sistema robótico se denominan *componentes CoolBOT* o *componentes*. Cada uno de estos elementos implementa y aporta una determinada funcionalidad al sistema en conjunto, para lo cual coopera con otros componentes consumiendo y/o produciendo datos. Cada uno de estos elementos se ejecuta individualmente como una *máquina de flujo de datos* [Arvind,1981], que se encuentra inactiva esperando datos en sus entradas, activándose sólo cuando llegan dichos datos para procesarlos, y seguidamente emitir datos procesados a través de sus salidas. Los componentes de *CoolBOT* son modelados como un autómata de puertos y representan unidades independientes de ejecución. Estos componentes, una vez diseñados y construidos, pueden ser instanciados cuantas veces sea necesario.

3.1. Modelado de componentes

Como ya se ha comentado, cada componente *CoolBOT* es modelado como un autómata con puertos de entrada y puertos de salida. Este modelo permite diferenciar claramente la funcionalidad interna del componente de la interfaz externa como puede verse en las figuras 3.1 y 3.2.

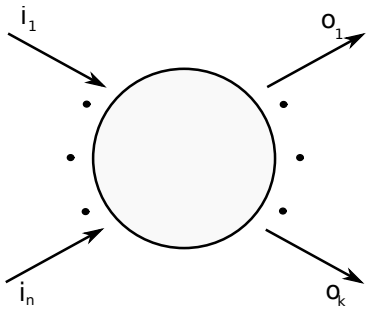

Figura 3.1: Vista externa de componente.

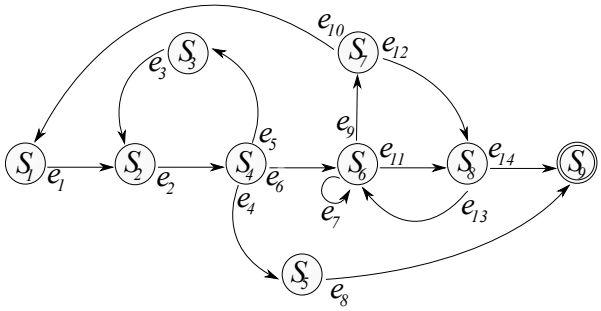

Figura 3.2: Vista interna de componente.

La figura 3.1 muestra la descripción externa de un componente, donde el componente en sí mismo es representado como un círculo, sus puertos de entrada como flechas entrantes al componente y sus puertos de salida como flechas salientes. En la figura 3.1 observamos una representación interna de un componente, esto es un autómata que modela su comportamiento. En la representación usada del autómata, cada estado es representado por un círculo (el estado final con doble línea), y las transiciones entre estados por flechas. Cada estado se encuentra etiquetado (S_n), así como las flechas de transiciones (e_i) indicando bajo qué condición interna al componente, o dato de entrada concreto, o ambos, se produce una transición entre estados.

3.2. Variables observables y controlables

Con el objetivo de proporcionar robustez y controlabilidad, *CoolBOT* proporciona dos conjuntos de variables: observables y controlables. Esto permite diseñar componentes que sean observables para así determinar y seguir su correcto funcionamiento, así como otorgarles cierto nivel de control sobre su modo de operación.

- Variables Observables: representan aspectos del componente de interés desde fuera del mismo.
- Variables Controlables: representan aspectos del componente que pueden ser controlados externamente.

CoolBOT garantiza la observabilidad y controlabilidad de cualquier componente, para lo cual introduce dos tipos de puertos por defecto en todo componente: un puerto de monitorización y un

puerto de control. La visión externa que tenemos de un componente en la figura 3.1 ahora se ve ampliada a la mostrada en la figura 3.3.

- El *puerto de monitorización* es un puerto público que permite publicar las *variables observables*.
- El *puerto de control* es un puerto público que permite modificar/actualizar las *variables de control*.

Así pues cualquier componente puede ser controlado y monitorizado por un *supervisor externo*, tal y como se ilustra en la figura 3.4, donde se observa la utilidad de ambos puertos.

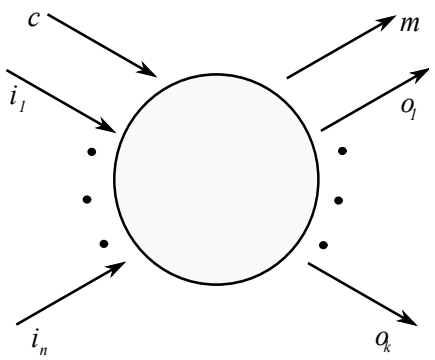

Figura 3.3: Vista externa de componente con puerto de control y monitorización.

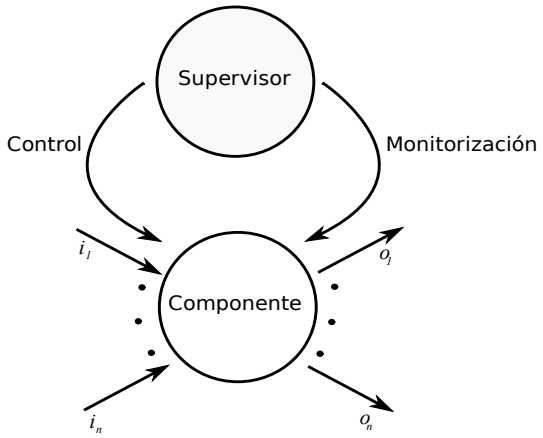

Figura 3.4: Bucle común de control.

Además *CoolBOT* proporciona a cada componente de una serie de variables observables y controlables por defecto. Estas variables se describen en la tablas 3.1 y 3.2.

Variables de monitorización por defecto	
Nombre	**Descripción**
state (s)	Estado del autómata donde se encuentra el componente.
priority (p)	Prioridad actual de ejecución del componente.
config (c)	Solicita un cambio de configuración supervisado, o confirma comandos de configuración.
result (r)	Resultado de ejecución.
error description (ed)	Descripción de error indicando una excepción local irrecuperable.

Cuadro 3.1: Variables de monitorización por defecto.

Variables de control por defecto	
Nombre	**Descripción**
new state (ns)	Estado del autómata al que se desea que el componente transite.
new priority (np)	Prioridad de ejecución a la que se desea que el componente se ejecute.
new exception (nex)	Excepción inducida externamente.
new config (nc)	La configuración del componente puede ser modificada y actualizada durante la ejecución a través de esta variable de control.

Cuadro 3.2: Variables de control por defecto.

3.3. Autómata por defecto

Como ya se mencionó con anterioridad, en *CoolBOT* todos los componentes se modelan a partir de un autómata por defecto. Podemos ver una representación del autómata por defecto en la figura 3.5, donde los diferentes estados se representan por un círculo (estado final con doble línea) y las transiciones entre los mismos por arcos etiquetados para indicar el evento que dispara la transición.

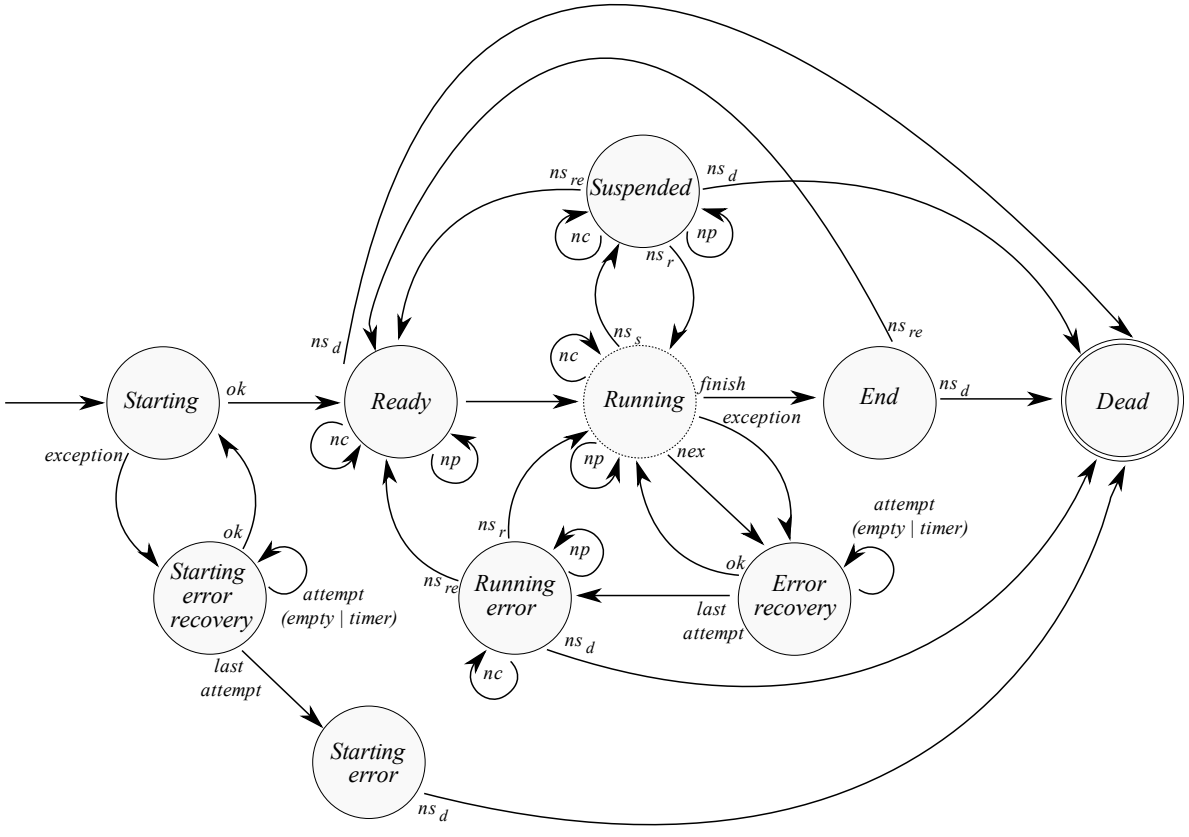

Figura 3.5: Autómata por defecto.

Algunos de los eventos que provocan transiciones son internos al componente (*exception, nex, attempt, ok, last attempt, finish*), el resto son eventos provocados por cambios en las variables de control por defecto ($ns_r, ns_{re}, ns_s, ns_d$, *np, nc y nex*), donde el subíndice indica a que estado del autómata ha sido comandado el componente: *r* (*running* state), *re* (*ready* state), *s* (*suspended* state), *d* (*dead* state). Los restantes eventos (*np*, *nc* y *nex*) indican que un supervisor externo ha cambiado la prioridad a la que el componente ha de ejecutarse (*np*), su configuración interna (*nc*), o bien, que ha inyectado una excepción (*nex*) para su comprobación

Como se puede observar en la figura 3.5, el estado *running* se encuentra indicado mediante un circulo con línea discontinua. En realidad el estado *running* es un *pseudoestado* que representa la porción del autómata donde se implementa la funcionalidad concreta del componente, por ello es llamado *autómata de usuario*. Obviamente el *autómata de usuario* varía entre componentes dependiendo de la funcionalidad que se requiera en cada caso, este autómata es definido durante la fase de creación del componente.

El resto de los estados del autómata por defecto organizan la vida de un componente en distintas fases:

- starting: Adquiere los recursos necesarios para la ejecución del componente.
- ready: el componente esta listo para la ejecución y se encuentra a la espera de que se le comande transitar hacía el autómata de usuario (ns_r).
- running: se ejecuta del *autómata de usuario*.
- suspended: el componente se encuentra suspendido a la espera de que se le comande transistar a otro estado.
- end: el componente ha acabado su tarea y finaliza su ejecución publicando el resultado (si lo hubiera) a través del puerto de monitorización.
- dead: finalización del componente.

Además existen dos estados concebidos para el tratamiento de fallos durante la ejecución del componente. *Starting error recovery* y *starting error* manejan errores durante la adquisisión de recursos, mientras que *error recovery* y *running error*, manejan los errores durante la ejecución.

3.4. Componentes multihilo

Los componentes *CoolBOT* son entidades independientes que se ejecutan concurrentemente para realizar y llevar a cabo sus propios objetivos y tareas. Cada componente se mapea en hilos,

ya sean Win32 o POSIX, según nos encontremos sobre Windows o GNU/Linux respectivamente[1].

Durante la ejecución de un componente *CoolBOT* este se encuentra en un bucle constante procesando paquetes de puertos que acarrean distintas acciones dependiendo del tipo o contenido del paquete que se reciba y del estado actual del autómata que modela al componente. En general son las llegadas de paquetes las que ocasionan las transiciones entre estados del autómata y en función de la frecuencia de llegada de los mismos, puede darse el caso de un bloqueo del componente a la espera de paquetes de puertos, esto es, los componentes se comportan como *máquinas de flujo de datos*, que procesan la información cuando disponen de ella en sus entradas y en otro caso esperan la llegada de datos a través de sus puertos de entrada.

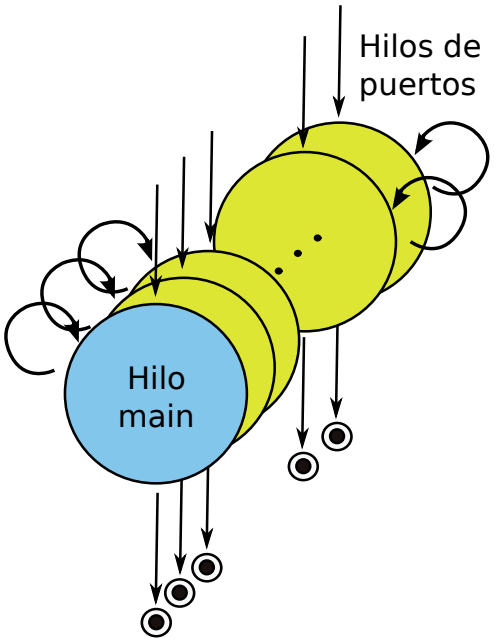

Figura 3.6: Componente multihilo.

Este bucle de procesamiento que constituye el núcleo de un componente puede descomponerse, o no, en unidades más ligeras de ejecución: hilos. De forma general, todo componente necesita para su ejecución de al menos un hilo, éste es el llamado *hilo main*. Sin embargo, con el objetivo de lograr que un componente sea más reactivo, es posible distribuir la atención de los puertos de entrada en múltiples hilos, como se ilustra en la figura 3.6. Estos son los llamados *hilos de puertos*. Estos hilos atienden conjuntos disjuntos de puertos de entrada del componente, y siguen el mismo paradigma de máquina de flujo de datos para el procesamiento de paquetes que lleguen a través de dichos conjuntos de puertos. En los casos de componentes donde aparezcan este tipo de hilos, es el *hilo main* el encargado de ejecutar el autómata del componente así como de controlar y monitorizar dichos hilos. Por otro lado, el *hilo main* es también el encargado de mantener la consistencia de las estructuras de datos internas del componente y sincronizar el acceso a dichos datos sin que se produzcan interbloqueos.

3.5. Intercomunicación entre componentes *CoolBOT*

De forma similar a la comunicación entre procesos *(IPC: Inter Process Communications)[Stevens,1999]* aportada por los actuales sistemas operativos, *CoolBOT* utiliza un sistema de comunicación entre componentes *(ICC: Inter Component Comunications)*. Este modelo estandariza las comunicaciones, permitiendo el trabajo cooperativo entre componentes a la par que manteniéndolos desacoplados, lo que favorece la reutilización y desarrollo independiente de componentes.

[1]Windows y GNU/Linux son los sistemas operativos sobre los que se soporta CoolBOT en su versión actual

El modelo de intercomunicación utilizado en *CoolBOT* se basa en los puertos de entrada y de salida de los componentes, realizando conexiones entre los mismos. Los datos se transmiten a través de estas conexiones en forma de paquetes de datos de distintos tipos denominados *paquetes de puertos* (*port packets*). Como norma general, los puertos de entrada y de salida sólo aceptan un conjunto limitado de esos tipos de paquetes de datos.

3.6. Tipos de puertos y conexiones

CoolBOT proporciona distintos tipos de puertos de entrada y salida, que determinan distintos protocolos de comunicaciones al establecer conexiones entre ellos. Esto permite que, combinándolos adecuadamente como conexiones entre puertos, se haga uso de diferentes protocolos de interacción entre componentes. Cabe destacar que las conexiones entre puertos sólo son posibles si los tipos de paquetes que aceptan ambos puertos coinciden, pero además ambos puertos deben constituir un par compatible. La tabla 3.3 muestra las conexiones posibles entre puertos de entrada y salida, así como una descripción del protocolo que se obtiene para cada conexionado.

Puerto de Salida	**Puerto de Entrada**	**Descripción**
OTick (t)	*ITick (t)*	*Conexiones tipo Tick*: Implementa un protocolo para señalizar eventos entre componentes
OGeneric (g)	*ILast (l)*	*Conexiones tipo Last, Fifo y Unbounded Fifo*: Hay una cola (fifo) de paquetes en el puerto de entrada
	IFifo (f)	
	IUFifo (uf)	
OPoster (p)	*IPoster (p)*	*Conexiones tipo Poster*: Hay una copia principal de paquetes en el puerto de salida, los puertos de entrada mantienen copias locales
OShared (s)	*IShared (s)*	*Conexiones tipo Shared*: Los componentes comparten una memoria residente en el puerto de salida. Implementa un protocolo de memoria compartida
OMultiPacket (mp)	*IMultiPacket (mp)*	*Conexiones tipo Multi Packet*: Acepta múltiples tipos de paquetes a través de la misma conexión entre puertos
OLazyMultiPacket (lmp)		
OPrioriy (pr)	*IPriorities (pr)*	*Conexiones tipo Priority* : Implementa un protocolo de envío con prioridad

Cuadro 3.3: Conexiones de Puertos.

Para manejar distintos tipos de paquetes los puertos *MultiPacket* se subdividen en *Slots*, cada uno de los cuales está dedicado a un tipo concreto de paquete de puerto de los que el puerto *MultiPacket* acepte (figura 3.7).

Un puerto *MultiPacket* permite por tanto que cada *Slot* se conecte a puertos *SinglePacket* (todos los tipos restantes de puertos) de acuerdo a los criterios de compatibilidad ilustrados en la figura 3.8, estableciendo *conexiones MultiPacket simples*.

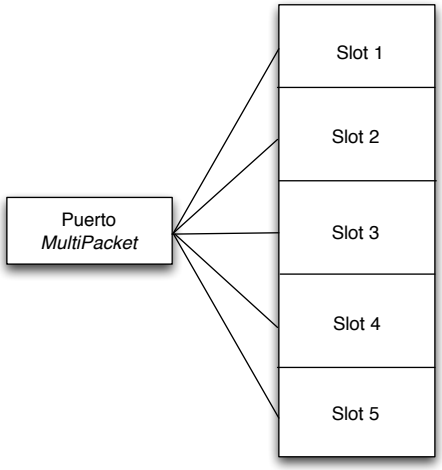

Figura 3.7: Puerto *MultiPacket*.

$$OMultiPacket \xrightarrow[n]{} \xrightarrow[m]{} ILast \qquad OLazyMultiPacket \xrightarrow[n]{} \xrightarrow[m]{} ILast$$

$$OMultiPacket \xrightarrow[n]{} \xrightarrow[m]{} IFifo \qquad OLazyMultiPacket \xrightarrow[n]{} \xrightarrow[m]{} IFifo$$

$$OMultiPacket \xrightarrow[n]{} \xrightarrow[m]{} IUFifo \qquad OLazyMultiPacket \xrightarrow[n]{} \xrightarrow[m]{} IUFifo$$

$$OGeneric \xrightarrow[n]{} \xrightarrow[m]{} IMultiPacket$$

Figura 3.8: Conexiones MultiPacket simples ($\forall\, n,m \in \mathbb{N}; n,m \geq 1$).

3.7. Componentes compuestos

Los componentes *CoolBOT* pueden ser descompuestos en dos tipos, componentes atómicos y componentes compuestos. Los primeros son componentes simples ideados para abstraer el hardware subyacente, implementar algoritmos genéricos o encapsular librerías o bibliotecas. Sin embargo, existe la posibilidad de crear componentes más complejos a partir de los atómicos. Los componentes compuestos tienen como atributos instancias de componentes simples y/o otros componentes compuestos, de forma que se establece una jerarquía aprovechando la modularidad que ofrece el concepto de componente usado por *CoolBOT*.

La idea principal es que los componentes compuestos implementan su funcionalidad apoyándose en las de componentes más simples. Cada componente compuesto supersiva, usando los puertos de control y monitorización, las acciones de los componentes que integra. Esta jerarquía de componentes se ilustra en la figura 3.9.

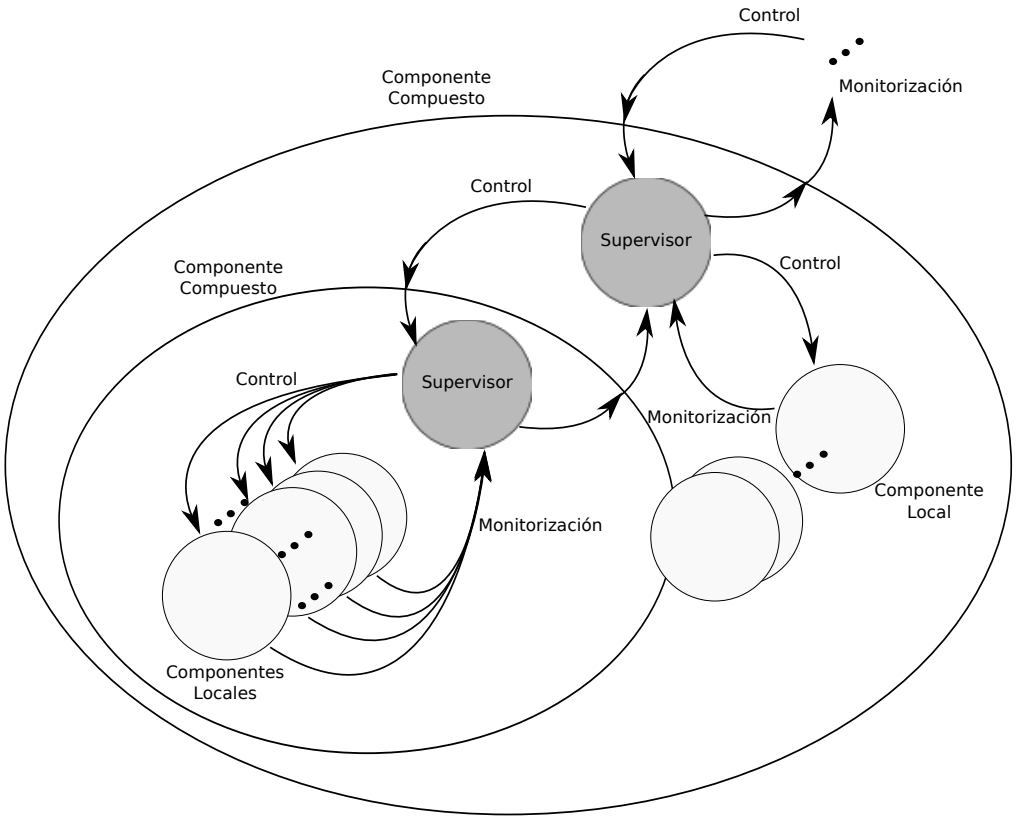

Figura 3.9: Jerarquía de componentes.

Capítulo 4

Metodología, recursos y plan de trabajo

4.1. Metodología

4.1.1. El proceso de desarrollo

Un proceso define *quién* está haciendo *qué*, *cuándo*, y *cómo* para alcanzar un determinado objetivo. En la ingeniería del software el objetivo es construir un producto software o mejorar uno existente. Un proceso efectivo proporciona normas para el desarrollo eficiente de software de calidad. Captura y presenta las mejores prácticas que el estado actual de la tecnología permite. En consecuencia, reduce el riesgo y hace el proyecto más predecible.

La metodología a utilizar durante el desarrollo de este proyecto será el Proceso Unificado (UP). El Proceso Unificado [Larman, 2006] es un proceso de desarrollo de software de gran éxito para la construcción de sistemas orientados a objetos. El Proceso Unificado combina las prácticas comúnmente aceptadas como "buenas prácticas", tales como ciclo de vida iterativo y desarrollo dirigido por los casos de uso, en una descripción consistente y bien documentada. El UP fomenta muchas buenas prácticas, pero una destaca sobre las demás: el desarrollo iterativo. El desarrollo se organiza en una serie de mini-proyectos cortos, de duración fija (alrededor de cuatro semanas) llamados iteraciones; el resultado de cada uno es un sistema que puede ser probado, integrado y ejecutado. Cada iteración incluye sus propias actividades de análisis de requisitos, diseño, implementación y pruebas. El ciclo de vida iterativo se basa en la ampliación y refinamiento sucesivos del sistema mediante múltiples iteraciones, con retroalimentación cíclica y adaptación como elementos principales que dirigen para converger hacia un sistema adecuado. El sistema crece incrementalmente a lo largo del tiempo, iteración tras iteración, y por ello, este enfoque también se conoce como **desarrollo iterativo e incremental** (ver Figura 4.1).

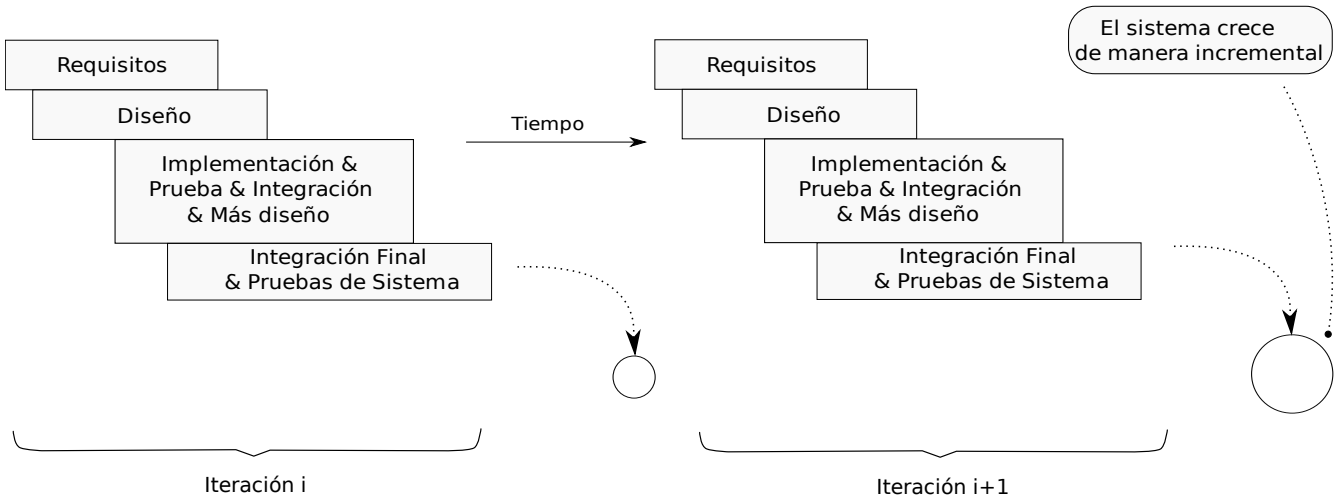

Figura 4.1: Desarrollo iterativo e incremental

El resultado de cada iteración es un sistema ejecutable, pero incompleto; no está preparado para ser puesto en producción. El sistema podría no estar listo para su puesta en producción hasta después de varias iteraciones (por ejemplo 10 ó 15).

La salida de una iteración no es un prototipo experimental o desechable. La salida es un subconjunto de la funcionalidad total con calidad de producción del sistema final.

Aunque, en general, cada iteración aborda nuevos requisitos y amplía el sistema incrementalmente, una iteración podría, ocasionalmente, volver sobre el software que ya existe y mejorarlo; por ejemplo, una iteración podría centrarse en mejorar el rendimiento de un subsistema, en lugar de extenderlo con nuevas características.

El ciclo de vida iterativo proporciona una serie de beneficios al proceso de desarrollo tales como:

- Mitigación tan pronto como sea posible de riesgos altos (técnicos, requisitos, objetivos, usabilidad y demás).
- Progreso visible en las primeras etapas.
- Una temprana retroalimentación, compromiso de los usuarios y adaptación, que nos lleva a un sistema refinado que se ajusta más a las necesidades reales del personal involucrado.
- Gestión de la complejidad; el equipo de desarrollo no se ve abrumado por la "parálisis del análisis o pasos muy largos y complejos".
- El conocimiento adquirido en una iteración se puede utilizar metódicamente para mejorar el propio proceso de desarrollo, iteración a iteración.

El Proceso Unificado se ha elegido como la metodología a seguir durante el desarrollo de este trabajo debido a que se ajusta a los proyectos de infraestructura. Es decir, se plantea un análisis inicial, se diseñan e implementan las primeras funcionalidades del sistema y se obtiene una versión prueba del mismo. A continuación se refinan los requisitos y se contempla en el diseño, para luego ser implementado. Este proceso se realiza durante varias iteraciones incrementando las capacidades del sistema. Se obtiene así un prototipo cada vez más elaborado que puede ser testeado independientemente para verificar la correctitud del diseño e implementación de cada iteración.

4.1.2. Aplicando el Proceso Unificado

Como se explicó en el apartado anterior, el Proceso Unificado se repite a lo largo de una serie de ciclos que constituyen la vida de un sistema. Cada ciclo produce una nueva versión del sistema y cada versión es un producto preparado para su entrega.

En este proyecto se ha dividido el ciclo de desarrollo en dos conjuntos de iteraciones:

El primer conjunto de iteraciones corresponde al desarrollo del protocolo de comunicación. En primer lugar se definen las operaciones que debe contemplar para lograr la funcionalidad esperada. Luego, tras diseñar las clases que lo integran, se implementan éstas y se hacen pruebas de envíos de paquetes sencillos. A posteriori se va refinando el diseño y se añaden operaciones adicionales que complementen la funcionalidad del protocolo. Adicionalmente, se prueban paquetes más complejos, similares a los que debe soportar el protocolo. Iteración tras iteración se obtiene una versión más completa del protocolo

Una vez obtenida la versión final del protocolo de comunicación, se realiza otro conjunto de iteraciones, correspondiente al desarrollo de la interfaz Web. Para ello es necesario determinar las funcionalidades que necesita el usuario operador de la aplicación para lograr la teleoperación con un robot remoto y los requisitos software demandados por la aplicación. En cada iteración se diseña, implementa y prueba una de las funcionalidades destacadas en la etapa de análisis, hasta satisfacer definitivamente el ciclo de desarrollo.

A modo de resumen, en la ilustración 4.2 se observa cómo se ha aplicado este proceso de desarrollo de software.

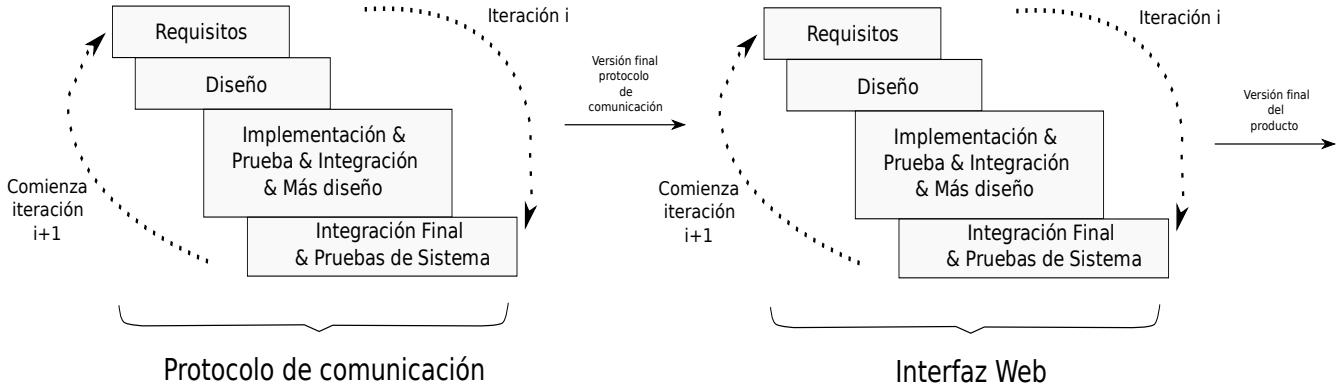

Figura 4.2: Proceso Unificado aplicado a este proyecto

4.2. Recursos necesarios

4.2.1. Recursos hardware

Durante el desarrollo del proyecto han sido utilizados los siguientes componentes hardware:

- MacBook *Macintosh* Core 2 Duo de Intel a 2 GHz, 2 GB RAM DDR3, 160 GB HD.
- Robot Pioner III.
- Cárama Logitech QuickCam USB
- Red inalámbrica.

4.2.2. Recursos software

Durante el desarrollo de este trabajo se han utilizado un gran número de herramientas software tanto para el sistema operativo GNU/Linux como para MacOSX. Concretamente las distribuciones utilizadas han sido:

- Ubuntu 8.10 Intrepid Ibex
- Mac OS X 10.5 Leopard

A continuación se puede observar una lista con algunas de las aplicaciones utilizadas durante el desarrollo de este trabajo (las más significativas), clasificadas según el género al que pertenecen.

- Entornos de desarrollo integrado
  - *Kdevelop*(GNU/Linux, MacOSX): Entorno de desarrollo integrado en C++ para el desarrollo de toda la infraestructura distribuida.
  - *NetBeans*(GNU/Linux, MacOSX): Entorno de desarrollo integrado para el desarrollo de la interfaz Web Java.
- Librerías
  - *AceLib*(GNU/Linux, MacOSX): Librería para el desarrollo de aplicaciones distribuidas.
  - *Swing*(GNU/Linux, MacOSX): Librería para la creación de interfaces gráficas de usuario en Java.
  - *Swig*(GNU/Linux, MacOSX): Librería para inclusión de código C y C++ en aplicaciones de lenguajes de *scripting* tales como PERL, PHP, PYTHON, TCL, RUBY o Java.
- Servidor de versiones
  - *cvs*(GNU/Linux): Servidor de versiones donde reside la versión de *CoolBOT* sin soporte de red.
  - *git*(GNU/Linux, MacOSX): Servidor de versiones donde se encuentra la nueva versión de *CoolBOT* con soporte de red así como todos los componentes *CoolBOT* y sus vistas.
- Editores LaTeX
  - *Kile*(GNU/Linux): Editor de código LaTex para la realización del presente documento.
  - *Texmaker*(MacOSX): Editor de código LaTex para la realización del presente documento.
- Gráficos vectoriales
  - *OmniGraffle*(MacOSX): Porgrama para la edición de gráficos vectoriales.
  - *Inkscape*(GNU/Linux, MacOSX): Programa de edición de gráficos vectoriales escalables (SVG).
  - *Xfig*(GNU/Linux): Programa para la edición de gráficos vectoriales.
- Compiladores
  - *gcc*(GNU/Linux, MacOSX): Compilador GNU del lenguaje C.
  - *g++*(GNU/Linux, MacOSX): Compilador GNU del lenguaje C++.
  - *pdfLatex*(GNU/Linux, MacOSX): Compulador código LaTex.
- Depuradores
  - *gdb*(GNU/Linux): Depurador GNU de código C/C++.

  - *cgdb*(GNU/Linux): Depurador GNU de código C/C++ con un front-end visual.

- Diagramas UML
  - *DIA*(GNU/Linux): Programa para la creación de diagramas UML con posibilidad de exportar a distintos formatos.

- Herramientas de red
  - *Wireshark*(GNU/Linux): Herramienta para la captura y análisis del tráfico por la red.

- Utilidades
  - *MacPorts*(MacOSX): Herramienta para la compilación e instalación de paquetes *open-source* para MacOSX.
  - *Fink*(MacOSX): Gestor de paquetes que intenta portar todo el software de Unix de código abierto a Dawrin y MacOSX. Permite tanto la descarga de binarios precompilados o de todo el código fuente.

4.3. Plan de trabajo

- Protocolo de comunicación.
  - Documentación y herramientas.
  - Análisis de requisitos del protocolo de comunicación.
  - Diseño del protocolo de comunicación.
  - Implementación del protocolo de comunicación.

- Interfaz Gráfica en Java.
  - Análisis de requisitos de la interfaz gráfica.
  - Diseño de la interfaz gráfica.
  - Implementación de la interfaz gráfica.

- Validación y conclusiones.
  - Test de validación.
  - Análisis de resultados y elaboración de conclusiones al respecto.

- Presentación y defensa del PFC.
  - Confección de la memoria del PFC.
  - Preparación y defensa oral del PFC.

Capítulo 5

Análisis

En este capítulo se presenta el sistema sobre el que se va a trabajar, descrito mediante un *modelo de dominio* que describe el entorno del problema que se piensa resolver. Una vez asentados los elementos más importantes del contexto del sistema se abordará la captura y descripción de requisitos mediante el *modelo de casos de uso*, para continuar con una descripción de cada una de las herramientas escogidas con el fin de satisfacer los requisitos de usuario y requisitos del software, presentados y justificados en las diferentes secciones de este capítulo.

5.1. Modelo de dominio

Un modelo del dominio captura los tipos más importantes de objetos en el contexto del sistema. Los objetos del dominio representan las "cosas" que existen o los eventos que suceden en el entorno en el que trabaja el sistema [Jacobson, Booch and Rumbaugh, 2000].

Muchos de los objetos del dominio o clases (para emplear una terminología más precisa) pueden obtenerse de una especificación de requisitos o mediante la entrevista con los expertos del dominio. Las clases del dominio particularizadas en el contexto de este proyecto aparecen en tres formas típicas:

- Objetos del negocio que representan cosas que se manipulan en el negocio, como *componentes*, *puertos*, *hilos de puertos*, etc.

- Objetos del mundo real y conceptos de los que el sistema debe hacer un seguimiento como *robot*, *sensores*, *interfaz usuario-máquina*, etc.

- Sucesos que ocurrirán o han ocurrido, como *la acción de comandar al robot*, *lecturas de los sensores*, etc.

El modelo del dominio se describe mediante diagramas UML[1] (especialmente diagrama de clases). Estos diagramas muestran a los clientes, usuarios, supervisores y a otros desarrolladores las clases del dominio y cómo se relacionan unas con otras mediante asociaciones.

A continuación se van a describir dos modelos de dominio para comprender mejor todo el contexto en el que se desenvuelve este proyecto. Uno de ellos correspondiente a las asociaciones que posee todo componente *CoolBOT* y otro ilustrativo de los requisitos o necesidades que se demandan para este trabajo

Modelo del dominio de un componente *CoolBOT*

La figura 5.1 representa el modelo de dominio de un componente *CoolBOT*. Todos los conceptos representados en este diagrama como clases han sido comentados en el capítulo 3. Ahora, con este modelo se proporciona información acerca de cómo se relacionan entre ellos desde el punto de vista del software.

La clase *Component* representa un componente *CoolBOT*. Un componente *CoolBOT* es una entidad que puede ser atómica o compuesta, es decir, formada por otros componentes. Lo más común es que sea lo primero, aunque todo depende de la complejidad del sistema que se desee desarrollar.

Un componente está dirigido por su autómata de estados, con el fin de que sea controlable. En todo momento el componente se encontrará en un único estado. Además, éste puede tener además del hilo principal del componente otros hilos o hilos de puertos[2] (*PortThread* según figura 5.1) que se ejecutarán concurrentemente con el primero.

Además de lo comentado en el párrafo anterior un componente posee puertos de entrada a través de los que recibirá datos desde otros componentes y puertos de salida, mediante los cuales enviará información hacia otros componentes conectados a él. Los puertos reciben o envían paquetes de puertos (*PortPacket* en la figura 5.1), que son los datos que manejan éstos. Así pues, para este modelo se han indicado como ejemplo dos paquetes de puertos: *OdometryPacket*, que almacena información relativa a la odometría del robot y *CameraImagePacket* para representar una imagen extraída de una cámara integrada con el robot. Todo componente posee al menos un puerto de entrada para poder ser controlado y uno de salida para ser monitorizado.

[1] *Unified Modeling Language* (UML). Es el lenguaje de modelado de sistemas software más conocido y utilizado en la actualidad

[2] Los hilos de puertos se asocian a puertos del componente para que los atiendan y ejecuten alguna función cuando haya actividad en ellos

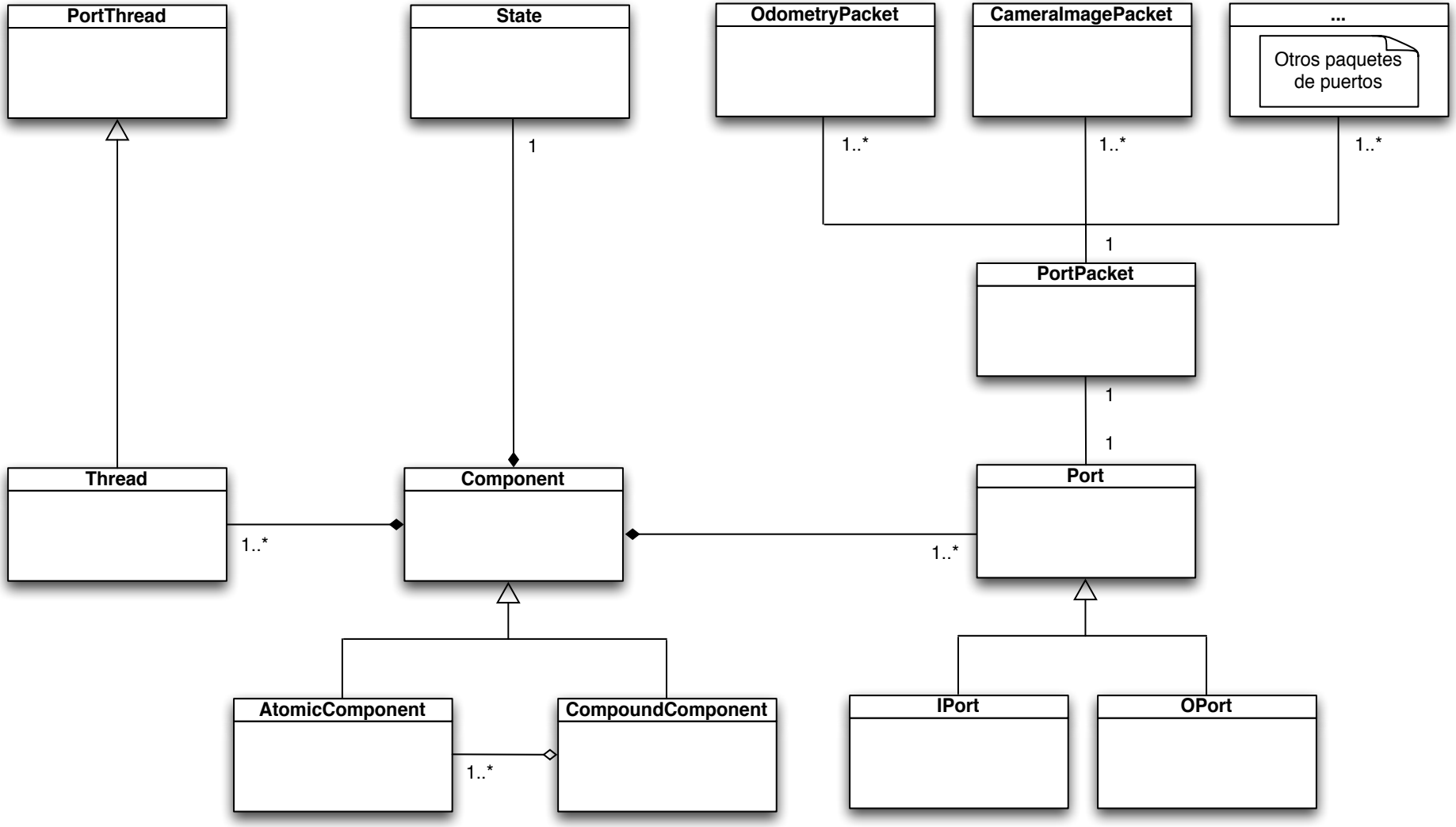

Figura 5.1: Modelo del dominio de un componente *CoolBOT*

Modelo del dominio del sistema distribuido *CoolBOT*

El sistema que se pretende desarrollar en este proyecto se corresponde a alto nivel con el diagrama de la figura 5.2. En esta imagen podemos ver que los usuarios que van a utilizar la aplicación (clientes WWW) se comunican con el robot a través de una interfaz Web. Esta interfaz está compuesta de *vistas* que recibirán datos desde un sistema distribuido y los presentarán gráficamente al cliente WWW. Asímismo, este usuario será capaz de enviar comandos hacia el robot por medio de las *vistas* integradas en la interfaz Web que éste utiliza.

El sistema distribuido presente en el diagrama 5.2 representa un sistema de componentes conectados a través de la red que reciben y envían información desde y hacia el cliente WWW para llevar a cabo el manejo y la monitorización de un robot móvil remoto.

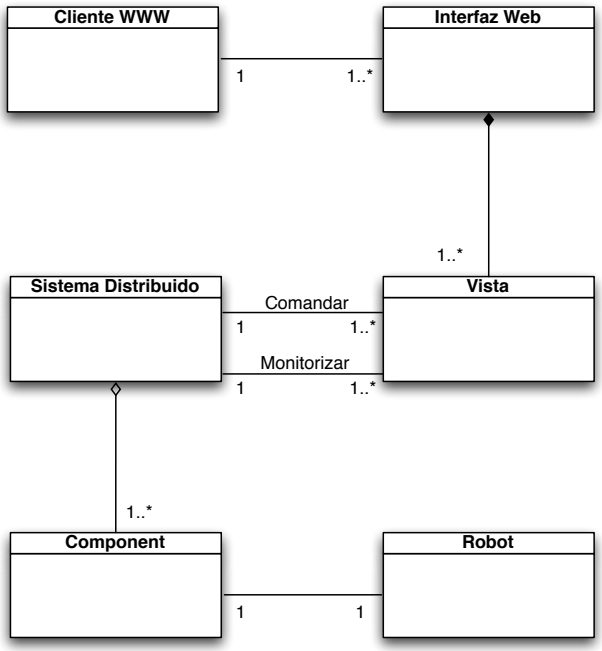

Figura 5.2: Modelo del dominio del sistema distribuido

Debido a que el objetivo del modelado del dominio es comprender y describir las clases más importantes dentro del contexto del sistema, evitando especificar los detalles relativos a la representación, los diagramas expuestos en este apartado no contienen métodos, responsabilidades ni atributos.

5.2. Modelo de casos de uso

El modelo de casos de uso [Jacobson, Booch and Rumbaugh, 2000] permite que los desarrolladores software y los clientes lleguen a un acuerdo sobre los requisitos, es decir, sobre, sobre las condiciones y posibilidades que debe cumplir el sistema. El modelo de casos de uso sirve como acuerdo entre clientes y desarrolladores y proporciona la entrada fundamental para el análisis, diseño y las pruebas.

Un modelo de casos de uso es un modelo del sistema que contiene actores, casos de uso y sus relaciones. En los siguientes apartados se definirán cada uno de estos conceptos y se especificarán cuáles son para este proyecto mediante los diagramas del lenguaje unificado de modelado (UML).

5.3. Actores

Normalmente, un sistema tiene muchos tipos de usuarios. Cada tipo de usuario se representa por un actor. Los actores utilizan el sistema interactuando con los casos de uso. Un caso de uso es una secuencia de acciones que el sistema lleva a cabo para ofrecer algún resultado de valor para un actor.

CoolBOT es un proyecto pensado en términos de software libre, por lo que no está dirigido a un usuario específico, sino a una amplia comunidad. Por ello, según el uso que se desee hacer de *CoolBOT*, podemos encontrar cuatro actores o tipos de usuarios divididos en tres grupos (imagen 5.3):

- Desarrollador del *framework CoolBOT*: Es aquel actor que añade nuevas funcionalidades a la infraestructura de *CoolBOT*, como por ejemplo, añadir un soporte de red extensible a todos los componentes *CoolBOT*.

- Desarrollador de sistemas robóticos *CoolBOT*: En este grupo se engloba a todos los posibles usuarios programadores de sistemas robóticos. Por un lado se encuentran los *desarrolladores de componentes*, que son los que programan los componentes que formarán parte del sistema robótico. Por otro lado se agrupan los *integradores de sistemas* para la construcción de sistemas robóticos a partir de una red de componentes.

- Operador: Es quien hace uso de los sistemas robóticos ya desarrollados con el fin de realizar pruebas o demos teleoperativas. Este actor hace uso de una interfaz para comandar y/o monitorizar robots.

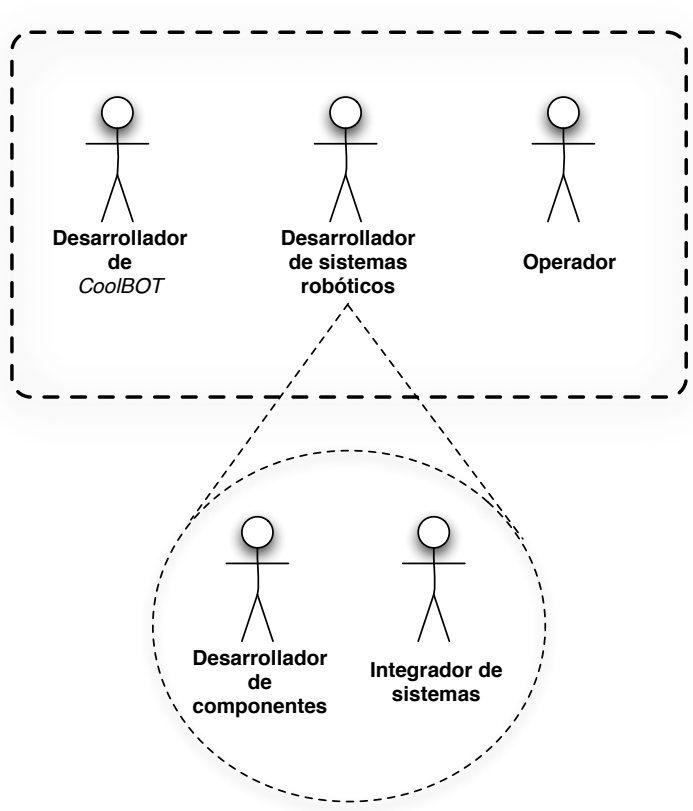

Figura 5.3: Actores

5.4. Casos de uso

Los casos de uso han sido adoptados casi universalmente para la captura de requisitos de sistemas software en general, y de sistemas basados en componentes en particular. Pero los casos de uso son mucho más que una herramienta para capturar requisitos, dirigen el proceso de desarrollo en su totalidad. Los casos de uso son la entrada fundamental cuando se identifican y especifican clases, subsistemas e interfaces, cuando se identifican y especifican casos de prueba, y cuando se planifican las iteraciones del desarrollo y la integración del sistema.

La captura de requisitos tiene dos objetivos: encontrar los verdaderos requisitos y representarlos de un modo adecuado para los usuarios, clientes y desarrolladores. Entendemos por "verdaderos requisitos" aquellos que cuando se implementen añadirán el valor esperado para los usuarios. Con "representarlos de un modo adecuado para los usuarios, clientes y desarrolladores" queremos decir en concreto que la descripción obtenida de los requisitos debe ser comprensible por usuarios y clientes.

A continuación vamos a indicar mediante diagramas de casos de uso el conjunto de casos de

uso que debe proporcionar la infraestructura de red que se va a añadir *CoolBOT* para cada uno de los actores señalados en el punto anterior.

5.4.1. Operador

Como ya se mencionó en el apartado anterior, el uso que este actor hace de *CoolBOT* se basa en la manipulación (en términos de envío de órdenes) y monitorización de un robot móvil. Por tanto, sus interacciones con el sistema se resumen en la figura 5.4.

El operador se comunica con un actor denominado "Sistema robótico soportado por CoolBOT" en forma de envío de comandos hacia el robot, y recibe desde éste información producida desde el servidor donde escucha el robot, relacionada con el mismo para monitorizarlo.

El actor "Sistema robótico soportado por *CoolBOT*" alude a todo el sistema robótico de componentes *CoolBOT* distribuidos.

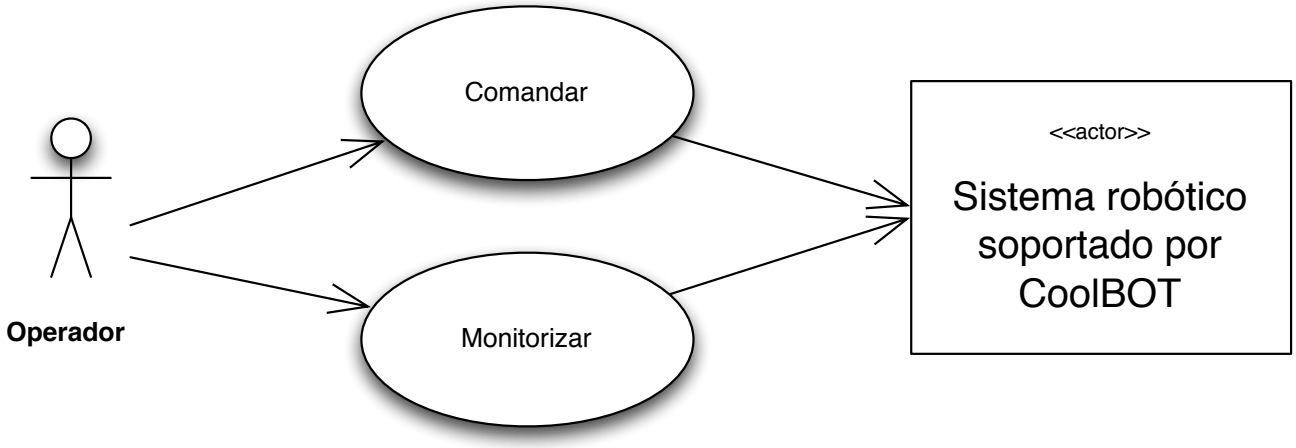

Figura 5.4: Diagrama de caso de uso para el actor *operador*

5.4.2. Desarrollador de sistemas robóticos

Dentro de desarrolladores de sistemas robóticos podemos diferenciar dos conjuntos de actores: los desarrolladores de componentes *CoolBOT* y los integradores de componentes.

El desarrollador de componentes *CoolBOT* es quien crea nuevos componentes distribuidos que podrán ser interconectados a través de los integradores de componentes. Por el hecho de que ambos son necesarios para poder desarrollar un sistema robótico e integrarlo adecuadamente se ha decidido mantenerlos en el mismo grupo.

Los casos de uso para el desarrollador de sistemas robóticos se pueden observar en la imagen 5.5. A continuación pasamos a enumerar y detallar cada uno de estos requisitos:

1. **Instanciar componentes que escuchen en un puerto TCP/IP**: Una vez que el usuario desarrollador de componentes *CoolBOT* haya creado un componente con sus entradas y sus salidas puede necesitar que éste sea capaz de comunicarse a través de la red con otros componentes. Para que un componente lance su nivel de red y escuche por él, será necesario instanciarlo indicando un puerto TCP/IP, que utilizarán otros componentes para identificarlo dentro de una máquina en concreto.

2. **Poner un componente en estado de RUNNING**: Como se comentó en el capítulo 3, el ciclo de vida de un componente viene definido por un autómata de estados. Un componente comienza a ejecutar su hilo *main* realmente cuando transita en su autómata al estado "*RUNNING*". Como requisito, es necesario que un componente que vaya a conectarse a la red, cuando transite a estado *RUNNING*, active su nivel de red. A partir de este momento el componente puede recibir y enviar datos desde/hacia la red.

3. **Conectar puertos de componentes**: Al igual que se realizan los conexionados de puertos entre diferentes componentes que residen en la misma máquina, es necesario proporcionar el mismo mecanismo para conectar componentes que están en diferentes máquinas, indicando como información adicional la dirección IP y el puerto donde escucha el componente remoto.

4. **Desconectar puertos de componentes**: De forma análoga al conexionado de puertos, debe existir un mecanismo que permita la desconexión de puertos de componentes remotos, al igual que se hace de forma local.

5. **Crear nuevas clases de datos que vayan dentro de paquetes de puertos**: Como se explicó en el capítulo 3, los componentes *CoolBOT* tienen puertos de entrada y de salida para llevar a cabo la recepción y el envío de paquetes de puertos. El desarrollador de componentes *CoolBOT* indicará el tipo de cada puerto de entrada o de salida, así como el/los paquetes de puertos que acepta, durante la creación del componente. En algún caso puede ser necesario añadir información adicional dentro de un paquete de puerto ya existente. Por ejemplo, imaginemos que existe un paquete de puerto que envía una imagen que representa el mapa que ha explorado un robot, en forma de matriz de píxeles siguiendo el modelo de color rojo, verde y azul (RGB). Supongamos que nos interesa saber cuánto se tarda en enviar ese mapa de un componente a otro (bien sea local o remoto). Podríamos almacenar clase que represente una marca de tiempo o *timestamp* en el paquete de puertos que representa el mapa.

   En el caso de que nuestro componente se conecte vía red, es necesario definir las operaciones de serialización (*marshalling*) para la nueva clase que hemos añadido al paquete de puerto. El proceso de serialización es necesario para que la nueva clase que se envía junto con el esto del paquete de puerto sea interpretada correctamente según el *byte-order* de la máquina donde se ejecuta el componente receptor.

6. **Crear nuevos paquetes de puertos**: En alguna ocasión puede que sea necesario crear un nuevo paquete de puerto en lugar de crear una nueva clase dentro de un paquete de puerto. Esto se debe a que cada desarrollador de componentes *CoolBOT* tiene sus propias necesidades y requiere unos paquetes de puertos acorde a los datos que maneja su componente.

   Al igual que comentamos en el punto anterior, en caso de que el componente vaya a conectarse a la red, es necesario que el nuevo paquete de puerto implemente sus propias funciones de serialización para que los datos que conforman el paquete sean interpretados en cualquier componente receptor independientemente de la arquitectura de la máquina donde éste se ejecute.

5.4.3. Desarrollador de *CoolBOT*

El desarrollador de *CoolBOT* es el principal artífice de añadir o mejorar las funcionalidades del *framework*, que serán utilizadas por el desarrollador de componentes *CoolBOT*. Los requisitos necesarios por este actor presentes en la figura 5.6 corresponden a aquellas funcionalidades necesarias para integrar el nivel de red dentro del *framework*.

1. **Añadir / modificar regla de procedimiento del protocolo de comunicación**: El protocolo de comunicacion utilizado por los componentes distribuidos debe ser fácilmente escalable para permitir a este actor la adición de nuevos paquetes al protocolo de comunicación o nuevas reglas de procedimiento. Esto se debe a que en algún momento un desarrollador del *framework* estima oportuno realizar alguna comprobación adicional en la secuencia de acciones del protocolo de comunicación, o segmentar alguna regla ya existente para hacerla más específica. Lo importante es que el protocolo sea lo suficientemente flexible como para permitir modificaciones a posteriori.

   Añadir un nuevo paquete al protocolo implica añadir o modificar una regla de procedimiento, debido a que, la necesidad del primero debe estar contemplada dentro de la secuencia de acciones del protocolo.

2. **Crear nuevos tipos de puertos de componentes**: En el capítulo 3 vimos que en *CoolBOT* existen diferentes tipos de puertos que gestionan de varias maneras los datos enviados y/o recibidos por el componente. Un desarrollador de *CoolBOT* podría pensar en una manera nueva de cómo recibir o enviar los paquetes de puertos, creando un tipo nuevo de puerto para los componentes. Esta funcionalidad permite a los desarrolladores de componentes *CoolBOT* pensar en nuevos componentes o modificar los ya existentes para que utilicen estos nuevos tipos de puertos.

   Crear nuevos tipos de componentes implica además establecer las compatiblidades con los tipos de puertos ya existentes, indicando aquellos tipos de puertos de entrada o de salida que pueden conectarse a un puerto del tipo nuevo.

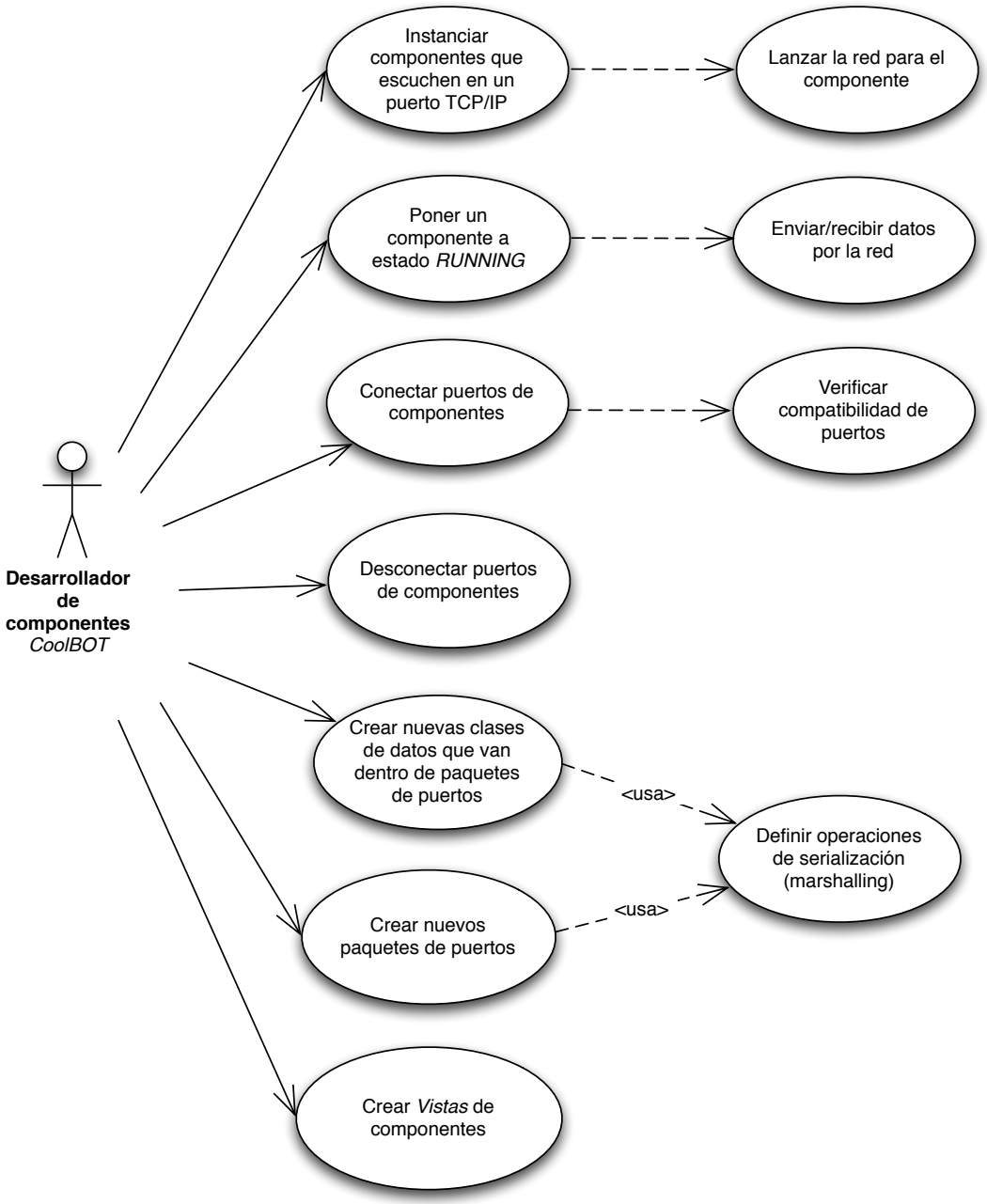

Figura 5.5: Diagrama de caso de uso para el actor *desarrollador de componentes* CoolBOT

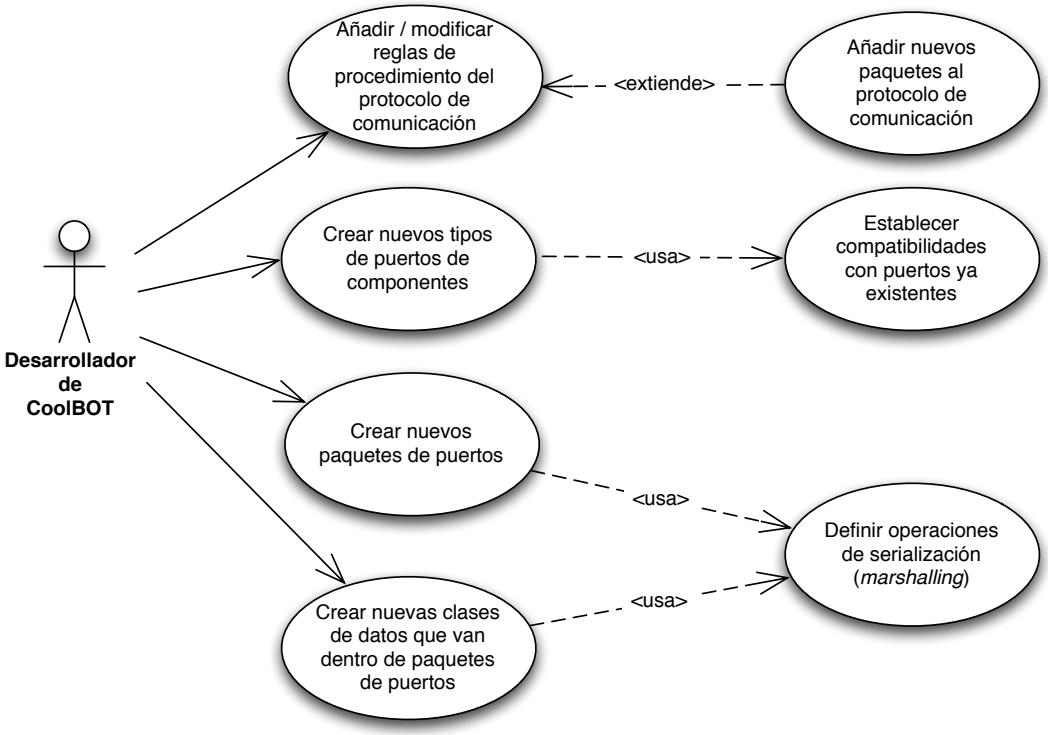

Figura 5.6: Diagrama de caso de uso para el actor *desarrollador de CoolBOT*

3. **Crear nuevas clases de datos que van dentro de paquetes de puertos**: Como ya se mencionó en el 5º caso de uso del desarrollador de componentes *CoolBOT*, esta funcionalidad permite añadir más datos a los paquetes de puertos, haciendo que estos puedan ser personalizados por usuarios desarrolladores en base a sus necesidades.

   Estas nuevas clases deben proporcionar métodos para su serialización cuando el componente se encuentre conectado a la red.

4. **Crear nuevos paquetes de puertos**: Aunque esta funcionalidad fue comentada en el último caso de uso descrito para el desarrollador de componentes *CoolBOT*, es una necesidad compartida por ambos tipos de desarrolladores. En particular el uso de los *templates* es una forma de aprovechar código orientado a datos genéricos que permite la utilización de métodos ya existentes con diferentes tipos de datos. Así, un usuario desarrollador del *framework* puede proporcionar una batería de paquetes de puertos a otros desarrolladores de componentes que quieran hacer uso de la funcionalidad dada con los tipos de datos que estos necesiten manejar.

   Como requisito para aquellos componentes distribuidos es necesario definir operaciones de *marshalling* y *demarshalling* para estos nuevos paquetes de puertos, de forma que los datos que almacenen se interpreten correctamente en el componente receptor.

5.5. Requisitos del sistema

En la sección 5.4 se presentaron los casos de uso para cada actor o usuario del sistema. A continuación se presenta un resumen de las características que debe tener el sistema a desarrollar en este proyecto con el fin de aportarles a cada uno de ellos la funcionalidad indicada en los casos de uso.

Elección del lenguaje de programación

CoolBOT se ha desarrollado utilizando orientación a objetos, en concreto está escrito en C++. Este hecho, unido a las deseables características que ofrece la orientación a objetos y a la existencia de una variedad de librerías y *frameworks* para la intercomunicación de procesos *Interprocess Communication (IPC)*, sienta las bases para tomar la decisión de que el lenguaje para la implementación de la infraestructura de red sea C++.

Por otro lado, debido al requisito de proporcionar una interfaz Web de teleoperación con el sistema descrito en el párrafo anterior, se hará uso de los *applets* de Java.

Debido a que el desarrollo del proyecto se llevará a cabo parte en Java y parte en C++ y será necesario integrar código de ambos lenguajes en una sola aplicación, se hará uso de una librería para el encapsulado de código C/C++ desde lenguajes interpretados como Java. Esta librería, *Swig*, se describe más detalladamente en la sección 5.10.

Requisitos de comunicación entre componentes distribuidos:

Para que los componentes *CoolBOT* puedan interaccionar de manera remota es necesario algún mecanismo que guíe la comunicación entre ellos. Debido a que el canal de comunicación (es decir, la red) es imperfecto, se requiere algún medio para el intercambio de datos fiable, como por ejemplo, un protocolo de comunicación.

Un protocolo de comunicación consiste en un conjunto de reglas normalizadas para el envío apropiado de información a través de un canal de comunicación. Este protocolo debe responder a unos requerimientos de fluidez en las comunicaciones, de manera que miniminize el *delay* presente en las comunicaciones remotas.

Además es necesario que el trasiego de paquetes vía red sea transparente para el usuario desarrollador de componentes *CoolBOT*. Es decir, este usuario debe programar la funcionalidad de su componente sin preocuparse de si el componente será instanciado local o remotamente. El componente seguirá funcionando en local como hasta ahora y en el caso remoto, será el protocolo de

comunicación el encargado de hacer llegar los datos al extremo remoto.

Se desarrollará por tanto un protocolo de comunicación que satisfaga los requerimientos indicados en el párrafo anterior. Las características de este protocolo están descritas en la sección 6.2. El hecho de comunicar componentes *CoolBOT* distribuidos, justifica el nombre elegido para el protocolo de comunicación: *Distributed CoolBOT Components Communication Protocol* (*DC3P*).

Para satisfacer los requisitos propuestos en este apartado, *DC3P* deberá hacer uso de un *middleware* que lo abstraiga de los detalles de las capas bajas del nivel de red y que sea portable entre diferentes máquinas. Es por esto que para el desarrollo de este proyecto se ha escogido un *middleware* para la programación de aplicaciones distribuidas, *Adaptive Communication Environment (ACE)*, descrito en la sección 5.8.

Representación intermedia de datos

El envío de información entre componentes situados en diferentes máquinas puede ocasionar un problema si la disposición de los datos en memoria varía de una a otra. Esto puede suceder entre máquinas con arquitecturas diferentes (*Intel*, *AMD*) o incluso entre máquinas con la misma arquitectura pero tamaños de palabra distintos. Para paliar este inconveniente es necesario hacer uso de una representación intermedia que codifique los datos dentro de un *buffer* de memoria y los transmita a través de una conexión de red como una serie de *bytes*. De esta manera se consigue que los datos enviados por la red sean correctamente interpretados en el extremo receptor. Esta representación intermedia hará uso de funciones de empaquetado (*marshalling*) y desempaquetado *demarshalling* de los datos. Estos conceptos al igual que un análisis en mayor profundidad sobre las representaciones intermedias de datos se explican con mayor profundidad en la sección 5.6. Para la representación intermedia de datos se ha utilizado *Common Data Representation (CDR)* de *CORBA* ya que es la representación empleada por el *middleware ACE*. En la sección 5.7 se explica con más detalle esta representación intermedia de datos.

Vistas de componentes *CoolBOT* para la interfaz Web

En los casos de uso (sección 5.4) definimos los requisitos para el actor operador. Éste utiliza el sistema para comandar y monitorizar a un robot móvil. Estas acciones las llevará a cabo a través de una interfaz que es la que comunica a este actor con el sistema de componentes *CoolBOT* conectado al robot.

Siguiendo la terminología adoptada en el modelo del negocio del sistema (sección 5.1) este actor se corresponde con el cliente WWW que hará uso de una interfaz Web para comunicarse con el robot remoto. Como explicamos en dicha sección, la interfaz Web estará compuesta de un conjunto de *vistas*, que no son más que representaciones gráficas de los datos producidos por los

componentes *CoolBOT* en función de la naturaleza de los mismos. Así, un componente que capture imágenes de una cámara *pan-tilt* integrada en un robot, puede tener una *vista* formada por una ventana donde mostrar la última imagen recibida y dos controles deslizables para controlar los movimientos de la cámara en horizontal (*pan*) y vertical (*tilt*).

Debido a que la interfaz de teleoperación tiene el requisito de ser una aplicación Web, es necesario hacer uso de un lenguaje que combine el desarrollo de interfaces con la posibilidad de integración de la misma como un servicio Web.

Por este requisito se ha seleccionado *Swing*, una librería escrita Java para el desarrollo de la interfaz Web y sus *vistas* cuya elección está justificada en la sección 5.9.

5.6. Representación intermedia, *Marshalling* y *Demarshalling*

El envío de datos a través de la red es una tarea no trivial debido a que tenemos que tener en cuenta las siguientes premisas:

- Los programas almacenan los datos en estructuras: Los programas trabajan con distintas estructuras de diversa complejidad que acceden a los datos de diferentes maneras.
- Los paquetes son una secuencia de bytes (sin estructura): Los datos se transmiten a través de la red como *streams* de bytes, por tanto, deben existir reglas conocidas por emisor y receptor para conocer la localización de los datos dentro del *stream*.
- Las estructuras de datos deben ser linealizadas para su transmisión: Las estructuras de datos se encuentran ubicadas en memoria, y es necesario un mecanismo de traducción que permita su envío por la red, sin perder información de tipos.

  Además de esto, las máquinas con diferente arquitectura almacenan los datos de diferentes maneras:

  - Tamaño de los datos (enteros de 16, 32, o 64 bits): Las máquinas con diferentes tamaños de datos trabajan con tamaños de palabra diferentes, así que es necesario que el receptor de los datos sepa traducirlos al tamaño de su bus de datos (Figura 5.7).
  - Acceso a los datos (Big-endian ó Little-endian): Por defecto, a nivel de red, el protocolo TCP/IP envía los datos ordenados como Big-endian. Ésto suscita un problema en la comunicación de máquinas con acceso a los datos Little-endian, condicionando a emisor y receptor a codificar y decodificar los datos a priori y a posteriori del envío, respectivamente (Figuras 5.8 y 5.9).

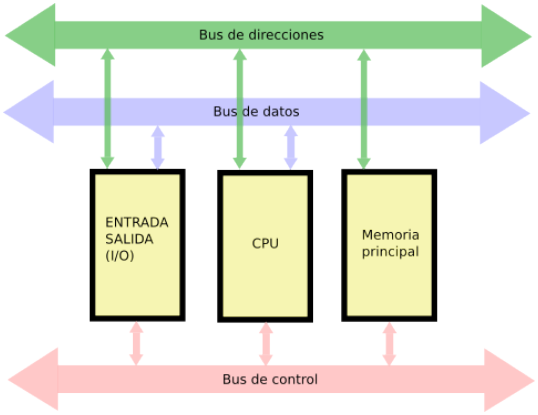

Figura 5.7: Ejemplo de Arquitectura Von Neumann

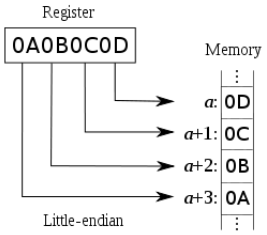

Figura 5.8: Organización Little-Endian.

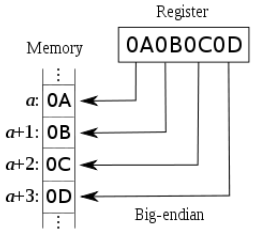

Figura 5.9: Organización Big-Endian.

- Codificación del texto ASCII/Unicode (UTF8/16/32): Máquinas con distintos sistemas operativos utilizan diferentes codificaciones de caracteres que requieren la conversión de los mensajes al sistema de representación del sistema operativo destino (figura 5.10).

Una solución a los problemas presentados previamente es el uso de una **representación intermedia** que convierta los datos en un formato común conocido tanto por emisor y receptor. Además, sería ideal enviar los datos en el formato propio (nativo) del emisor haciendo uso de etiquetas que indiquen el formato de los datos dentro del paquete.

Dentro del contexto de la comunicación entre máquinas (bien se trate de máquinas distintas o de procesos diferentes que operan dentro de la misma máquina), es necesario hablar de dos conceptos claves, *marshalling* y *demarshalling*.

Marshalling es el proceso de tomar un conjunto de datos y ensamblarlos de una manera adecuada para su transmisión en un mensaje. Es decir, convertir datos nativos en una representación de datos externa.

Por otro lado, el **demarshalling** consiste en desempaquetar una secuencia de bytes de un mensaje con el objetivo de producir un conjunto de datos en el destinatario. Es decir, convertir

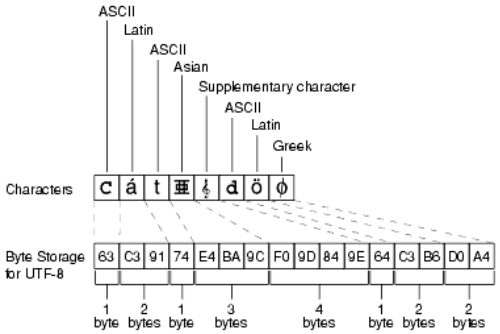

Figura 5.10: Bytes necesarios para almacenar diferentes tipos de caracteres en la representación UTF-8. Los caracteres ASCII (C,t y d) requieren sólo un byte. Los caracteres latinos y griegos (á, ö, y Ø) requieren 2 bytes. Los caracteres asiáticos requieren 3 bytes. Los caracteres adicionales (clave de sol) requieren 4 bytes de almacenamiento.

desde una representación externa a datos nativos.

En la actualidad existen varias tecnologías que hacen uso de estos conceptos. Algunos ejemplos son:

- CORBA Common data representation (Binario).
- Java's object Serialization (Binario).
- XML or eXtensible Markup Language (Texto).

5.7. CDR: CORBA's Common data representation

5.7.1. Introducción

La sintaxis de *CDR* indica un formato en el que representar distintos tipos de datos en una ristra (*stream*) de bytes. En definitiva un *stream* de bytes corresponde normalmente a un *buffer* de memoria que es enviado a otro proceso a través de una red de comunicación. A continuación se describen las principales características que establecen ciertas restricciones a la hora de crear una representación común a distintas máquinas y cómo son indicadas al pasar los datos a *CDR*, representación intermedia escogida para el envío de datos de componentes.

- ***Byte order* variable** - Máquinas con el mismo *byte order* pueden intercambiar mensajes sin necesidad de realizar un *byte swapping*. Cuando las máquinas que forman parte de la comunicación tienen diferente *byte order*, el emisor del mensaje determina el *byte order* del mensaje y el receptor es el responsable de realizar el *swapping* de bytes para convertirlo a su orden nativo. Cada encapsulado CDR contiene un flag que indica el *byte order* apropiado.
- **Tipos primitivos alineados** - Los tipos de datos primitivos OMG IDL (*Object Management Group Interface Definition Language*) están alineados en sus límites naturales dentro de los mensajes GIOP (*General Inter-ORB Protoco*), permitiendo que los datos sean manejados eficientemente por arquitecturas que fuerzan el alineamiento de datos en memoria.
- **Mapeo IMG IDL completo** - CDR describe representaciones para todos los tipos de datos OMG IDL, incluyendo pseudo-objetos transferibles tales como *TypeCodes*. Donde sea necesario CDR define representaciones para aquellos tipos de datos cuyas representaciones no están definidas.

5.7.2. Sintaxis de transferencia CDR

La sintaxis de transferencia CDR es el formato en el que los GIOP representan tipos de datos OMG IDL en un stream de octetos.

Un stream de octetos es un término abstracto que normalmente corresponde a un buffer de memoria que va a ser enviado a otro proceso o máquina a través de algún mecanismo IPC (*Inter Process Communication*) o vía red. Entederemos que un stream de octetos es una secuencia arbitrariamente larga (pero finita) de octetos (8 bits) con un comienzo bien definido. Los octetos dentro del stream están enumerados de *0* a *n-1*, donde *n* es el tamaño del stream. La posición numérica de un octeto en el stream es conocida como *índice*. Los índices de los octetos son utilizados para calcular los límites del alineamiento.

GIOP define dos tipos distintos de streams de octetos, *mensajes* y *empaquetados*. Los mensajes son las unidades básicas de intercambio de información en los GIOP.

Los empaquetados son streams de octetos dentro de los cuales las estructuras de datos OMG IDL pueden ser serializadas independientemente, indistintamente de cualquier contexto del mensaje. Una vez una estructura de datos ha sido encapsulada, el stream de octetos puede ser representado como el tipo de datos OMG IDL opaco **sequence<octet>**, que puede ser *serializado* nuevamente dentro de un mensaje u otro empaquetado.

5.7.3. Tipos primitivos

Los tipos de datos primitivos están especificados tanto para máquinas big-endian como little-endian. Los formatos de los mensajes incluyen etiquetas en las cabeceras de los mensajes que indican el *byte ordering* en el mensaje. Los empaquetados incluyen un *flag* que indica el *byte ordering* dentro del mismo. El *byte ordering* de cualquier empaquetado puede ser diferente del mensaje o del empaquetado dentro del cual está anidado. Es responsabilidad del receptor del mensaje traducir el *byte ordering* en caso de que sea necesario. Los tipos de datos primitivos están codificados en múltiplos de octetos.

5.7.4. Tipos construidos

Las primitivas que conforman el tipo construido son añadidas en un orden estricto:

- **sequence** - longitud (unsigned long) seguida de los elementos en orden.
- **string** - longitud (unsigned long) seguida de los caracteres en orden (16 bits de representación).
- **array** - elementos en orden (la longitud es conocida).
- **structure** - elementos en orden de declaración.
- **enumerated** - elemento (unsigned long).
- **union** - etiqueta de tipo seguida del elemento.

El *marshalling* se realiza de manera automática a partir de la especificación de tipos. Dichas especificaciones son dadas por el Lenguaje de Descripción de Interfaces (IDL). En la figura 5.11 se muestra el ejemplo de una estructura de datos la cual va a ser serializada siguiendo la descripción IDL de especificaciones de tipos y encapsulada dentro de un CDR (tabla 5.1).

```
struct Person {
      string name;                "Smith"
      string place;               "London"
      unsigned long year;         1934
}
```

Figura 5.11: Ejemplo de *marshalling* de código según especificación IDL

Bytes	Significado	Comentarios
00 00 00 05	5	Longitud de la string (5 caracteres)
53 6D 69 74	"Smith"	La string
68 00 00 00	"h"	Rellenado con "0" para alinear el siguiente long
00 00 00 00	6	Longitud de la string (alineado en límite de 4-bytes)
4C 6F 6E 64	"Lond"	La string
6F 6E 00 00	"on"	"0" añadidos para alinear el siguiente long
00 00 00 00	1934	unsigned long

Cuadro 5.1: Representación a nivel de bytes de datos construidos en CORBA.

5.7.5. Alineamiento

Con el objetivo de permitir que los datos primitivos sean introducidos y desalojados del stream de octetos con instrucciones especificamente diseñadas para estos tipos de datos primitivos, en CDR todos los tipos de datos deben estar alineados a sus límties naturales (p.e., el límite alineamiento de un dato primitivo es igual al tamaño del dato en octetos). Cualquier dato primitivo de n octetos debe empezar en un índice del stream de octetos múltiple de n. En CDR, n es 1,2,4 u 8. En la tabla 5.2 se indican los alineamientos para cada tipo de datos primitivos.

Esta restricción en *CDR* supone que existan ocasiones en las que al mover un dato a un *stream* de bytes sea necesario saltar ciertas posiciones para así colocar el dato debidamente alineado. Este hueco (*gap*) en el *stream* de bytes será el mínimo necesario para que el dato comience en una posición múltiplo del límite correspondiente para cada tipo de dato primitivo. La tabla 5.2 indica los alineamientos para cada tipo de dato primitivo considerado por *CDR*, dichos alineamientos son relativos al primer byte del *stream* que es indexado como 0. Cualquier dato insertado en el *stream* deberá empezar en una posición múltiplo de su valor de alineamiento.

Tipo de datos	Alineamiento de octetos (bytes)
char	1
wchar	1,2 ó 4
octet	1
short	2
unsigned short	2
long	4
unsigned long	4
long long	8
unsigned long long	8
float	4
double	8
long double	8
boolean	1
enum	4

Cuadro 5.2: Requisitos de alineamiento para tipos de datos primitivos OMG IDL.

5.8. Entorno de comunicación adaptativo: ACE

5.8.1. Introducción

ACE (Adaptive Communication Environment) es un framework de código abierto orientado a objetos que implementa muchos de los patrones centrales para el desarrollo de software de comunicación. ACE proporciona un amplio conjunto de capas escritas en código C++ reutilizable, y componentes framework que optimizan las tareas de las aplicaciones de comunicación entre plataformas, consiguiendo que el mismo código compile y funcione en diferentes combinaciones de sistema operativo y compilador de C++. Las tareas de las aplicaciones de comunicación provistas por ACE incluyen manejador de eventos y señales, inicialización de objetos y servicios, comunicación entre procesos (IPC), manejador de memoria compartida, enrutamiento de mensajes, (re)configuración dinámica de servicios distribuidos, ejecución concurrente y sincronización.

ACE está destinada a desarrolladores de aplicaciones y servicios de comunicación de alto rendimiento en tiempo real. Este entorno simplifica el desarrollo de aplicaciones interconectadas orientadas a objetos y los servicios que utilizan comunicación entre procesos, manejador de eventos, enlazador dinámico explícito y concurrencia. Además, ACE automatiza la configuración y reconfiguración del sistema, enlazando dinámicamente los servicios de las aplicaciones en tiempo de ejecución y ejecutando estos servicios en uno o más procesos o hilos.

ACE proporciona una API de acceso a los recursos de los sistemas operativos, un conjunto de

clases que encapsulan dichas APIs, varios Frameworks e incluso servicios y aplicaciones distribuidas en red listos para ser usados directamente.

Para separar las diferentes áreas y reducir la complejidad, ACE está diseñado utilizando una arquitectura en capas, desde las más cercanas al sistema operativo hasta las de más alto nivel, como podemos ver en la Ilustración 5.12

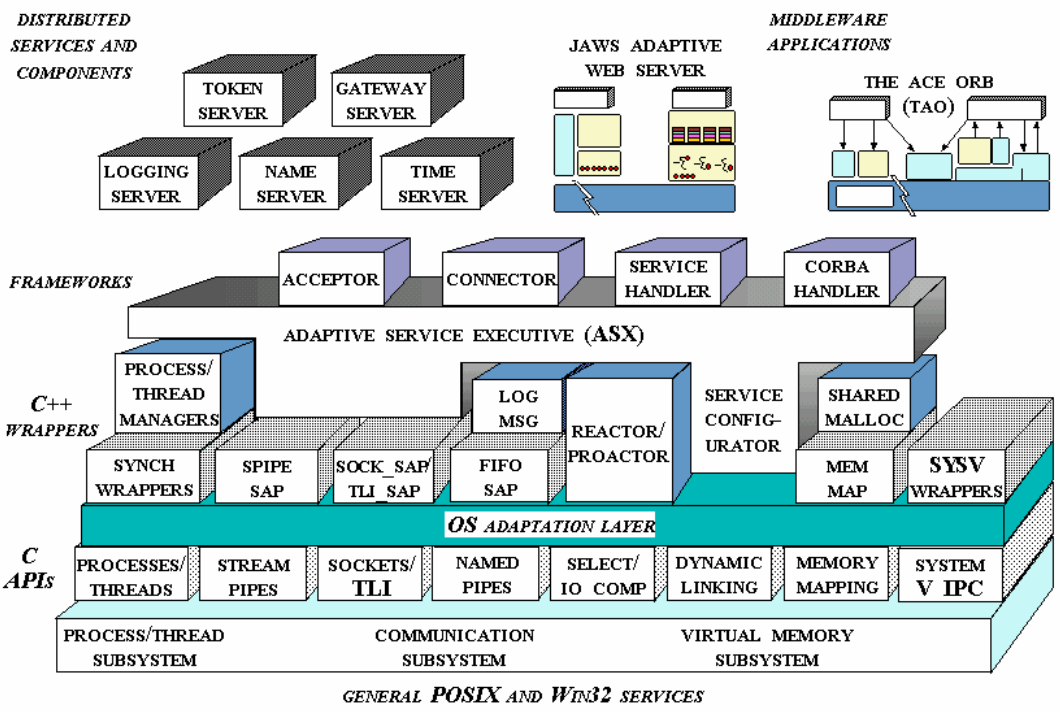

Figura 5.12: Estructura y funcionalidad de ACE [ACE,2007]

Las características que ofrecen cada una de estas capas y las funcionalidades que aportan para desarrollar aplicaciones en red son las siguientes:

- *Capa de adaptacion al sistema operativo*: Esta capa proporciona un conjunto de funciones para las operaciones y las llamadas al sistema operativo más comunes y se sitúa entre las APIs nativas de llamadas al sistema y el resto de ACE. Constituye aproximadamente un diez por ciento del total de ACE, y consiste en una clase llamada ACE_OS que alberga más de 500 métodos estáticos de C++. Son estos métodos los que encapsulan las APIs del sistema y nos ocultan los detalles específicos de cada plataforma, proporcionándonos un interfaz uniforme que podemos usar directamente como hacen las capas más altas de la arquitectura. Mediante la Capa de Adaptación ACE_OS, aseguramos la portabilidad y la mantenibilidad de nuestro código.
- *Capa de envoltorios C++ de ACE o wrapper facades*: Se pueden programar aplicaciones altamente portables utilizando la capa anterior, pero esta capa simplifica el desarrollo de

aplicaciones encapsulando y mejorando los servicios del sistema operativo con interfaces robustas usando los tipos del lenguaje C++. Por definición un Wrapper Facade consiste en una o más clases que encapsulan funciones y datos dentro de un interfaz con comprobación de tipos y orientado a objetos. La utilidad de esta capa reside en el empaquetado de clases C++ de las funciones aisladas estilo C de la capa anterior, de forma que se reduce significativamente el esfuerzo de aprender y utilizar ACE correctamente.

Podemos utilizar ACE en cualquiera de sus capas, pero es en ésta donde realmente comenzamos a beneficiarnos de las ventajas del conjunto, y los beneficios continúan según avanzamos en capacidad de abstracción. Al usar esta capa, las operaciones no permitidas con los tipos de datos, se detectan en tiempo de compilación en lugar de hacerlo en tiempo de ejecución como ocurría al utilizar APIs en C.

- *Capa de Frameworks de ACE*: Los componentes de esta capa son el más alto nivel disponible como bloques para la construcción de aplicaciones. Estos componentes se basan en patrones de diseño específicos del dominio de software de comunicación distribuido. En general los frameworks de ACE facilitan el desarrollo de software de comunicación que puede ser actualizado y extendido en funcionalidad sin tener que modificar, recompilar, reenlazar e incluso sin volver a lanzar los procesos. Este alto grado de flexibilidad se consigue en ACE combinando las posibilidades que nos ofrece el lenguaje C++ como la herencia, la sobrecarga de métodos, el uso de plantillas y el enlace dinámico con la aplicación e implementación particular de patrones de diseño.

5.8.2. ¿Qué proporciona ACE en comparación con las API's de C?

A pesar de su ubicuidad, la API para el manejo de sockects en C no es portable. A continuación se enumeran algunas divergencias entre plataformas:

- *Nombres de función*: Las funciones *read()*, *write*, y *close()*, básicas para la lectura, escritura, y cierre de descriptores de sockets, no son portables para todos los sistemas operativos. Por ejemplo, Windows define un conjunto diferente de funciones (*ReadFile()*, *WriteFile()*, y *Closesocket()*) que proporcionan estos comportamientos.

- *Semántica de funciones*: Algunas funciones tienen diversos comportamientos en diferentes plataformas. Por ejemplo. en UNIX y en Win32, la función *accept()* puede recibir un puntero nulo en los campos de la dirección del cliente y la longitud de éste. En algunas plataformas en tiempo real tales como VxWorks, pasarle un puntero nulo a *accept ()* puede acabar con la aplicación.

- *Tipos de los manejadores de sockets*: Diferentes plataformas utilizan diferentes representaciones para los manejadores de sockets. Por ejemplo, en las plataformas UNIX, los manejadores

de sockets son de tipo entero, mientras que en Win32 están implementados como punteros a enteros.

- *Archivos de cabecera*: Diferentes combinaciones compilador - sistema operativo, emplean diferentes nombres para los archivos de cabecera que contienen los prototipos de las funciones de la API de sockets.

Otro problema con la API de sockets de C es que sus docenas de funciones carecen de una uniformidad en sus nombres, que hace difícil determinar el rango de actuación de la API. Por ejemplo, no es claramente obvio que las funciones *socket()*, *bind()*, *accept()*, y *connect()* pertenezcan a la misma API. Otras APIs de red palian este problema precediendo de cada función un prefijo común. Por ejemplo, en la API TLI cada función es precedida de *t_*.

5.8.3. Manejo de memoria en ACE

El *framework* ACE proporciona clases para el manejo eficiente, tanto de la memoria dinámica, como la memoria compartida entre procesos. Nos centraremos en los detalles referentes a la memoria dinámica local, es decir para un solo proceso. La clase principal que sustenta en concreto este manejo de memoria es *ACE_Allocator*, que proporciona flexibilidad y escalabilidad en la gestión de la memoria. La flexibilidad la aporta el hecho de que los objetos de esta clase pueden cambiar en tiempo de ejecución, sin embargo tal característica se implementa con funciones virtuales de C++, con lo que la resolución de indirecciones conlleva una cierta pérdida en rendimiento.

Existen distintos tipos de *Allocators*, objetos de la clase *ACE_Allocator*, según la política de manejo de memoria que se quiera seguir, aunque son similares en funcionalidad. El uso de estos mecanismos propios de *ACE* no sólo encapsula el manejo de la memoria y minimiza el uso de llamadas al sistema, sino que proporciona buen rendimiento y operaciones previsibles, algo muy deseable en sistemas de tiempo real o, en general, en sistemas con necesidades de tiempos de respuesta bajos.

Para lograr y facilitar el uso eficiente de la memoria *ACE* proporciona la clase *ACE_Message_Block*, que permite manipular mensajes: almacenarlos en buffers cuando se reciben por la red, añadir o eliminar cabeceras, reordenar secuencias de mensajes, etc. Para todas estas funciones la clase *ACE_Message_Block* usa la clase *ACE_Allocator*, evitando copias inecesarias de datos y el *overhead* del manejo de la memoria dinámica.

ACE Message Blocks

La clase *ACE_Message_Block* [Huston,2003], se encuentra implementada en base al patrón de diseño *Composite* [Gamma, 1995], y presenta las siguientes características:

- Cada objeto *ACE_Message_Block* proporciona punteros a tipos *ACE_Data_Block*, de los que se mantiene una cuenta de referencias a cada bloque de datos, permitiendo la compatición de los mismos sin necesidad de copia.
- Se proporcionan los mecanismos para encadenar mensajes en una lista compuesta, facilitando el manejo de posibles formatos con cabecera y cuerpo.
- Encapsula la sincronización y el manejo de la memoria.

La interface de la clase permite manejar dos tipos de mensajes: simples y compuestos.

- **Simple**: contiene un único *ACE_Message_Block*, figura 5.13.
- **Compuesto**: contiene múltiples *ACE_Message_Block* enlazados, figura 5.14. Esto proporciona una estructura que permite agregar elementos de forma recursiva.

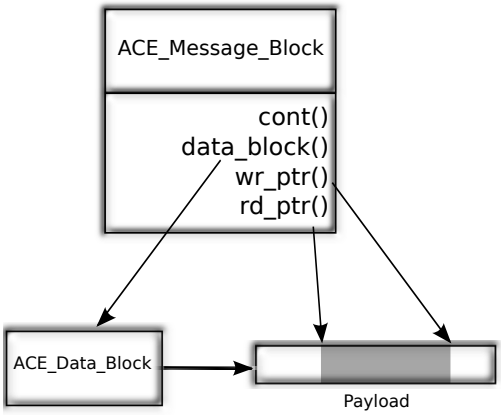

Figura 5.13: Mensaje simple.

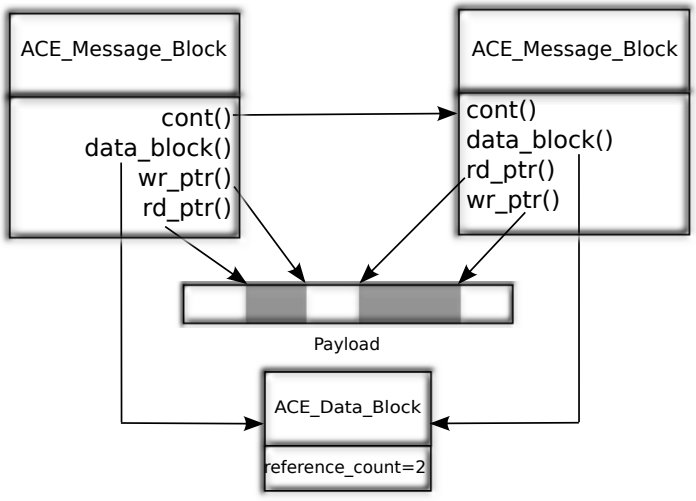

Figura 5.14: Mensaje compuesto.

5.8.4. *CDR Streams* en *ACE*

Las aplicaciones de red suelen requerir linearización para convertir datos construidos, como vectores o listas, a *streams* de bytes y viceversa. Además las operaciones de *marshalling* y *demarshalling* de datos son operaciones cruciales cuando se intercomunican máquinas con diferentes *bytes order* o distintos límites de alineamiento en memoria.

ACE proporciona estas características encapsuladas en dos clases: *ACE_OutputCDR* y *ACE_InputCDR*. Estas clases utilizan la representacion *CDR* 5.7, del estándar *CORBA* [OMG,2002], [OMG,2004]. En concreto, la clase *ACE_OutputCDR* crea un *buffer CDR* para hacer *marshalling* de datos, mientras que la clase *ACE_InputCDR* realiza el *demarshalling* los datos desde un *buffer CDR*.

Estas dos clases porporcionan operaciones para realizar el *marshalling/demarshalling* tanto de datos primitivos como de vectores de los mismos. Internamente utilizan la clase *ACE_Message_Block*, evitando copias de datos y optimizando la conversión a la representación intermedia *CDR*, utilizando incluso instrucciones de bajo nivel según la plataforma (pe: intel X86).

5.8.5. Ventajas de *ACE*

Las características descritas con anterioridad hacen de ACE un *framework* que aporta grandes ventajas al desarrollo de software de comunicaciones.

El primer punto fuerte de ACE es la portabilidad, atributo que justifica con creces la elección de este *framework* para este desarrollo. ACE está soportado por una gran cantidad, en expansión, de sistemas operativos, lo que nos permite migrar nuestras aplicaciones a distintas plataformas cuando sea necesario, sin necesidad de recompilar nuestro software. Como se comentó con anterioridad esto es muy deseable en software para sistemas robóticos, donde se involucran distintos tipos de hardware y plataformas muy variadas.

Como segunda característica en favor de ACE podemos destacar la calidad del software que se obtiene. El uso de patrones ampliamente probados, contrastados y difundidos repercute directamente en la reusabilidad y modularidad de nuestras aplicaciones.

Por último, pero no menos importante, cabe destacar la eficiencia que este *framework* aporta, punto de gran valor para aplicaciones de teleoperación, donde minimizar las latencias es un factor determinante para obtener software de calidad.

5.9. Interfaces gráficas en Java: *Swing*

Antes de comentar *Swing*, es necesario introducir dos términos, *AWT* y *JFC*.

La librería predecesora de *Swing* se denomina *Abstract Window Toolkit (AWT)*. *AWT* es la parte de Java diseñada a la creación de interfaces de usuario y pintado de gráficos e imágenes. Es un conjunto de clases orientadas a proporcionar todo lo que un desarrollador necesite para crear una interfaz gráfica para cualquier aplicación o applet Java.

Las clases de la fundación Java ó *Java Foundation Classes (JFC)* consisten en 5 partes principales: *AWT*, *Swing*, accesibilidad, Java 2D, y *Drag and Drop*[3]. Java 2D ha llegado a convertirse en una parte integrada de *AWT*, *Swing* está construida por encima de *AWT*, y el soporte de accesibilidad está incluido en *Swing*. Las cinco partes de *JFC* no son curiosamente mutuamente exclusivas, y se espera que *Swing* se integre más profundamente con *AWT* en futuras versiones de Java. Además, *AWT* es uno de los núcleos de *JFC*, conviertiéndolo en una de las librerías más importantes en Java 2.

Swing [Robinson, 2003] consiste en un amplio conjunto de componentes que abarca desde los más simples, tales como etiquetas, a los más complejos, como tablas o árboles. Casi todo componente *Swing* deriva de una sola clase padre llamada *JComponent* que hereda de la clase *AWT Container*. Por esta razón, *Swing* se describe mejor como una capa por encima de *AWT* más que una sustituta de ésta.

5.9.1. Independencia de la plataforma

La característica más destacable de los componentes *Swing* es que están escritos en código 100 % Java y no dependen de componentes escritos en código nativo, como ocurre con la mayoría de componentes *AWT*. Esto significa que un botón *Swing* o un área de texto puede parecerse y funcionar identicamente en Macintosh, Solaris, Linux y plataformas Windows. Este diseño reduce la necesidad de testear y depurar las aplicaciones para cada plataforma.

5.9.2. *Modelo Vista Controlador (MVC)*

El diseño de los componentes *Swing* se ha desarrollado siguiendo el patrón *modelo-vista-controlador (MVC)*. La arquitectura *MVC* descompone a las aplicaciones en tres partes:

[3] *Drag and Drop* alude a una característica propia de las interfaces gráficas actuales en la que se permite a un usuario interactuar con la misma arrastrando (*drag*) y soltando (*drop*) componentes

- Un *modelo* que representa los datos para la aplicación.
- La *vista* que consiste en la representación gráfica de los datos.
- Un *controlador* que captura datos de usuario a partir de la *vista* y los traduce en cambios de estado en el *modelo*.

Una de las mayores ventajas que la arquitectura *MVC* proporciona a *Swing* es la capacidad de configurar el *Look & Feel* de un componente sin modificar el modelo.

5.9.3. Ventajas de usar *Swing* frente a *AWT*

- El diseño 100 % Java posee menos limitaciones de plataforma.
- Proporciona una sola*API* capaz de soportar múltiples *Look & Feel* asi que los desarrolladores y usuarios finales no tienen que limitarse a un *Look & Feel* único. El aspecto de las interfaces puede personalizarse o utilizarse alguno de los que vienen por defecto (Metal Mac, Basic Motif, Window Win32).
- Soporte de accesibilidad: Se facilita la generación de interfaces que ayuden a la accesibilidad de discapacitados.
- Hoy en día el desarrollo de componentes *Swing* es más activo en comparación con *AWT*.

5.10. Encapsulado de código nativo, *Swig*

Swig es un *wrapper*[4] de código o compilador de interfaces que conecta programas escritos en C y C++ con lenguajes de *scripting* tales como Perl, Python, Ruby y Tcl, etc. Funciona encontrando declaraciones en archivos de cabecera C/C++ y utilizándolos para generar código encapsulado que los lenguajes de *scripting* necesitan para acceder al código C/C++ que se encuentra debajo. Además, *Swig* proporciona una variedad de características configurables que ayudan a confeccionar los procesos de encapsulados más adecuados para cada aplicación.

5.10.1. Lenguajes soportados

Swig actualmente genera código encapsulado para 18 lenguajes diferentes:

[4]Los *wrappers* o *envoltorios* son un mecanismo que proporcionan una interfaz para el uso de funciones de bajo nivel a partir de un lenguaje de más alto nivel o con mayor nivel de abstracción.

- Allegro CL
- C#
- CFFI
- CLISP
- Guile
- Java
- Lua
- Modula-3
- Mzscheme
- OCAML
- Octave
- Perl
- PHP
- Python
- R
- Ruby
- Tcl
- UFFI

  Además de esto, el árbol de análisis sintático (*parser*) puede ser exportado como XML y expresiones Lisp.

5.10.2. ANSI C

Swig es capaz de encapsular todo el estándar ANSI C. Estas características incluyen:

- Manejo de todos los tipos de datos ANSI C.
- Funciones globales, variables globales y constantes.

- *Structs* y *Unions*.
- Punteros.
- *Arrays* de cualquier dimensión.
- Punteros a funciones.
- Argumentos de longitud variables.
- *Typedef*.
- *Enums*.

5.10.3. ANSI C++

Swig proporciona soporte para encapsular la mayoría de las funcionalidades definidas en ANSI C++:

- Todos los tipos de datos C++.
- Referencias.
- Punteros a miembros.
- Clases.
- Herencia y herencia múltiple.
- Sobrecarga de funciones.
- Sobrecarga de operadores.
- Miembros estáticos.
- *Namespaces*.
- Templates miembros.
- Especialización de templates.
- Punteros inteligentes.
- Soporte de librería C++ para *strings* y la *STL*.

5.10.4. Preprocesador

Swig proporciona un preprocesador completo de C con las siguientes características:

- Expansión de macros.
- Automático encapsulado de sentencias *#define* como constantes (cuando sea aplicable).
- Soporte para C99.

5.10.5. Opciones configurables

Swig proporciona control sobre la mayoría de los aspectos de generación de *wrappers*. La mayoría de estas opciones configurables están completamente integradas en sistema de tipos C++ haciendo fácil aplicar configuraciones a través de jerarquías de herencias, instanciaciones de templates, y más. Estas características incluyen:

- Conversiones de tipos (*marshalling)* personalizables.
- Manejo de excepciones.
- Extensión de clases o *structs*.
- Manejo de memoria.
- Resolución de ambigüedades.
- Instanciación de templates.
- Importar archivos y enlazado de módulos cruzados.
- Manejo de macros *Swig* extendido.

Capítulo 6

Diseño

Este capítulo se centra en exponer las actividades y resultados de la fase de diseño de este proyecto. Se incluye el diseño de las operaciones de marshalling de datos que se enviarán por la red. Posteriormente se muestra en detalle la especificación del protocolo de comunicaciones diseñado en respuesta a los requerimientos de comunicaciones entre componentes. Seguidamente se presentan los diagramas de clases y patrones de diseño utilizados para la implementación realizada de dicho protocolo y su integración en *CoolBOT*. Finalmente este capítulo describe la estructura interna del nuevo modelo de componente distribuido de *CoolBOT* y la estructura genérica de una *vista* de componente.

6.1. Operaciones de *Marshalling* de datos

Las operaciones de *marshalling* y *demarshalling* de datos al formato definido por *CDR* se encuentran encapsuladas, de forma que el usuario desarrollador (tanto de *CoolBOT* como de sistemas robóticos) sólo accede a una interfaz simple y común para cualquier tipo de dato (ya sea primitivo o construido). Esta decisión de diseño se fundamenta en facilitar el manejo y ocultar al usuario detalles de bajo nivel. De esta forma el programador de componentes *CoolBOT* sólo necesita conocer una interfaz sencilla de *marshalling/demarshalling* de los datos que decida enviar por la red, abstrayéndose totalmente del uso concreto de funciones de comunicaciones de la librería *ACE* y de particularidades relativas a los tipos de datos y la representación *CDR* (alineamientos, tamaños de datos, tamaños de buffers, etc).

Otra de las ventajas de la interfaz de serialización de datos suministrada al usuario es su fácil aplicación a tipos de datos definidos como clases *template*. Las particularidades de generalizar una clase con este recurso del lenguaje C++ y que ciertos tipos de atributos no se concreten hasta una instanciación, supone que las operaciones de *marshalling/demarshalling* no puedan

definirse hasta no conocer los tipos de los datos. Es por esto que la interfaz suministrada se ha implementado utilizado también funciones *template*, de forma que, las operaciones concretas para los tipos generados tras la instanciación de las clases *template* que el usuario pudiera crear, se instancian a la vez que dichas clases. La implementación de la interfaz se describe en detalle en el apéndice 8.2.

Con estas funciones suministradas como herramienta básica para realizar el *marshalling/demarshalling* de datos, el usuario es capaz de convertir cualquier tipo de dato a la representación *CDR* para su envío. Esto es posible ya que, además de las funciones para datos básicos, se proporciona al usuario una interfaz para definir sus propias funciones de *marshalling/demarshalling*. Esta interfaz no es más que una clase abstracta que define una serie de operaciones (para más detalle ver *PackingInterface*, apéndice 8.1.3. Como norma de diseño estas operaciones deben estar definidas para toda estructura de datos que el usuario pretenda enviar a un componente remoto. De cara al usuario esto se traduce en que toda clase que se pretenda enviar por la red debe heredar de la interfaz de empaquetado (*PackingInterface*).

6.2. Diseño del protocolo DC3P

6.2.1. Introducción al diseño de protocolos

El diseño de un protocolo es un problema tan antiguo como las comunicaciones en sí mismas. Solamente cuando la interpretación de las reglas de protocolo fueron automatizadas mediante máquinas de alta velocidad fue descubierto que el diseño del protocolo en sí mismo puede ser un gran desafío. Los protocolos que están siendo desarrollados hoy en día son más amplios y sofisticados que sus predecesores. Intentan ofrecer una mayor funcionalidad y fiabilidad, pero como resultado, han visto incrementados su tamaño y complejidad. El problema al que un diseñador se enfrenta es fundamental: cómo diseñar un amplio conjunto de reglas para intercambiar información que sean las mínimas, lógicamente consistentes, completas, e implementables eficientemente.

6.3. Especificación del protocolo *DC3P*

En esta sección se presenta la especificación del protocolo de comunicaciones *DC3P (Distributed CoolBOT Components Communication Protocol)* diseñado para soportar el servicio de paso de datos entre componentes del *framework CoolBOT*. La especificación que aquí se expone constará de cinco puntos básicos [Holzmann,1991]:

1. Descripción del servicio proporcionado.

2. Descripción del entorno de ejecución.

3. Tipos de mensajes y vocabulario especifíco.

4. Formato de los mensajes.

5. Reglas de procedimiento y consistencia en el intercambio de mensajes.

Se seguirá una linea descriptiva, iniciándose con un nivel de abstracción alto y se procederá introduciendo mayor nivel de detalle y formalismo, obteniéndose así una especificación completa y consistente del servicio de comunicaciones entre componentes y del protocolo *DC3P* que lo sustenta [Sunshine,1979].

6.3.1. Descripción general del servicio

El servicio de comunicaciones entre componentes permite el trasiego de los datos que pasan de unos componentes a otros durante la ejecución de un sistema robótico concreto. Este servicio proporciona al usuario una abstracción, de forma que sea transparente el hecho de que un componente se comunique con otros de forma local o remota. El usuario de *CoolBOT* (desarrollador de sistemas robóticos) sólo debe tener presente las colaboraciones entre componentes, para determinar así el conexionado de los puertos de entrada y salida.

La información intercambiada entre componentes fluye desde un puerto de salida del componente emisor de los datos hasta un puerto de entrada de un componente receptor de los mismos. Por tanto, el servicio *DC3P* proporciona transferencia de mensajes en un solo sentido, desde un puerto de salida hasta un puerto de entrada. *DC3P* procura mecanismos para comprobar las compatibilidades de conexionado entre puertos de componentes descritas en la sección 3.5, así como para testear si un componente ha quedado bloqueado durante su ejecución.

Todo esto es proporcionado de forma transparente para el usuario de sistemas robóticos, de forma que éste sólo necesita aportar las *IPs* y puertos *TCP* de escucha de los componentes que pretende interconectar, actuando la capa del servicio *DC3P* como una caja negra, figura 6.1

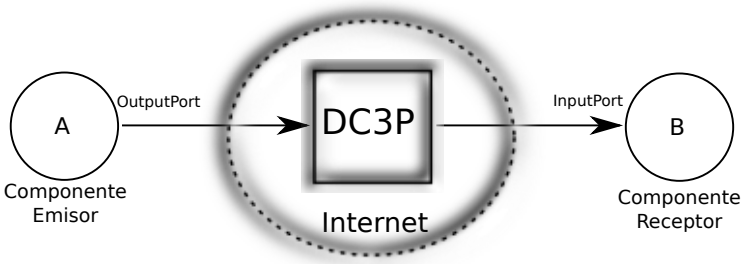

Figura 6.1: Visión a alto nivel del servicio DC3P

Con mayor concreción, las funcionalidades proporcionadas al usuario por el servicio de comunicación *DC3P* son las citadas a continuación:

- Conexionado/desconexionado de puertos de componentes.

  Operaciones para permitir la realización de conexiones de puertos de salida con puertos de entrada de componentes. Existen mecanismos tanto para que desde un componente local conecte sus puertos con los de otro componente remoto, así como la posibilidad de que un tercero indique a dos componentes remotos que se conecten entre sí.

- Control de compatibilidad conexiones de componentes.

  Mecanismos de control para la compatibilidad de conexiones entre puertos de componente.

- Envío y recepción de paquetes de datos.

  Se proporcionan los mecanismos para que un componente que genera ciertos datos los envíe hacia otro componente remoto y éste los reciba adecuadamente.

- Control de bloqueo de componentes.

  Mecanismos de control para la detección de componentes que hayan finalizado su ejecución inesperadamente o se encuentren bloqueados.

6.3.2. Descripción del entorno de ejecución

Las entidades que participan en el intercambio de mensajes son los componentes *CoolBOT*, que pueden actuar con dos roles distintos en la comunicación según ejerzan como *emisor* de datos o *receptor* de datos. Se define como *entidad emisora* aquel componente que genera ciertos datos como salida y que son emitidos a través de uno de sus puertos de salida (*Output Port*) hacia un puerto de entrada (*Input Port*) de otro componente. De forma opuesta, la *entidad receptora*

se define como aquel componente que usa determinados datos que llegan a través de uno de sus puertos de entrada (*Input Port*).

El interconexionado de componentes en *CoolBOT* pasa a convertirse en una o más conexiones a través de la red *TCP/IP*, usando el servicio proporcionado por la capa de transporte. Se crearán conexiones *TCP*, concretamente una por cada par de componentes conectados y por sentido de la emisión de datos. Por tanto, cada conexión TCP creada se encargará del intercambio de datos entre una serie de pares de puertos *CoolBOT* de componente (*OutputPort-InputPort*). La figura 6.2 ilustra la creación de diferentes conexiones entre componentes remotos. Como podemos ver los componentes se encuentran distribuidos y realizan diferentes conexionados de puertos. Estos conexionados de puertos *CoolBOT* se mapean en la misma conexión TCP siempre que se dirijan en el mismo sentido de la comunicación entre pares de componentes remotos.

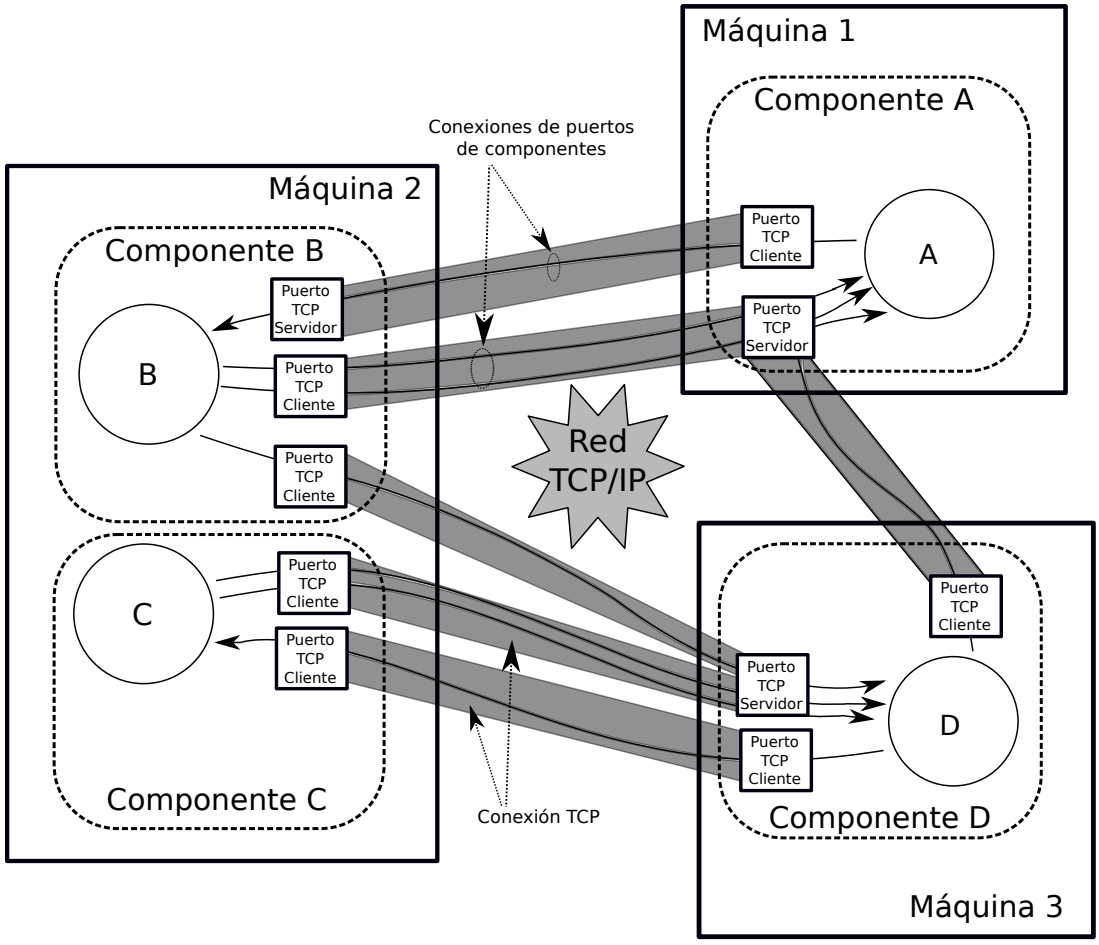

Figura 6.2: Conexiones TCP entre componentes distribuidos

El componente que inicia la conexión con otro, es decir el que juega un papel activo en el inicio de la comunicación, es el que actúa como emisor de los datos. Por tanto, el *emisor* debe conocer

a priori la dirección *IP* y puerto *TCP* de escucha del componente *receptor*, de forma que pueda efectuar una solicitud de conexión con el mismo. Cuando un componente *receptor* recibe alguna solicitud de conexión obtendrá la dirección *IP* y puerto *TCP* que haya enviado el *emisor* en la solicitud.

Una vez establecidas las conexiones que fueran necesarias para la comunicación entre componentes, estos estarán dispuestos para el envío de datos generados durante la ejecución del *emisor* y usados por el componente *receptor*. Como se comentó en la sección 6.3.1, a pesar de que el flujo de datos entre componentes *CoolBOT* es de un solo sentido, de *emisor* a *receptor*, ciertas reglas de procedimiento del protocolo (descritas en detalle en la sección 6.3.5), contemplan el intercambio de mensajes de control del protocolo en ambos sentidos, para así proporcionar correctamente todas las funcionalidades descritas para el servicio.

6.3.3. Vocabulario y tipos de mensajes

El protocolo *DC3P* proporciona conjuntos de mensajes cada uno de ellos orientados a suministrar una funcionalidad concreta de las descritas en la sección 6.3.1.

El vocabulario del protocolo (figura 6.3) define seis grupos distintos de mensajes: *Echo, PortInfo, Data, Connect/Disconnect, RemoteConnect/RemoteDisconnect, Ack/Reject*. Para los mensajes del tipo *Echo y PortInfo*, existen las variantes *Request* y *Response*, en caso de ser una solicitud o una respuesta, respectivamente. Para que el protocolo de soporte a los posibles conexionados de puertos de componentes que un usuario requiere, los grupos de mensajes *Connect/Disconnect* y *RemoteConnect/RemoteDisconnect* ofrecen las variantes *Single-Single, Single-Multi, Multi-Single y Multi-Multi*, donde la palabra antes del guión se refiere al tipo de puerto de salida y la posterior al tipo de puerto de entrada. Los paquetes *PortInfo* se utilizan para pedir información sobre un puerto de entrada, por tanto, existen los tipos *Single* y *Multi* en función del tipo de puerto del que se solicita o recibe información. Los paquetes del protocolo utilizados para enviar paquetes de puertos de componente *Data* ofrecen también las posibilidades *Single* y *Multi* dependiendo del tipo de puerto de entrada al que se dirige el paquete de puerto.

6.3.4. Formato de los mensajes

Los mensajes del protocolo *DC3P* han sido diseñados de manera que transmitan la mayor cantidad de información útil, es decir, que minimicen los datos de relleno o *padding*.

El formato de los mensajes del protocolo *DC3P* es de longitud variable, aunque todas las variantes de mensajes comparten una cabecera común, que es de una longitud fija de 8 bytes.

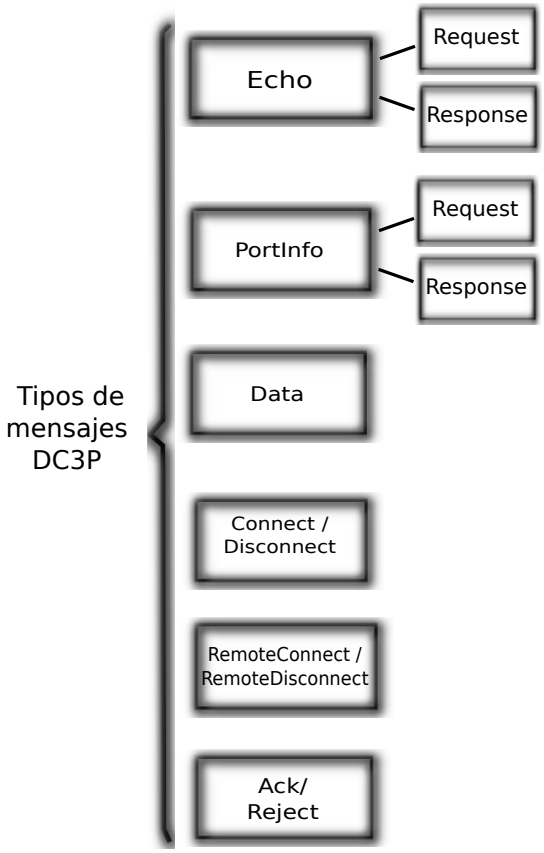

Figura 6.3: Tipos de mensajes *DC3P*

La cabecera, mostrada en la figura 6.4, está formada por dos campos: *ByteOrder* y *CommandType*. El primer byte indica el *byte order* de la máquina donde se encuentra el componente emisor del mensaje. Este valor es 1 ó 0 para *little-endian* o *big-endian* respectivamente. El segundo byte de la cabecera indica el tipo de mensaje del protocolo que viene a continuación. Este campo permite identificar cómo interpretar adecuadamente el mensaje.

Los valores que puede tomar el campo *CommandType* para los distintos tipos de mensajes del protocolo son los indicados en la tabla 6.1.

Los mensajes del protocolo tienen una longitud variable y van siempre precedidos de la cabecera anteriormente descrita. A continuación se detallan los formatos para cada uno de los tipos de mensajes.

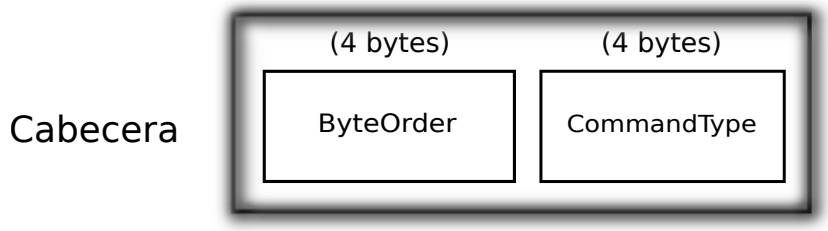

Figura 6.4: Formato de la cabecera *DC3P*

Tipo Mensaje		**CommandType**
Connect	Single-Single	0
	Single-Multi	1
	Multi-Single	2
	Multi-Multi	3
RemoteConnect	Single-Single	4
	Single-Multi	5
	Multi-Single	6
	Multi-Multi	7
Disconnect	Single-Single	8
	Single-Multi	9
	Multi-Single	10
	Multi-Multi	11
RemoteDisconnect	Single-Single	12
	Single-Multi	13
	Multi-Single	14
	Multi-Multi	15

Tipo Mensaje		**CommandType**
PortInfo	Request Single	16
	Request Multi	17
	Response Single	18
	Response Multi	19
Data	Single	20
	Multi	21
Echo	Request	23
	Response	24
Ack		25
Reject		26

Cuadro 6.1: Valores del campo CommandType.

Connect/Disconnect

Como se indica en la sección de puertos y componentes (3.6), en *CoolBOT* existen dos grupos de tipos de puertos de componentes: aquellos con soporte para un único tipo de paquetes de puerto (*SinglePacket*) y aquellos que soportan un conjunto de tipos de paquetes (*MultiPacket*).

Por tanto, para el caso de las solicitudes de conexión/desconexión existen cuatro tipos de mensajes en cada caso, en función de la conexión que se pretenda realizar al combinar los diferentes tipos de puertos. Las principales diferencias entre estos mensajes radican en la identificación del puerto de paquete. En caso de tratarse de una conexión que involucre a algún puerto *SinglePacket* bastará con indicar el identificador del puerto con un valor entero. En caso de conexiones/desconexiones que involucren a un puerto *MultiPacket*, además de el identificador anterior habrá que aportar un identificador que concreta el *Slot* para ese puerto determinado.

Como se detalla a continuación, no existen diferencias en los formatos de *Connect* con respecto al *Disconnect*. Las figuras 6.5, 6.6, 6.7, 6.8 ilustran los diferentes formatos de mensajes connect y disconnect.

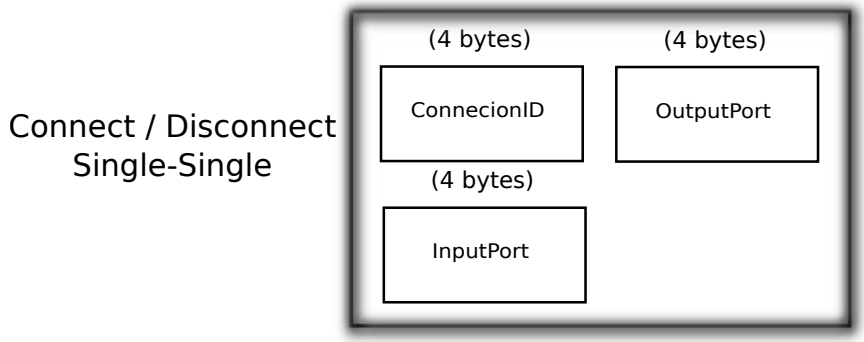

Figura 6.5: Formato de mensajes *Connect/Disconnect Single-Single*.

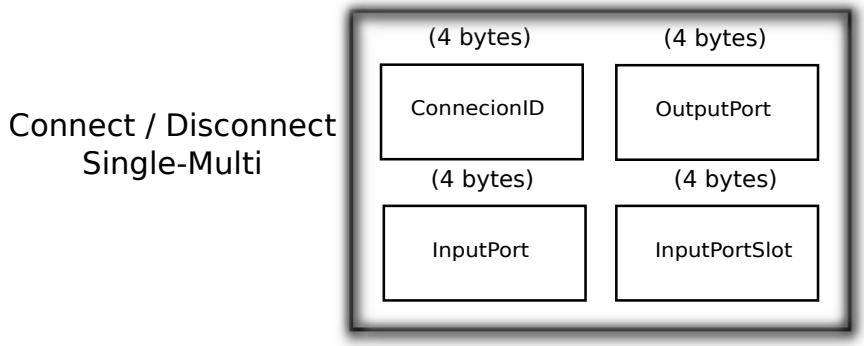

Figura 6.6: Formato de mensajes *Connect/Disconnect Single-Multi*.

RemoteConnect/RemoteDisconnect

Se trata de un conjunto de mensajes destinados a indicar a un componente remoto que inicie la conexión con otro componente. Existen las mismas variantes que para los formatos de paquetes *Connect* y *Disconnect*.

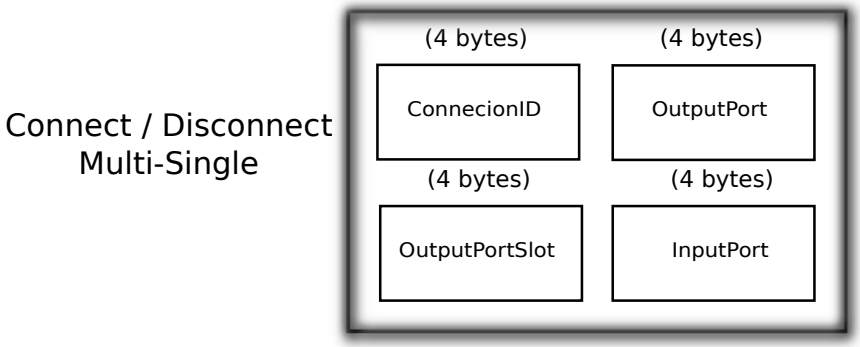

Figura 6.7: Formato de mensajes *Connect/Disconnect Multi-Single*.

PortInfo

Este tipo de mensaje está destinado a la obtención de información referente a un puerto de entrada de un componente *CoolBOT*.

Según los dos grupos de puertos soportados en *CoolBOT* (*SinglePacket* o *MultiPacket*), el formato de solicitud de información sobre un puerto determinado y la respuesta asociada varía. La diferencia radica, de nuevo, en el campo identificativo del puerto en cuestión.

SinglePacket

En el caso de puertos *SinglePacket* los formatos de mensajes son los indicados en la figura 6.13, para el caso de un *Request* y en la figura 6.14, para el caso de un *Response*.

En concreto, un mensaje *PortInfo Request Single*, se especifica, con un campo de 4 bytes, un valor entero que identifica al puerto de componente para el que se solicita información.

Para el mensaje *PortInfo Response Single*, se añade al *Request* una ristra de respuesta conteniendo la información solicitada. El tamaño de esta ristra es variable según la información respectiva de cada puerto de componente, con lo que ésta va precedida por el campo *NameLength* de 4 bytes, que indica la longitud en bytes del campo *NameString*.

MultiPacket

Para puertos *Multipacket* los formatos de mensajes son los indicados en la figura 6.15, para el caso de un *Request* y en la figura 6.16, en caso de un *Response*.

Los mensajes *PortInfo Request/Response Multi* contienen, además de un campo *InputPort*, similar al de los paquetes *PortInfo Request/Response Single*, un campo adicional de 4 bytes, que concreta el identificador del *Slot*.

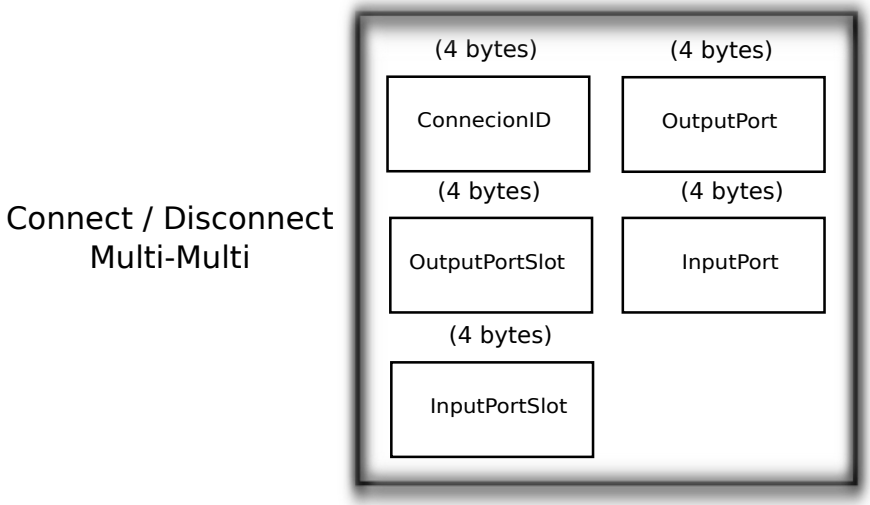

Figura 6.8: Formato de mensajes *Connect/Disconnect Multi-Multi*.

La respuesta a en este tipo de mensajes es similar a la de los paquetes *PortInfo Request Single*, se trata de una ristra, *NameString*, con la información solicitada, precedida de un campo *NameLength*, que indica su longitud.

Echo

Se trata de un tipo de mensaje destinado a verificar si un componente se encuentra en línea y activo. Tiene dos variantes: *Request* y *Response*. La particularidad de este tipo de mensaje es que consiste exclusivamente en un envío de la cabecera (figura 6.4), sin ser añadido ningún campo adicional. La figura 6.17 detalla el formato de este tipo de mensaje.

Data

Este formato de mensaje esta destinado al envío de datos (paquetes de puerto) que se intercambian entre componentes *CoolBOT*. De nuevo, en función del tipo de puerto de entrada del componente remoto, existe un formato adecuado a cada tipo.

SinglePacket

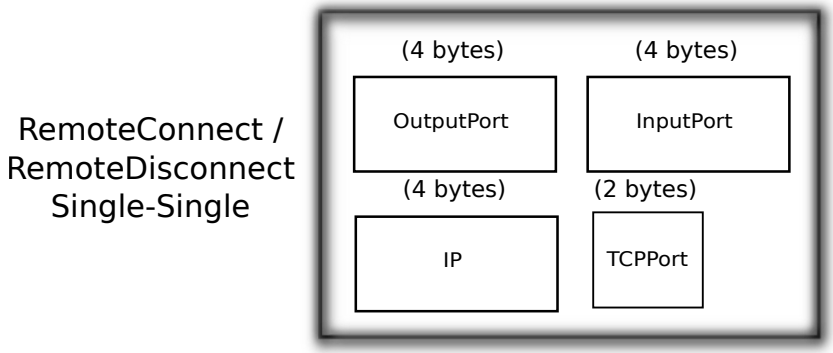

Figura 6.9: Formato de mensajes *RemoteConnect/RemoteDisconnect Single-Single*.

El formato para el envío de datos en el caso de puertos de entrada *SinglePacket* se refleja en la figura 6.18.

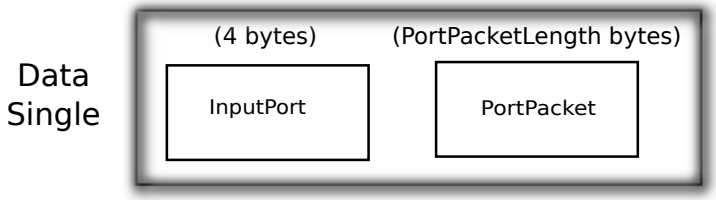

Figura 6.18: Formato de mensajes de *Data Single*

MultiPacket En caso de envío de datos hacia puertos de entrada de tipo *MultiPacket* el formato es el representado en la figura 6.20.

Ack/Reject

Son dos paquetes destinados a indicar confirmación o rechazo, respectivamente. Tienen una estructura igual a los paquetes *Echo*, es decir consisten exclusivamente en un envío de la cabecera.

6.3.5. Reglas de procedimiento

El control del flujo es una técnica para asegurar que la entidad de transmisión no sobrecargue a la entidad receptora con una excesiva cantidad de datos. La entidad receptora reserva generalmente

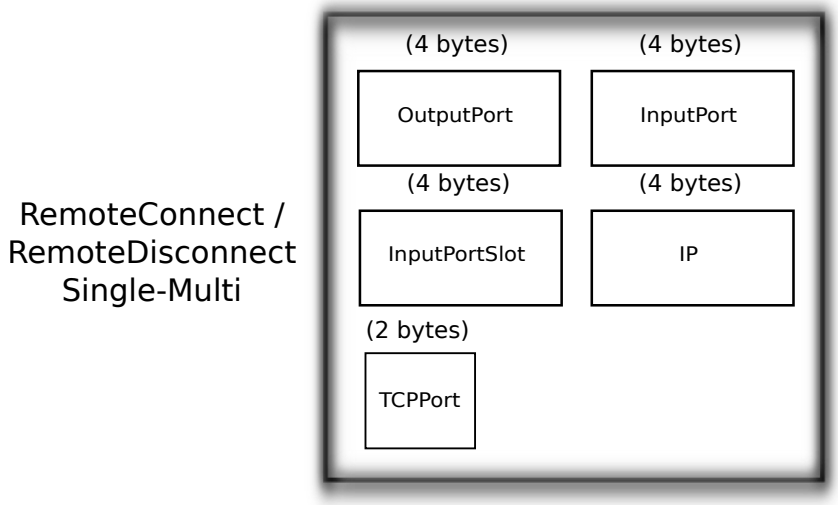

Figura 6.10: Formato de mensajes *RemoteConnect/RemoteDisconnect Single-Multi*.

una zona de memoria temporal para la transferencia. Cuando se reciben los datos, el receptor debe realizar cierta cantidad de procesamiento antes de pasar los datos al software de los niveles superiores. Si no hubiera procedimientos para el control del flujo, la memoria temporal del receptor se podría llenar y potencialmente desbordarse mientras se estuvieran procesando datos anteriores.

El control de flujo se representa mediante **reglas de procedimiento**. Estas reglas describen a lo largo del tiempo las tramas de datos enviadas entre dos entidades (origen y destino) necesarias para llevar a cabo la comunicación entre ambas (ver figura 6.21).

Las técnicas de control de flujo y control de errores suelen introducir una complejidad añadida en el diseño de los protocolos de comunicación debido a que deben garantizar la llegada correcta y en orden de las tramas enviadas, sin recepción de duplicados, y garantizando la ausencia de errores en las mismas, produciendo una serie de asperezas que hay que limar para abordar un diseño eficiente del protocolo en cuestión.

En este caso, todos estos desafíos ya se encuentran contemplados en el protocolo de la capa de transporte TCP/IP, así que, el protocolo DC3P debe preocuparse exclusivamente de la secuencia de paquetes intercambiados entre origen y destino, garantizando que sea la correcta.

A continuación se describen las reglas de procedimiento del protocolo *DC3P* para cada una de las posibles situaciones en el intercambio de información entre componentes remotos.

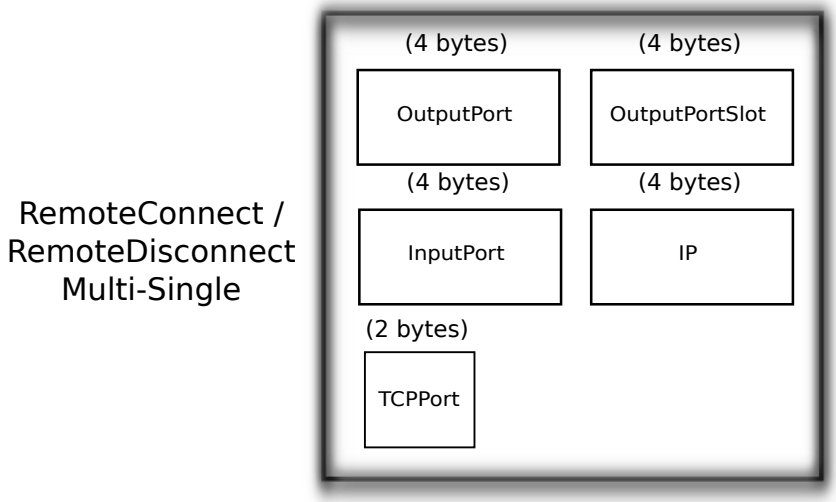

Figura 6.11: Formato de mensajes *RemoteConnect/RemoteDisconnect Multi-Single*.

- **Conexión.**

  Cuando un determinado componente desea iniciar una comunicación desde uno de sus puertos de salida a un puerto de entrada de otro componente debe establecer una conexión. En el establecimiento de conexión, siempre iniciado por el componente emisor de datos, intervienen tres tipos de paquetes del protocolo: *PortInfo* (variantes *Single* o *Multi*), *Connect* y *Ack/Reject*. La figura 6.22 presenta un diagrama de secuencia del establecimiento de una conexión entre dos puertos genéricos de los componentes A y B. Los paquetes *PortInfo*, tanto *Request* como *Response*; y *Connect* se usarán en sus variantes *Single* o *Multi*, según el tipo de los puertos que se desea conectar.

  El componente *B* puede enviar una respuesta confirmando la conexión (*Ack*) y por tanto indicando que se encuentra a la espera de la llegada de paquetes de puerto (datos) por el puerto de entrada que haya sido conectado; o bien puede rechazar la conexión indicándolo al componente *A* con el envío de una respuesta negativa (*Reject*).

- **Desconexión.**

  La desconexión de puertos al igual que la conexión es iniciada siempre por el componente emisor. Como se observa en la imagen 6.23 el componente *A* inicia la desconexión enviando el paquete *Disconnect* específico en función de los tipos de puertos a desconectar (*SinglePacket* o *MultiPacket*), indicando su puerto de salida y el puerto de entrada del componente *B* que desea desconectar. El componente *B* confirma la desconexión respondiendo con un *Ack*. El componente al que se solicita una desconexión está obligado a realizarla siempre, sin embargo, debe notificar la realización de la misma al otro extremo. Si el componente *A*

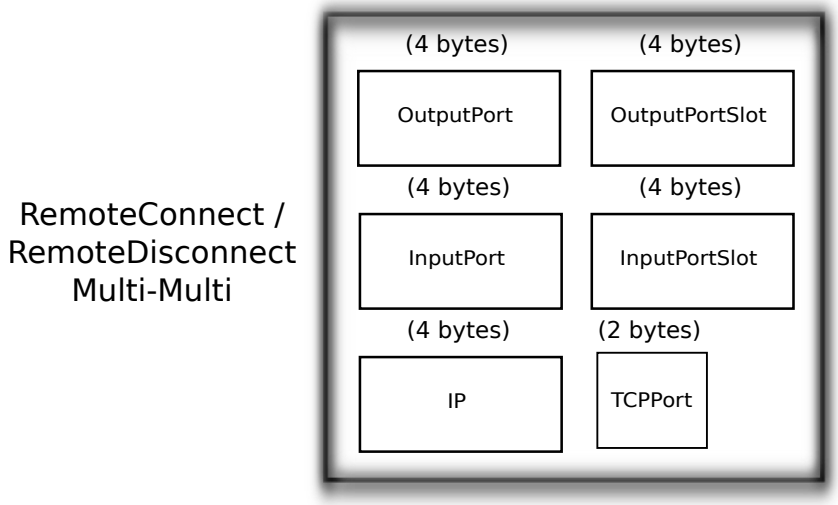

Figura 6.12: Formato de mensajes *RemoteConnect/RemoteDisconnect Multi-Multi.*

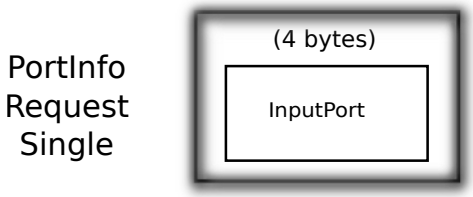

Figura 6.13: Formato de mensajes *SinglePacket PortInfo Request*

pretendiera desconectar dos puertos entre los que no existe conexión, el paquete *Disconnect* no tendrá ningún efecto en el componente *B*, que responderá siempre con una confirmación.

- **Conexión Remota.**

  El protocolo ofrece la posibilidad de que un extremo local indique a otro extremo remoto que inicie una secuencia de conexión con un tercero. El diagrama de la figura 6.24 representa el intercambio de paquetes *DC3P* para una conexión remota de componentes. El extremo local puede ser cualquier proceso, tanto un componente como una interfaz de teleoperación. Este extremo utiliza un paquete *RemoteConnect* que envía al componente *B*, indicándole un puerto de salida de *B* y un puerto de entrada de un componente remoto C, así como la *IP* y el puerto *TCP* de escucha de *C*. El componente *B* iniciará una secuencia de conexión con *C* cuyo resultado (*Ack* o *Reject*) retransmite al extremo local al finalizar la secuencia de conexionado.

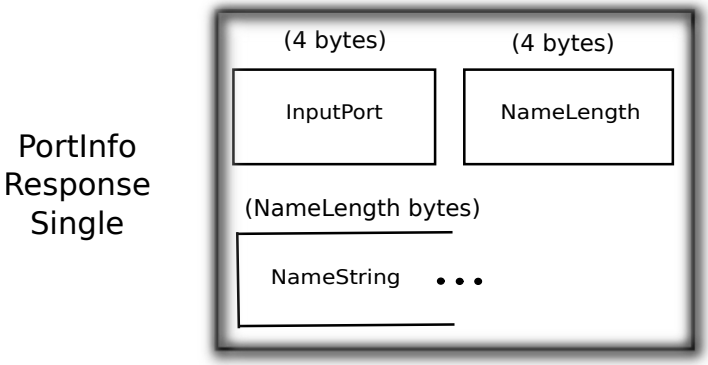

Figura 6.14: Formato de mensajes *SinglePacket PortInfo Response*

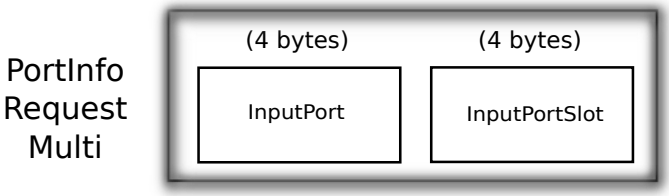

Figura 6.15: Formato de mensajes *PortInfo Request Multi*

- **Desconexión Remota.**

  Al igual que la secuencia de conexión remota, existe una operación de desconexión remota mediante la cual, como ilustra el diagrama de la figura 6.25, un extremo local A indica a un componente remoto B que inicie una secuencia de desconexión de un tercer componente C. Como respuesta a esta acción el extremo local puede recibir de B un *Ack* que confirma que la desconexión se ha realizado con éxito o un *Reject*, indicando que hubo algún tipo de problema para realizar la desconexión, como por ejemplo que el paquete *Ack* de C a B no se haya recibido.

- **Envío de datos.**

  Los envíos de paquetes de puertos siempre tienen lugar desde un puerto de salida de un componente hacia un puerto de entrada de otro componente. La secuencia de envío de datos *DC3P* utiliza paquetes *Data*, en sus versiones *Single* o *Multi* según el tipo de puerto de entrada al que se envía. La figura 6.26 refleja el envío de varios paquetes de puertos, cada uno encapsulado en el paquete *Data* correspondiente.

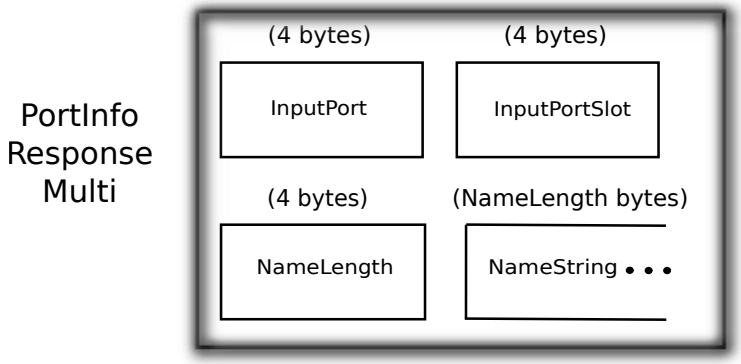

Figura 6.16: Formato de mensajes *PortInfo Response Multi*

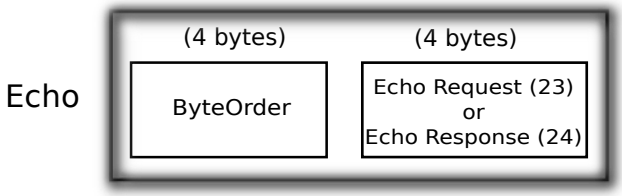

Figura 6.17: Formato de mensajes de *Echo*

6.4. Componentes *Coolbot* con soporte de red

Una vez disponible un protocolo que recoje los requisitos de comunicación entre componentes *CoolBOT*, el siguiente paso para lograr un modelo de componentes distribuido es integrar la infraestructura necesaria para que un componente haga uso de dicho protocolo. Una vez logrado dicho hito se dispondrá de un modelo de componente que permite comunicarse a través de una red TCP/IP. A continuación se presentan los detalles referentes al modelo actual de componentes, para a posteriormente reflejar los cambios de diseño introducidos de forma que soporten las comunicaciones vía red.

6.4.1. Modelo de componente sin red

Como se describe en la sección 3, los componentes *CoolBOT* funcionan como máquinas de flujo de datos. Siguiendo este modelo un componente se encuentra inactivo, procesando únicamente cuando existen datos en sus puertos de entrada, y emitiendo los resultados de su procesado por sus puertos de salida. En un componente *CoolBOT* los puertos se organizan dentro de estructuras

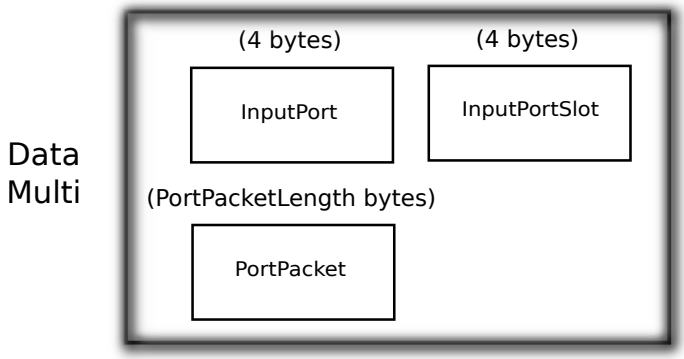

Figura 6.19: Formato de mensajes de *Data Multi*

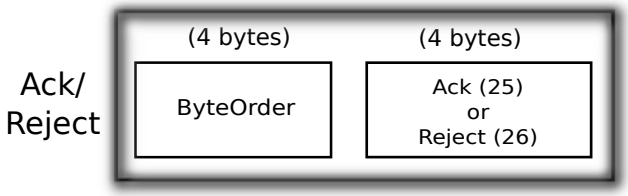

Figura 6.20: Formato de mensajes *Ack/Reject*

denominadas *Box*. En el caso de que el *Box* contenga puertos de salida se denomina *IBox* y en el caso de contener puertos de entrada se denomina *OBox*. La figura 6.27 refleja estas estructuras en concreto para un grupo de 5 puertos de salida y 5 puertos de entrada.

Los *IBox* y *OBox* son estructuras señalizables, es decir que responden ante determinados eventos que sucedan en los mismos.

Cada componente *CoolBOT* procesa la información que llega a sus puertos de entrada, para lo cual existe un hilo destinado a tal fin denominado *hilo main*. Este hilo está constantemente en estado suspendido hasta que exista algún dato en algún puerto de entrada del *IBox* del componente. Cuando a un puerto de entrada llega algún dato, este puerto queda señalizado y el hilo *main* cambia a estado activo e inicia el procesamiento. Una vez el hilo main lee los datos del puerto de entrada señalizado este se marca como atendido y los datos obtenidos como resultado del procesamiento se envían a un puerto de salida. Si no existe ningún puerto señalizado el hilo *main* vuelve al estado suspendido. De forma similar un *OBox* permite que los puertos de salida se señalicen cuando se escribe un dato en ellos, de esta forma el puerto de salida se encarga de que los datos lleguen a los puertos de entrada de otro componente a los que estuviera conectado. Además del hilo *main*, cada componente dispone de un hilo *timer*. Este hilo está destinado a realizar tareas que requieran ejecutarse cada vez que transcurra un cierto periodo de tiempo. La figura 6.28 aclara la organización

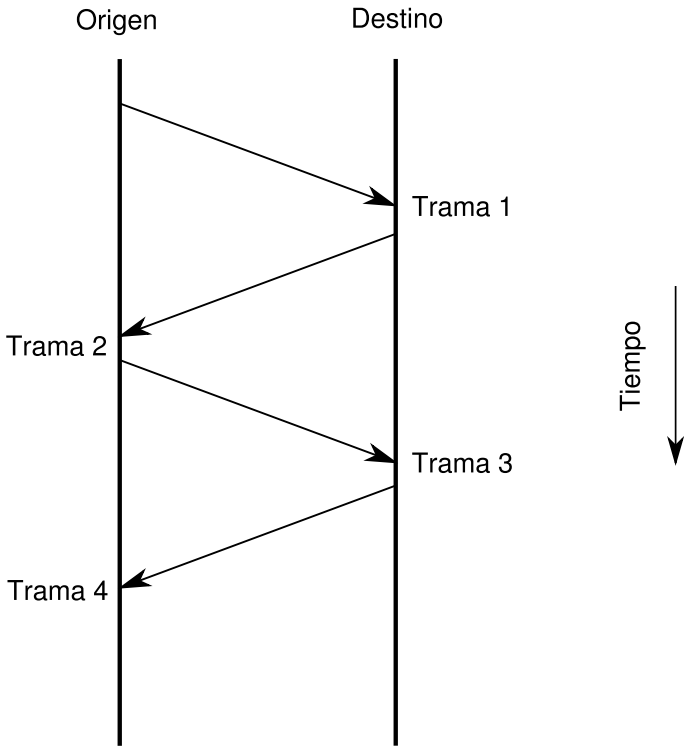

Figura 6.21: Modelo de diagrama de secuencia

interna de un componente.

6.4.2. Modelo de componente con red

El soporte de red en un componente debe ser capaz de ejecutar todas las acciones de comunicación que existen en el modelo de componente sin red. El comportamiento del componente se mantiene exactamente igual, una máquina de flujo de datos, donde el hilo *main* se active ante llegadas de datos a los puertos de entrada y emita los resultados por los puertos de salida. El soporte de red debe actuar cuando sea necesario, es decir cuando las conexiones de puertos se hayan realizado entre componentes remotos. Este modelo permite que un componente pueda realizar las conexiones a través de la red o no, según el tipo de conexionado requerido.

Cuando un componente A se conecta a otro componente B, el envío de datos se realiza en un sólo sentido, desde el emisor A hasta el receptor B. Si esta actividad se realiza a través de la red el componente B debe estar escuchando por un puerto *TCP*, con lo que actúa como servidor en la comunicación, mientras que A actúa como cliente. Un componente siempre actuará como servidor ya que debe escuchar por la red cualquier posible petición de conexión/desconexión de puertos y envíos de datos. Cuando dicho componente requiera conectarse a otro o enviar datos actuará como

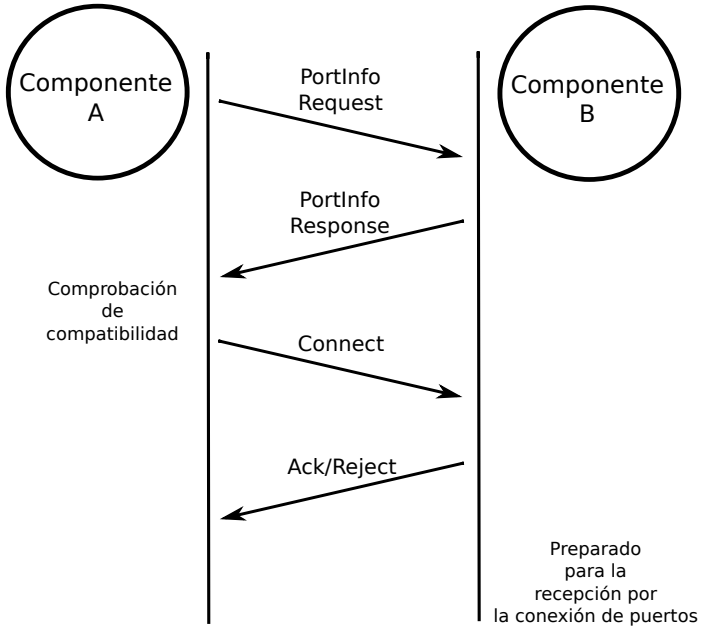

Figura 6.22: Secuencia de conexión

cliente y enviará a través de la red si la conexión del puerto de salida se hubiera realizado con un componente remoto.

Para escuchar en la red de comunicaciones y detectar la llegada de cualquier paquete del protocolo *DC3P*, un componente usa un puerto *TCP*. Tras la llegada de algún paquete del protocolo debe ser capaz de responder al mismo en base a las secuencias de mensajes *DC3P*. Además debe llevar algún tipo de control de las conexiones de puertos que se realicen a través de la red, así como establecer algún mecanismo que entregue los datos al puerto de entrada correspondiente en el *IBox*. Para añadir esta funcionalidad a los componentes *CoolBOT* se ha utilizado un nuevo hilo interno, este es el denominado *Input Network Thread (INT)*.

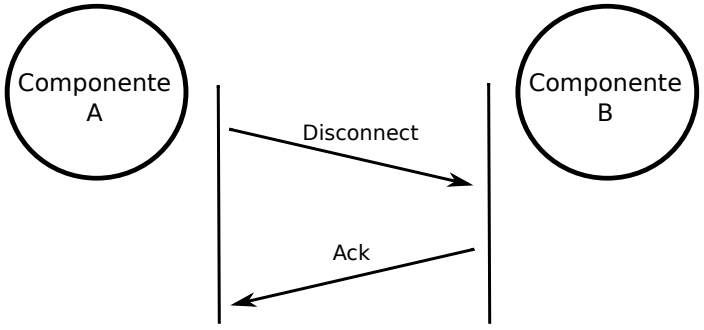

Figura 6.23: Secuencia de desconexión

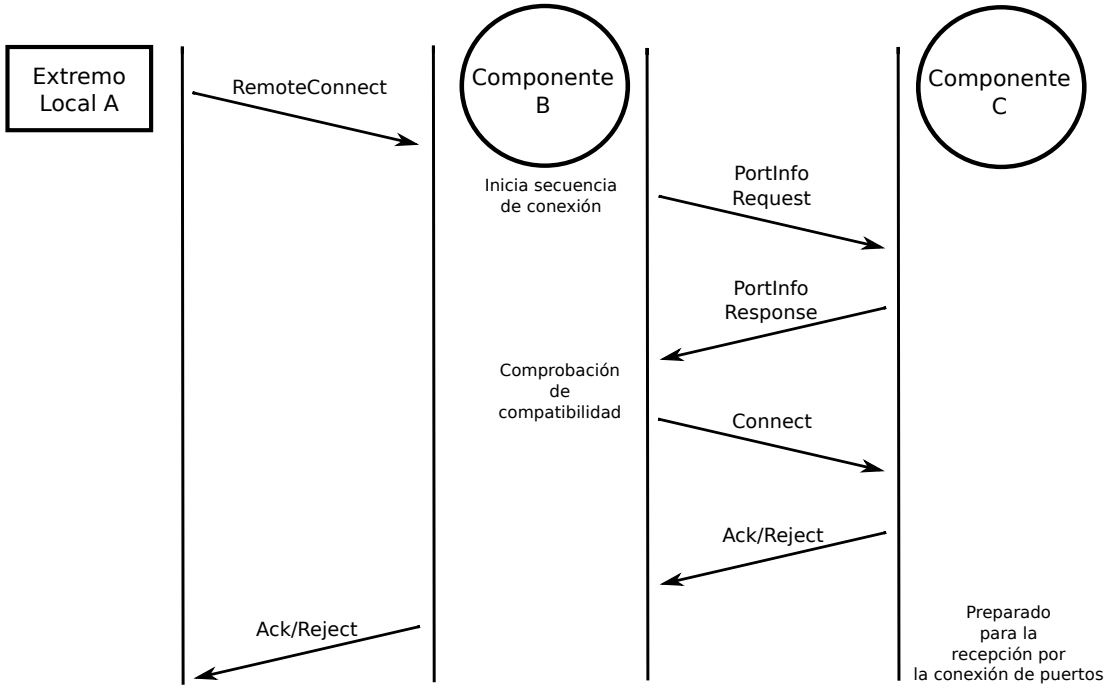

Figura 6.24: Secuencia de conexión remota

Cuando el hilo *INT* detecta una solicitud de conexión *TCP* la acepta ya la almacena en un conjunto de conexiones activas. El hilo *INT* se encuentra suspendido y se activa cada cierto tiempo chequeando si existe actividad por la red, tanto de conexiones *TCP* nuevas como alguna actividad por las conexiones activas. Este comportamiento se identifica con el bucle habitual que ejecuta un servidor en cualquier sistema distribuido. Además de gestionar las conexiones *TCP* entrantes, el hilo *INT* emite por las mismas las respuestas adecuadas cuando se recibe un paquete *DC3P* por una conexión activa. Todas las conexiones de puertos en un mismo sentido se multiplexan en una conexión *TCP*, para más detalle ver el apartado 6.3.2.

Cuando se haya efectuado con éxito una secuencia de conexión el hilo *INT* debe entregar al puerto correspondiente en el *IBox* del componente los datos que lleguen en paquetes *Data DC3P*. Para lograr esto el hilo *INT* dispone de un *OBox* con puertos de salida complementarios a los puertos de entrada del *IBox* del componente. Los puertos de salida del hilo *INT* se conectan o desconectan del correspondiente puerto de entrada del componente cuando se lleve a cabo una secuencia *DC3P* de conexión o desconexión respectivamente. Cada conexión entre un puerto de salida del hilo *INT* y un puerto de entrada del componente se realiza la primera vez que se reciba una solicitud de conexión a ese puerto de entrada del componente y para las siguientes solicitudes hacia ese puerto de entrada determinado se incrementa un contador de conexiones realizadas. Utilizando este contador el hilo *INT* desconectará su puerto de salida del puerto concreto de entrada del componente cuando se reciba la última secuencia de desconexión de ese puerto. Por tanto las conexiones entre el *OBox* del hilo *INT* y el *IBox* del componente se realizan bajo demanda. La

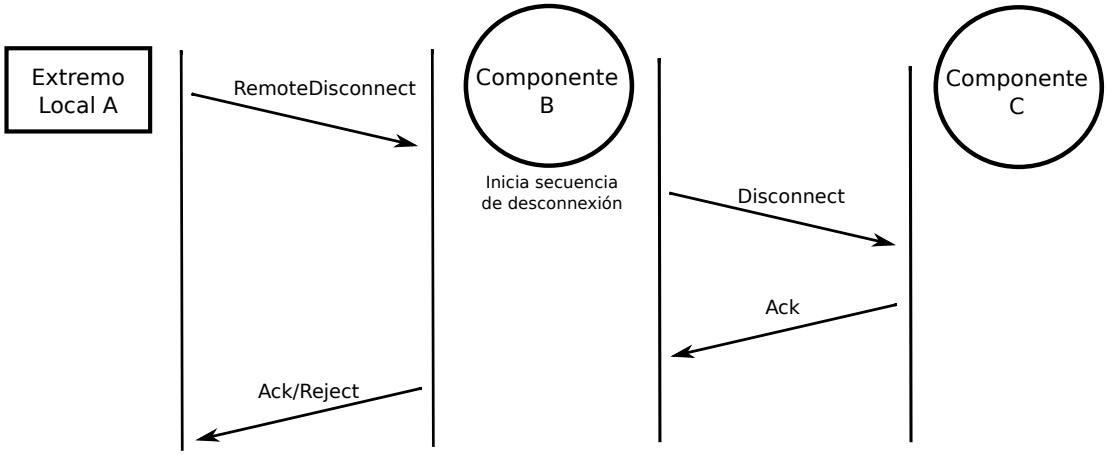

Figura 6.25: Secuencia de desconexión remota

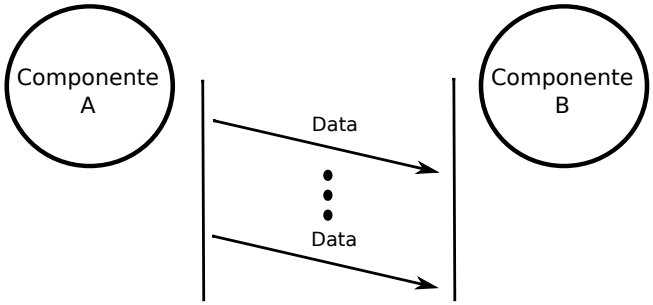

Figura 6.26: Secuencia de envío de datos

figura 6.29 refleja cómo es la organización interna del hilo *INT* dentro de un componente con red. En la figura se muestran todos los puertos de salida del hilo *INT* conectados a los correspondientes puertos de entrada del componente.

Los envíos por la red que realiza un componente se resuelven de manera similar a las recepciones. Existe un hilo *Output Network Thread (ONT)* encargado de dicha tarea. Sin embargo este hilo no ejecuta un bucle de acciones cada cierto tiempo al igual que el *INT*, sino que se encuentra constantemente en suspensión y sólo se activa cuando existe actividad en algún puerto de salida que esté conectado por la red.

El *ONT* dispone de un *IBox* con puertos de entrada complementarios a los puertos de salida del componente. Las conexiones/desconexiones de puertos entre el *OBox* del componente y el *IBox* del *ONT* se realizan bajo demanda, de forma que cuando se lleva a cabo una secuencia *DC3P* de conexión o desconexión se realiza la conexión o desconexión del puerto correspondiente del componente con el complementario del hilo *ONT*. El hilo *main* es el encargado de la tarea de realizar las secuancias *DC3P* de conexión/desconexión, así como del conexionado de los puertos

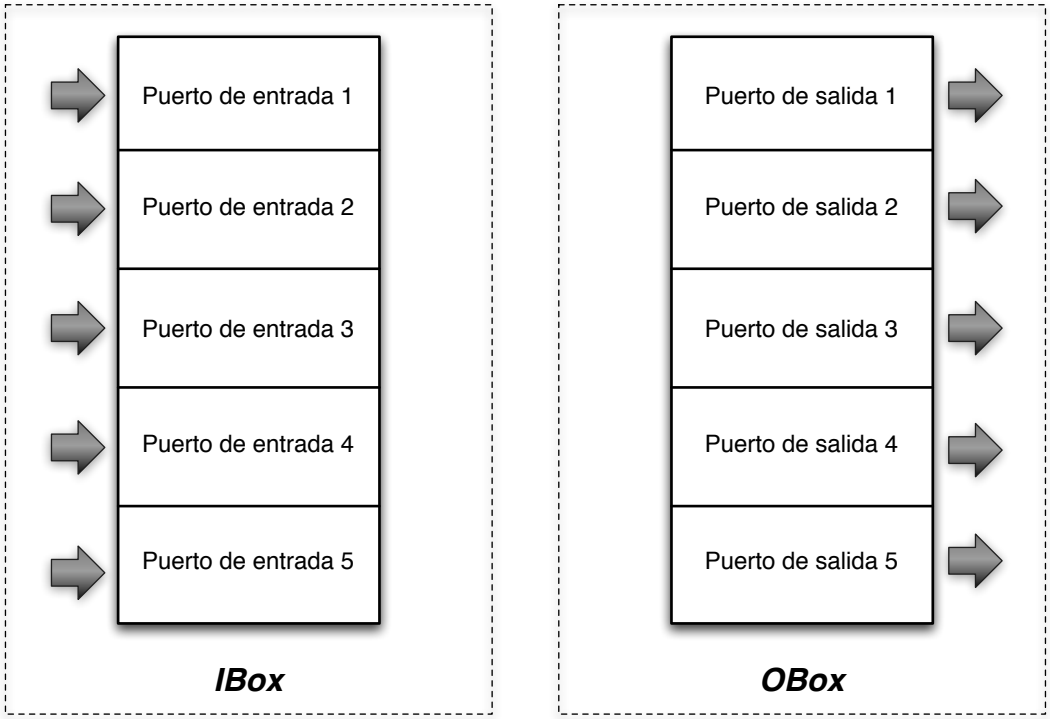

Figura 6.27: Ejemplo de IBox y OBox

de salida del componente con los puertos de entrada del hilo *ONT*. Para esto se mantiene una cuenta de cuantas veces ha sido conectado un puerto de salida del componente de forma que entre el *OBox* del componente y el *IBox* del *ONT* sólo existan las conexiones demandadas. La figura 6.30 clarifica la estructura interna del hilo *ONT* dentro de un componente. Se muestran todos los puertos de salida del componente conectados a los puertos de entrada del hilo *ONT*.

El hilo *ONT* es el encargado de envíar todos los paquetes de puertos que el componente emita por los puertos de salida que estén conectados por la red. El hilo *ONT* se activará ante cualquier llegada de datos en su *IBox* y emitirá el paquete de puerto a los puertos de entrada de los componentes remotos a los que estuviera conectado el componente.

Cabe destacar que los paquetes del protocolo *DC3P* viajan a través de la red según la especificación *CDR*, detallada en 5.7. Esto es transparente para los hilos de red (*INT* y *ONT*) ya que CoolBOT suministra las operaciones de envío y recepción por la red y son estas las encargadas de realizar el *marshalling/demarshalling* de paquetes *DC3P*.

Con este modelo el comportamiento del componente es exactamente como en el modelo sin red, de forma que para un componente es transparente el hecho de que los datos se hayan recibido desde un componente remoto o no.

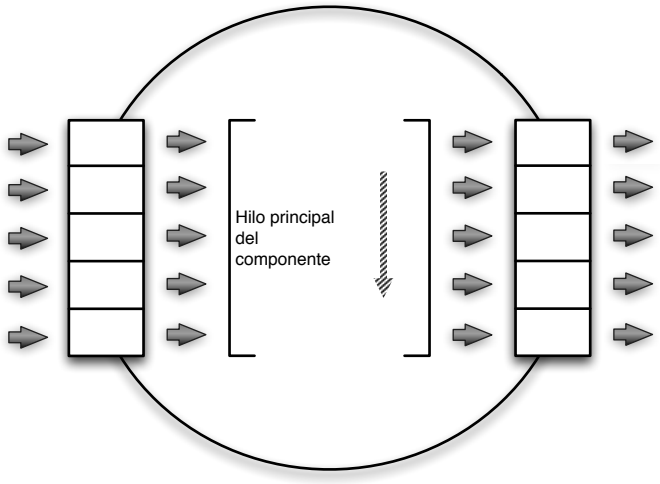

Figura 6.28: Hilo *main*, IBox y OBox de un componente

6.5. *Sondas de componentes*

Según la terminología técnica, una **sonda** es un objeto de manipulación remota cuya misión es llegar a un objetivo prefijado y realizar algún tipo de acción o mandar información. En función del contexto o entorno que se necesite investigar podemos hablar de diversos tipo de **sondas**:

- **Sonda espacial**. Son enviadas al espacio con el fin de estudiar cuerpos de nuestro Sistema Solar, tales como planetas, satélites, asteroides o cometas.
- **Sonda náutica**. Determinan la distancia vertical entre el fondo del lecho marino y una parte de determinada del casco de una embarcación.
- **Sonda molecular**. Usadas en biología molecular como herramienta para detectar la presencia de ADN.

Utilizaremos en este trabajo el concepto **sondas de componentes** para referirnos a un tipo de mecanismo que proporciona acceso a la interfaz de un componente de manera local o remota.

Las *sondas* representan la imagen especular de un componente. Constan de un *IBox* de entrada que tendrá tantos puertos como puertos de salida tenga el componente y de un *OBox* de salida, con tantos puertos como puertos de entrada tenga el componente. Esto queda más claro a la vista de la imagen 6.32, donde se puede contemplar una vision gráfica de cómo se lleva a cabo el conexionado entre el componente y su *sonda*.

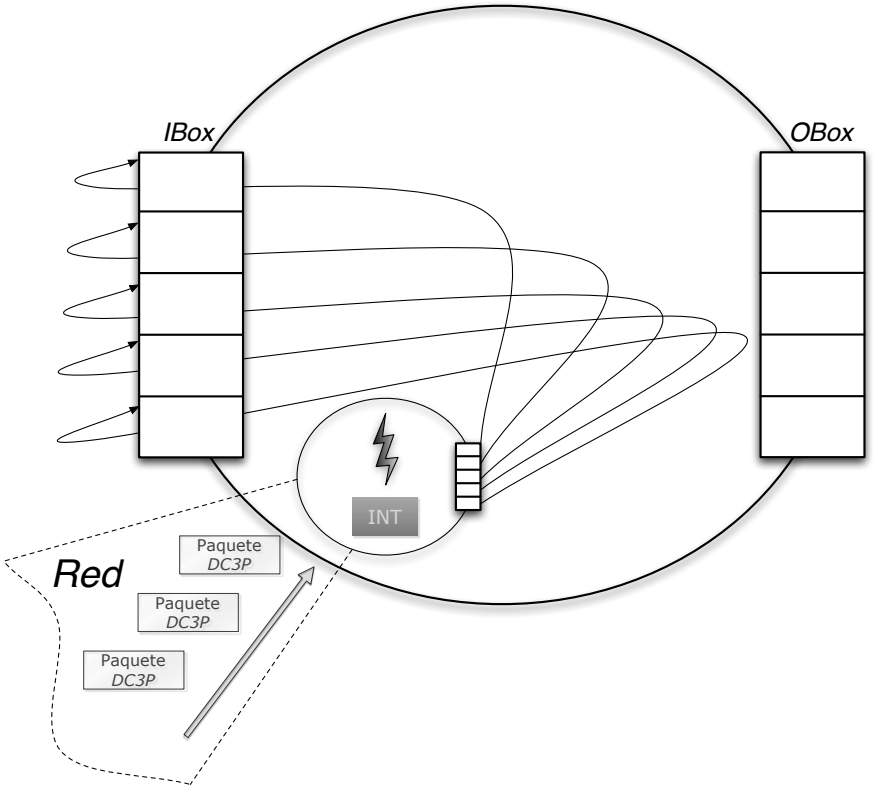

Figura 6.29: Hilo *INT* de un componente

Aunque pueda llevar a confusión la representación utilizada para la *sonda* en el diagrama 6.32 debido a que la disposición de su *IBox* y su *OBox* aparece intercambiada con respecto a la representación utilizada hasta ahora para los componentes, es simplemente por una razón de simplicidad y claridad gráficas, ya que, del otro modo, se hubiesen cruzado las líneas de conexión, resultando engorroso de cara al lector.

6.6. Construcción de *vistas* mediante *sondas* de componentes

El diseño de las *sondas* permite desacoplar por completo las *vistas* (presentación de datos) del sistema róbotico que esté siendo monitorizado a través de las mismas, de manera similar a como el patrón *Modelo-Vista-Controlador (MVC)* separa el control y los datos de la representación de estos últimos.

El uso de *sondas* facilita el desarrollo de aplicaciones de monitorización en robótica en tan-

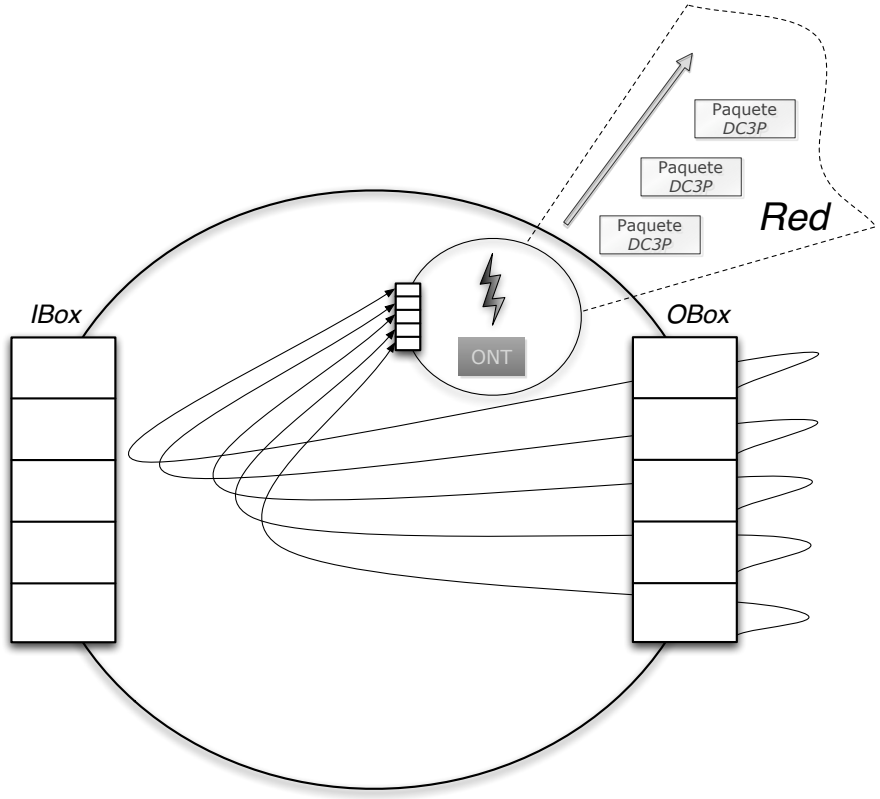

Figura 6.30: Hilo *ONT* de un componente

to en cuanto permite configurar qué datos extraer del sistema que se comunica con el robot y presentarlos de la manera más apropiada de cara al operador del sistema. Asimismo esto encaja en entornos teleoperados ya que sirve perfectamente para labores de monitorización pero también de manipulación de robots, bien sea en local o remotamente, capacitando así la construcción de software para visitas guiadas remotas.

Por ejemplo, un desarrollador de sistemas robóticos podría estar interesado en probar su componente para verificar su correctitud sin necesidad de utilizar un *front-end* de ventana, sino directamente lanzando en una máquina su componente y en otra una aplicación test a través de consola que lance su *sonda*, conecte los puertos de interés, mande comandos y posteriormente imprima por pantalla la información de los *paquetes de puertos* que se quieran analizar. En la imagen 6.33 podemos observar gráficamente este ejemplo. A continuación se va a explicar detalladamente cómo se comunica una aplicación de consola con un componente que está en otra máquina a través de su *sonda*.

En primer lugar se va a describir qué hace el componente remoto con el que la aplicación de consola desea comunicarse (representado por el número 1). Este componente tiene un *IBox* con un único puerto de entrada a través del cual le llegarán unos *paquetes de puerto* de un tipo

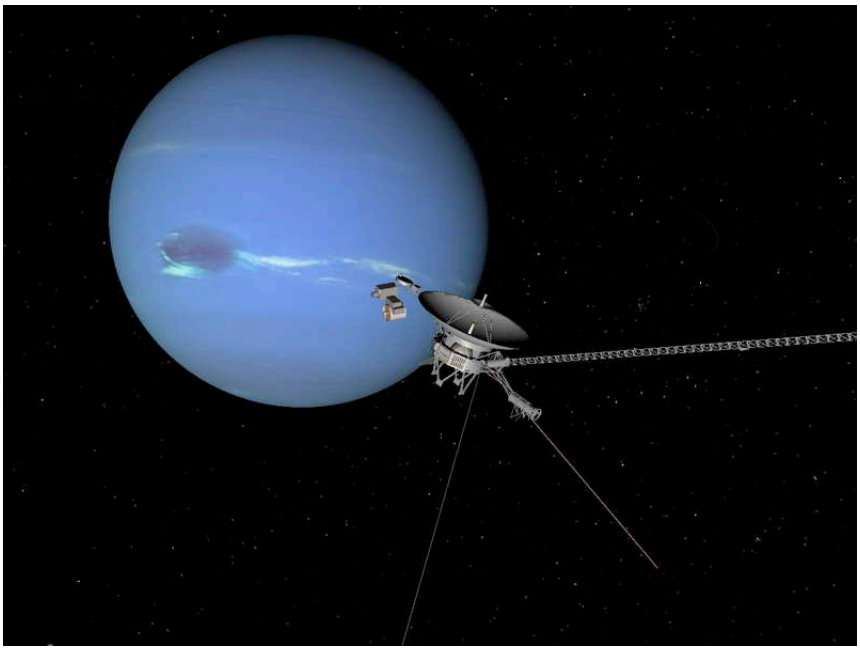

Figura 6.31: Sonda Voyager II. Utilizada en 1989 para captar imágenes alrededor de la órbita de Neptuno, descubrió seis de sus ocho lunas y confirmó la existencia de los anillos casi invisibles que se encuentran alrededor de él.

"Coordenadas XY", que imaginamos consiste en dos valores enteros que representan un punto en coordenadas cartesianas. El componente también tiene un *OBox* de salida con dos puertos; uno envía *paquetes de puerto* del tipo "Coordenadas XY" y el otro envía paquetes de tipo "ImageRGB". Los paquetes de este tipo representan una imagen en *RGB* obtenida a través de una cámara integrada con el robot. El funcionamiento de este componente es el siguiente: el componente recibe un paquete de tipo "Coordenadas XY" y lo traduce a un punto en un mapa donde está situado el robot. A continuación el componente obtiene la posición del robot y lo comanda hacia el punto que se le indicó. Acto seguido, el componente encapsulará la posición actual del robot en un paquete de tipo "Coordenadas XY" y lo enviará a través de su *puerto de componente* de salida. En este ejemplo el único puerto que está conectado es el que envía "Coordenadas XY", así que estos serán los únicos paquetes que emitirá el componente. Una vez entendido el funcionamiento del componente pasamos a explicar el funcionamiento de su *sonda*.

La *sonda* (representado por el número 2) es una imagen especular del componente anterior, de manera que tiene un *IBox* con dos puertos de entrada y un *Obox* con un solo puerto de salida. Estas estructuras están conectadas con las de su componente homónimo a través de la red. De esta manera, cuando el componente (representado por el número 1) envía un paquete *DC3P*, el hilo de entrada de la *sonda* que estará escuchando en un puerto TCP/IP, recibirá actividad por uno de sus puertos (en este caso sólo hay uno conectado) y desencapsulará el paquete *DC3P*, que será impreso por pantalla según la representación escogida para este tipo de datos.

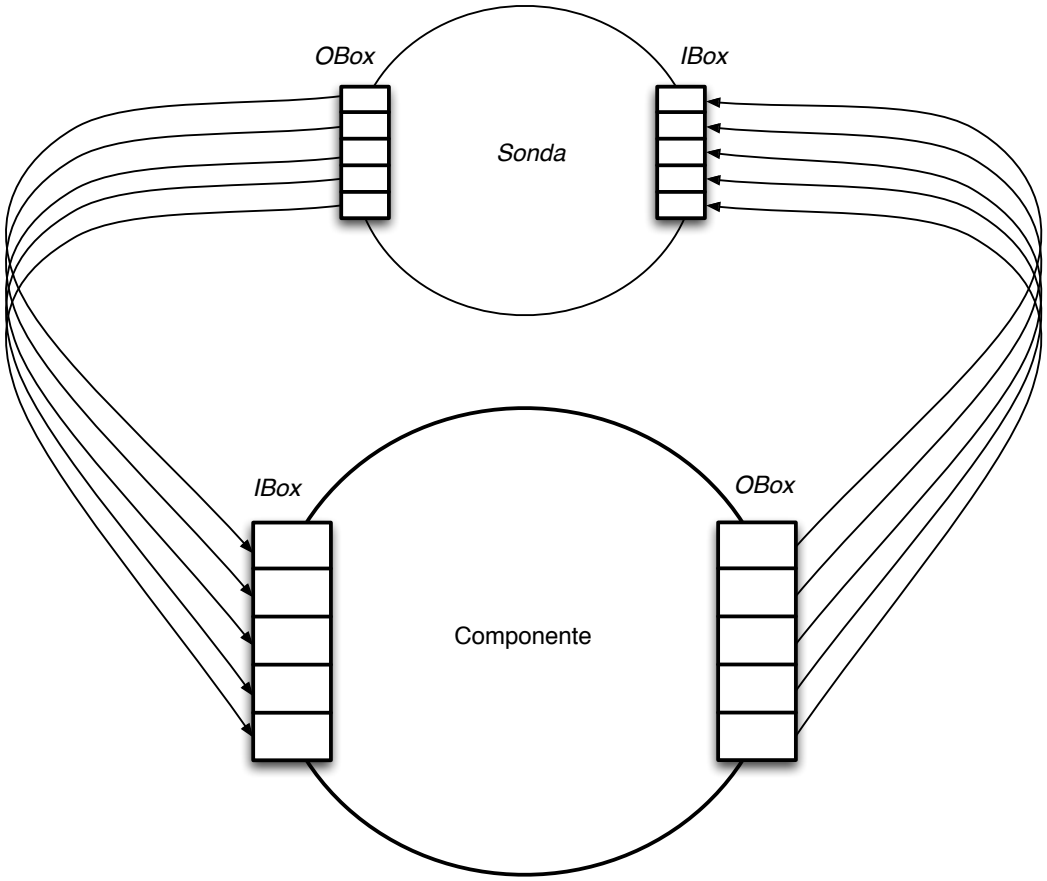

Figura 6.32: Conexionado entre un componente y su *sonda*

Cuando a través de consola se desee comandar el robot, se rellenará un paquete de tipo "Coordenadas XY" con el punto (x,y) a donde se desea dirigir el robot, y será el componente *Probe* el encargado de encapsular el dato para que pueda viajar por la red y hacérselo llegar al puerto de entrada del componente "sondeado". De esta forma se puede realizar desde una aplicación de consola muy sencilla un front-end no gráfico que permita a un operador comandar a un robot y presentar los datos según sus necesidades.

A través de este ejemplo se observa claramente el desacople existente entre la aplicación por consola y la *sonda* (representado por el número 3), que realmente sirve de interfaz a las funciones de un componente instanciado en la misma u otra máquina. Esto permite desarrollar aplicaciones y *front-end* gráficos orientados hacia la presentación de resultados y hacia la usabilidad y accesibi-

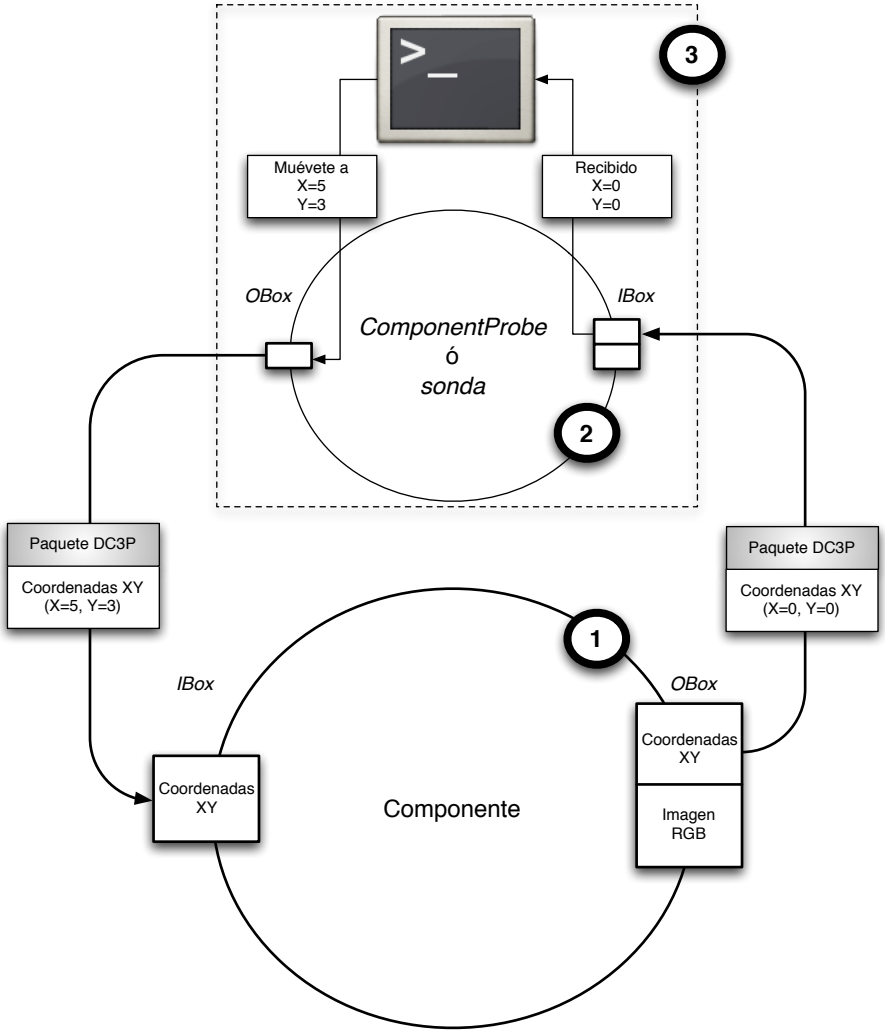

Figura 6.33: Aplicación de consola para comandar y monitorizar un robot.

lidad de la aplicación, donde las *sondas* sean las encargadas de llevar a cabo toda la comunicación con los componentes y de proporcionar los datos susceptibles de ser representados.

En la imagen 6.34 se aprecia ya un sistema de teleoperación más elaborado con una interfaz de control y de monitorización (representada con un número 1) que utilizará funciones propias de interfaces gráficas tales como dibujo de formas primitivas, composición de gráficos, presentación de imágenes *RGB*, inserción de elementos de interacción con la interfaz (botones, paneles, pestañas, barras deslizables), etc. mejorando la experiencia teleoperativa del operador. En esta imagen podemos observar cómo se intentan comandar los movimientos de un robot a través de la interfaz, que traducirá los eventos producidos en ella en acciones de la *sonda* para que envíe o reciba datos del componente. La *sonda* (señalizada con el número 2) está suscrita a los puertos del componente que gobiernan el movimiento del robot y monitorizan su posición, enviando y recibiendo por ellos los

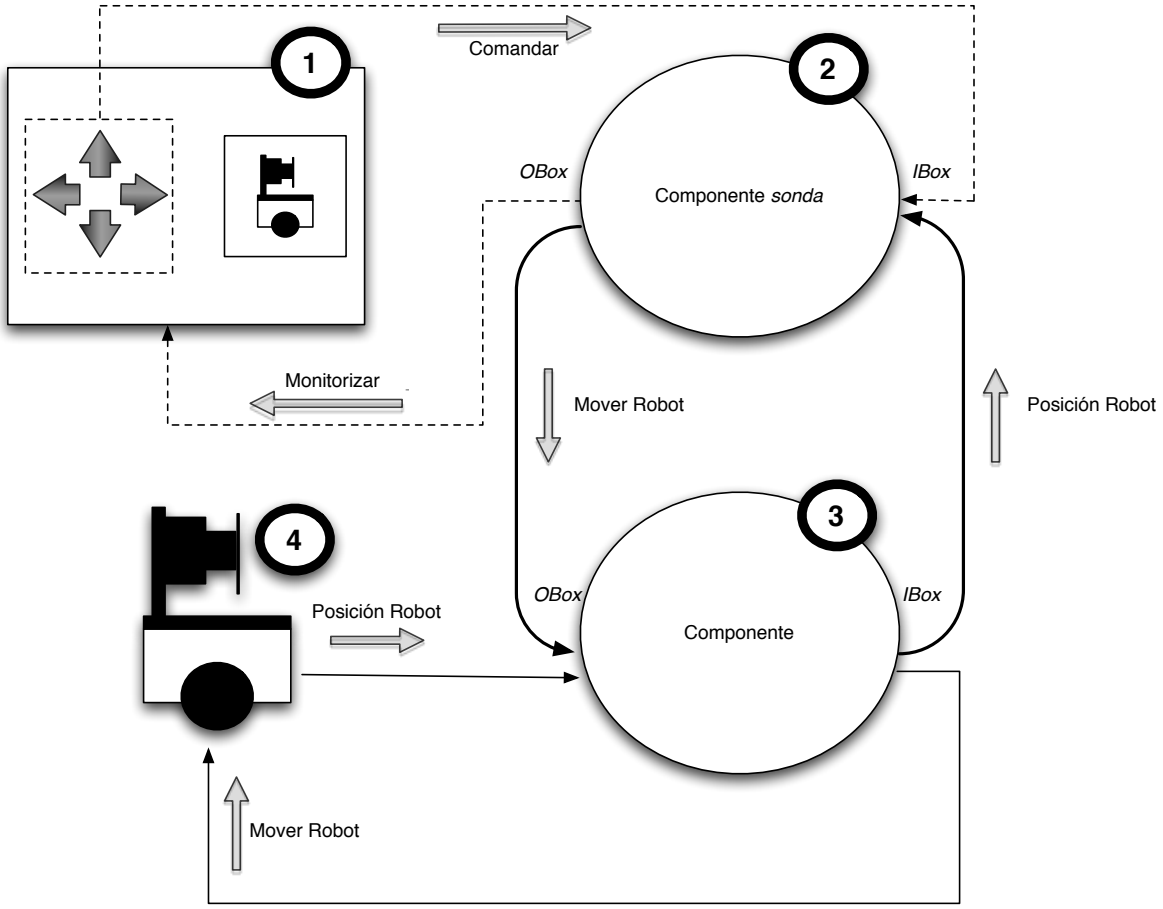

Figura 6.34: Componente *Probe* aplicado a una *Vista*

paquetes que precise en cada instante. En otro extremo remoto estará el componente (representado por un 3) dormido en sus puertos de entrada esperando por la *sonda* o por el robot. Por la primera se despertará cuando ésta le inyecte algún paquete, que, en este caso, servirán para dirigir al robot hacia alguna posición. Por el robot (representado con un número 4) se despertará cuando le envíe su posición a través de uno de sus puertos de entrada.

Debido a que en este ejemplo el sistema robótico está formado por un sólo componente, la *vista* solo necesita hacer uso de una *sonda*. Sin embargo, esto no siempre es así. Una vista puede querer representar datos provenientes de diferentes componentes por tanto será necesario "sondear" esos componentes para obtener la información que se desee dibujar. Para tal fin se requiere que esa vista disponga de un *sonda* para cada uno de los componentes que se representan en ella.

En caso de que un operador esté muy familiarizado con un lenguaje de programación de interfaces determinado o necesite la funcionalidad que aportan ciertos lenguajes orientados a la realización de este tipo de aplicaciones como (*Java, GTK, Qt, wxWidgets, Tcl/Tk, etc.*) tiene la posibilidad

de construir un sistema teleoperado utilizando alguna librería para *wrapping* de código y el uso de *sondas* para la comunicación remota con los componentes que integran el sistema robótico. Para este trabajo ha sido necesario el uso de una librería de estas características y de las *sondas* para elaborar una interfaz gráfica en Java compuesta de *vistas* de componentes.

6.7. Interfaz de teleoperación Java

La interfaz de teleoperación desarrollada en este trabajo consiste en un servicio Web que proporciona el acceso a un *applet* en *Java* compuesto de un conjunto de *vistas* que hacen uso de componentes *sondas*.

El conjunto de *vistas* integradas en la interfaz capacita la comunicación con un sistema robótico destinado a la realización de visitas guiadas remotas. La interfaz desarrollada en este trabajo a modo de prototipo ilustrativo está estructurada de la siguiente forma (figura 6.35):

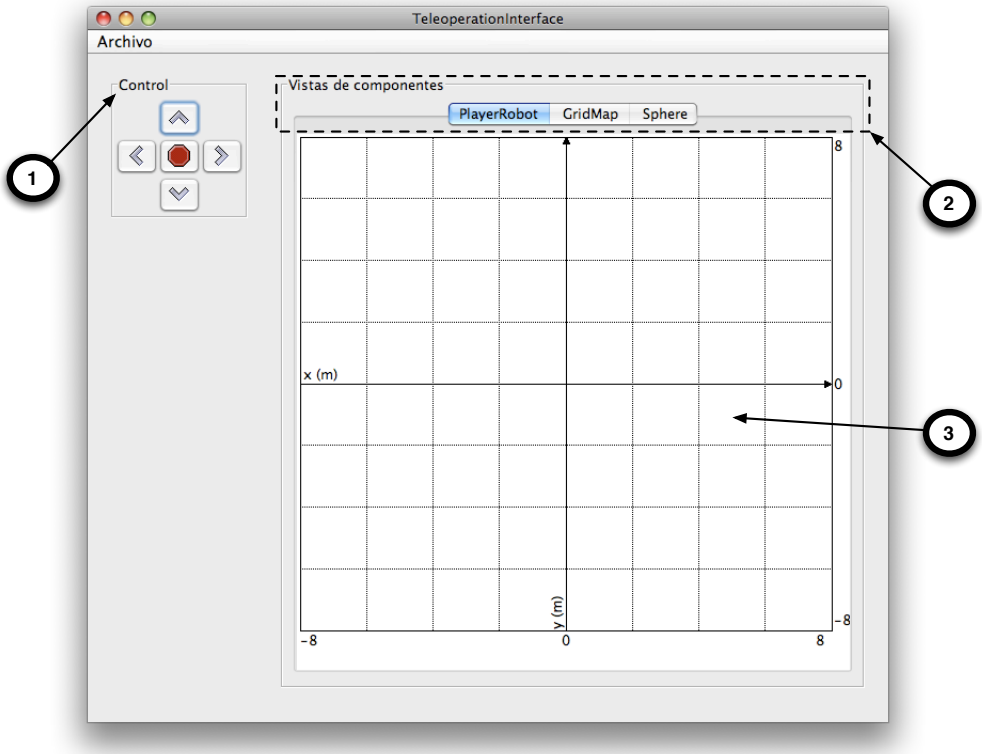

Figura 6.35: Interfaz de teleoperación

1. **Control**. Permite comandar al robot o detenerlo si es necesario (botón rojo central). Las

acciones que pueden llevarse a cabo son:

⇑: Avanzar hacia el frente.

⇓: Retroceder.

⇐: Girar sobre sí mismo hacia la izquierda.

⇒: Girar sobre sí mismo hacia la derecha.

Estas acciones son comandadas a través del puerto de entrada *Commands* del componentes *Player-Robot*.

2. **Vistas de componentes**. Conforman un conjunto de pestañas en la que cada una se corresponde con la *vista* de un componente.
3. **Área de dibujo**. Representa la zona donde cada *vista* realizará la presentación de sus datos. En función de la naturaleza de estos pueden dibujarse caracteres alfanuméricos, líneas, polígonos, imágenes, etc. al igual que se comentó en la sección 6.6.

Todos los elementos que forman parte de la interfaz son componentes *Swing*, que como se comentó en el capítulo 5 es una librería para el desarrollo de aplicaciones gráficas en *Java*. La aplicación que conforma la interfaz es en sí un *JApplet*, que contiene principalmente elementos *JButton* para el control del robot, y un *JTabbedPane* que contiene todas las *vistas* gráficas de los componentes necesarios para teleoperar en entornos remotos con el robot. En el capítulo (cap:pruebas-resultados) se describe la información que representa cada vista con más detalle. A continuación se van a explicar las directrices seguidas para diseñar en general las *vistas*:

- Cada *vista* debe hacerse heredar de la clase *JPanel*.
- El número de *sondas* que debe tener cada *vista* coincide con el número de componentes que emiten datos que serán representados por la vista.
- Debe redefinirse el método *PaintComponent* para cada *vista* en función de cómo se quieren presentar los datos para cada una de ellas.
- Cada *vista* debe contener un hilo *timer* para ordenar cada cierto tiempo a las *sondas* que comprueben si algún componente ha emitido algún dato nuevo con el fin de repintar el área de dibujo de la *vista*. El período con el que se configura el *timer* dependerá de la frecuencia de refresco deseada para la *vista*.

Durante el proceso de diseñar una *vista* se ha observado que uno de los elementos que toda *vista* de un componente debe tener es una o varias *sondas*. Esto conlleva a la necesidad de poder acceder a código escrito en C++ desde código Java, por lo que se precisa alguna herramienta orientada al *wrapping* de código, como es el caso de *Swig* (ver sección 5.10). En la sección siguiente se indicará cómo utilizar *Swig* para permitir a las *vistas* acceder a los datos captados por las *sondas*.

6.8. Acceso a las *sondas* a través de *Swig*

Swig utiliza su propia extensión para la definición de ficheros de interfaces (extensión **.i**). En estos ficheros se especifican aquellas clases y métodos para los que se requiere una interfaz, es decir, todo aquel código C++ que necesite ser accedido desde Java.

En estos ficheros suelen definirse métodos para el acceso a los puertos de las *sondas*, bien sea para recibir o para enviar datos. De esta manera, cada vez que una *vista* necesite representar un dato, llamará a una función para la recepción de un paquete desde un puerto de la *sonda*. De igual manera es necesario poder enviar paquetes por los puertos de las *sondas* cada vez que se produzca un evento en la interfaz relacionado con esa *vista* que obligue a realizar alguna acción sobre el sistema robótico. Un ejemplo de este último caso sería el de un evento producido por un operador que oprime el botón ⇑ con el fin de comandar hacia delante el robot. Este evento debería de tener un *callback* asociado que cree un paquete que sea capaz de interpretar el componente que controla al robot y lo envíe a través del puerto de salida de la *sonda* asociado a este tipo de paquetes. De esta manera el paquete llega al componente a través de la *sonda*, haciendo que éste se despierte por el puerto al que llegó el comando y dialogue con el robot para que avance en línea recta.

Capítulo 7

Detalles de diseño

7.1. Diseño de los mensajes del protocolo DC3P

Como ya hemos introducido previamente en la sección 6.2, el diseño de un protocolo no es una tarea trivial, sino que conlleva un gran esfuerzo que debe traducirse en funcionalidad y fiabilidad esperadas. En este caso, el diseño del protocolo DC3P alberga dos premisas claras: sencillez y rapidez. Cuanto menores sean las reglas de procedimiento del protocolo y el tamaño de los mensajes, menor será la predisposición hacia los errores y mayor será la rapidez del mismo.

Como hemos visto en el capítulo 6.3.3, el protocolo de comunicación *D3CP* consta de 6 tipos diferentes de mensajes, que se extienden a 18 si tenemos en cuenta sus variantes, para llevar a cabo la comunicación entre componentes *CoolBOT* de manera remota. Aplicando patrones al diseño de los comandos que definirán la funcionalidad del protocolo, se ha realizado el diseño observable en la figura 7.1.

En este diagrama de clases observamos que todo comando (*DC3PCommand*) del protocolo está formado por una cabecera (*PacketHeader*) y un cuerpo de mensaje (*PacketBody*). El tiempo de vida de estas dos últimas clases está condicionado por el tiempo de la primera, ya que no tiene sentido la existencia de una cabecera o de un cuerpo de mensaje aisladas de un *DC3PCommand*.

Además, tanto las clases *DC3PCommand* como *PacketHeader* heredan una interfaz de depuración (*DebuggingInterface*) y una interfaz que permita que sean empaquetadas/desempaquetadas para su correcto envío y recepción por la red respectivamente (*PackingInterface*). Estas interfaces son explicadas con más detalle en el apartado 8.1. Más adelante podremos observar que la clase *PacketBody* hereda también de estas dos interfaces para proporcionar a sus clases hijo estas funcionalidades y otras comentadas a posteriori en la imagen 7.3.

En la imagen 7.2 podemos contemplar con más detalle el diagrama relacional de la clase *PacketBody*. Como podemos apreciar en la imagen, esta clase implementa el patrón *prototype* [Gamma, 1995], permitiendo manejar de forma polimórfica las distintas clases que heredan de ella, es decir, a todos los paquetes del protocolo DC3P que tengan cuerpo. De esta manera ahorramos replicar código, ya que podemos utilizar un único objeto susceptible de ser *clonado*, dependiendo del paquete que necesitemos instanciar en cada momento.

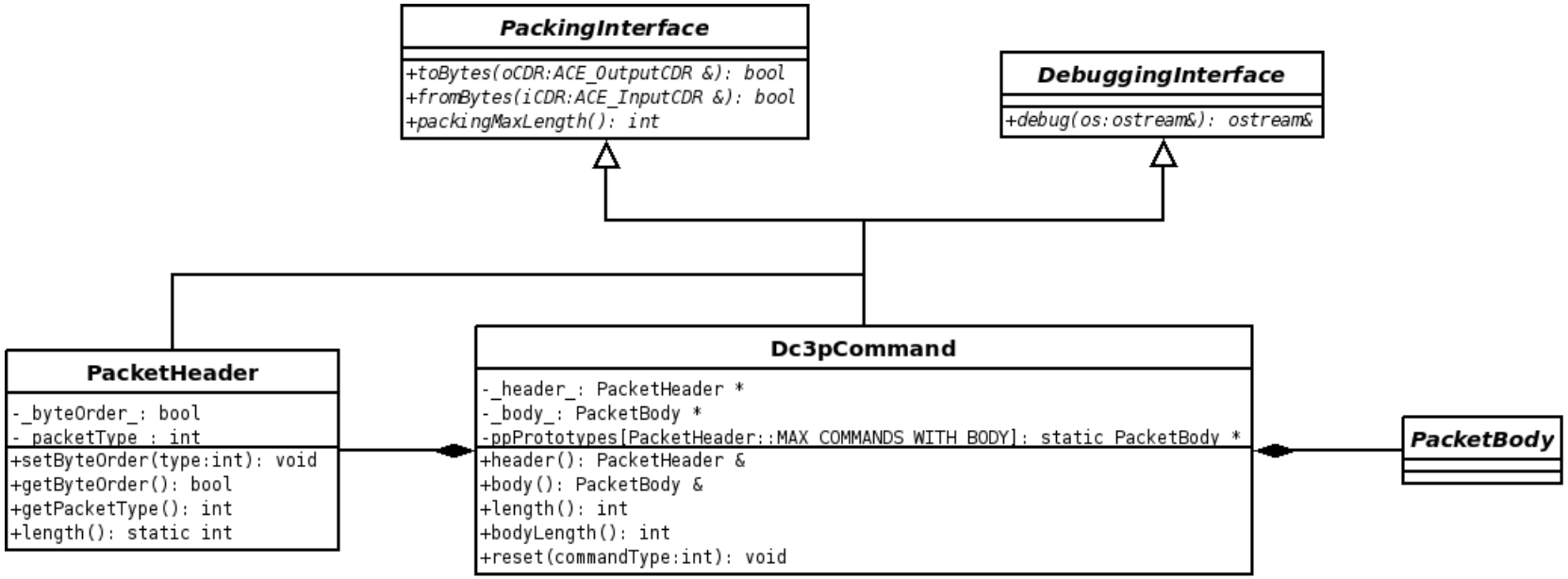

Figura 7.1: Diseño del protocolo de comunicación D3CP

Como podemos ver en la imagen 7.2, los paquetes del protocolo que heredan de *PacketBody* son aquellos que contienen cabecera y cuerpo, en lugar de cabecera exclusivamente, como es el caso de los paquetes de *Echo* y *Ack*.

La clase *PacketBody* proporciona una serie de interfaces que van compartir los paquetes que hereden de ella y que deben ser implementadas en todos y cada uno de ellos, dotando al diseño de estos paquetes de las funcionalidades descritas brevemente en la ilustración 7.3.

7.2. Patrones de diseño utilizados

En esta sección se presenta una descripcción de los patrones que han sido utilizados y cómo han sido adaptados en cierta medida a las necesidades de diseño del proyecto.

7.2.1. Prototype

- **Propósito.** Permite especificar tipos concretos de objetos que se pueden crear a través de una instancia prototípica. La creación de los objetos se hace a través de una función que copia el prototipo.
- **Aplicabilidad.** Este patrón se aplica de forma general cuando un sistema o subsistema requiere:
  - una abstracción e independencia de cómo se crean, componen y representan una serie de objetos que maneja.
  - las clases que se instancien puedan ser especificadas en tiempo de ejecución.
  - una clase tiene muchas variedades de estados distintos sin embargo, sólo se van a manejar un conjunto reducido de ellos.

  Este patrón se ha utilizado en el diseño de la estructura de los mensajes del protocolo *DC3P* (clase *Dc3pCommand*), así como en la estructura de los paquetes de datos que intercambian los componentes (clase *PortPacket*).
- **Estructura.**

  La estructura general de este patrón se ha adaptado según muestra el diagrama *UML* de la figura 7.4.

7.2.2. Patrón Modelo Vista Controlador: MVC

- **Propósito.**

Permite desacoplar los modelos de datos de su presentación gráfica y del control. De esta forma cualquiera de estos elementos puede ser modificado sin necesidad de cambios en los otros dos.

- **Aplicabilidad.**

  Este patrón se aplica de forma general cuando un sistema o subsistema requiere:

  - Necesidad de representaciones gráficas.
  - Necesidad de distribución de los elementos siguiendo modelos cliente-servidor.

  Este patrón se ha utilizado en el diseño de las vistas *CoolBOT* y de la interfaz de teleoperación.

- **Estructura.**

  La estructura general de este patrón se ha adaptado según muestra el diagrama *UML* de la figura 7.5. El control está imbuído en las estructuras de *IBox* y *OBox*, ya que son estructuras señalizables. La *vista CoolBOT* se corresponde con la vista del patrón *MVC* y el modelo de datos con los paquetes de puertos *CoolBOT*.

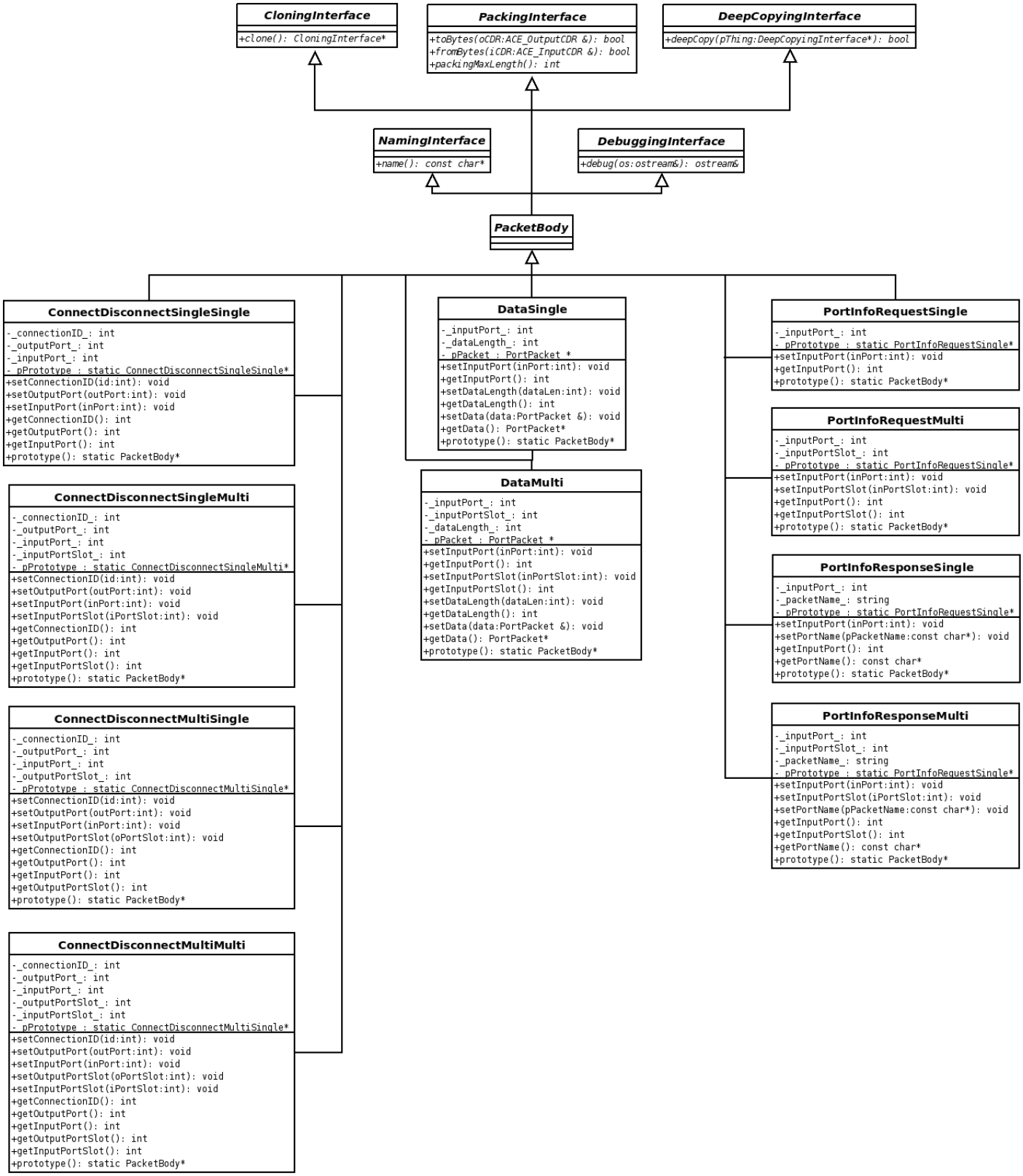

Figura 7.2: Estructura del cuerpo los mensajes del protocolo DC3P

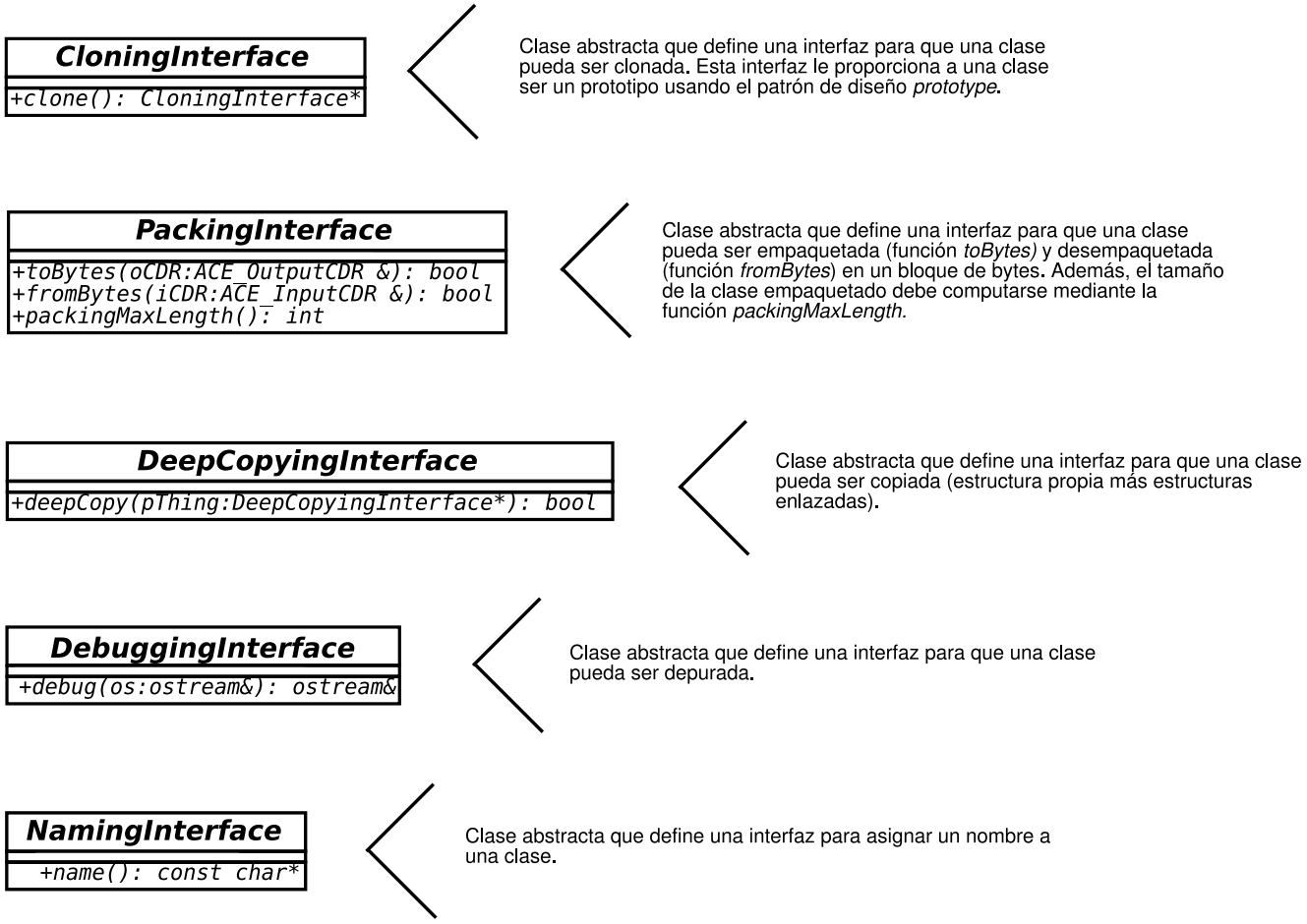

Figura 7.3: Interfaces proporcionadas por la clase *PacketBody*

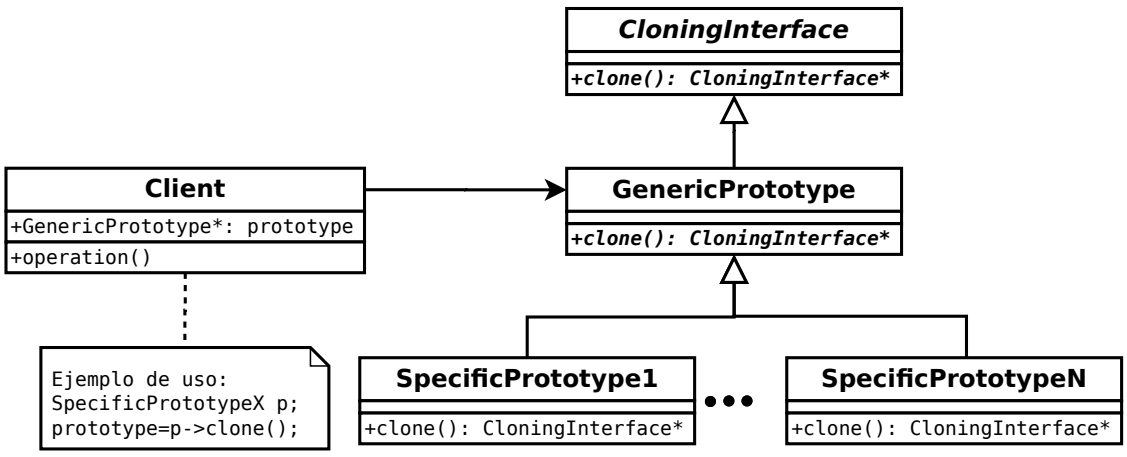

Figura 7.4: Diagrama de clases del patrón *prototype*

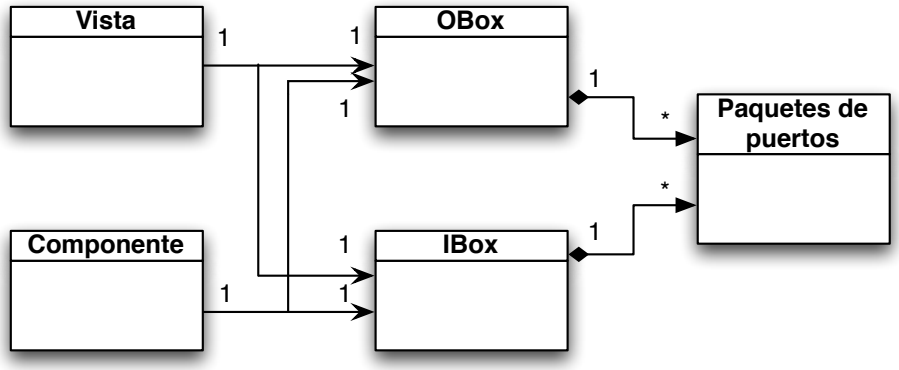

Figura 7.5: Diagrama de clases del patrón *MVC*

Capítulo 8

Detalles técnicos sobre la implementación del proyecto

8.1. Diseño de las clases *PortPacket* y *PacketBody*

Como se ha explicado en varios capítulos de este documento, las clases *PortPacket* y *PacketBody* son clases "padre" de las que heredan los paquetes de datos de usuario, y los mensajes que poseen cuerpo del protocolo DC3P, respectivamente. Aunque en otros apartados se ha expuesto una visión del diseño de ambas clases por separado, puede ser necesario señalarlo de manera conjunta, para que se vea claramente que ambas clases comparten el mismo diseño, como se puede apreciar en la imagen .

Para aclarar en cierta forma la funcionalidad que proporcionan estas dos clases, se va a explicar con un nivel somero de detalle cada una de las clases de las que heredan tanto *PortPacket* como *PacketBody*.

8.1.1. CloningInterface

Esta interfaz es necesaria para aplicar el patrón *Prototype* a aquellas clases que interesen ser clonadas. Cuando queremos instanciar una subclase dependiendo del código que se ejecute, podemos emplear un único objeto, que sea del tipo del padre y darle la "forma" de la subclase que deba instanciarse en cada momento. Es decir, podríamos copiar o "clonar" una instancia de la subclase adecuada. Esta instancia es conocida como *prototipo*. En nuestro *framework*, si todos los paquetes de datos o todos los mensajes del protocolo poseen una función de clonado (como es el caso), podremos clonar cualquiera de ellos a través de un objeto *PortPacket* ó *PacketBody*.

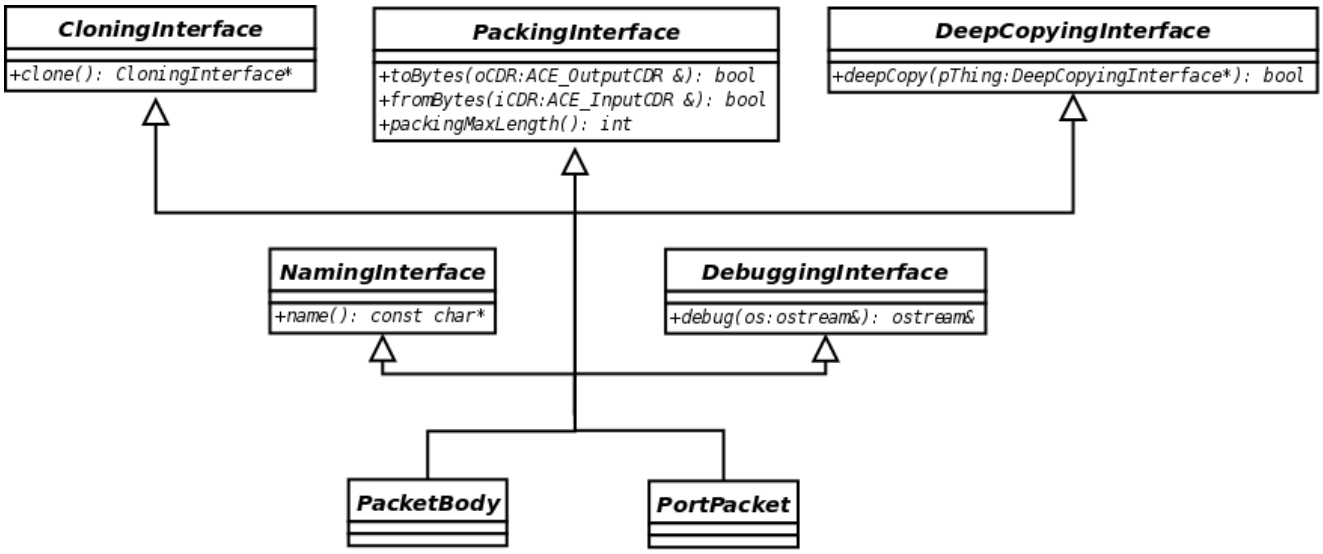

Figura 8.1: Vista del diseño de las clases *PortPacket* y *PacketBody*

El patrón *Prototype* [Gamma, 1995] puede ser usado para reducir el número de clases del sistema dramáticamente. Este patrón resulta útil emplearlo cuando un sistema deba ser independiente de cómo sus clases sean creadas, compuestas, representadas; y

- cuando las clases a instanciar sean especificadas en tiempo de ejecución, por ejemplo, mediante carga dinámica; *o*
- cuando las instancias de una clase puedan tener una de varias combinaciones de estados. En ese caso puede ser conveniente instalar un número adecuado de *prototypes* y clonarlos más que instanciar las clases manualmente cada vez con el estado apropiado.

8.1.2. DeepCopyingInterface

En ocasiones puede ser interesante una interfaz que permita copiar íntegramente una clase junto a sus estructuras, más aún, cuando la clase que deseemos copiar contenga punteros a otras estructuras. De esta forma, mediante la interfaz *DeepCopyingInterface*, obtendremos una copia de todas las estructuras que contenga la clase susceptible de ser copiada, así como una copia de los miembros que se encuentren apuntados desde dicha clase. De esta forma, si una clase A, que contenga una serie de punteros a estructuras, la copiamos desde otra clase B, a través de la interfaz *DeepCopyInterface*, las modificaciones que hagamos en B sobre cualquier miembro, no repercutirá sobre A, puesto que todas las estructuras y punteros contenidos en A han sido copiadas en otra zona de memoria a la que apunta B.

8.1.3. PackingInterface

El sentido de *PackingInterface* es poder enviar y recibir datos por la red, sin necesidad de conocer cómo se encapsulan (*Marshalling*) o desencapsulan (*Demarshalling*) la información en la estructura de datos proporcionada por **CORBA**, *CDR*.

De esta forma, si un usuario desea enviar datos por la red, tan sólo tendría que seguir los siguientes pasos para encapsularlos en un *CDR*:

- Para cada uno de los atributos que deseen enviarse por la red vamos a hacer una llamada a *PackingMaxLength* para computar el tamaño máximo del *CDR*. Estas llamadas devolverán un tamaño en bytes, que se sumarán para calcular el tamaño total de los datos más *padding*:

```
int packingMaxLength()
{
    return PackingMaxLength(atributo_1) +
           PackingMaxLength(atributo_2) +
           ... +
           PackingMaxLength(atributo_N);
}
```

- Crearse una variable de tipo *ACE_OutputCDR*, que vamos a denotar en este ejemplo como *oCDR*, dónde se encapsularán los datos a enviar por la red. Se hará una llamada a la función *toBytes* por cada dato a serializar:

```
bool toBytes(ACE_OutputCDR &oCDR)
{
    toBytes(atributo_1,oCDR);
    toBytes(atributo_2,oCDR);
    ...
    toBytes(atributo_N,oCDR);
}
```

- Ahora que hemos encapsulado los datos en la estructura *oCDR*, podremos desempaquetarla en otra estructura del tipo *ACE_InputCDR*, que denominaremos aquí *iCDR*:

```
bool toBytes(ACE_InputCDR &iCDR)
{
    fromBytes(atributo_1,iCDR);
    fromBytes(atributo_2,iCDR);
    ...
    fromBytes(atributo_N,iCDR);
}
```

Con lo que tendremos en las variables *atributo_1, atributo_2, ..., atributo_N* el contenido de aquellas variables que hemos encapsulado previamente.

Para más información acerca de cómo utilizar estas interfaces proporcionadas por *PackingInterface* es conveniente consultar la guía de creación de comandos del protocolo DC3P (ver capítulo A.1) y para entrar en detalles de su implementación puede consultarse el anexo 8.2).

8.1.4. NamingInterface

Esta interfaz proporciona un método para asignar un nombre a una clase. Esto nos puede ser útil a la hora de depurar nuestra aplicación, ya que podremos preguntar por dicha interfaz y nos devolverá el nombre de la subclase que haya sido instanciada.

8.1.5. DebugginInterface

Esta clase nos proporciona un método para poder configurar la depuración de cualquier clase que herede de ella. De esta forma nososotros podremos indicar qué información deseamos ver cuando preguntemos por un objeto al que se le haya definido esta interfaz.

Un ejemplo de implementación de esta interfaz sería:

```
ostream& debug(ostream& os) const  //printing packet for debugging
{
    os << "CoolBOT::ClaseDeEjemploDeDepuracion" << endl;
    os << "atributo_entero = "<< atributo_1 <<endl;
    os << "atributo_ristra = "<< atributo_2 <<endl;
    ...
    return os;
}
```

8.2. Empaquetado de datos de forma genérica y transparente al usuario

Como vimos en el capítulo 5.7.5, ACE encapsula los datos para su envío en una estructura CDR, la cual acepta todos los tipos primitivos OMG IDL indicados en la tabla 5.2 además de

strings. Así pues, ACE proporciona funciones para encapsular / desencapsular datos de los tipos previamente mencionados, además de vectores de los mismos. Con el objetivo de evitar que el usuario llame en cada momento a la función correspondiente para empaquetar cada dato primitivo (o construido), se ha decidido dotar a todos los paquetes de información que tengan que viajar por la red, de dos interfaces comunes (una para el empaquetado y otra para el desempaquetado), heredadas de *PackingInterface*. Dichas interfaces se definen de la siguiente manera:

- **bool toBytes (Dato, OutputCDR)**

  Esta función lee el valor de la variable *Dato* y lo almacenará en *OutputCDR* devolviendo en una variable booleana si se llevó a cabo correctamente el empaquetado del dato o no.

- **bool fromBytes (Dato, InputCDR)**

  Esta función lee el siguiente dato de *InputCDR* y lo volcará en la variable *Dato*, devolviendo en una variable booleana si se llevó a cabo correctamente el desempaquetado del dato o no.

  Las dos funciones anteriores están *templatizadas*, es decir, utilizando esta simple interfaz con cualquier dato primitivo, se llamará a la función que corresponda a la especialización del tipo en concreto, y en el caso de no tratarse de un tipo primitivo se llamará a la función genérica que invocará a la función implementada para dicho tipo. En el caso de arrays de datos primitivos las anteriores interfaces cambian sensiblemente:

- **bool toBytes (Array, DimensionArray, OutputCDR)**

  La función anterior recibe un array de datos primitivos, *Array*, y su dimensión como segundo parámetro (*DimensionArray)*, y devuelve en el último parámetro, *OutputCDR*, la estructura CDR de salida con el vector encapsulado en su interior.

- **bool fromBytes (Array, DimensionArray, InputCDR)**

  Esta función lee el contenido de InputCDR, desempaquetando en *Array* el vector encapsulado previamente, y su longitud en *DimensionArray*.

A continuación pueden observarse algunas implementaciones de las funciones *template* comentadas previamente para distintos tipos de datos primitivos. Para no ser redundantes, no se han incluído las especializaciones de todos los tipos de datos, sino sólo la de aquellos que tienen una implementación un tanto diferente, intentando abarcar toda la casuística existente.

- **Short**

```
template <>
    inline bool toBytes(short &packet, ACE_OutputCDR &oCDR)
    {
        oCDR.write_short(packet);
```

```
        return oCDR.good_bit();
    }

template<>
    inline bool fromBytes(short &packet, ACE_InputCDR &iCDR)
    {
        iCDR.read_short(packet);
        return iCDR.good_bit();
    }
```

En este caso sólamente es necesario invocar a las funciones que proporciona ACE para la escritura / lectura de enteros cortos (short) sin realizar ningún tipo de conversión de tipos previa.

- **Integer**

```
template <>
    inline bool toBytes(int &packet, ACE_OutputCDR &oCDR)
    {
        oCDR.write_long(packet);
        return oCDR.good_bit();
    }

template <>
    inline bool fromBytes(int &packet, ACE_InputCDR &iCDR)
    {
        iCDR.read_long(packet);
        return iCDR.good_bit();
    }
```

Los datos de tipo entero se encapsulan / desencapsulan como si de tipos long se tratase, ya que ambos ocupan 4 bytes. En este caso tampoco ha sido necesario ningún tipo de conversión previa a las llamadas a las funciones de escritura / lectura del CDR.

- **Long**

```
template <>
    inline bool toBytes(long &packet, ACE_OutputCDR &oCDR)
    {
        oCDR.write_long(packet);
        return oCDR.good_bit();
    }
```

```
template <>
    inline bool fromBytes(long &packet, ACE_InputCDR &iCDR)
    {
        ACE_CDR::Long p;
        iCDR.read_long(p);
        packet=p;
        return iCDR.good_bit();
    }
```

El desempaquetado de datos de tipo long requiere el uso de una variable intermedia donde se realizará el *demarshalling* del dato previamente para hacer un cast a posteriori del tipo *ACE_CDR::Long* a *long*, debido a que la función *read_long* no está sobrecargada para este tipo de datos. Cabe destacar que todos los tipos básicos que proporciona ACE son *typedef* de los tipos primitivos de C++.

- **Long double**

```
template<>
    inline bool toBytes(long double &packet, ACE_OutputCDR &oCDR)
    {
        ACE_CDR::LongDouble p;
        p.assign(packet);
        oCDR.write_longdouble(p);
        return oCDR.good_bit();
    }

template<>
    inline bool fromBytes(long double &packet, ACE_InputCDR &iCDR)
    {
        ACE_CDR::LongDouble p;
        iCDR.read_longdouble(p);
        packet=p;
        return iCDR.good_bit();
   }
```

Para el tipo *Long double* también es necesario utilizar una variable intermedia del tipo *ACE_CDR::LongDouble* que es el tipo aceptado por las funciones de lectura / escritura. Para ésta última es necesario además llamar a la función *assign* puesto que se trata de una clase.

- **Unsigned char**

```
template<>
    inline bool toBytes(unsigned char &packet, ACE_OutputCDR &oCDR)
    {
        ACE_CDR::Char c;
        c=packet;
        oCDR.write_char(c);
        return oCDR.good_bit();
    }

template<>
    inline bool fromBytes(unsigned char &packet, ACE_InputCDR &iCDR)
    {
        ACE_CDR::Char c;
        iCDR.read_char(c);
        packet=c;
        return iCDR.good_bit();
    }
```

Para este tipo ha sido necesario realizar un cast de *unsigned char* a *char* porque las funciones que incorpora el CDR para leer y escribir en el mismo reciben un dato del tipo de éste último. No supone mayor inconventiente porque no se produce ningún tipo de pérdida de información mediante la conversión.

- **String**

```
template<>
    inline bool toBytes(string &packet, ACE_OutputCDR &oCDR)
    {
        oCDR.write_string(packet.c_str());
        return oCDR.good_bit();
    }


template<>
    inline bool fromBytes(string &packet, ACE_InputCDR &iCDR)
    {
        ACE_CDR::Char *buff=0;
        iCDR.read_string(buff);
        packet=(string)buff;
        return iCDR.good_bit();
    }
```

En este caso, las funciones que incorpora ACE para el manejo de ristras de caracteres exigen que las variables pasadas sean del tipo *char ** ó *ACE_CDR::Char **, por lo que, en el caso

de la escritura se realiza la conversión correcta mediante la función *c_str()*, y en la lectura se realiza un cast a *string* tras volcar la rista en la variable *buff*.

- **Función generica**

```
template <class T>
    inline bool toBytes(T &packet, ACE_OutputCDR &oCDR)
    {
        return packet.toBytes(oCDR);
    }

template <class T>
    inline bool fromBytes(T &packet, ACE_InputCDR &iCDR)
    {
        return packet.fromBytes(iCDR);
    }
```

Como el tipo que aceptan estas interfaces es un tipo genérico T, todas las funciones que las llamen para un tipo de dato primitivo o construido coincidirán con su prototipado, pero sólo entrarán en ellas aquellas para las que no exista una especialización para el dato llamador. En este caso se llamarán a las funciones *toBytes* ó *fromBytes* implementadas para dicho tipo.

- **Array de short**

```
template<>
    inline bool toBytes(short array[], int arrayLength, ACE_OutputCDR &oCDR)
    {
        return oCDR.write_short_array(array,arrayLength);
    }

template <>
    inline bool fromBytes(short array[], int arrayLength, ACE_InputCDR &iCDR)
    {
        return iCDR.read_short_array(array,arrayLength);
    }
```

Para el caso de vectores de tipos primitivos, ACE también proporciona funciones para realizar el empaquetado / desempaquetado de los mismos de una manera más eficiente que llamando a la función de cada elemento por separado. Como se comentó anteriormente, el prototipo de éstas varía sensiblemente, indicando en primer lugar un puntero al primer elemento del array, a continuación el tamaño del mismo, y por último el CDR de entrada o de salida.

- **Array de long**

```
template<>
    inline bool toBytes(long array[], int arrayLength, ACE_OutputCDR &oCDR)
    {
        for(int i=0; i<arrayLength;i++)
            toBytes(array[i],oCDR);
        return oCDR.good_bit();
    }

template <>
    inline bool fromBytes(long array[], int arrayLength, ACE_InputCDR &iCDR)
    {
        for(int i=0; i<arrayLength; i++)
            fromBytes(array[i],iCDR);
        return iCDR.good_bit();
}
```

En este caso ocurre algo similar a lo que ocurría en el caso del tipo de datos *long*. Es decir, las funciones *read_long_array* y *write_long_array* no aceptan un puntero a un array de datos de tipo *long*, así que sería necesario realizar un cast previo, el cual ya se hace en la función del dato primitivo *long*, así que simplemente se llamará a la función *toBytes* y *fromBytes* de dicho tipo para cada uno de los elementos del vector.

8.3. Encapsulado de operaciones para el cálculo de longitud de empaquetado: *packingMaxLength*

Como se puede comprobar en la interfaz de *marshalling* de datos, un parámetro requerido es el objeto textitCDR Stream en el que los datos serán introducidos/extraídos en base a la especificación *CDR* (Sección 5.7). *ACE* ofrece dos clases para crear este tipo de objeto: *ACE_OutputCDR*, en el caso de un *CDR Stream* de salida (*marshalling*); *ACE_InputCDR*, para uno de entrada (*demarshalling*). En ambos casos estos objetos requieren que se defina el tamaño los mismos en bytes, antes de ser utilizados.

CoolBOT ofrece un conjunto de operaciones para calcular los tamaños que un tipo de datos básico ocupará en el formato *CDR*. Nótese que este cálculo no es trivial, puesto que además de los bytes que ocupa un tipo de dato en sí, se debe contar con el tamaño añadido para mantener el alinemamiento de datos, siendo este valor dependiente de los datos que previamente pueden haber sido insertados en el *CDR Stream*. Para no restringir el orden en que un usario empaquete los atributos de sus clases (especificando la función *toBytes*), el cálculo del *padding* de alineamiento es el máximo posible, para cada tipo de dato.

Este conjunto de operaciones se ha implementado de forma similar a las funciones *toBytes* y *fromBytes* (Apéndice 8.2), es decir con el uso de *templates* de C++ y una serie de especializaciones.

```
template <class T>
inline int packingMaxLength(T &packet)
{
        return packet.packingMaxLength();
}

template <class T>
inline int packingMaxLength(T *packet)
{
        return packet->packingMaxLength();
}
```

Para cualquier tipo generico T se llama a la función *packingMaxLength* propia del tipo. Esta versión genérica sólo se instanciará cuando el usuario pretenda calcular el tamaño de empaquetado de un tipo construido, para el que se habrá especificado sus funciones concretas *toBytes* y *fromBytes* (Apéndice 8.2).

Algunas especializaciones concretas de los *templates* anteriores son las mostradas a continuación:

- **Short**

  ```
  template <>
  inline int packingMaxLength(short &packet)
  {
          return 2*sizeof(short) -1;
  }
  ```

- **Bool**

  ```
  template <>
  inline int packingMaxLength(bool &packet)
  {
          return sizeof(bool);
  }
  ```

  También existen sobrecargas de la función para el cálculo de longitudes en el caso de vectores, con sus versión genérica, en la que de nuevo se llamará a la especificada para el dato construido, y versiones especializadas.

```
template <class T>
inline int packingMaxLength(T packet[], int arrayLength)
{
        int result=0;
        for(int i=0; i<arrayLength; i++)
                result= result + packet[i].packingMaxLength();
        return result;
}
```

- **Int**

```
template <>
inline int packingMaxLength(int packet[], int arrayLength)
{
        return packingMaxLength(packet[0]) + (arrayLength -1) * sizeof(int);
}
```

Capítulo 9

Pruebas y resultados

Para las pruebas realizadas en este proyecto se ha partido de un conjunto de componentes *CoolBOT* cada uno de los cuales implementa una funcionalidad diferente. A cada uno se le ha añadido el soporte para el manejo de la red, descrito en el apartado 6.4.2, así como las operaciones de inicio del componente por la red, conexionado y desconexionado de puertos con componentes remotos. El interconexionado de estos componentes permite diseñar un sistema robótico teleoperado para la realización de visitas guiadas remotas. Además se han empleado *sondas* que permitan a las *vistas* el acceso a los puertos de los componentes, construyendo con ellas una interfaz que permite teleoperar este sistema.

En este capítulo se exponen los componentes que constituyen el sistema robótico creado, así como las diferentes *vistas* Java que se utilizan en la interfaz de teleoperación diseñada. Finalmente se presentan una serie de escenarios y distribuciones de componentes con las que se ha probado el sistema final.

9.1. Componente *PlayerRobot*

Se trata de un componente diseñado para representar a un robot móvil y permitir el acceso a sus diferentes elementos: sensores y actuadores. Este componente funciona internamente como un cliente de *Player (sección 2.1)*, por tanto se conecta al servidor *Player* que podrá estar ejecutándose con un robot real o bien utilizando *Stage* para simularlo. El robot dispone de varios sónars, un láser, *bumpers*, una cámara *Pan Tilt* de abordo y batería. Además ofrece lecturas de su configuración inicial y de la odometría, así como las posiciones de los diferentes sensores con respecto al robot (geometría). La figura 9.1 muestra la interfaz externa de este componente.

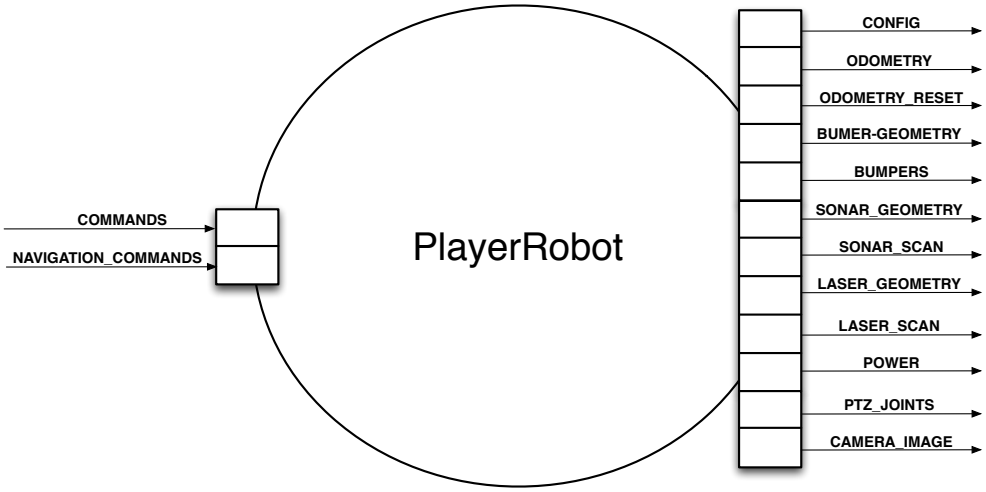

Figura 9.1: Componente *PlayerRobot*

9.2. Componente *GridMap*

Este componente ha sido diseñado para la construcción de un mapa del entorno del robot. Este mapa se va actualizando a medida que el robot navega por diferentes zonas. Para ello el componente *GridMap* hace uso de las lecturas de datos provenientes del robot, concretamente sónars, láser y odometría. Como salida emite una imagen en escala de grises que representa el entorno actual conocido por el robot. La figura 9.2 ilustra con detalle los puertos de entrada y salida de este componente.

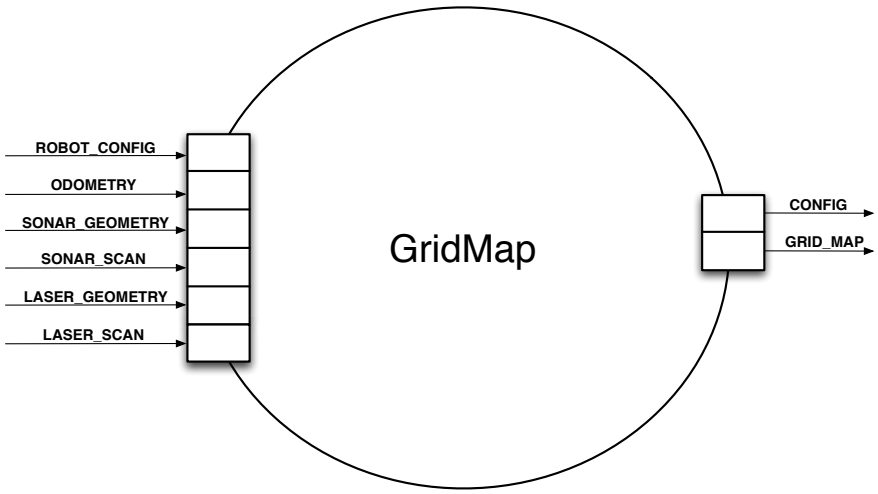

Figura 9.2: Componente *GridMap*

9.3. Componente *NDNavigation*

NDnavigation es un componente que implementa el algoritmo de navegación *Nearness Diagram (ND) Navigation* [Minguez,2004]. Este algoritmo permite seleccionar las zonas a las que se puede mover un robot de forma segura, esto es, evitando las colisiones con los obstáculos existentes en el entorno para alcanzar una zona objetivo. Este algoritmo es capaz de adaptarse a entornos dinámicos. El componente *NDNavigation* hace uso del mapa construido por el componente *GridMap*, la configuración del robot y los comandos de navegación que indican el punto al que se desea desplazar el robot. Como salidas de este componente se obtienen los comandos de navegación para el robot, es decir el siguiente movimiento a efectuar en cada momento. Además este componente emite una serie de datos para la depuración y seguimiento del algoritmo durante su ejecución. La figura 9.3 aclara la interfaz de este componente.

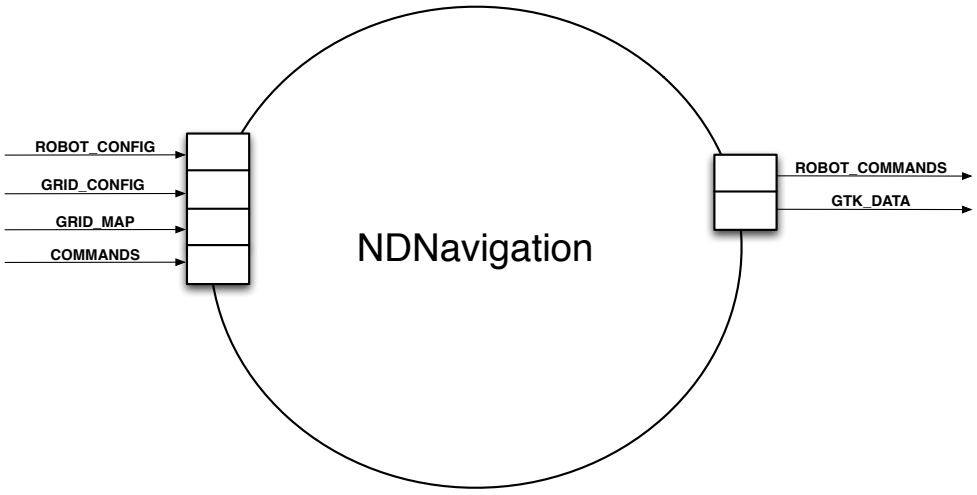

Figura 9.3: Componente *NDNavigation*

9.4. Componente *ShortTermPlanner*

Este componente *CoolBOT* implementa un planificador a corto plazo. Se trata de un componente dedicado a obtener una ruta a seguir en la navegación hasta un objetivo final (*goal*). Para tal fin el componente *ShortTermPlanner* determina la ruta hasta un objetivo, dividiéndola en subobjetivos. Para planificar, este componente hace uso de la configuración del robot en cada momento, además de su odometría y del mapa creado por el componente *GridMap* y el *goal* que se pretende alcanzar. Como resultados *ShortTermPlanner* emite una ruta a seguir como un conjunto de subobjetivos, además de un mapa planificación basado en el creado por *GridMap* en el que se interpretan con diferentes colores las zonas del mapa: esqueleto básico del mapa, obstáculos

y subobjetivos. La planificación que este componente emite se va reajustando a medida que va obteniendo nuevos datos del componente *GridMap* y de *PlayerRobot*, de forma que en ocasiones realiza una replanificación si encuentra una ruta mejor. La figura 9.4 describe las entradas y salidas de este componente.

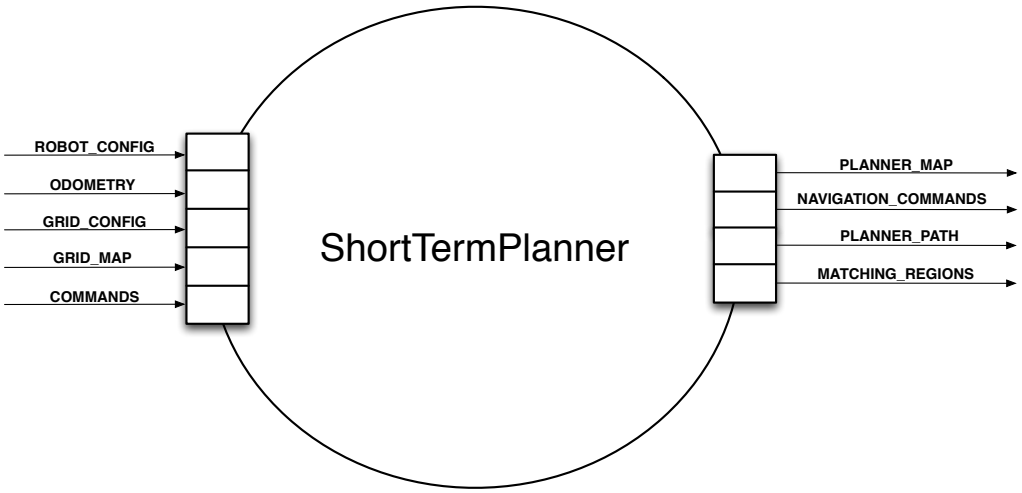

Figura 9.4: Componente *ShortTermPlanner*

9.5. Vista PlayerRobotJavaView

Se trata de una *vista* compuesta de un eje de coordenadas centrado en el robot sobre el que se dibujan líneas representando los datos de los puertos *laser_scan* y *sonar_scan*. Estos valores se calculan en base a la geometría de colocación de estos sensores en el robot recibida desde los puertos *sonar_geometry* y *laser_geometry*. Esta vista además muestra los datos de la odometría y la batería del robot, recibidos desde los puertos *odometry* y *power*. La vista no permite ninguna interacción con el sistema. La figura 9.5 muestra la interfaz de puertos de la *vista playerrobotview*. La imagen de la figura 9.6 muestra la interfaz con la pestaña que muestra la *vista PlayerRobotGtk*. En azul se representa el barrido del láser del robot, mientras que los barridos de los sonars se representan en naranja.

9.6. Vista SphereJavaView

Esta *vista* permite visualizar los datos de una cámara web *QuickCam Sphere* de *Logitech*. Los datos que recibe esta vista son las imágenes de la cámara situada en el robot y los valores de posición de la misma (*Pan, Tilt y Zoom*). Además esta *vista* permite mover la cámara en vertical

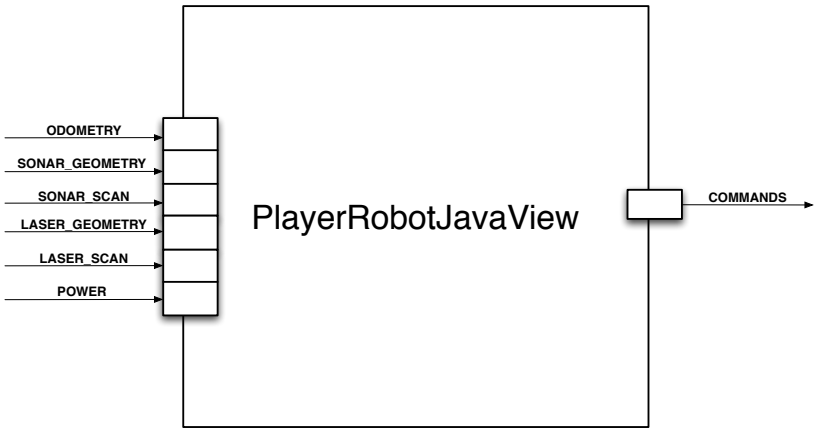

Figura 9.5: Interfaz de puertos de *PlayerRobotJavaView*

y horizontal así como colocarla en el origen (*home*). Para ello, la *vista* ofrece dos barras deslizables (*sliders*) en vertical y horizontal, además de un botón *home*. La imagen de la figura 9.7 refleja una recepción de imágenes en esta vista. En la figura 9.7 se observa la interfaz de puertos de la *vista*.

9.7. Vista GridJavaView

En la figura 9.10 se observa la interfaz de teleoperación con la *vista GridJavaView*. Esta *vista* muestra el mapa elaborado por el componente *GridMap*. En este mapa las zonas en negro son zonas inexploradas por el robot durante la navegación, en gris se observan las zonas descubiertas en el entorno y las paredes y obstáculos se perfilan en blanco. Además, la *vista* realiza sobre el mapa un *raytracing* en color cian, es decir, dibuja segmentos con origen en la posición del robot hasta las zonas de obstáculos en el mapa. En amarillo, la *vista* representa la ruta planificada por el componente *ShortTermPlanner* indicando con un círculo el *goal* final y con cuadrados los subobjetivos. La *vista GridJavaView* ofrece al operador la opción de comandar al robot un objetivo haciendo *clic* sobre el mapa con el botón izquierdo del ratón. Esta acción delega el control al planificador, con lo que el robot iniciará la navegación de forma semiautónoma, pudiendo ser parado en cualquier momento al hacer *clic* con el botón derecho del ratón sobre el mapa. De cara a poder realizar diferentes pruebas, la *vista* permite que la navegación se realice sin que se haga planificación a corto plazo, es decir, sin que el componente *ShortTermPlanner* calcule una ruta hasta el objetivo final. Para ello se debe hacer *clic* con el botón izquierdo del ratón en un punto objetivo sobre el mapa a la vez que se pulsa la tecla *shift*.

La figura 9.9 ilustra los puertos de *GridJavaView*.

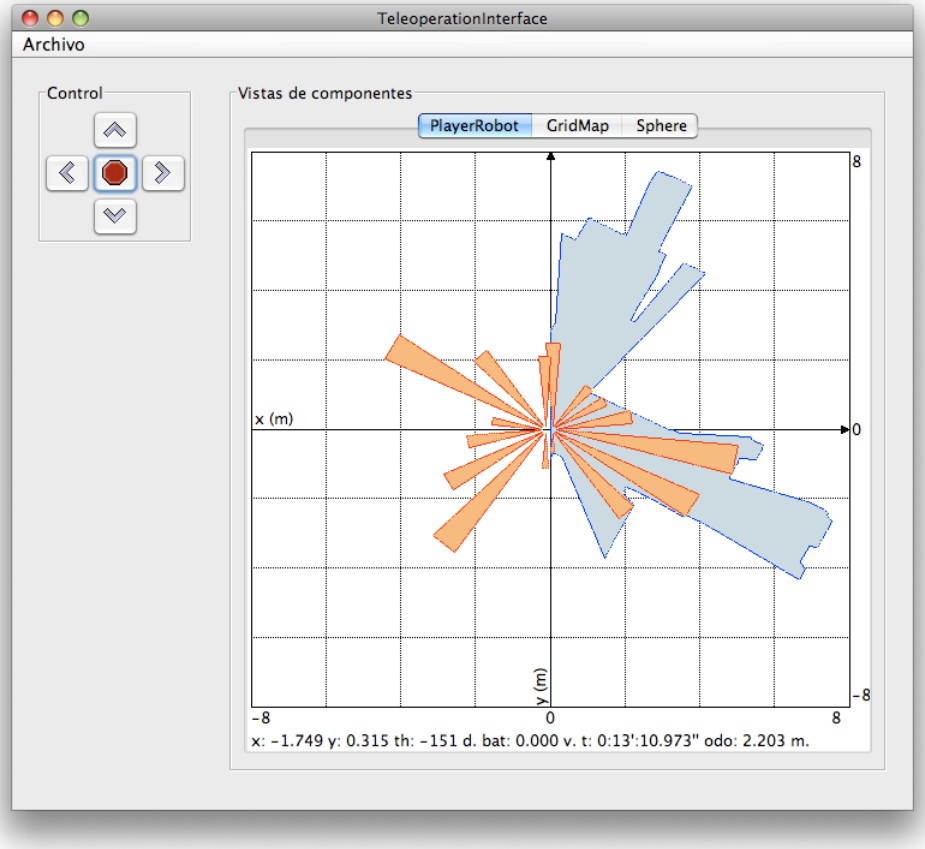

Figura 9.6: Vista *PlayerRobotJavaView*

9.8. Conexionado del sistema completo

La figura 9.11 muestra las conexiones de puertos entre todos los componentes del sistema semiautónomo desarrollado, no se muestran las conexiones con las *vistas*.

9.9. Pruebas realizadas

Con el sistema robótico presentado anteriormente se han realizado diferentes pruebas. Principalmente se ha variado la distribución de los componentes en distintas máquinas, así como de la interfaz de teleoperación.

En ambas pruebas se ha lanzado el sistema completo y su interfaz. Se ha comandado al robot a

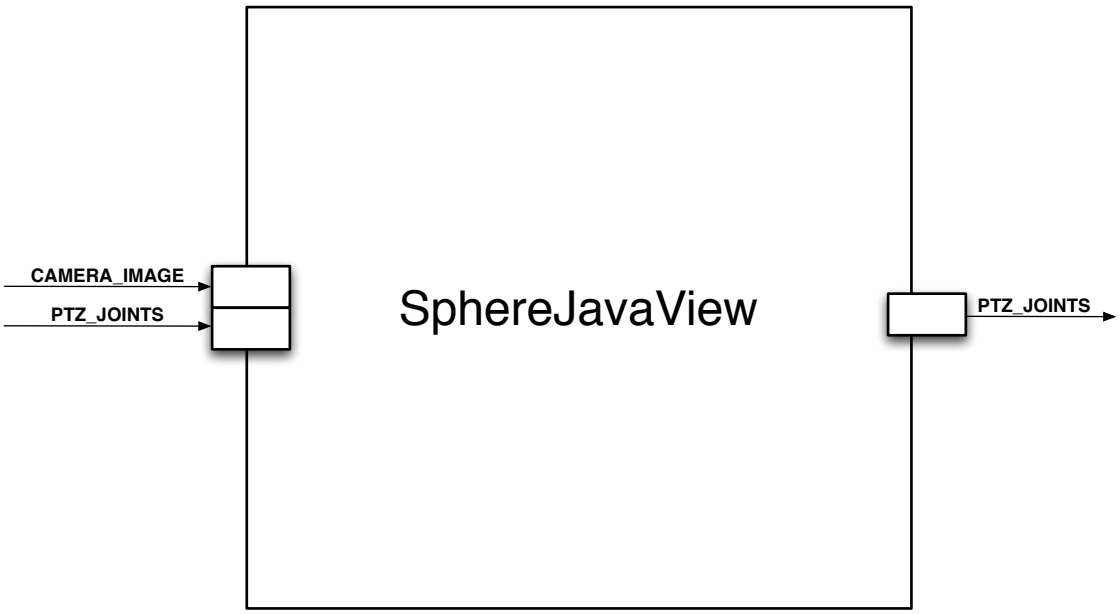

Figura 9.7: Interfaz de puertos de *SphereJavaView*

través de la *vista GridMapJavaView* marcando un objetivo en una zona no descubierta en el mapa e indicando que se haga uso del planificador *ShortTermPlanner*. La tarea comandada es efectuada por el robot de forma autónoma siendo supervisada desde la interfaz.

A continuación se describen los diferentes test realizados.

9.9.1. Test1: Sistema sin uso de red

En esta prueba los componentes del sistema y sus *vistas* residen en una misma máquina. Tanto el conexionado entre los componentes como el conexionado con las *vistas* se ha realizado sin hacer uso de la red *TCP/IP*, con lo que ningún componente o *vista* inicia su soporte de red. El diagrama de la figura 9.12 ilustra la configuración realizada.

9.9.2. Test2: Sistema teleoperado con componentes en local

En este caso se ha optado por una distribución en dos máquinas.

En una primera máquina se localizan todos los componenes del sistema: *PlayerRobot, ND-Navigation, GridMap, ShortTermPlanner*. En esta máquina las conexiones entre componentes se

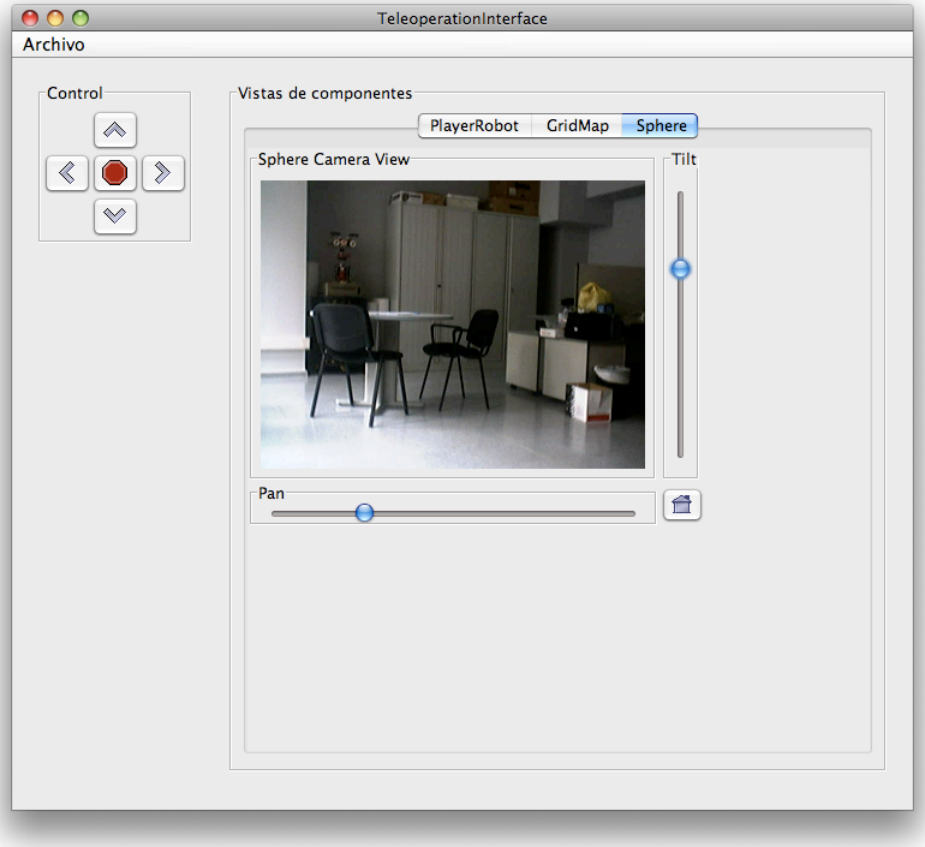

Figura 9.8: Vista *SphereJavaView*

realizarán sin uso de la red *TCP/IP*.

En una segunda máquina reside la interfaz de teleoperación y todas las *vistas* del sistema. Desde la interfaz se realiza el conexionado saliente de las *vistas* con los componentes y el conexionado remoto entrante a las *vistas* desde los componentes.

La figura 9.13 refleja la distribución realizada para este test.

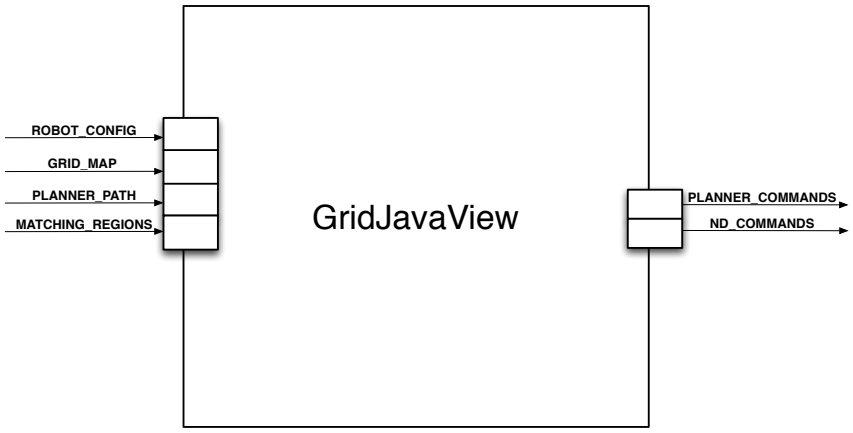

Figura 9.9: Interfaz de puertos de *GridJavaView*

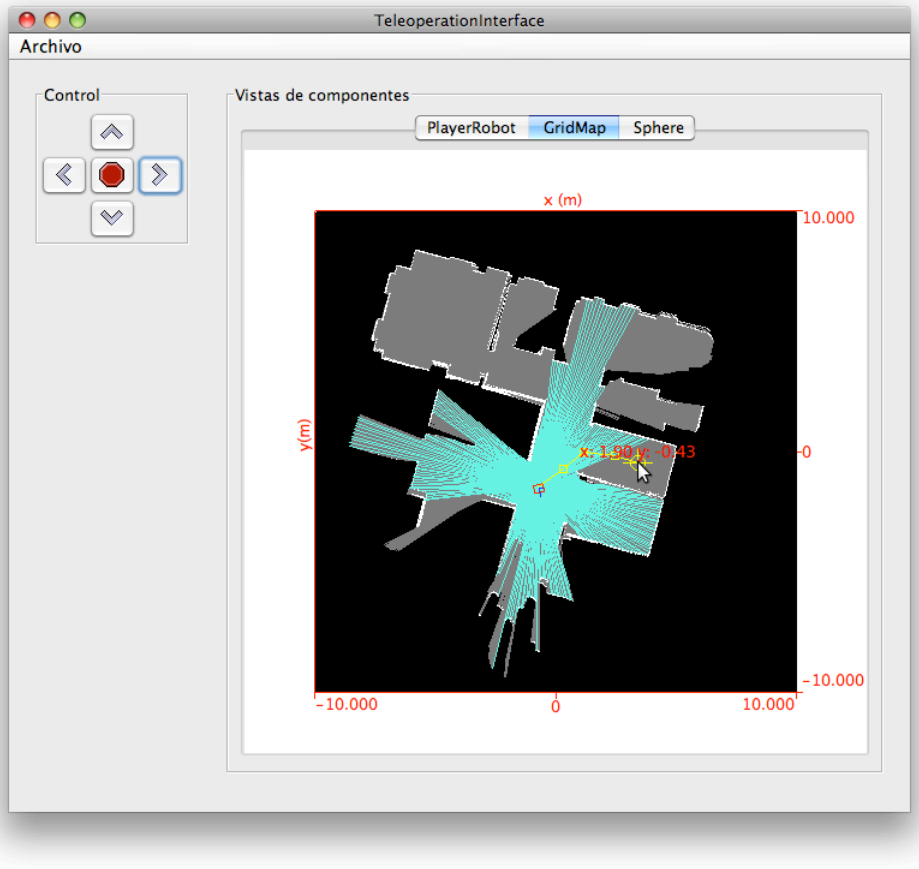

Figura 9.10: Vista *GridJavaView*

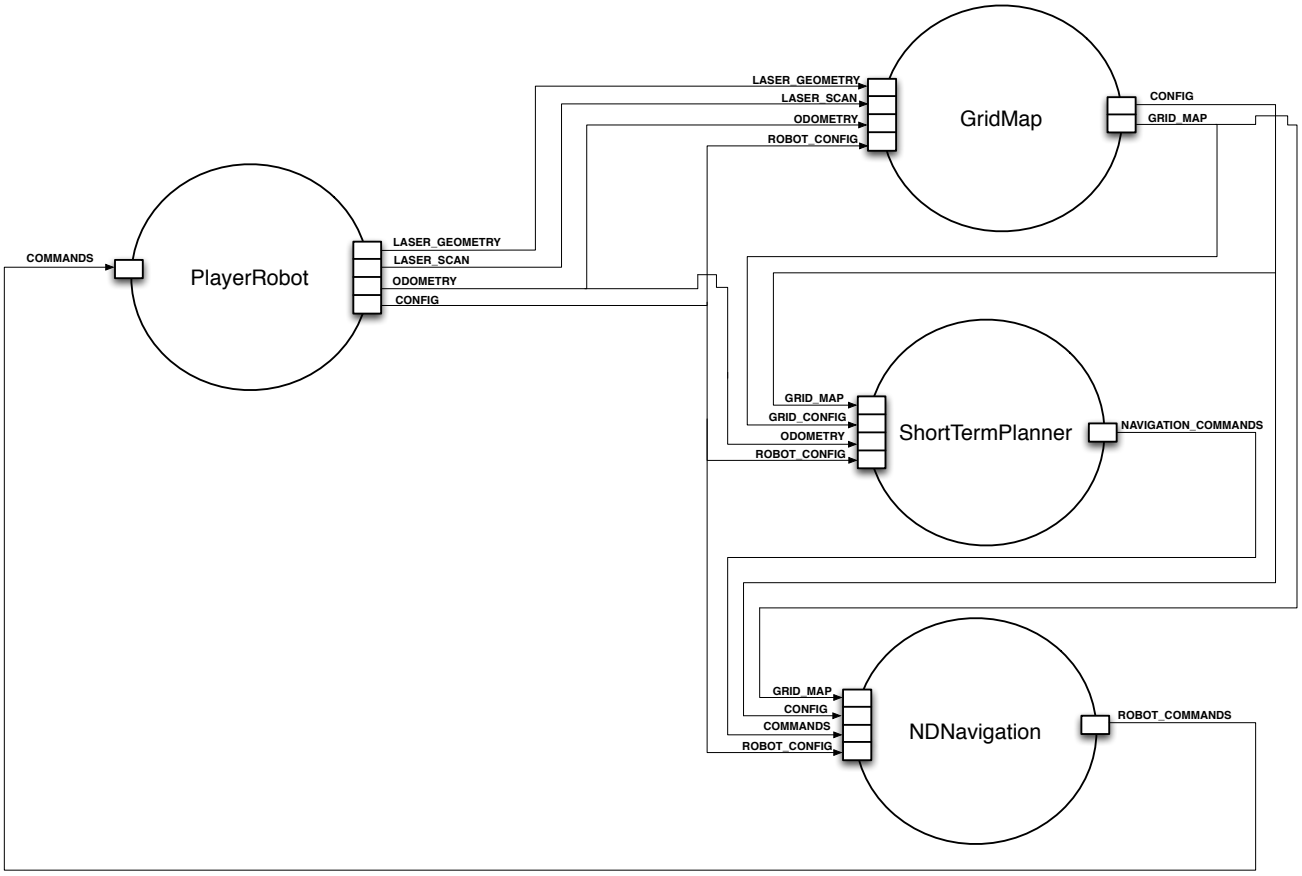

Figura 9.11: Conexionado de componentes del sistema

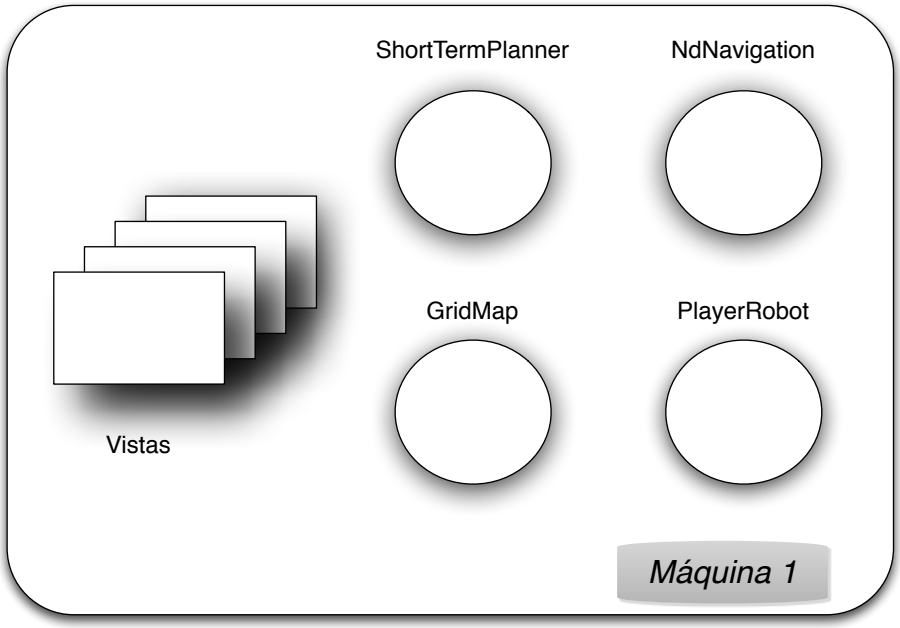

Figura 9.12: Configuración Test1

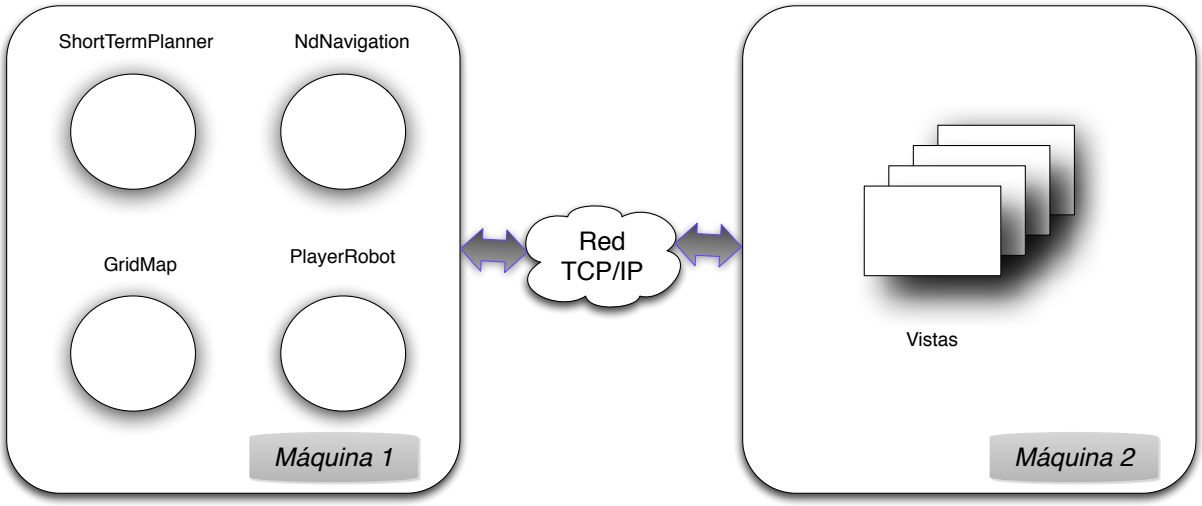

Figura 9.13: Configuración Test2

Capítulo 10

Conclusiones y trabajo futuro

En este capítulo se presentan las conclusiones obtenidas y posibles líneas de desarrollo futuras en base al trabajo llevado a cabo durante su desarrollo, y a las distintas pruebas realizadas durante la evolución del mismo.

10.1. Conclusiones

Las conclusiones extraídas de este proyecto se presentan separadas en dos grupos: conclusiones generales y conclusiones relacionadas con las pruebas realizadas, y el objetivo explícito de este proyecto que era el de poner en funcionamiento un sistema robótico teleoperado para la realización de visitas guiadas remotas.

10.1.1. Conclusiones generales

Diseño y Desarrollo de una Infraestructua Software de Teleoperación y Computación Distribuida

El mecanismo de diseño aplicado al sistema realizado reduce la complejidad en el tratamiento de problemas relacionados con la robótica móvil. En general se trata de un tarea de divide y vencerás (*divide and conquer*). Las acciones y problemas necesarios a resolver en la creación de un sistema teleoperado para poder llevar a cabo visitas guiadas remotas se han descompuesto en componentes. Cada componente está encargado de resolver un problema determinado en la conducta del sistema. Una de las conductas destacables es la posibilidad de ser teleoperado. La interfaz diseñada satisface esta necesidad del sistema, permitiendo en todo momento una monitorización de toda la actividad relevante, así como comandar al robot alrededor de un

escenario (bien sea a través del control directo por parte del usuario o indicando una zona del mapa a donde deberá desplazarse de manera semiautónoma).

Con esto se ha logrado satisfacer el objetivo final de este proyecto: obtener un sistema teleoperado para la realización de visitas guiadas remotas.

Portabilidad.

Los mecanismos de comunicación utilizados para permitir la teleoperación y distribución del sistema han sido los proporcionados por el *framework ACE* que ofrece una capa de abstracción del sistema operativo. Esto facilita la portabilidad del sistema desarrollado capacitándolo para ser extensible a todos los sistemas operativos que *ACE* soporta. El uso de los componentes software del sistema (componentes *CoolBOT*) en máquinas con diferentes arquitecturas se ve favorecido por el empleo de una representación intermedia de los datos intercambiados en las comunicaciones. La implementación de operaciones de *marshalling* desarrolladas permiten serializar los datos representándolos según el estándar *CDR* de *CORBA*.

Eficiencia.

Con el objetivo de satisfacer la necesidad de comunicaciones remotas se ha diseñado el protocolo *DC3P*. Este protocolo maneja un conjunto de mensajes simples. Cada paquete ofrece la información imprescindible para los requerimientos de comunicación entre componentes *CoolBOT*. Las reglas de procedimiento *DC3P* que requieren un mayor trasiego de paquetes se reducen a cuatro, destinadas al conexionado y desconexionado de puertos de componentes. Sin embargo, las operaciones de conexionado/desconexionado no acontecen con alta frecuencia, principalmente tienen lugar al inicio y finalización de la ejecución del sistema, aunque también esporádicamete durante su funcionamiento. Por otro lado, las operaciones que se efectúan reiteradamente durante la ejecución son las de envíos de paquetes de puertos. Estas operaciones involucran un único paquete por cada envío de datos. El soporte de comunicaciones por red añadido a los componentes *CoolBOT* involucra a dos hilos de puertos: *Input Network Thread (INT)* y *Output Network Thread (ONT)*. Cada componente con soporte de red supone por tanto la ejecución de un mínimo de 4 hilos: hilo *main*, hilo *timer*, *INT* y *ONT*. Sin embargo los hilos añadidos para el soporte de red no suponen gran costo, al seguir éstos un modelo de ejecución de máquina de flujo de datos. El hilo *ONT* se encuentra siempre suspendido y sólo se activa cuando sea requerido algún envío de paquetes de puerto por la red. En cuanto al hilo *INT* se encuentra suspendido, activándose cada cierto tiempo para chequear la actividad entrante por la red. Esta tarea es muy simple y en cuanto existe actividad el hilo *INT* se limita a entregar los datos al *IBox* del componente, siendo el hilo *main* el encargado del procesado de la información.

Escalabilidad y modularidad.

- Escalabilidad y modularidad a nivel de protocolo *DC3P*

  En caso de requerimientos futuros en las comunicaciones, el diseño del protocolo *DC3P* basado en el patrón *prototype*, facilita añadir nuevos tipos de mensajes y la modificación de los existentes.

- Escalabilidad y modularidad a nivel de *marshalling/demarshalling*

  Los mensajes de datos se adaptan, sin necesidad de cambio alguno, a cualquier nuevo paquete de puerto que pudiera diseñarse. Como requerimiento mínimo en el diseño de nuevos paquetes de puertos se establece la definición de funciones que preparen los datos para su envío por la red (*marshalling/demarshalling*): *toBytes* y *fromBytes*. Como ejemplo de la modularidad, un usuario puede fácilmente aplicar algoritmos de compresión de los datos antes y después del marshalling y demarshalling. Usando las funciones de librerías como *libJPEG* es posible añadir a las funciones de *marshalling/demarshalling* la capacidad de que automaticen la compresión de imágenes manteniendo la misma interfaz.

  sin requerir modificiaciones en las operaciones como puede ser la compresión de imágenes en diferentes formatos

Transparencia de las comunicaciones.

Es de gran importancia lograr abstraer a un usuario teleoperador de las comunicaciones que se llevan a cabo. Esto permite que para dicho usuario utilizar un sistema distribuido o un sistema en local requiera las mismas acciones, sin distinción alguna por el hecho de que el sistema se localice en diferentes máquinas. El proyecto realizado obtiene un sistema que cumple estas características. Además, como valor añadido, para un desarrollador de sistemas robóticos con componentes *CoolBOT*, la infraestructura de red es transparente. Las operaciones que pudiera requerir en el desarrollo se encuentran encapsuladas abstrayendo totalmente el manejo del protocolo de comunicaciones y el *marshalling* de datos.

Sistematización en la creación de componentes.

El soporte de red creado es totalmente adaptable a cualquier tipo de componente. Para crear la infraestructura de red de un componente, es necesario conocer únicamente su interfaz de puertos. Además, para iniciar la red se necesita un puerto *TCP* por el que el componente escuchará. A partir de estos datos se pueden construir componentes de manera completamente sistemática.

Vistas de componentes.

Para la creación de *vistas* de componentes se han diseñado un modelo basado en el uso de *sondas*. Se ha conseguido desacoplar la *vista* del *modelo* y del *control*, siguiendo la arquitectura *Modelo Vista Controlador (MVC)*. Esto permite la creación de diferentes *vistas* sin la necesidad de variar el modelo de datos (paquetes de puerto), ni el control (imbuído en las estructuras señalizables *IBox* y *OBox* de los componentes). Además la creación de *vistas*, al igual que la de componentes con red, es una tarea perfectamente sistematizable.

Portabilidad del *framework CoolBOT* a MacOSX

Aunque portar *CoolBOT* a MacOSX no era un objetivo propuesto durante la etapa de inicio del desarrollo de este proyecto, se ha llevado a cabo debido a que es uno de los sistemas operativos soportados por el equipo utilizado durante este trabajo. Además del *framework*

CoolBOT, se han portado todos los componentes y las *sondas*, haciendo que el sistema robótico sea más portable. En la actualidad está soportado por los sistemas operativos GNU/Linux, Win32 y MacOSX.

10.1.2. Conclusiones experimentales

Mediante la integración de un sistema de navegación segura para un robot móvil que realiza visitas guiadas remotas, se han puesto de manifiesto las capacidades de teleoperación y computación distribuida que proporciona el soporte de red DC3P y su integración el framework *CoolBOT* que se ha realizado en este proyecto. En base a la experiencia recogida a lo largo de estas pruebas consideramos de interés incidir en las siguientes conclusiones relacionadas con la realización experimental en la fase de pruebas.

Componentes Distribuidos La integración del soporte de red DC3P en *CoolBOT* supone la posibilidad de organizar un sistema distribuido de computación donde los componentes que integran el software de un sistema determinado pueden situarse o instanciarse arbitrariamente en algunas de las diversas máquinas que pueden constituir el sistema.

Interfaces de Teleoperación Igualmente es posible diseñar y desarrollar vistas *CoolBOT* que integran dicho soporte red, de manera que éstas también constituyen interfaces de monitorización y control de un sistema integrados por componentes. Al igual que los componentes, estas vistas pueden vincularse a cualquier máquina que conforme el entorno de computación distribuida de dicho sistema, de manera que dicha infraestructura de red convierta a cada una de ellas en un interfaz potencial de teleoperación para un sistema que las integre.

Diseño e Integración de Sistemas basados en Componentes Existen multitud de escenarios y distribuciones posibles de sistemas integrados por componentes *CoolBOT*. El desarrollador e integrador de sistemas robóticos *CoolBOT* tiene total libertad para decidir la mejor distribución del sistema en base a los recursos de computación distribuidos de los que disponga. Así, dado un sistema robótico específico a integrar, se diseñará una distribución que se adapte a las necesidades del mismo. Sin embargo, pueden existir componentes que integrados en determinadas circunstancias en algún sistema impliquen requerimientos demasiado exigentes para su funcionamiento en diferentes máquinas, como por ejemplo el del bucle reactivo de control de un algoritmo de evitación de obstáculos que debe ejecutarse a frecuencias de funcionamiento de al menos 5 Hz. Todas estas consideraciones, además de otras como los tipos de red que integran el entorno distribuido, así como la potencia computacional de las distintas máquinas disponibles en el sistema, junto con los requerimientos de funcionamiento del mismo determinan el diseño e integración final que presenta el software de control de un sistema robótico integrado por componentes y vistas *CoolBOT*.

10.2. Trabajo futuro

En este proyecto se describe el desarrollo de la infraestructura de red *DC3P* y su integración en el framework *CoolBOT*, además de la utilización de la misma para la construcción de *vistas* y componentes *CoolBOT* distribuidos. Todo ello se ha ilustrado integrando un sistema compuesto por componentes y vistas que permite la teleoperación de un sistema robótico orientado a la realización de visitas guiadas remotas. A partir de este trabajo surgen diferentes e interesantes líneas de desarrollo futuras que pasamos a enumerar y detallar brevemente a continuación.

Automatización de la generación de esqueletos C++ componentes, sondas y *vistas CoolBOT*

Actualmente existe una herramienta para la generación de esqueletos de componentes *CoolBOT* [Santana-Jorge,2007] [ref-TMF-Francisco]. La generación de código que esta herramienta proporciona no contempla el soporte de red ni las *sondas* de componentes. Para incrementar la facilidad de creación de estos elementos se plantea la posibilidad de que esta herramienta automatice también la construcción de *sondas*, así como la inclusión de la insfraestrutura de red para los componentes *CoolBOT*.

Estudio y caracterización de retardos de comunicación

Este proyecto se ha centrado en la elaboración de una infraestructura de computación distribuida integrada en el framework orientado a componentes *CoolBOT*. Como posible estudio futuro se propone la posibilidad de analizar y caracterizar formal y experimentalmente los retardos de comunicación en función de los distintos tipos redes que puedan constituir un sistema distribuido dado.

Servicio de nombres de componentes distribuidos

El modelo actual de identificación de componentes remotos requiere el conocimiento previo de las direcciones *IP* o nombres DNS de la máquina y de los puertos de escucha *TCP* de los componentes *CoolBOT* que conforman un sistema. Un desarrollo de gran interés puede ser la realización de un servicio de nombres que proporcione un registro de las instancias de componentes que se encuentren activos en la red, permitiendo identificar a éstos por un nombre. De la misma forma dicho servicio de nombres podría almacenar información acerca de la interfaz externa de puertos de entrada y salida de cada componente, así como de una descripción de los mismos.

***Grid Computing*.**

A pesar de que *CoolBOT* es un *framework* destinado al desarrollo de sistemas robóticos, el modelo de construcción de estos sistemas a partir de componentes software se asemeja al utilizado en la computación basada en *grids* o *clusters*. Por tanto, el uso de *CoolBOT* es extensible como herramienta para diseñar y construir cualquier tipo desistema distribuido. Se considera interesante la posible utilización de *CoolBOT* en el desarrollo de aplicaciones distribuidas siguiendo un modelo de computación basado en *grids*.

Apéndice A

Manuales de usuario

A.1. Guía de creación de comandos del protocolo DC3P

El protocolo DC3P ha sido diseñado con el objetivo de que sea pequeño, simple y escalable. Las dos primeras premisas aluden a la eficiencia en la transferencia de mensajes mientras que la tercera busca la fácil integración de nuevos comandos del protocolo para un usuario programador que desee ampliar las funcionalidades del mismo.

En esta sección se van a indicar aquellos ficheros que deben ser modificados para añadir un nuevo comando al protocolo DC3P, así como dónde incluir los ficheros de usuario que contengan la implementación del nuevo comando.

Se va a suponer que se ha seguido la guía de instalación de *CoolBOT* (capítulo A.3), así que el directorio raíz del que cuelgan los distintos componentes se supondrá

```
\$HOME/cvs-projects
```

A.1.1. Directorio

Los ficheros que conforman el protocolo DC3P se encuentran en el directorio

```
\$HOME/cvs-projects/coolbot/network
```

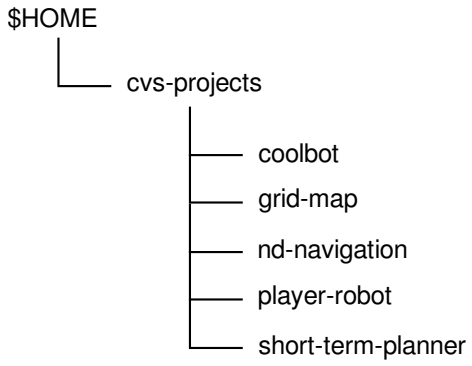

Figura A.1: Estructura del árbol de directorios de CoolBOT

En este directorio es donde el usuario deberá incluir los ficheros relativos al nuevo comando que ha desarrollado para el protocolo DC3P. Los nombres de los ficheros que se aconsejan utilizar para seguir la línea de estilo de *CoolBOT* son:

- coolbot_UserCommand.h
- coolbot_UserCommand.cpp

Nota: En esta guía se empleará el término UserCommand como nombre de clase ejemplo y UserCommandAttribute como atributo de la misma para ilustrar y facilitar al usuario la creación de su propio comando del protocolo DC3P.

A.1.2. Clase UserCommand - Fichero .h

Inclusión de ficheros

En el fichero de cabecera debemos incluir los ficheros necesarios para heredar las interfaces comunes a todos los paquetes del protocolo DC3P y para permitir la inicialización de variables estáticas. Así, las inclusiones que debe contener este fichero son al menos:

```
#include "network/coolbot_commandbody.h"
#include "util/coolbot_staticinit.h"
```

Herencia y namespace *CoolBOT*

La clase que represente dicho comando debe heredar de forma pública de *PacketBody*, para que herede así las interfaces de *CloningInterface*, *DeepCopyInterface*, *PackingInterface*, *NamingInterface* y *DebuggingInterface*. Además, en su parte pública debe declararse una *MACRO* ó *Enum* donde indicarán los bytes que ocupan los atributos de la clase que se enviarán por la red, utilizando como referencia los tamaños de los tipos que aparecen en la tabla 5.2. Es conveniente incluir todo el código de la clase dentro del *namespace* de **CoolBOT** para que forme parte del mismo ámbito de éste espacio de nombres (ver imagen A.2).

```
namespace CoolBOT
{
   class UserCommand: public PacketBody
   {
      public:
            enum{USER_COMMAND_LENGTH = /* tamaño del cuerpo en bytes */}

            UserCommand(); //Constructor

            ~UserCommand(); //Destructor

             CloningInterface* clone() const { ... }

             bool deepCopy(DeepCopyingInterface* pThing) { ... }

             const char* name() { ... }

             UserCommand & operator=(const UserCommand & packet) { ... }

             bool toBytes(ACE_OutputCDR &oCDR) { ... }

             bool fromBytes(ACE_IntputCDR &iCDR) { ... }

             int packingMaxLength() { ... }

            ...
   };
}
```

Figura A.2: Vista general de las funciones miembro públicas de un comando del protocolo DC3P

Las interfaces heredadas de *PacketBody* deben implementarse de la siguiente manera:

- CloningInterface

Clase abstracta que define una interfaz para que una clase pueda ser clonada. Esta interfaz le proporciona a una clase ser un prototipo usando el patrón de diseño *prototype*.

```
CloningInterface* clone() const
{
   return new COMMAND_USER(*this);
}
```

Figura A.3: Interfaz de clonado

- DeepCopyingInterface

  Clase abstracta que define una interfaz para que una clase pueda ser copiada (estructura propia más estructuras enlazadas).

```
bool deepCopy(DeepCopyingInterface* pThing)
{
    if(!pThing) return false;
        COMMAND_USER* pPacket=static_cast<COMMAND_USER*>(pThing);
    if(this!=pPacket) _assign_(*pPacket);
        return true;
}
```

Figura A.4: Interfaz de copia

- PackingInterface

  Clase abstracta que define una interfaz para que una clase pueda ser empaquetada (función *toBytes*) y desempaquetada (función *fromBytes*) en un bloque de bytes. Además el tamaño de la clase empaquetada debe computarse mediante la función *packingMaxLength*.

  Como vimos en la imagen 7.3 del capítulo 7.1 las clases que heredan de *PackingInterface*, implementan las funciones siguientes:

  - *toBytes()* - Donde haremos una llamada a esta función para cada uno de los atributos pertenecientes a la clase **COMMAND_USER** que deben viajar por la red. Por regla general estos atributos coincidirán con los necesarios para construir un objeto idéntico de la misma clase en el receptor. Es decir, que aquellos atributos que puedan obtenerse a partir de otros sería conveniente no enviarlos para no sobrecargar el canal de comunicación con información superflua.
  - *fromBytes()* - Donde realizaremos una llamada a esta función para desempaquetar cada uno de los atributos que hemos empaquetado previamente con la función *toBytes()*. Como comentamos previamente, puede que no todos los atributos de la clase se asignen directamente con las llamadas a esta función, sino que, para aquellos que se obtengan a partir de otros, existirán funciones que nos permitan obtenerlos cuyo valor devuelto sea asignado al atributo en cuestión.

```
bool toBytes(ACE_OutputCDR &oCDR)
{
    CoolBOT::toBytes(_COMMAND_USER_ATRIBBUTE_,oCDR);
    return oCDR.good_bit();
}

bool fromBytes(ACE_InputCDR &iCDR)
{
    CoolBOT::fromBytes(_COMMAND_USER_ATRIBBUTE_,iCDR);
    return iCDR.good_bit();
}

int packingMaxLength()
{
    return COMMAND_USER_LENGTH // + Length_of_array_attributes;
}
```

Figura A.5: Interfaz para el envío/recepción de datos por la red

- *packingMaxLength()* - Esta función debe devolver el tamaño de los atributos de la clase que van a enviarse por la red. Como hemos observado previamente en la imagen A.2, es necesario declarar un *enum* al que se le asignará el tamaño de aquellos atributos que se enviarán por la red. La nomenclatura que se aconseja seguir para el nombre del elemento de este *enum* es *COMMAND_USER_*LENGTH.

- NamingInterface

  Clase abstracta que define una interfaz para asignar un nombre a una clase.

```
const char* name() { return "CoolBOT::COMMAND_USER"; }
```

Figura A.6: Interfaz de *naming*

- DebuggingInterface

```
ostream& debug(ostream& os) const
{
    os << "CoolBOT::COMMAND_USER" << endl;
    os << "COMMAND_USER_ATRIBBUTE = "<< _COMMAND_USER_ATRIBBUTE_ <<endl;
    return os;
}
```

Figura A.7: Interfaz de depuración

La interfaz *DebuggingInterface* nos permite definir una interfaz de depuración para cada uno de los objetos que hereden de la misma. En este caso, volcaremos sobre el *stream* de salida

"*os*" aquella información que nos sirva para depurar nuestro código. Se han seguido unas directrices para homogeneizar y simplificar la depuración de todas y cada una de las clases:

- CoolBOT::*Nombre de la clase*
- *Nombre del atributo*
- *Valor del atributo*

Parte privada, función *assign*

Como vimos en la sección A.1.2, existen un conjunto de funciones miembro públicas que son necesarias implementar en cada una de las clases que representan un comando del protocolo DC3P.

Una de ellas, corresponde al operador de asignación, cuya implementación para todas y cada una de las clases comando desarrolladas podemos observar en la imagen A.8, donde, si se cumple la condición, se hace una llamada a una función privada *_assign_()*. Sabemos que esta función es privada porque su nombre comienza y termina en guión bajo ("_"), y ésta es la nomenclatura indicada en la guía de estilo de *CoolBOT* (REFGUIADEESTILOCOOLBOT) para miembros y atributos privados.

```
USER_COMMAND& operator=(const USER_COMMAND& packet)
  { if(this!=&packet) _assign_(packet); }
```

Figura A.8: Operador de asignación

La función *_assign_(const USER_COMMAND&)* se encarga de asignar los atributos de un objeto de tipo *(const USER_COMMAND&)* al objeto que estemos manejando en ese momento. Su implementación sería similar a:

```
void _assign_(const USER_COMMAND& packet)
{
   _USER_COMMAND1_=packet._USER_COMMAND1_;
   _USER_COMMAND2_=packet._USER_COMMAND2_;
   ...
    _USER_COMMANDN_=packet._USER_COMMANDN_;
}
```

Figura A.9: Assign

Parte privada, inicialización de variables estáticas

Como se comentó en la sección A.1.2, una de las interfaces que heredan de *PacketBody* es *CloningInterface*. Aunque se comentó en dicho capítulo que se explicaría más detalladamente en la

sección 8, es necesario introducir brevemente su utilidad para comprender el resto de funciones que deben implementarse en la parte privada de la clase del nuevo comando para el protocolo DC3P.

Con la función *clone*, proporcionada por la clase *CloningInterface*, podemos realizar el clonado de objetos utilizando un objeto ya creado en nuestra aplicación. Así, con la función *prototype()* indicada al final de la sección A.1.2, podemos crear un "prototipo", es decir, un puntero a una zona de memoria estática que apunta a un objeto de tipo padre (*PacketBody*), y mediante la función *clone*, obtendremos una réplica del objeto hijo sobre el que estamos trabajando, de tal manera, que podremos manejar un objeto hijo a partir de una instancia del padre, reduciendo drásticamente el número de objetos que deben crearse y el número de inicializaciones que deben llevarse a cabo.

Por todo esto es necesario funciones que lleven a cabo la inicialización y destrucción de variables estáticas, las cuales pueden observarse en la figura A.10

```
private:
    static USER_COMMAND* _pPrototype_;

    static void _staticInitialization_() { _pPrototype_=new USER_COMMAND(); }

    static void _staticFinalization_() { if(_pPrototype_) delete _pPrototype_; }

     friend class StaticInit<USER_COMMAND>;
```

Figura A.10: Inicialización y finalización de variables estáticas

Para garantizar que la inicialización de las variables estáticas se lleva a cabo antes de usar el objeto de la clase *USER_COMMAND*, hacemos una llamada a la función de inicialización comentada previamente, desde fuera de la clase, tal y como se ve en la ilustración A.11

A.1.3. Clase USER_COMMAND - Fichero .cpp

En el fichero *USER_COMMAND.cpp* es donde debemos inicializar todas las variables estáticas que hayamos definido en nuestra clase. Como mínimo este fichero contendrá la inicialización de la variable *_pPrototype_* a nulo, para poder instanciar un prototipo de la clase *USER_COMMAND* que pueda ser clonado posteriormente, como vimos en la sección A.1.2. Así, las inclusiones de ficheros y las inicializaciones estáticas dan como resultado un fichero de implementación similar al observable en la imagen A.12

```
namespace CoolBOT
{
   class USER_COMMAND: public PacketBody
   {
      public:
            ...

      private:
            ...

   };
}

namespace:
{
    CoolBOT::StaticInit<CoolBOT::USER_COMMAND> JOIN2(staticInit,USER_COMMAND);
}
```

Figura A.11: Inicialización de variables estáticas

A.2. Guía de creación de paquetes de datos

Como se ha descrito previamente en diversas partes de este documento lós ítems de información que se comunican los componentes entre sí se denominan *port packets*. Para que un usuario pueda crear su propio paquete de puerto (*port packet*) debe seguir los siguientes pasos:

Imaginemos que el usuario va a crear un nuevo *port packet* cuyo nombre va a ser *NewUserPacket*. Además, esta clase va a contener tres atributos:

- data: atributo de tipo *Integer*.
- boolean: atributo de tipo *Bool*.
- myClass: atributo de un tipo construido por el usuario que denominaremos *UserClass*.

En primer lugar nos creamos un fichero cabecera con el nombre *coolbot_newuserpacket.h* en el directorio

```
\$HOME/cvs-projects/coolbot/component/
```

En este fichero será donde el usuario definirá los métodos y atributos que tendrá su clase. Antes de empezar a implementarla, es necesario incluir los siguientes ficheros:

```
#include "coolbot_compiler_adjustments.h"

#ifndef COOLBOT_USER_COMMAND_CPP
#define COOLBOT_USER_COMMAND_CPP
#include "network/coolbot_USER_COMMAND.h"

namespace CoolBOT
{
    USER_COMMAND* USER_COMMAND::_pPrototype_=NULL;

    ...

}

#endif //COOLBOT_COMMAND_USER_COMMAND_CPP
```

Figura A.12: Fichero de implementación USER_COMMAND.cpp

- `#include "ports/coolbot_portpacket.h"`

  Añade un conjunto de interfaces que deben compartir todos los *port packets*.

- `#include "util/coolbot_staticinit.h"`

  Incluye funciones para la inicialización de variables estáticas.

Como se comentó anteriormente en el apartado A.1.2, es necesario implementar todas las interfaces heredadas de las clase *PortPacket*, así que se recomienda la lectura de dicho apartado para entender cómo y porqué deben ser implementadas en nuestra clase *NewUserPacket*. Así pues, nuestra clase deberá heredar de *PortPacket* e implementar todas las interfaces que nos proporciona esta clase, más los métodos necesarios para lograr la funcionalidad esperada de la misma. De esta manera, la parte pública de nuestra clase resulta algo similar a:

```
#ifndef COOLBOT_NEWUSERPACKET_H
#define COOLBOT_NEWUSERPACKET_H
#include "ports/coolbot_portpacket.h"
#include "util/coolbot_staticinit.h"
#include "userclass.h"

namespace CoolBOT
{

    class NewUserPacket: public PortPacket
    {
```

```
public:

    NewUserPacket( ... ) { ... }

    ~NewStatePacket() {}

    CloningInterface* clone() const { ... }

    bool deepCopy(DeepCopyingInterface* pThing) { ... }

    bool toBytes(ACE_OutputCDR &oCDR)
    {
        CoolBOT::toBytes(_data_,oCDR);
        CoolBOT::toBytes(_boolean_,oCDR);
        CoolBOT::toBytes(_myClass_,oCDR);
    }

    bool fromBytes(ACE_InputCDR &iCDR)
    {
        CoolBOT::fromBytes(_data_,iCDR);
        CoolBOT::fromBytes(_boolean_,iCDR);
        CoolBOT::fromBytes(_myClass_,iCDR);
    }

    int packingMaxLength()
    {
        return CoolBOT::packingMaxLength(_state_) +
               CoolBOT::packingMaxLength(_boolean_) +
               CoolBOT::packingMaxLength(_myClass_);
    }

    ostream& debug(ostream& os) const
    {
        return os << "CoolBOT::NewUserPacket" << endl
                  << "_data_= " << _data_ << endl
                  << "_boolean_= " << _boolean_ << endl
                  << "_myClass_= " << _myClass_ << endl;
    }

    const char* name() { return "CoolBOT::NewUserPacket"; }

    void setData( ... ) { ... }
```

```
int getData() { return _data_; }

void SetBoolean() { ... }

bool getBoolean() { return _boolean_; }

static PortPacket* prototype() { return _pPrototype_; }
```

Es importante entender cómo implementar las interfaces heredadas de *PackingInterface*, explicadas también en el apartado anteriormente citado (A.1.2). La clase definida por el usuario *UserClass*, deberá definir análogamente sus funciones *toBytes, fromBytes y packingMaxLength*, para poder llamarlas directamente cómo se puede ver desde el código expuesto previamente.

Asímismo, los métodos para poder observar/modificar una variable de tipo *UserClass* suponemos que se encuentran definidos de manera pública en el fichero "*userclass.h*".

Respecto a la parte privada de la clase, se añadirán los atributos propios de la clase *NewUserPacket* (en este caso, *_data_, _boolean_ y _myClass_*, así como la variable estática *_pPrototype_* y sus funciones de inicialización y finalización al igual que como se explicó en el apartado A.1.2. Análogamente, en el fichero "*userclass.cpp*" deben incluirse todas las inicializaciones de las variables estáticas, incluida la variable *_pPrototyope_*.

Un ejemplo del fichero de implementación, *userclass.cpp*, sería:

```
#include "coolbot_compiler_adjustments.h"

#define COOLBOT_NEWUSERPACKET_CPP
#include "component/newuserpacket.h"

namespace CoolBOT
{

       NewUserPacket* NewUserPacket::_pPrototype_=NULL;

}
```

A.3. Guía de instalación de *CoolBOT* y componentes

En esta sección se especifican los pasos a seguir para instlar el *framework CoolBOT* y un conjunto de componentes que implementan un sistema de evitación de obstáculos con planificador a corto plazo.

Este sistema de evitación hace uso de insfraestructura de red, por lo que es necesaria la librería *ACE* (libace). Además se requiere añadir al *.bashrc* una variable de entorno para la compilación del sistema:

```
ACE_ROOT=<directorio instalacion ACE, usualmente /usr/share/ace>
export ACE_ROOT
```

Tanto *CoolBOT* como el resto de componentes está en el servidor *git new-mozart.dis.ulpgc.es/home/git-user/git-repository/*. A continuación se pasa a mostrar como ser realiza la instalación de *CoolBOT* y de los componentes. Se aconseja crear un directorio y dentro del mismo realizar las instalación de todo el software bajado desde el servidor *GIT*. En el siguiente documento dicho directorio se va a suponer que dicho directorio es *git-projects* normalmente situado en el directorio raíz de la cuenta de usuario, normalmente indicado por la variable de entorno *$HOME* .

A.3.1. *CoolBOT*

Descarga

Situarse en el directorio *git-projects*, y ahí bajarse *CoolBOT* desde el servidor *GIT* con la siguiente orden:

```
git clone ssh://<usuario>new-mozart.dis.ulpgc.es:/home/git-user/git-repository/ \
coolbot.git
```

Esta orden crea el directorio *coolbot* en *git-projects*.

Variable de entorno

Es preciso incluir en el *.bashrc* la variable de estado *COOLBOT_PATH* indicando donde se encuentra *CoolBOT* y donde se encontrará la librería de *CoolBOT* una vez se compile. Lo que hay

que introducir es lo siguiente:

```
COOLBOT_PATH=$HOME/git-projects/coolbot
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$COOLBOT_PATH/lib
export COOLBOT_PATH LD_LIBRARY_PATH
```

A continuación es preciso inicializar el shell para actualizar las variables de estado:

```
source .bashrc
```

Compilación de *CoolBOT*

En el directorio de de donde se ha bajado *CoolBOT* indicado por la variable de entorno *COOLBOT_PATH* se encuentran varios ficheros *make*, con extensión *.mak* que se utilizarán para realizar la compilación de *CoolBOT*.

- Generación de reglas de compilación

  El primer fichero se denomina *makeMakeObjects.mak* y se utiliza para realizar la generación de las reglas de compilación de *CoolBOT*. Es preciso utilizarlo la primera vez que *CoolBOT* se compila y cuando se añade un nuevo fichero o directorio a *CoolBOT*. Para ejecutarlo emitimos la siguiente orden en línea de comandos:

  ```
  make extraFlags="-ggdb -O0" -f makeMakeObjects.mak
  ```

  si se quiere hacer una compilación en modo depuración (*debug*), y si no es modo *debug* emitiremos:

  ```
  make -f makeMakeObjects.mak
  ```

  Como se observa en la opción de depuración la variable *extraFlags* permite añadir cualquier flag válido para *g++* en la compilación.

  Como resultado en el directorio *obj* se habrán creado todas las reglas de compilación necesarias.

- Compilación de la Librería Dinámica *CoolBOT*

  Lo siguiente es compilar la librería de *CoolBOT*. Ello lo hacemos emitiendo usando el fichero make *coolbot-lib.mak* emitiendo el siguiente comando:

```
make extraFlags="-ggdb -O0" -f coolbot-lib.mak
```

para la compilación en modo de depuración, y para una compilación en modo de no depuración:

```
make -f coolbot-lib.mak
```

Como resultado de la compilación se creará la librería *CoolBOT* en el directorio *lib*, concretamente el fichero *libcoolbot.so*.

- Compilación de Ejemplos de Uso

  El resto de ficheros make en el directorio de *CoolBOT*, son útiles para compilar los ejemplos de uso de *CoolBOT* que se encuentran en la carpeta *examples*, concretamente todos los ficheros con extensión .cpp que se pueden encontrar en *examples*. Cada fichero make se corresponde con un ejemplo de uso. La forma de compilar es idéntica a la ya vista en las anteriores sección sólo que utilizando el fichero make correspondiente, de manera que por ejemplo el fichero, *component-test.mak* compilará el ejemplo *component-test.cpp* del directorio *examples*. Para comprobar que todo ha ido bien compile dicho ejemplo y ejecútelo, si no hay errores *CoolBOT* ya se encuentra instalado. Tenga en cuenta que los ficheros ejecutables de los ejemplos, una vez compilados, se colocan en el directorio *bin*.

A.3.2. Componente GridMap

Descarga

Situarse en el directorio *git-projects*, y ahí bajarse el componente GridMap desde el servidor *GIT* con la siguiente orden:

```
git clone ssh://<usuario>new-mozart.dis.ulpgc.es:/home/git-user/git-repository/ \
grid-map.git
```

Esta orden crea el directorio *grid-map* en *git-projects*.

variable de entorno

Es preciso incluir en el .bashrc la variable de estado *GRID_MAP_PATH* indicando donde se encuentra el componente GridMap y donde se encontrará la librería del componente una vez se compile. Lo que hay que introducir es lo siguiente:

```
GRID_MAP_PATH=$HOME/git-projects/grid-map
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$GRID_MAP_PATH/lib
export GRID_MAP_PATH LD_LIBRARY_PATH
```

A continuación es preciso inicializar el shell para actualizar las variables de estado.

Compilación del componente

En el directorio de de donde se ha bajado el componenteindicado por la variable de entorno se encuentran varios ficheros make, con extensión .mak que se utilizarán para realizar *GRID_MAP_PATH* la compilación del componente.

- Generación de reglas de compilación

  El primer fichero se denomina *makeMakeObjects.mak* y se utiliza para realizar la generación de las reglas de compilación del componente. Es preciso utilizarlo la primera vez que el componente se compila y cuando se añade un nuevo fichero o directorio al componente. Para ejecutarlo emitimos la siguiente orden en línea de comandos:

  ```
  make extraFlags="-ggdb -O0" -f makeMakeObjects.mak
  ```

  si se quiere hacer una compilación en modo depuración (debug), y si no es modo debug emitiremos:

  ```
  make -f makeMakeObjects.mak
  ```

  Como se observa en la opción de depuración la variable *extraFlags* permite añadir cualquier flag válido para *g++* en la compilación.

  Como resultado en el directorio *obj* se habrán creado todas las reglas de compilación necesarias.

- Compilación de la Librería Dinámica del componente

  Lo siguiente es compilar la librería del componente que nos permitirá luego enlazarlo en los programas donde queramos utilizarlo. Ello lo hacemos emitiendo usando el fichero make *grid-map-lib.mak* emitiendo el siguiente comando:

  ```
  make extraFlags="-ggdb -O0" -f grid-map-lib.mak
  ```

  para la compilación en modo de depuración, y para una compilación en modo de no depuración:

```
make -f grid-map-lib.mak
```

Como resultado de la compilación se creará la librería del componente en el directorio *lib*, concretamente el fichero *libgrid-map.so*.

- Compilación de Ejemplo

  El fichero *grid-map-test.mak* se utiliza para compilar *grid-map-test.cpp* el ejemplo que se encuentra en el fichero *examples* y que al compilarlo nos permite probar si el componente si la librería del componente se ha generado correctamente. El ejemplo no hace nada, sólo se lanza el componente, se espera unos segundo, y luego se lo destruye.

A.3.3. Componente NDNavigation

Descarga

Situarse en el directorio *git-projects*, y ahí bajarse el componente *NDNavigation* desde el servidor *GIT* con la siguiente orden:

```
git clone ssh://<usuario>new-mozart.dis.ulpgc.es:/home/git-user/git-repository/ \
nd-navigation.git
```

Esta orden crea el directorio *nd-navigation* en *git-projects*.

variable de entorno

Es preciso incluir en el .bashrc la variable de estado *ND_NAVIGATION_PATH* indicando donde se encuentra el componente *NDNavigation* y donde se encontrará la librería del componente una vez se compile. Lo que hay que introducir es lo siguiente:

```
ND_NAVIGATION_PATH=$HOME/git-projects/nd-navigation
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ND_NAVIGATION_PATH/lib
export ND_NAVIGATION_PATH LD_LIBRARY_PATH
```

A continuación es preciso inicializar el shell para actualizar las variables de estado.

Compilación del componente

En el directorio de de donde se ha bajado el componente indicado por la variable de entorno se encuentran varios ficheros make, con extensión .mak que se utilizarán para *ND_NAVIGATION_PATH* realizar la compilación del componente.

- Generación de reglas de compilación

  El primer fichero se denomina *makeMakeObjects.mak* y se utiliza para realizar la generación de las reglas de compilación del componente. Es preciso utilizarlo la primera vez que el componente se compila y cuando se añade un nuevo fichero o directorio al componente. Para ejecutarlo emitimos la siguiente orden en línea de comandos:

  ```
  make extraFlags="-ggdb -O0" -f makeMakeObjects.mak
  ```

  si se quiere hacer una compilación en modo depuración (debug), y si no es modo debug emitiremos:

  ```
  make -f makeMakeObjects.mak
  ```

  Como se observa en la opción de depuración la variable *extraFlags* permite añadir cualquier flag válido para *g++* en la compilación.

  Como resultado en el directorio *obj* se habrán creado todas las reglas de compilación necesarias.

- Compilación de la Librería Dinámica del componente

  Lo siguiente es compilar la librería del componente que nos permitirá luego enlazarlo en los programas donde queramos utilizarlo. Ello lo hacemos emitiendo usando el fichero make *nd-navigation-lib.mak* emitiendo el siguiente comando:

  ```
  make extraFlags="-ggdb -O0" -f nd-navigation-lib.mak
  ```

  para la compilación en modo de depuración, y para una compilación en modo de no depuración:

  ```
  make -f nd-navigation-lib.mak
  ```

  Como resultado de la compilación se creará la librería del componente en el directorio *lib*, concretamente el fichero *libnd-navigation.so*.

- Compilación de Ejemplo

  El fichero *nd-navigation-test.mak* se utiliza para compilar *nd-navigation-test.cpp* el ejemplo que se encuentra en el fichero *examples* y que al compilarlo nos permite probar si el componente si la librería del componente se ha generado correctamente. El ejemplo no hace nada, sólo se lanza el componente, se espera unos segundo, y luego se lo destruye.

A.3.4. Componente ShortTermPlanner

Descarga

Situarse en el directorio *git-projects*, y ahí bajarse el componente *ShortTermPlanner* desde el servidor *GIT* con la siguiente orden:

```
git clone ssh://<usuario>new-mozart.dis.ulpgc.es:/home/git-user/git-repository/ \
short-term-planner.git
```

Esta orden crea el directorio *short-term-planner* en *git-projects*.

variable de entorno

Es preciso incluir en el .bashrc la variable de estado *SHORT_TERM_PLANNER_PATH* indicando donde se encuentra el componente *ShortTermPlanner* y donde se encontrará la librería del componente una vez se compile. Lo que hay que introducir es lo siguiente:

```
SHORT_TERM_PLANNER_PATH=$HOME/git-projects/short-term-planner
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$SHORT_TERM_PLANNER_PATH/lib
export SHORT_TERM_PLANNER_PATH LD_LIBRARY_PATH
```

A continuación es preciso inicializar el shell para actualizar las variables de estado.

Compilación del componente

En el directorio de de donde se ha bajado el componente indicado por la variable de entorno se encuentran varios ficheros make, con extensión .mak que se utilizarán *SHORT_TERM_PLANNER_PATH* para realizar la compilación del componente.

- Generación de reglas de compilación

  El primer fichero se denomina *makeMakeObjects.mak* y se utiliza para realizar la generación de las reglas de compilación del componente. Es preciso utilizarlo la primera vez que el componente se compila y cuando se añade un nuevo fichero o directorio al componente. Para ejecutarlo emitimos la siguiente orden en línea de comandos:

```
make extraFlags="-ggdb -O0" -f makeMakeObjects.mak
```

si se quiere hacer una compilación en modo depuración (debug), y si no es modo debug emitiremos:

```
make -f makeMakeObjects.mak
```

Como se observa en la opción de depuración la variable *extraFlags* permite añadir cualquier flag válido para *g++* en la compilación.

Como resultado en el directorio *obj* se habrán creado todas las reglas de compilación necesarias.

- Compilación de la Librería Dinámica del componente

  Lo siguiente es compilar la librería del componente que nos permitirá luego enlazarlo en los programas donde queramos utilizarlo. Ello lo hacemos emitiendo usando el fichero make *short-term-planner-lib.mak* emitiendo el siguiente comando:

  ```
  make extraFlags="-ggdb -O0" -f short-term-planner-lib.mak
  ```

  para la compilación en modo de depuración, y para una compilación en modo de no depuración:

  ```
  make -f short-term-planner-lib.mak
  ```

  Como resultado de la compilación se creará la librería del componente en el directorio *lib*, concretamente el fichero *libshort-term-planner.so*.

- Compilación de Ejemplo

  El fichero *short-term-planner-test.ma*k se utiliza para compilar *short-term-planner-test.cpp* el ejemplo que se encuentra en el fichero *examples* y que al compilarlo nos permite probar si el componente si la librería del componente se ha generado correctamente. El ejemplo no hace nada, sólo se lanza el componente, se espera unos segundo, y luego se lo destruye.

A.3.5. Componente PlayerRobot

Descarga

Situarse en el directorio *git-projects*, y ahí bajarse el componente *PlayerRobot* desde el servidor *GIT* con la siguiente orden:

```
git clone ssh://<usuario>new-mozart.dis.ulpgc.es:/home/git-user/git-repository/ \
player-robot.git
```

Esta orden crea el directorio *player-robot* en *git-projects*.

variable de entorno

Es preciso incluir en el .bashrc la variable de estado *PLAYER_ROBOT_PATH* indicando donde se encuentra el componente PlayerRobot y donde se encontrará la librería del componente una vez se compile. Lo que hay que introducir es lo siguiente:

```
PLAYER_ROBOT_PATH=$HOME/git-projects/player-robot
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PLAYER_ROBOT_PATH/lib:/usr/local/lib
export PLAYER_ROBOT_PATH LD_LIBRARY_PATH
```

A continuación es preciso inicializar el shell para actualizar las variables de estado.

Compilación del componente

En el directorio de de donde se ha bajado el componente indicado por la variable de entorno *PLAYER_ROBOT_PATH* se encuentran varios ficheros make, con extensión .mak que se utilizarán para realizar la compilación del componente.

- Generación de reglas de compilación

  El primer fichero se denomina *makeMakeObjects.mak* y se utiliza para realizar la generación de las reglas de compilación del componente. Es preciso utilizarlo la primera vez que el componente se compila y cuando se añade un nuevo fichero o directorio al componente. Para ejecutarlo emitimos la siguiente orden en línea de comandos:

  ```
  make extraFlags="-ggdb -O0" -f makeMakeObjects.mak
  ```

  si se quiere hacer una compilación en modo depuración (debug), y si no es modo debug emitiremos:

  ```
  make -f makeMakeObjects.mak
  ```

Como se observa en la opción de depuración la variable *extraFlags* permite añadir cualquier flag válido para *g++* en la compilación.

Como resultado en el directorio *obj* se habrán creado todas las reglas de compilación necesarias.

- Compilación de la Librería Dinámica del componente

  Lo siguiente es compilar la librería del componente que nos permitirá luego enlazarlo en los programas donde queramos utilizarlo. Ello lo hacemos emitiendo usando el fichero make *player-robot-lib.mak* emitiendo el siguiente comando:

  ```
  make extraFlags="-ggdb -O0" -f player-robot-lib.mak
  ```

  para la compilación en modo de depuración, y para una compilación en modo de no depuración:

  ```
  make -f player-robot-lib.mak
  ```

  Como resultado de la compilación se creará la librería del componente en el directorio *lib*, concretamente el fichero *libplayer-robot.so*.

- Compilación de Ejemplo

  En el directorio del componente *PlayerRobot* hay varios ficheros make para realizar compilaciones de ejemplos contenidos en el directorio *examples* del componente. En concreto el fichero *player-robot-gtk-test.mak* compila el ejemplo *player-robot-gtk-test.cpp* que integra los componentes *PlayerRobot, GridMap, NDNavigation* y *ShortTermPlanner* que integran un evitador de obstáculos utilizando el algoritmo *ND* con planificador a corto plazo. La implementación del algoritmo *ND* es propia del IUSIANI.

  El otro fichero make de interés es el *player-robot-with-vfh-and-nd-gtk-test.mak* que se corresponde con el fichero *player-robot-with-vfh-and-nd-gtk-test.cpp* que hay en el directorio *examples* del componente. Este ejemplo integra los componentes *PlayerRobot, GridMap* y *NDNavigation* e integra un evitador de obstáculos con las implementaciones realizadas en *Player/Stage* de los algoritmos *VFH* y *ND*.

Bibliografía

[George,2001] George T. Heineman; William T. Councill *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley Professional, Reading 2001

[Huston,2003] Huston, Stephen D.; Johnson, James CE; Syyid, Umar. *The ACE Programmer's Guide: Practical Design Patterns for Network and Systems Programming.* Addison Wesley, 2003.

[ACE,2007] The Adaptive Communication Environment. *http://www.cs.wustl.edu/ schmidt/ACE.html.* (2007).

[Domínguez-Brito, 2003] Domínguez-Brito, A.C. *CoolBOT: a Component-Oriented Programming Framework for Robotics.* PhD thesis, Dpto. Informática y Sistemas, Universidad de Las Palmas de Gran Canaria (mozart.dis.ulpgc.es/Publications/publications.html).

[Gamma, 1995] Erick Gamma, Helm Richard, Johnson Ralph, and Vlissides John *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[Goldberg and Siegwart, 2001] Goldberg, Ken and Siegwart, Roland *Beyod Webcams. An introduction to online robots.* The MIT Press, 2001.

[Jacobson, Booch and Rumbaugh, 2000] Jacobson, Ivar, Booch Grady and Rumbaugh James *El proceso unificado de desarrollo de software.* Addison-Wesley, 2000.

[Lamport, 1986] Lamport, Leslie *LaTex : A document Preparation System.* Addison-Wesley, 1986.

[Larman, 2006] Larman, Craig *UML y Patrones: Una introducción al análisis y diseño orientado a objetos y al proceso unificado.* Addison-Wesley, 1986.

[Stevens,1999] Stevens, Richard. *UNIX Network Programming, Volume 2, Second Edition: Interprocess Communications.* Prentice Hall, 1999.

[Robinson, 2003] Robinson, Matthew *Swing.* Manning, 2003.

[Roland, 1983] Rolland, Christian *LaTex guide pratique.* Addison-Wesley, 1993.

[GTK+ 2.0 Web Page] GTK+ 2.0 Tutorial: *http://library.gnome.org/devel/gtk-tutorial/stable/c24.html.*

[STL] Silicon Graphics, Inc. Standard template library programmers's guide (http://www.sgi.com/tech/stl/).

[Stroustrup,1997] Stroustrup, B. The C++ programming language: Special Edition. *Addison Wesley, 3ª edición, 1997.*

[Sunshine,1979] Sunshine, Carl A. Formal methods for communication protocols specification and verification. *Rand note for ARPA/NBS, 1979.*

[Vandevoorde,2003] Vandevoorde, D; Josuttis, N. M. C++ Templates: the complete guide. *Addison Wesley, 2003.*

[Player-Stage,2010] The Player Project. http://playerstage.sourceforge.net. *(2010).*

[Santana-Jorge,2007] Santana-Jorge, Fco Jesús. PFC: Compilador/generador de esqueletos C++ para componentes CoolBOT. *Facultad de informática, Universidad de Las Palmas de Gran Canaria, 2007.*

[Schmidt,2001] Schmidt, Douglas C.; Huston, Stephen D. C++ Network programming Volume I: Mastering complexity with ACE and patterns. *Addison Wesley, 2001.*

[Marina,2007] Marina, J. L. ACE: desarrollo multiplataforma de aplicaciones en red. *Linux+ (32):32-39, 7/2007.*

[Marina2,2007] Marina, J. L. Introducción a ACE: Teoría. http://www.jlmarina.net/publish/. *(2007).*

[Minguez,2004] Minguez, J.; Montano, L. Nearness Diagram Navigation (ND): Collision Avoidance in Troublesome Scenarios IEEE Transactions on Robotics and Automation, Volume 20, Issue 1, Pages:45 - 59, 2004.

[Holzmann,1991] Holzmann, Gerard J. Design and validation of computer protocols. *Prentice Hall, 1991.*

[Domínguez-Hernández,2007] Domínguez-Brito, A. C.; Hernández-Sosa, D.; Isern-González, J.; Cabrera-Gámez, J. CoolBOT: a Component Model and Software Infrastructure for Robotics. *Springer Tracts in Advanced Robotics Series, Vol. 30, pp. 135-142, Brugali, Davide (Ed.), 490 p, 2007.*

[OMG,2004] Object Management Group. Common Object Request Broker Architecture: Core Specification. (http://www.omg.org/cgi-bin/doc?formal/04-03-01). *2004.*

[OMG,2002] Object Management Group. The Common Object Request Broker: Architecture and Specification. (http://www.omg.org/cgi-bin/doc?formal/02-06-01). *Cap 15, Secciones 1 a la 3, (2002).*

[KhepOnTheWeb,1997] Olivier Michel, Patrick Saucy and Francesco Mondada. KhepOnTheWeb: An Experimental Demonstrator in Telerobotics And Virtual Reality*. 1997.*

[Arvind,1981] Arvind, Vinod Kathail. A multiple processor data flow machine that supports generalized procedures*. Proceedings of the 8th annual symposium on Computer Architecture, p.291-302, May 12-14, 1981, Minneapolis, Minnesota, United States*