ROBOT MÓVIL SEMIAUTÓNOMO CON TELECONTROL PARA INSPECCIÓN DE INTERIORES

SORAYA SANTANA DE LA FE

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: Robot Móvil Semiautónomo con Telecontrol para Inspección de Interiores
Apellidos y nombre del alumno: Santana de la Fe, Soraya
Fecha: Marzo de 2010

Tutor: Antonio Carlos Domínguez Brito
Tutor: Jorge Cabrera Gámez
Tutor: Antonio Falcón Martel

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

Agradecimientos

El trabajo que aquí se presenta es el resultado de muchas horas de dedicación y duro esfuerzo, es el reflejo del lugar al que todo estudiante pretende llegar cuando inicia su andadura universitaria. Pero alcanzar Ítaca no es tan importante como las experiencias vividas y, sobretodo, las personas que contribuyen durante el camino. Me gustaría expresar mi profundo agradecimiento a todas ellas.

A mis tutores de proyecto, especialmente a Antonio, sin sus directrices y ayuda este trabajo no sería una realidad. Han sido sin duda alguna una fuente de inspiración e ilusión, contagiandome de optimismo y ganas en todo momento.

A mis hermanos, mil gracias por su apoyo y confianza. Son los pequeños detalles los que marcan la diferencia y todos estos años de esfuerzo han estado repletos de ellos. A Eli, por su constante interés en temas plagados de palabrejas técnicas y por haberme hecho poner el listón un poco más arriba tras cada logro. Has hecho que tenga confianza en mí misma cuando flaqueaba, gracias. A Saray, por estar cerca siempre y por su apoyo. A Martín, por los ánimos con su particular sentido del humor.

A mi compañero y amigo Jose, con el que he compartido el desarrollo de este proyecto con risas y alguna noche sin dormir: *legendario*.

En general a mis compañeros y amigos de facultad durante estos años de estudio. Por sus palabras de ánimo, algún *Red Bull* en el momento justo y muchos ratos compartiendo cafés y almuerzos, gracias.

A todos los que no están pero también son parte de esto. A todos ellos, un bucle infinito de gracias.

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

Índice general

Índice de figuras

Índice de tablas

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

Capítulo 1

Introducción

La programación de sistemas robóticos teleoperados requiere abordar numerosas cuestiones: implementación de conductas en el robot, acceso remoto al sistema o presentación de datos para la monitorización, entre otros.

Históricamente el desarrollo de los sistemas robóticos tiene su inicio en la industria de manufacturación. Los robots son vistos exclusivamente como sistemas mecánicos capaces de ser programados para realizar tareas repetitivas en cadenas de producción en masa. La construcción de robots se inicia por tanto en busca de automatizar mecanismos de producción, donde esencialmente no se requiere que el sistema sea capaz de captar información de su entorno.

Sin embargo, la investigación espacial planteó una visión diferente de los sistemas robóticos. A diferencia de la industria de la manufacturación, donde la actividad es siempre perfectamente conocida y sistematizable, las misiones espaciales se enfrentaban a escenarios y situaciones desconocidas e impredecibles. Por tanto, se requieren sistemas capaces de captar datos del entorno y reaccionar ante los mismos. Minimamente un sistema explorador espacial debe disponer de entradas sensoriales, capacidad de interpretación de los datos captados como entrada y modificar sus acciones de una forma reactiva a dichas entradas. Sin embargo, en los inicios de la misión espacial, no existian sistemas con una adaptabiliad en tiempo real como se requería. Esta carencia se compensó con mecanismos que permitieran a un humano controlar todo o partes del robot de forma remota. Estos mecanismos son englobados dentro del término *teleoperación*.

Si bien los inicios en el desarrollo de sistemas teleoperados estuvieron ligados a la industria y a la investigación espacial, hoy por hoy se han convertido en sistemas con numerosas y variadas aplicaciones. Desde exploraciones submarinas y desactivación de bombas hasta sistemas más simples y accesibles a través de Internet como ueden ser robots para el guiado en visitas remotas a museos. Un ejemplo significativo del desarrollo e interés que han despertado los sistemas teleoperados es el sistema *Xavier* [Goldberg,2002]. Se trata de un robot autónomo con acceso vía Web. Más de 30.000 personas han teleopreado al sistema desde sus hogares, tan sólo con disponer de un navegador y acceso a Internet. Este es un ejemplo trivial que ilustra la teleoperación, pero la autonomía del sistema *Xavier* no se ha aplicado a la realización de tareas concretas. Existen hoy día multitud de escenarios donde un robot teleoperado es de gran utilidad: aplicaciones médicas a distancia (telemedicina), teleconferencias, manejo de sustancias peligrosas, y un largo etcétera.

1.1. Presentación del problema

La aparición de Internet ha favorecido enormemente el desarrollo de sistemas teleoperados. Estos sistemas tienen multitud de aplicaciones en diferentes entornos para la realización de tareas varias. Sin embargo, la robótica móvil es un campo joven en el que aún no se han establecido paradigmas definitivos para el desarrollo de este tipo de sistemas. La construcción de sistemas robóticos que implementen cierto grado de autonomía entraña además multitud de problemas: integración de hardware variado, manejo de diferentes elementos software, dificultad para la reutilización de componentes ya operativos, elección de estrategias de control, etc. A estos factores hay que añadir el manejo de las comunicaciones cuando se pretende obtener un sistema teleoperado. No es una tarea trivial lograr comunicaciones eficientes en este tipo de sistemas distribuidos, y éstas se presentan como una limitación importante en la mayoría de estos sistemas.Un sistema totalmente teleoperado, es decir controlado en su totalidad por un operador remoto, requiere altos anchos de banda debido a la gran cantidad de mensajes de control/monitorización que se intercambian. Ante estas situaciones resulta interesante que el sistema robótico disponga de ciertas capacidades que le permitan un mínimo de autonomía, con lo que sólo se requiera al operador en determinados momentos. Un sistema que ofrezca estas características tiene un campo interesente de aplicación en la inspección de interiores, donde el sistema pueda navegar de forma semiautónoma con una intervención mínima del teleoperador. Dicho sistema facilitaría la navegación a través de zonas de difícil acceso para personas, como tubos de ventilación.

En este proyecto se propone la creación de un sistema que satisfaga estas características, implementando para ello conductas que doten al sistema de cierta autonomía y que se permita la teleoperación del mismo.

A continuación se presentan ciertos conceptos íntimamente relacionados con este desarrollo y que clarifican el problema a abordar.

1.2. Teleoperación

Llamamos *teleoperación* a las acciones de control que un operador humano ejerce sobre un robot que se encuentra a una cierta distancia lo suficientemente grande como para que el operador no vea lo que el robot hace. Generalmente el operador trabaja desde una estación con algún tipo de interfaz. Normalmente la terminología se refiere al operador como el *extremo local* y al robot como el *extremo remoto*. El extremo local suele tener algún tipo de mecanismo de monitorización y control, mientras que en el extremo remoto se dispone de sensores y actuadores. El tercer elemento clave en este tipo de sistemas es el *enlace de comunicación*, componente crítico en sistemas teleoperados.

1.3. Control semiautónomo

El control semiautónomo es considerado como un modo particular de teleoperación, también denominado control de supervisión (*supervisory control*). Existen dos vertientes en el control semiautónomo: control compartido (*shared control*), y control delegado (*control trading*).

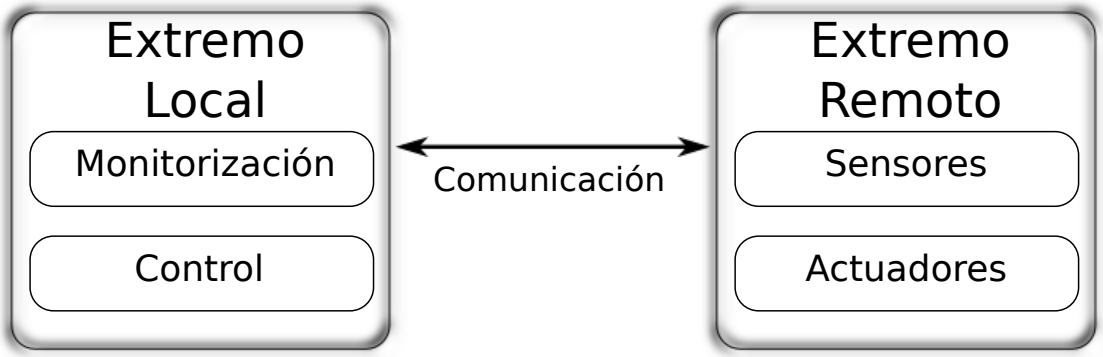

Figura 1.1: Organización de un entorno de teleoperación.

En el caso del control compartido, el extremo local puede tanto delegar una tarea al robot como realizarla mediante un control directo. Aún comandando una tarea para que el robot la realice por sí mismo, el operador monitoriza las acciones realizadas asegurándose de que todo se efectúa correctamente y pudiendo intervenir si algo fuera mal e indicando uno tras otro los comandos al robot.

En el control delegado el operador humano comanda una tarea al extremo remoto para que este la realice de forma autónoma. El operador sólo interactúa con el robot para indicar una nueva tarea o bien para cambiar una actualmente en ejecución por otra. En el modelo de control delegado se asume que el robot es capaz de realizar las tareas comandadas de forma autónoma, sin necesidad de recurrir a un control continuo, donde se requiere un mayor ancho de banda y aparecen mayores retrasos en las comunicaciones.

1.4. 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 los objetivos específicos del PFC.

1.4.1. Objetivos académicos

Como objetivos académicos 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.

1.4.2. Objetivos específicos del proyecto

El objetivo final de este desarrollo es obtener un sistema robótico semiautónomo orientado a la navegación e inspección de interiores. Además el sistema contará con una interfaz que permita la teleoperación. Para construir este sistema es necesario descomponer el desarrollo en subobjetivos, orientados a hacer frente a los diferentes requisitos de un sistema de estas características.

En este proyecto se ha optado por la utilización de un marco de programación de sistemas robóticos y el aprovechamiento de componentes software ya desarrollados. Este marco es *CoolBOT*, herramienta C++ para la construcción de sistemas robóticos a partir de componentes software.

- **Portabilidad.**

  Uno de los objetivos clave en este proyecto es desarrollar la infraestructura de comunicaciones entre componentes software que se ejecutarán de forma remota en máquinas con, posiblemente, distintos sistemas operativos y/o arquitecturas. Es crucial, por tanto, desarrollar un sistema con alta portabilidad, sobretodo tratándose de herramientas que deben adaptarse a sistemas que, generalmente, involucran una gran variabilidad de hardware y software (sensores, actuadores, drivers de dispositivos, sistemas operativos, etc).

- **Eficiencia.**

  Otra característica deseable es lograr un sistema de comunicaciones eficiente. Minimizar el trasiego de mensajes y datos contenidos en los mismos resulta de gran importancia a la hora de desarrollar las comunicaciones en sistemas distribuidos.

- **Escalabilidad y modularidad.**

  Obtener un sistema de fácil mantenimiento se presenta como otro objetivo importante del proyecto. Por tanto se busca lograr un software modular y con una estructura escalable y de fácil adaptación a modificaciones futuras.

- **Transparencia de las comunicaciones.**

  Sería de gran interés para un usuario teleoperador de sistemas abstraerse de detalles de bajo nivel a la hora de manejar de forma remota un sistema robótico. Una interfaz de teleopreación debe ofrecer la funcionalidad estrictamente necesaria para un operador, sin la necesidad de que este conozca en detalle mecanismos de comunicación de bajo nivel.

- **Sistematización en la creación de componentes software.**

  Resulta muy interesante poder facilitar al usuario de *CoolBOT* la creación de componentes software distribuidos sin más que indicar ciertas características básicas. Se busca que la infraestructura desarrollada favorezca la sitematización de estos procesos.

- **Vistas de componentes software.**

  Otro objetivo del presente proyecto es desarrollar una interfaz que permita teleoperar un sistema robótico, interactuando de forma tanto local como remota con los componentes del sistema. Esta interfaz debe ofrecer mecanismos de monitorización de la actividad del sistema y de control sobre el mismo. Además debe permitir comandar determinadas tareas para ser realizadas de forma semiautónoma con la posible intervención del operador en cualquier momento.

1.4.3. Visión del sistema final

Un ejemplo real de sistema teleoperado es el que se observa en la figura 1.2. En este sistema existe un usuario teleoperador que utiliza una interfaz para controlar y monitorizar a un robot móvil. El operador se encuentra separado físicamente del sistema robótico y las comunicaciones entre ambos se realizan a través de *Internet*.

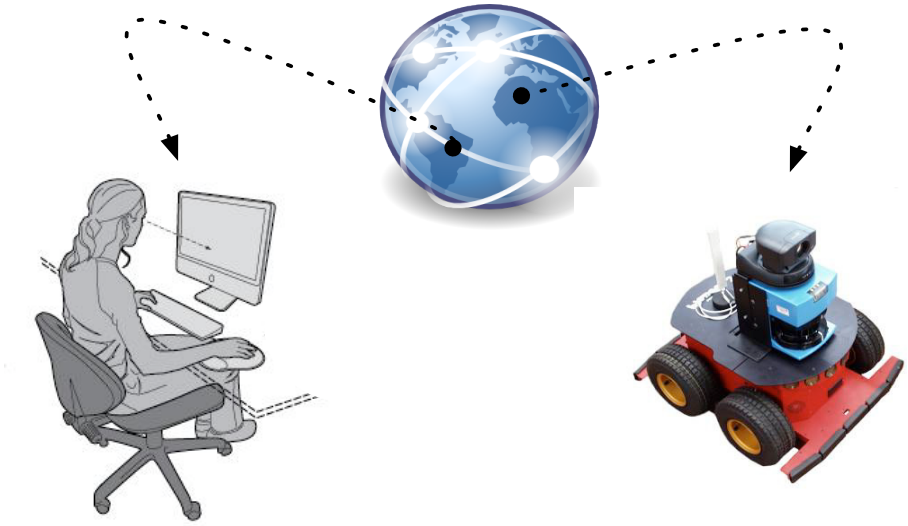

Figura 1.2: Sistema teleoperado

El sistema final que se pretende obtener en este proyecto es un robot semiautónomo teleoperado construido a partir de componentes software, donde cada uno implemente un funcionalidad concreta. La figura 1.3 ofrece una visión a alto nivel de la estructura de un sistema con estas características. Como se observa existen múltiples dispositivos interconectados a través de Internet, entre ellos un robot móvil. En la *máquina A* y *máquina B* así como en el *robot* residen uno o más componentes software. En varias de las máquinas se dispone de interfaces (*Interfaz A* e *Interfaz B*) que permiten observar la actividad de los componentes software que se encuentran distribuidos en el resto de máquinas. Estas interfaces pueden ofrecer la posibilidad de controlar los elementos que componen el sistema.

El diagrama de la figura 1.4 refleja, de forma esquemática, una posible representación del sistema de la figura 1.3. Cada una de las máquinas se representa por una base hardware sobre

Figura 1.3: Visión a alto nivel del sistema teleoperado

la que reside el software: componentes o vistas del sistema. Cada una de estas vistas ofrece una representación diferente del sistema. Las interfaces *A* y *B* están constituidas por un conjunto de vistas. Cada componente o vista de un sistema como el ilustrado se ejecuta de forma distribuida, utilizando la infraestructura de protocolos *TCP/IP* para las comunicaciones.

1.5. Estructura de este documento

Este documento se organiza en 7 capítulos y tres apéndices:

- Capítulo 1: Introducción.

  En este capítulo se presenta la introducción al presente documento. Se presenta el problema a resolver y conceptos básicos relacionados con el mismo. Además se exponen los objetivos de este desarrollo y se presenta la estructuración de este documento.

- Capítulo 2: Estado actual del tema.

  En este capítulo se pretende situar al lector en el estado actual del desarrollo de sistemas semiautónomos teleoperados. Se presentan los dos temas más importantes tratados durante este desarollo: los sistemas robóticos semiautónomos y el desarrollo de sistemas distribuidos.

- Capítulo 3: *CoolBOT*.

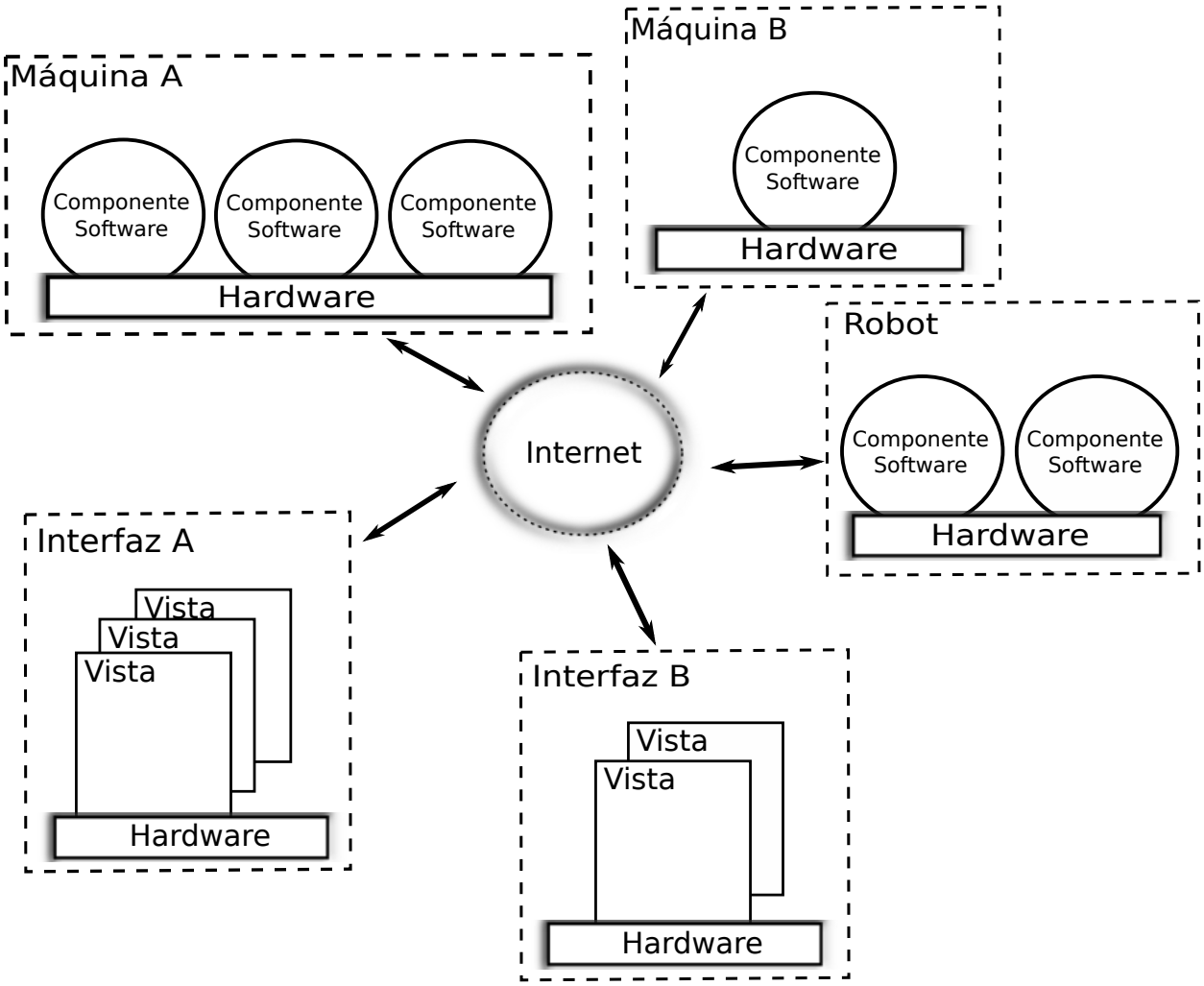

Figura 1.4: Visión esquemática del sistema teleoperado

En este capítulo se describe el marco de programación de sistemas robóticos *CoolBOT*. Esta herramienta ha sido utilizada como punto de partida en el desarrollo de este proyecto.

- Capítulo 4: Análisis.

  Este capítulo detalla las tareas de análisis realizadas durante el desarrollo. Como primer punto se describe la fase inicial del desarrollo. Además, se presenta el modelo de dominio del problema, así como los casos de uso y la especificación de requerimientos obtenida. Finalmente se analizan y comparan diferentes herramientas software para el desarrollo de sistemas distribuidos.

- Capítulo 5: Diseño.

  En este capítulo se presentan los resultados de la fase de diseño del proyecto. Se describen las

operaciones diseñadas para abordar las comunicaciones distribuidas de un sistema robótico: *marshalling* de datos y especificación de un protocolo de comunicaciones. Finalmente se presenta el modelo de componente software a partir del cual se construye el sistema robótico semiautónomo objetivo de este proyecto, así como la interfaz de teleoperción del mismo.

- Capítulo 6: Pruebas y resultados.

  Este capítulo presenta todos los componentes software que componen el sistema robótico desarrollado y su interfaz de teleoperación. Asimismo se reflejan los tests realizados.

- Capítulo 7: Conclusiones y trabajo futuro.

  Se presentan las conclusiones estraidas del desarrollo del proyecto. Además se plantean posibles líneas de desarrollo futuro a partir del trabajo realizado.

- Apéndice A: Manuales de usuario.

  Este apéndice incluye: manual de instalación del sistema desarrollado, Guía de creación de paquetes de puerto y Guía de creación de mensajes del protocolo desarrollado.

- Apéndice B: Detalles de diseño.

  Se describen los patrones de diseño utilizados y el diseño del protocolo.

- Apéndice C: Detalles de implementación.

  Se describen características propias de la implementación realizada.

Capítulo 2

Estado actual del tema

Asociados a los sistemas robóticos teleoperados aparecen multitud de dificultades como son la implementación de estrategias de control semiautónomo o el desarrollo de sistemas distribuidos. Este capítulo pretende presentar una visión general de la construcción de sistemas robóticos semiautónomos teleoperados, analizando los problemas que plantea y algunas herramientas dedicadas a afrontarlos.

2.1. Sistemas robóticos semiautónomos

Diseñar un sistema robótico semiautónomo requiere abordar diferentes tareas:

- Manejo de sensores y efectores

  Un sistema robótico que implemente cierto grado de autonomía requiere conocer su entorno. Para tal fin los robots incorporan una serie de sensores que le permiten capturar datos (lásers, sonars, cámaras, etc). Además estos sistemas permiten interactuar en cierta medida con su medio, para tal fin incorporan una serie de efectores (principalmente motores). El manejo de estos dispositivos requiere el conocimiento de sus particularidades y de herramientas para su programación (*Aplication Programming Interface: API*). Lidiar con el hardware de estos dispositivos es una tarea complicada devido a la gran variedad existente.

- Concurrencia y paralelismo.

  Es frecuente en la construcción de sistemas robóticos la necesidad de resolución de problemas de forma concurrente y/o paralela. En ocasiones, la necesidad de sincronización e interacción entre diferentes unidades del sistema convierten la programación de estos sistemas en una tarea muy compleja.

- Implementación de conductas autónomas.

  Un robot semiautónomo teleoperado bajo una estrategia de control delegado requiere la implementación de algoritmos para resolver determinadas tareas. De esta forma se logra que el robot sea capaz de afrontar de forma autónoma ciertos cometidos, o fracciones simples de los mismos. La complicación de esta labor de implementación reside en dividir el problema a

tratar, y elegir en consecuencia la combinación adecuada de conductas que permitan al robot cierto grado de autonomía.

- Teleoperación

  Cuando se pretende que un sistema robótico actúe de forma semiautónoma se requiere que exista un operador encargado de controlar al sistema en última instancia. De esta forma parte del control se traslada al usuario operador. Para tal fin se requiere de algún mecanismo que permita al usuario controlar el sistema de forma remota. Además el operador necesita disponer de información del estado del robot, con el fin de tomar las decisiones de control pertinentes.

En los siguientes apartados se presentan una serie de herramientas destinadas a facilitar la creación de sistemas robóticos. Estas herramientas abordan de distinta manera, con mayor o menor efectividad, el tratamiento de los problemas anteriormente descritos.

2.1.1. Player/Stage

El proyecto *Player/Stage* [Player-Stage,2010] es una infraestructura de software para la abstracción y simulación de hardware de sistemas robóticos o sistemas percepto efectores. Se compone del servidor de abstracción hardware denominada *Player*, y del simulador *Stage*. A continuación se realiza un pequeño esbozo de ambas herramientas software.

Player

Player es un software que proporciona una capa de abstracción sobre una gran variedad de hardware sensorial y diferentes modelos de robots. La arquitectura de este software sigue un modelo cliente-servidor sobre una red *TCP/IP*. El servidor *Player* se ejecuta en un robot, permitiendo a un programa cliente (o varios concurrentemente) leer datos de los sensores y enviar comandos a los actuadores del robot.

Player proporciona diferentes librerías cliente en distintos lenguajes de programación (C++, Tcl, Java y Python), lo que permite escribir programas cliente en cualquiera de estos lenguajes y ejecutarlos en cualquier máquina que disponga de una conexión de red con el robot donde se ejecute el servidor *Player*. Actualmente *Player* está soportado en *GNU-Linux*, *Solaris* y *BSD*.

Stage

Stage es un software de simulación de sistemas robóticos y sistemas percepto-efectores que evolucionan también en un entorno simulado, normalmente de interior, de dos dimensiones. *Player* puede ejecutarse utilizando *Stage* como sistema robótico simulado, en lugar de un sistema real. Los clientes interactuan con el servidor *Player* de la misma forma que lo harían con el sistema real. Esto facilita enormemente el desarrollo de aplicaciones para este tipo de sistemas, y de hecho es una de las principales bazas que han hecho que sea uno de los proyectos software en robótica más utilizados. En la figura 2.1 se observan varios robots en un entorno simulado utilizando *Player/Stage*.

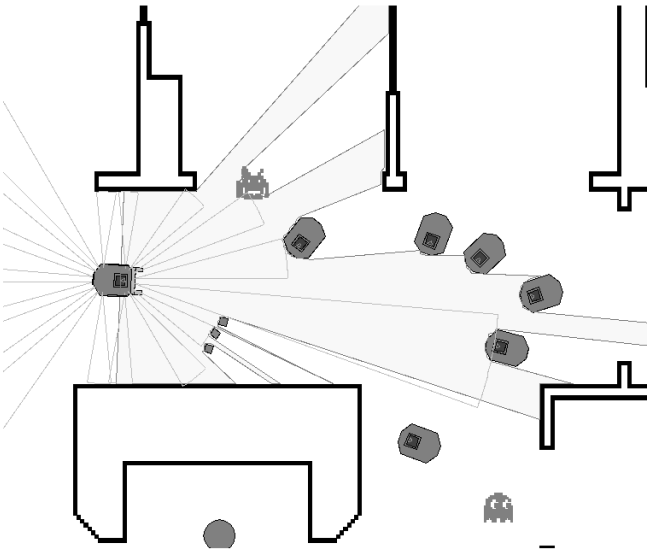

Figura 2.1: Entorno y robots *Pioneer* simulado con *Player/Stage*.

2.1.2. 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.

2.1.3. 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.

2.1.4. 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.

2.1.5. CoolBOT

En el *Instituto Universitario SIANI* (Sistemas Inteligentes y Aplicaciones numéricas en Ingeniería) se ha desarrollado *CoolBOT* [Domínguez,2003],[Domínguez,2007], 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 [Steenstrup83-jcss,1983] [Stewart97-ieee-tse,1997] 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 Linux. En el capítulo 3 se detalla en más profundidad el diseño y funcionalidad de este *framework*, ya que se ha seleccionado como herramienta para la realización del presente proyecto. La motivación de esta elección es, además de por la disponibilidad de la herramienta en el laboratorio del Instituro Universitario SIANI, por sus características y posibilidades a la hora de utilizar componentes software ya existentes y operativos.

2.2. Desarrollo de sistemas distribuidos

Los sistemas distribuidos presentan una serie de caracteristicas ventajosas frente a aplicaciones centralizadas (no distribuidas o *standalone*). Estas ventajas son aportadas principalmente por cinco características: rendimiento, confiabilidad, escalabilidad, flexibilidad y ahorro [Coulouris,2001].

- Rendimiento.

  Existen determinados problemas cuya solución demanda un gran costo computacional. Los recursos en una red de ordenadores se multiplican frente a los disponibles en una sóla máquina. Por tanto, disponer de un sistema donde muchos elementos de cómputo colaboren y se comuniquen se plantea como la arquitectura ideal para este tipo de casos.

- Confiabilidad.

  En el caso de sistemas distribuidos cada componente procesa de forma independiente haciendo uso de las comunicaciones para operar como parte del sistema. Cuando se produce un fallo en una unidad de proceso, ese fallo sólo involucra a esa entidad, de forma que si una máquina *cae* el sistema puede recuperarse con relativa facilidad y continuar funcionando.

- Escalabilidad.

  Este tipo de sistemas permite añadir o eliminar unidades de proceso sin más que incorporarla o quitarla de la red de comunicaciones del sistema. Esto permite reemplazar unidades que hayan tenido algún fallo o incluso crecer o reducir el sistema en función de la demanda de recursos.

- Flexibilidad.

  La flexibilidad de estos sistemas está justificada por su escalabilidad. Es por esto que presentan la posibilidad de que la carga de trabajo se distribuya dinámicamente. En un momento dado durante la ejecución de una tarea pueden demandarse o liberarse recursos, el sistema podría reorganizar las tareas entre las unidades disponibles.

- Ahorro.

  Como última característica cabe destacar el ahorro y eficacia en el uso de recursos que permiten este tipo de sistemas. Esto se debe al hecho de poder compartir los recursos al estar inerconectados en una red.

2.2.1. Middleware para comunicaciones

El desarrollo de aplicaciones distribuidas conlleva una serie de requerimientos que complican las tareas de diseño en comparación a aplicaciones homónimas de tipo *standalone*. Trabajar a un alto nivel permite obtener mayor flexibilidad y eficiencia, frente a las complicaciones que se presentan al programar a un nivel bajo de abstracción, donde las hetereogenidades de los sistemas de red deben ser manejadas por el programador. Es aquí donde puede entrar en juego una capa de abstracción software denominada *middleware*.

El *middleware* proporciona una capa que ofrece servicios orientados a la conectividad de aplicaciones, logrando tener una *API* común e independiente de la plataforma sobre la que una aplicación distribuida se ejecute. A su vez cada *middleware* puede estar estructurado en un conjunto de capas, cada una de las cuales proporciona una funcionalidad específica.

Como muestra la figura 2.2, el *middleware* es una capa software que reside entre el sistema operativo subyacente y las aplicaciones. Cada *middleware* puede estar estructurado de varias formas, pero lo habitual es que a su vez se subdivida en varias capas. En la figura se presenta un ejemplo ilustrativo que se subdivide en cuatro capas:

- Servicios específicos del dominio.
- Servicios comunes.
- Modelos de programación distribuida.
- Infraestructura básica.

La capa de más bajo nivel es la capa que proporciona acceso a las funciones nativas del sistema operativo (*SO*). Estas funciones son todas aquellas relacionadas con mecanismos de comunicación

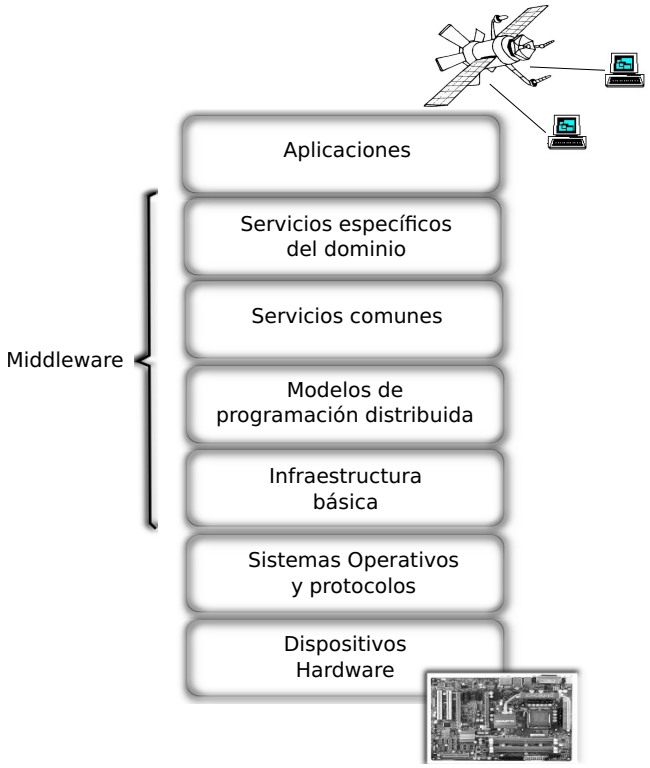

Figura 2.2: Estructura de un middleware.

y concurrencia (sockets, hilos, procesos, cerrojos, variables condición, ficheros, etc.). De esta forma se abstraen aspectos dependientes del *SO*, con lo que es la capa orientada a otorgar la portabilidad que ofrece el *middleware*.

La siguiente capa es la que porporciona una serie de modelos del alto nivel para la programación de aplicaciones distribuidas. Estos modelos se construyen con una serie de componentes reutilizables y comunes en aplicaciones de red, como pueden ser la gestión de conexiones y la serialización de datos también denominada por el término en inglés *marshalling*,

A continuación aparece una capa en el *middleware* orientada a aportar servicios comunes como pueden ser el control de concurrencia, notificaciones de eventos, servicios de *logging* o registro de sucesos.

La última capa de este *middleware* de ejemplo contribuye añadiendo servicios específicos de un dominio determinado de aplicaciones: comercio electrónico, telecomunicaciones, aplicaciones de control de procesos, etc. Esta capa de servicios es la menos madura en los *middlewares* actuales, debido en ciertos aspectos a la falta de capas de servicios comunes suficientemente desarrolladas y estables sobre las que se creen capas de servicios específicos.

Existe multitud de paquetes de software *middleware* para el desarrollo de aplicaciones distribuidas, cada uno de los cuales aporta una serie de servicios. La decisión de utilizar uno u otro dependerá de los requerimientos de cada aplicación. En el capítulo 4 se presenta el análisis de

requisitos del sistema y los criterios seguidos en la selección del *middleware* para el desarrollo de este proyecto.

Capítulo 3

CoolBOT

CoolBOT [Domínguez,2003][Domínguez,2007] es un *framework* o marco de programación orientado a componentes [Szyperski,1999] 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][Szyperski,1999].

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][Szyperski,1999], 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.

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

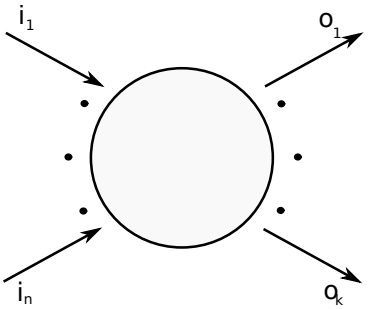

Figura 3.1: Vista externa de componente.

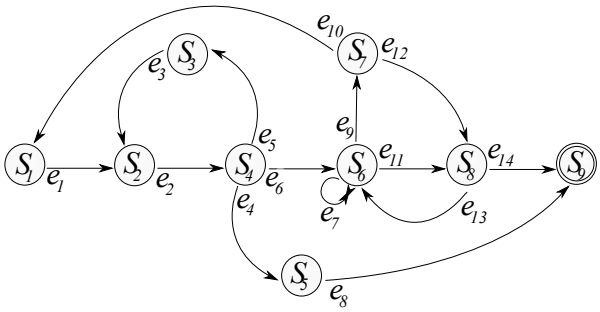

Figura 3.2: Vista interna de componente.

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.

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.

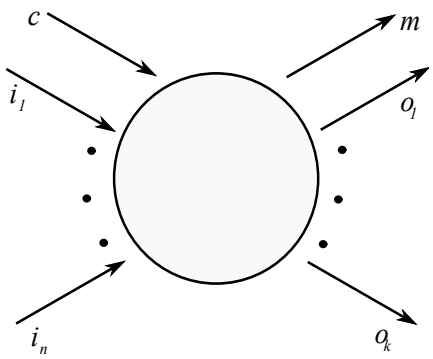

Figura 3.3: Vista externa de componente con puerto de control y monitorización.

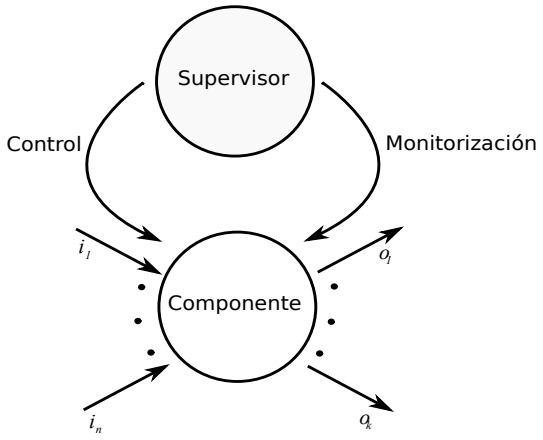

Figura 3.4: Bucle común de control.

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.

Tabla 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.

Tabla 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. 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.

Algunos de los eventos que provocan transiciones son internos al componente (*exception, nex,*

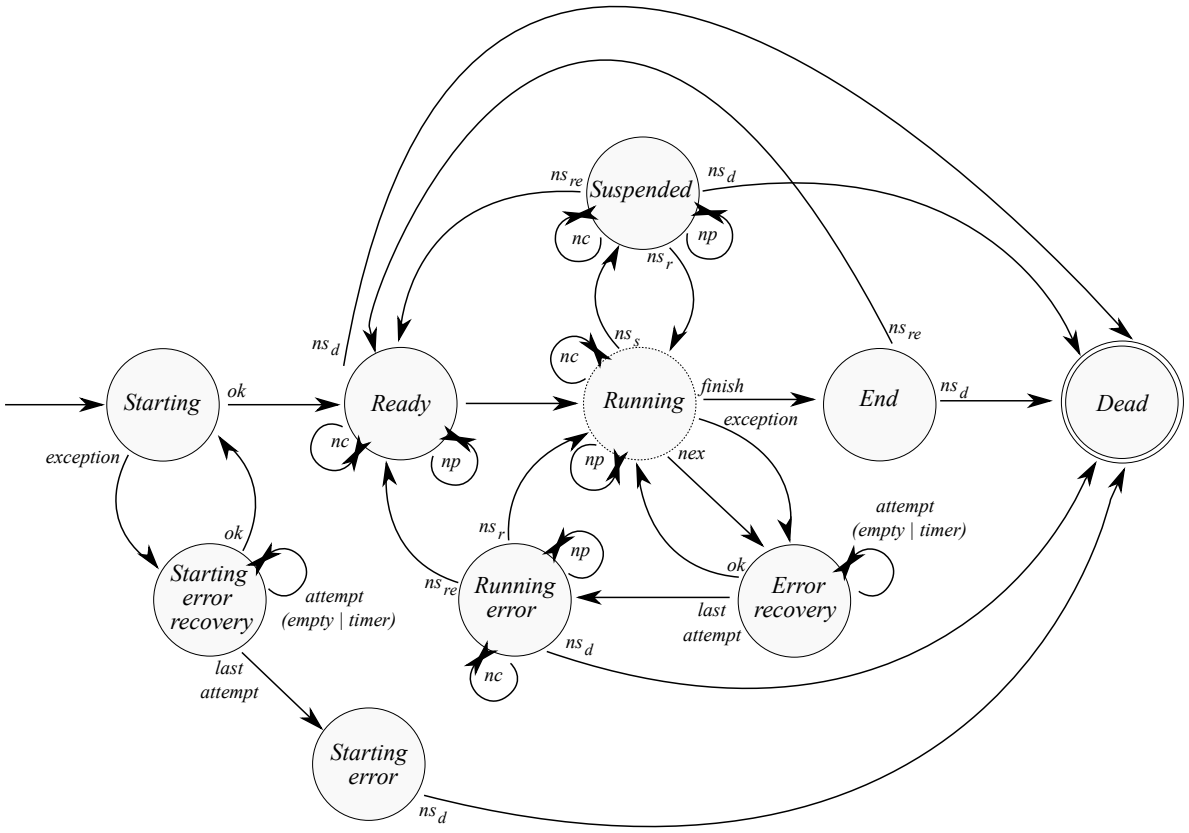

Figura 3.5: Autómata por defecto.

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.

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

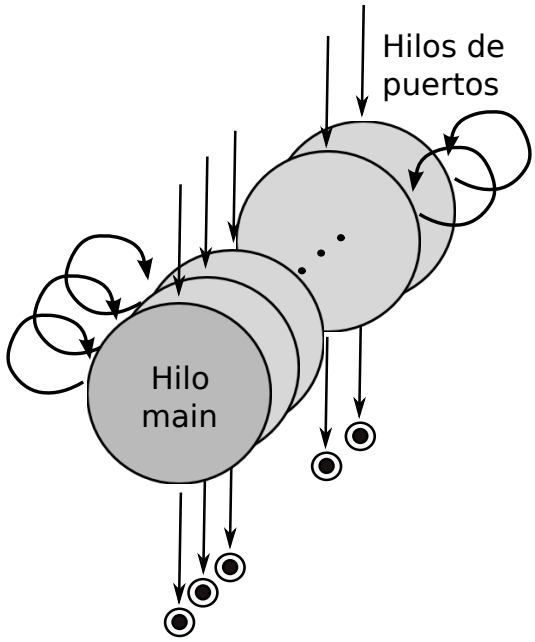

Figura 3.6: Componente multihilo.

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.

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

Tabla 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*.

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 hard-

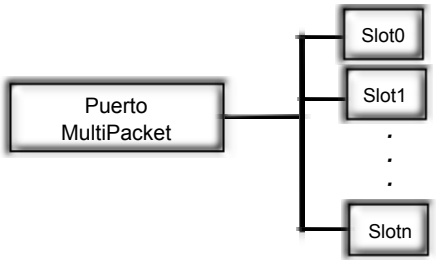

Figura 3.7: Puerto *MultiPacket*.

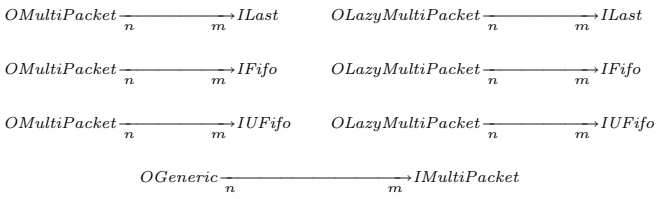

Figura 3.8: Conexiones MultiPacket simples ($\forall\, n,m \in \mathbb{N}; n,m \geq 1$).

ware 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.

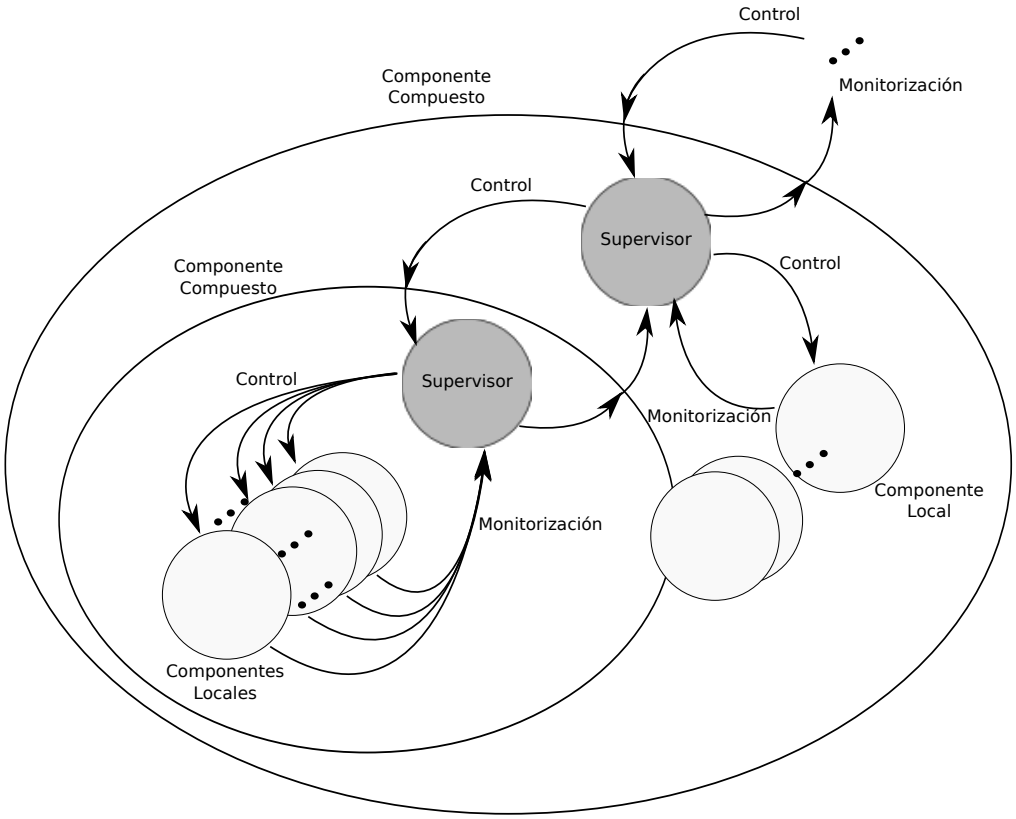

Figura 3.9: Jerarquía de componentes.

Capítulo 4

Análisis

En este capítulo se presentarán las tareas de análisis llevadas a cabo durante la realización de este proyecto. Se presentará la fase de inicio del proyecto donde se comenta la metodología de desarrollo aplicada, los recursos necesarios del proyecto y el plan de trabajo seguido. En la siguiente sección se realizará la especificación de requisitos partiendo de los casos de uso elaborados.

Finalmente se describen en detalle una serie de herramientas analizadas para ser utilizadas con el fin de satisfacer los requisitos del sistema. En cada una de estas descripciones se incide en las ventajas que presentan estas herramientas, argumentando su elección. Concretamente se presenta una posible representación intermedia de datos:*Common Data Representation (CDR)* [OMG,2002], utilizada en la *Common Object Request Broker Architecture (CORBA)* [OMG,2004]; y el análisis de varios *middleware* orientados a objetos para el desarrollo de sistemas distribuidos. Además se describe brevemente la librería gráfica *GTK* como herramienta para la creación de *vistas* de componentes.

4.1. Fase de inicio

4.1.1. Metodología de desarrollo

La ingeniería del software detalla una serie de metodologías para estructurar y organizar el proceso de desarrollo de software, lo que sienta las directrices para lograr productos que cumplan unos requisitos, tanto funcionales como de calidad, planteados en el inicio de todo proyecto. En este caso se ha seguido el proceso unificado de desarrollo (del inglés *Unified Process (UP)* [Jacobson,2000]), que proporciona un marco de trabajo genérico, dirigido por casos de uso, centrado en la arquitectura, iterativo e incremental. El *UP* define un proceso que porporciona una guía para ordenar las actividades de desarrollo, dirige las tareas, especifica artefactos necesarios y ofrece criterios de control y valoración, tanto de los productos como de las actividades. Para el desarrollo del proyecto de infraestructura software que nos ocupa se han utilizado y ajustado al contexto los elementos que el *UP* presenta.

La primera de las características citadas sobre el *UP* es que está dirigido por casos de uso y centrado en la arquitectura. Los casos de uso representan los requisitos funcionales del sistema y ayudan a identificar los puntos clave para los usuarios, entendiendo como usuarios tanto a personas

físicas como a otros sistemas software. La arquitectura software, en cambio, nos proporciona las vistas del sistema en construcción, incluyendo los aspectos estáticos y dinámicos del mismo. Serán los casos de uso los que proporcionen un hilo conductor durante todo el proceso de desarrollo, encajándose en la arquitectura que ya presenta el *framework CoolBOT*. Así los casos de uso nos darán la nueva funcionalidad que se pretende añadir, mientras la arquitectura nos proporcionará la estructura.

Como última característica clave del *UP* cabe destacar su metodología de desarrollo iterativa e incremental. Con este enfoque el desarrollo se organiza en iteraciones, que darán como resultado un sistema que podrá ser probado, integrado y ejecutado como parte del *framework CoolBOT*. En cada iteración obtenemos un sistema incompleto, pero que permite la prueba y por tanto una retroalimentación para sucesivas ampliaciones y mejoras. Es decir, como resultado de cada fase de refinamiento, se tiene un subconjunto con calidad de lo que será el sistema final [Larman,2003]. En general, las iteraciones en el ciclo de vida iterativo abordan nuevos requisitos y amplian el sistema, sin embargo pueden centrarse en una mejora del subsistema, por ejemplo de rendimiento. La retroalmentación que se logra con esta metodología aporta un conocimiento práctico de gran valor. Una realimentación temprana en el desarrollo de un producto software permite, además de clarificar requisitos, identificar si el diseño y la implementación parcial se están encauzando correctamente. A nivel práctico esto se traduce en que se atacan primeramente las partes críticas del sistema, logrando una arquitectura estable de forma temprana, que se va completando en iteraciones posteriores [Jacobson,2000]. Así hemos podido resolver y probar en una fase inicial las decisiones cruciales de diseño, que en el proyecto que nos ocupa están ligadas a la eficiencia de envío de datos por la red y su repercusión en las frecuencias de funcionamiento de los componentes *CoolBOT*, así cómo a la facilidad de mantenimiento y uso del sistema final.

Como comienzo del proyecto se afronta una fase de inicio previa, figura 4.1. Durante esta fase se realiza un análisis inicial que ayuda a realizar la planificación temporal del desarrollo y a la comprensión del contexto. Tras esta fase se habrá realizado una lista de características describiendo el contexto y alcance del sistema, favoreciendo la identificación de los riesgos críticos y por tanto determinando la viabilidad del proyecto. En el caso que nos ocupa se describirá el dominio del sistema, haciendo uso de casos de uso críticos que proporcionarán una primera visión a alto nivel. Partiendo del modelo obtenido y priorizando los requisitos identificados se estructurarán las siguientes fases en iteraciones, proporcionandose en cada fase una serie de artefactos [Jacobson,2000], entre los que cabe destacar las representaciones con diagramas *UML: Unified Modeling Language* de los modelos empleados como descripción de las distintas vistas del sistema en cada momento.

Figura 4.1: Fase inicial del desarrollo.

Las figuras 4.2, 4.3 y 4.4 ilustran, a nivel general, las diferentes iteraciones del desarrollo y la estructura de las estapas que se seguirán en este proyecto.

- Desarrollo de un protocolo de comunicaciones.

  Las dos primeras iteraciones se emplearán en la creación de un protocolo de comunicaciones entre componentes *CoolBOT* (figura 4.2). En la primera concretamente se desarrollarán las funcionalidades principales de dicho protocolo, refinándose y completándose, en una segunda iteración, en la que se integrará en el *framework CoolBOT*. De esta forma, el *subsistema 2* representado en la figura 4.2, dispondrá de la funcionalidad que permite las distribución de componentes *CoolBOT* en distintas máquinas y la comunicación entre los mismos usando el protocolo diseñado para tal efecto.

- Integración y prueba del soporte de red.

  En una tercera iteración, representada en la figura 4.3, se integrará el soporte de red diseñado a un componente de prueba y se realizarán los testeos necesarios. A continuación se procederá a integar la red a un sistema de componentes existente, obteniendo así un sistema de demostración (*demo*) distribuido.

- Interfaz de teleoperación.

  En las siguientes dos iteraciones se pretenderá crear una interfaz de teleoperación que, haciendo uso del protocolo implementado, permita a un usuario del *framework* monitorizar y controlar componentes *CoolBOT* (figura 4.4), en concreto para los componentes del sistema de demostración portado en la iteración anterior. Estas iteraciones aportarán una realimentación muy importante al desarrollo, ayudando a clarificar la visión de la interfaz no sólo en cuanto a usabilidad y funcionalidad, sino también sobre factores clave para los sistemas teleoperados como son fiabilidad, rendimiento y soporte.

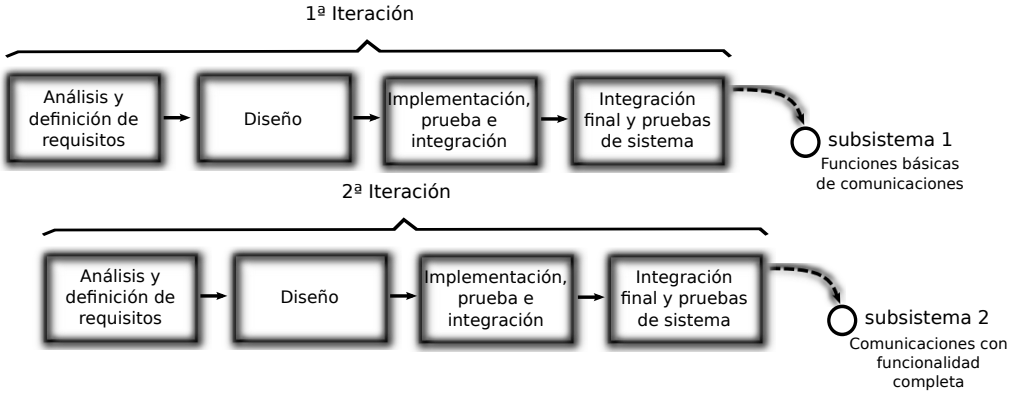

Figura 4.2: Iteraciones del desarrollo del protocolo.

Por supuesto, por las propias características previamente descritas de la metodología de desarrollo aplicada, en cualquiera de las múltiples iteraciones planificadas podrán añadirse o modificarse requisitos, así como realizarse mejoras sobre el subsistema del que se disponga en cada fase.

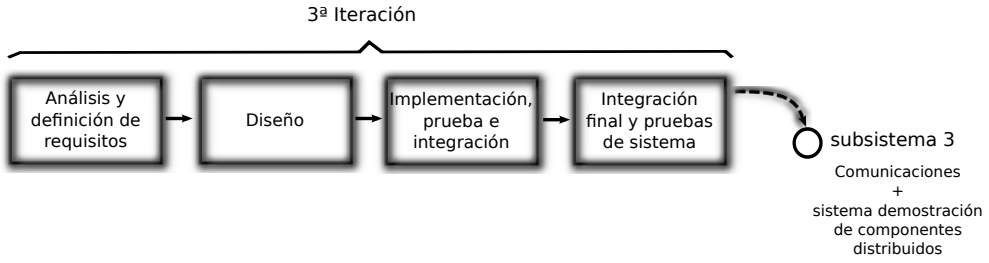

Figura 4.3: Iteraciones de la integración y creación de un sistema *demo*.

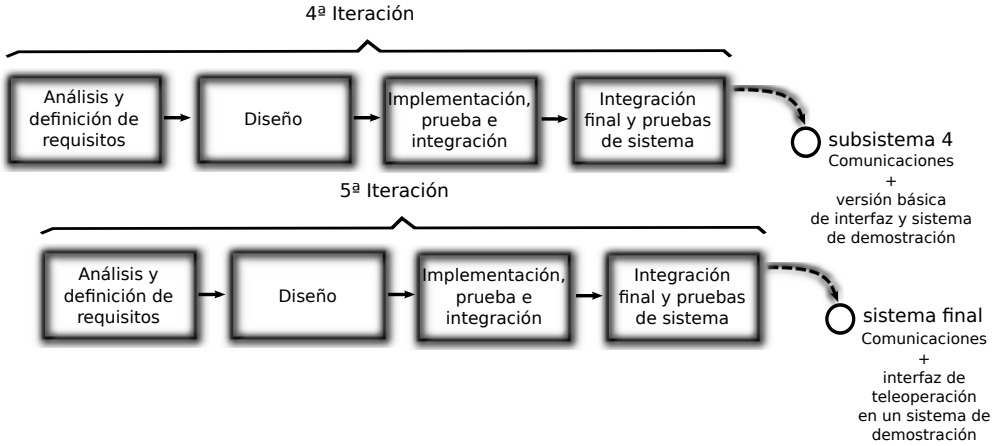

Figura 4.4: Iteraciones del desarrollo de la interfaz de teleoperación.

4.1.2. Recursos necesarios

Durante el desarrollo de este proyecto se han utilizado una serie de recursos tanto software como hardware, que se presentan a continuación.

Hardware

- Robot Pioneer 3-DX, que forma parte del sistema robótico móvil sobre el que se ilustra la operatividad del proyecto desarrollado.
- Cámara web USB, integrada en el robot Pioneer.
- PC sobre GNU/Linux, para el desarrollo con C++, prueba y simulación de software, así como la redacción de documentación.
- Red de comunicaciones, tanto wifi como Ethernet.
- Distintos equipos servidores para pruebas remotas (ejecución de software cliente-servidor).

Software

- Proyecto *Player/Stage*.
- Framework *CoolBOT*, para el desarrollo de sistemas robóticos. [Descrito en
- *IDE (Integrated Development Environment)* Kdevelop.
- Software de control de versiones.
- Software para la producción documental. 3]
- *Middleware ACE*, para el desarrollo de software de comunicaciones. [Descrito en 4.4.3]

En los siguientes apartados de describirán brevemente las herramientas software utilizadas.

IDE Kdevelop

Kdevelop un entorno integrado de desarrollo (*Integrated development Environment: IDE*), de software libre, desarrollado para el entorno de escritorio *KDE*.[1]

Este *IDE* funciona con distintos lenguajes de programación como C, C++, Java, Ada, SQL, Python, Perl, Pascal, bash script, etc. Entre sus características más destacables figuran paneles de navegación por los ficheros o por las clases del código, herramientas de edición, resaltado e indentado automático de código fuente, y completado de código. Es un editor bastante versátil y de fácil apredizaje. La versión utilizada ha sido *Kdevelop* 3.9.95.

Control de versiones CVS y GIT

En cualquier proyecto software de cierta envergadura se hace indispensable el uso de programas de control de versiones. Este tipo de software ofrece funcionalidades para llevar un control de los cambios que se efectúan sobre el software (bien código fuente o documentación). Permiten en todo momento deshacer cambios volviendo a versiones previas y facilitan el desarrollo en proyectos colaborativos.

En este proyecto se han utilizado dos sistemas distintos para el control de versiones, descritos brevemente a continuación.

CVS: Concurrent Versions System

En las primeras fases de implementación, desarrollo de clases básicas de insfraestrutura, se ha utilizado *CVS: Concurrent Versions System* [CVS]. Es un programa de software libre que utiliza una arquitectura cliente servidor. El servidor mantiene una copia maestra del proyecto en desarrollo y un historial de cambios. Los clientes pueden solicitar copias locales completas del proyecto, actualización a los ultimos cambios, historiales de cambios, modifcar la copia maestra a partir de los cambios realizados en las copias locales. Además *CVS* puede administrar diferentes

[1] Programas del entorno de escritorio *KDE* pueden ser utilizados en un entorno de escritorio *Gnome* instalando las librerías gráficas necesarias (*qt*)

ramas o línea de desarrollo de un proyecto. Por ejemplo es posible mantener una rama general mientras existe otra en la que se controlen cambios para añadir una funcionalidad nueva al software.

Las versión utilizada ha sido la 1.12.13.

GIT

Durante las fases de integración de código de este desarrollo se migraron los proyectos existentes sobre *CVS* a otro software de control de versiones denominado *GIT* [GIT]. Al igual que *CVS*, *GIT* es un proyecto de software libre que mejora sustancialmente las características del primero. Gestiona de una forma más eficiente diferentes líneas de desarrollo o ramificaciones en los proyectos que gestiona y permite una gestión distribuida de los cambios, es decir cada cliente tiene un copia local del historial de cambios. Esto permite, por ejemplo, poder unificar las ramas de historiales locales sin necesidad de cambiar la versión del servidor central. Además *GIT* permite usar varios protocolos de comunicación entre clientes y el servidor (http, ssh, git...). Como última mejora destacable frente a *CVS* se presenta el hecho de que *GIT* gestiona mucho mejor los renombrados, creación y eliminación de ficheros nuevos en proyectos, además de permitir la gestión de cambios no sólo de ficheros de tipo texto en un proyecto, sino de ficheros con cualquier tipo de formato.

La versiones utilizadas de *GIT* tanto para el servidor como para los clientes ha sido la 1.6.3.3.

Herramientas de producción documental

Como parte indispensable de todo sofware de calidad no puede ser olvidada la documentación. Por esto las herramientas orientadas hacia la creación de la misma resultan de gran ayuda e importancia. A continuación se presentan las que han sido empleadas en el desarrollo de este proyecto.

Latex

El paquete Latex [Latex] constituye un proyecto de software libre para la generación de documentos, principalmente técnicos. Utiliza macros para definir la extructura y formato de un documento y facilita enormemente la creación y modificación de escritos de gran tamaño, obteniéndose resultados de gran calidad. Este sistema se estructura en torno a un núcleo central formado por un compilador de documentos que ofrece las funcionalidades principales. Como añadido se utilizan multitud de paquetes que proporcionan características determinadas como la inclusión de cierto tipos de imágenes, tipografías, idiomas, etc.

Con *Latex* a diferencia de procesadores tipo *WYSIWYG: del inglés What you see is what you get*, la edición se dirige por el contenido y no por la forma, utlizando para esto los comandos y dejando la gestión del aspecto al compilador.

Existen distintos editores para *Latex*, en concreto se ha utilizado *Kile* versión 2.0.83, editor de software libre para el entrono de escritorio *KDE*, y disponible en los repositorios de la mayoría de la distribuciones de GNU/Linux.

Dia

Siguiendo el Proceso Unificado (*Unified Process: UP*) de desarrollo, la documentación del software utiliza *UML* en ciertas partes, con lo que es necesaria alguna herramienta para la creación de este tipo de diagramas. *Dia* [DIA] es un editor de diagramas desarrollado como software libre. Permite la edición de multitud de tipos de diagramas: de flujo, de redes y de circuitos electrónicos entre otros, además de diagramas *UML*. Se ha utilizado este editor pues tras comprobar algunas otras alternativas (*VioletUML, ArgoUML, Umbrello*), *Dia* era más intuitivo, de sencillo aprendizaje y se obtienen diagramas *UML* bien estructurados y de buena calidad.

La versión de *Dia* utilizada ha sido la 0.97.

Inkscape

Inkscape [InkScape] es un editor de gráficos vectoriales licenciado como software libre. Soporta formas, texto, lineas, uso de capas, gradientes y agrupamientos de objetos. Se trata de una herramienta de dibujo potente e intuitiva en su uso y permite crear fácilmente cualquier diagrama utilizando tanto formas básicas como en combinacion con imágenes. *Inkscape* ha sido utilizado en su versión 0.47 para la creación de todos los diagramas explicativos utilizados en este documento.

4.1.3. Plan de trabajo

A continuación de presenta el plan de trabajo a seguir durante este desarrollo.

1. Estudio de herramientas de desarrollo de software de comunicaciones. Se estudiarán algunas librerías y *middleware* para la gestión de los recursos en la creación de software distribuido.

2. Análisis de requisitos para el desarrollo del protocolo de comunicaciones. Se analizarán las necesidades en las comunicaciones entre componentes *CoolBOT* para obtener los requisitos necesarios.

3. Diseño del protocolo de comunicaciones. En base a los requisitos previamente identificados se diseñará un protocolo del nivel de aplicación que usarán los componentes *CoolBOT* en sus comunicaciones sobre *TCP/IP*.

4. Implementación del protocolo de comunicaciones en C++ e integración del mismo en *CoolBOT*. Se implementarán en C++ las clases necesarias para la implementación del protocolo diseñado así como las operaciones que hayan sido establecidas.

5. Creación de componentes *CoolBOT* de prueba. Se diseñará e implementará un componente básico al que se le implementará el soporte necesario para añadir la operatividad vía red *TCP/IP*. Realización de pruebas de funcionamiento.

6. Integración de las comunicaciones en un sistema robótico de demostración sobre *CoolBOT*. Se integrarán todos los recursos y operaciones del protocolo diseñado en un sistema robótico ya existente utilizando la versión operativa anterior de *CoolBOT*. Adaptación de cada componente *CoolBOT* del sistema de demostración a la nueva versión del *framework CoolBOT*

con el soporte de red distribuido desarrollado en las fases previas. Realización de pruebas de funcionamiento de todo el sistema de forma distribuida.

7. Análisis de requisitos para el desarrollo de la interfaz de teleoperación. Se estudiará las actuales interfaces gráficas de los componentes del sistema portado en la etapa anterior y la librería gráfica con la que han sido desarrolladas (*GTK*).

8. Diseño de la interfaz de teleoperación. Se diseñarán los cambios necesarios para que las interfaces gráficas de componentes se comuniquen vía red con el sistema de demostración de componentes distribuidos.

9. Implementación de la interfaz de teleoperación. Se implementarán las operaciones y estructuras necesarias de comunicaciones en las interfaces gráficas de componentes del sistema de demostración actual.

10. Pruebas y depuración. Se realizarán pruebas en distintos escenarios, es decir distribuyendo el sistema de distintas formas.

11. Generación del documento de Proyecto Fin de Carrera. Se redactarán las versiones finales del documento de memoria, haciendo uso de toda la documentación generada durante el proceso de desarrollo.

4.2. Modelo de dominio

En esta sección se presenta el modelo del dominio en el que se enmarca el desarrollo de este proyecto. Se trata de una descripción de los conceptos y sus relaciones en *CoolBOT* (presentados en el capítulo 3) necesarios para la elaboración de los casos de uso.

El diagrama de la figura 4.5 refleja los conceptos *puerto de entrada* y *puerto de salida*. Se trata de las dos modalidades de puertos existentes en *CoolBOT*. A partir de estas dos posibilidades existen tipos específicos de puertos de entrada (*IPortSinglePacket* e *IPortMultiPacket*) y salida (*OPortSinglePacket* y *OPortMultiPacket*). Los puertos del tipo *Single* (*IPortSinglePacket*, *OPortSinglePacket*) aceptan un único tipo de *paquete de puerto*, que almacena datos. Por otro lado los puertos del tipo *Multi* (*IPortMultiPacket*, *OPortMultiPacket*) tienen una serie de *Slots* a través de los cuales pueden aceptar múltiples tipos de *paquetes de puerto*, uno por *Slot*. Un *componente CoolBOT* tiene una serie de tipos de puertos, tanto de entrada como de salida, a través de los cuales puede intercambiar *paquetes de puerto* con otros *componentes*. Cada *componente* puede tener una serie de *vistas* que lo representa gráficamente. Por otro lado un *Sistema Robótico* está constituido por un conjunto de *componentes*, así como una *Interfaz de Teleoperación* tiene varias *vistas* de componentes que le permiten monitorizar el sistema. Desde la *Interfaz de Teleoperación* el sistema robótico puede ser controlado.

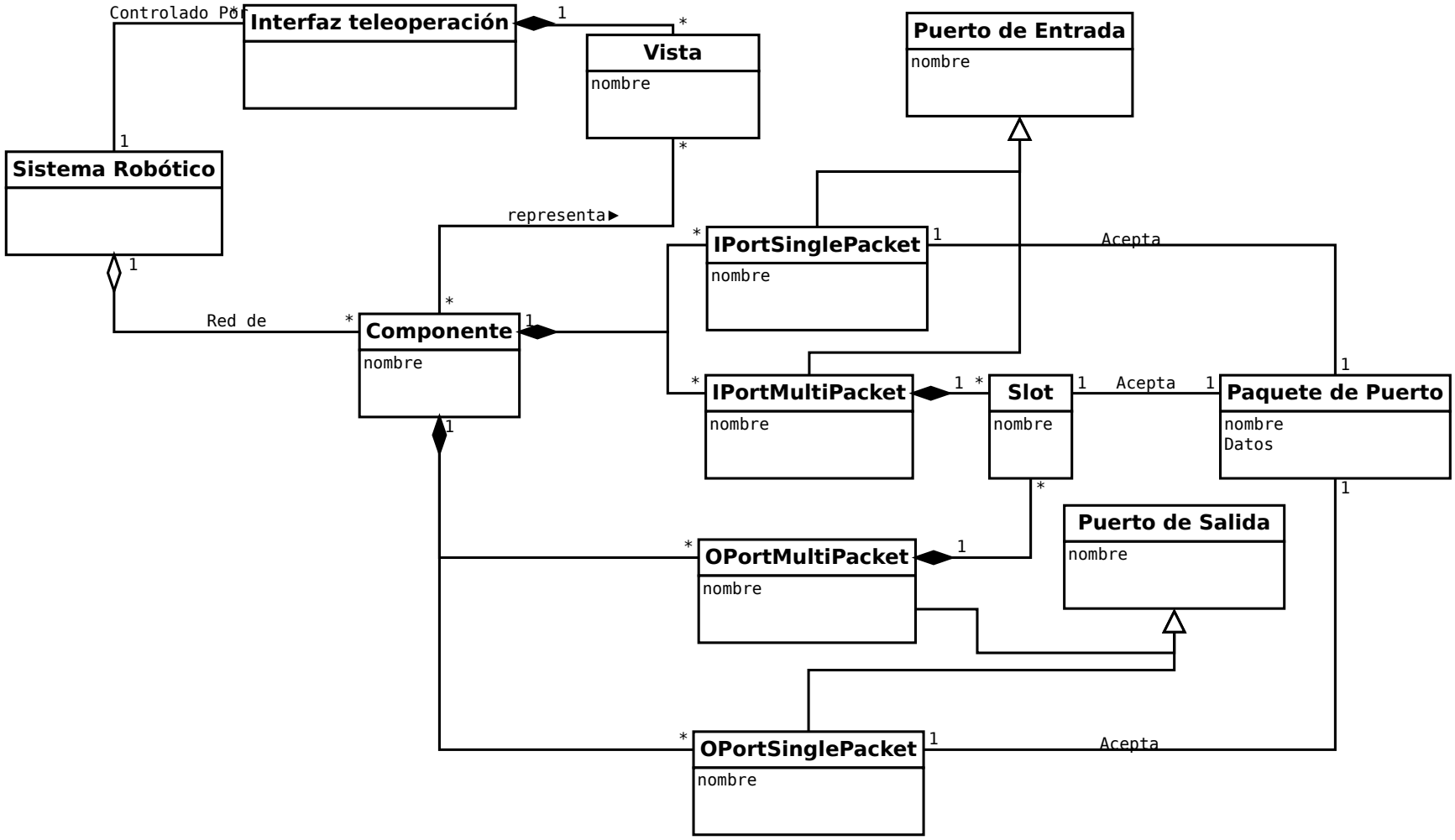

Figura 4.5: Modelo de dominio de CoolBOT.

A continuación se describe brevemente como ejemplo ilustrativo un sistema de evitación de obstáculos basado en campos de potenciales [Koren,1991] implementado utilizando *CoolBOT*, lo que nos proporciona una visión de cómo operan los diferentes elementos dentro de un sistema robótico completo. Se trata de un sistema sencillo de cuatro componentes, figura 4.6. El componente *Pioneer* integra los sensores y actuadores proporcionados para un robot Pioneer de *ActiveMedia*. El componente *PF Fusion* se encarga de tomar una desición de movimiento en base a las formas que se presenten en el entorno, mientras que el componente *Strategic PF* transforma comandos de movimiento de alto nivel en su correspondiente comando de *campo-de-potencial* (*potential field*). Por último, el componente *Joystick Navigation* permite controlar al robot usando un joystick. La integración de estos cuatro componentes permite crear un sistema en el que el robot se moverá en base a unas ordenes de alto nivel, los comandos que recibe el componente Strategic PF, comandos de dirigirse a un punto específico de manera autónoma evitando obstáculos.

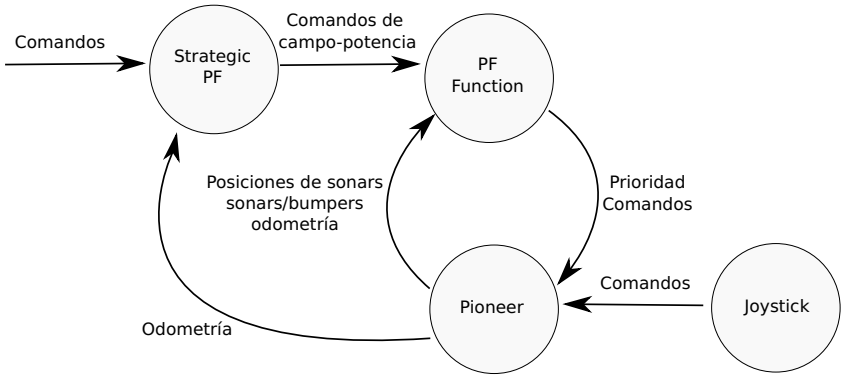

Figura 4.6: Sistema evitador.

La figura 4.6 ilustra el interconexionado entre los componentes que conforman el sistema de control del robot móvil para realizar la tarea descrita previamente, así como la información se distribuye en paquetes de puertos entre los componentes que operan como máquinas de flujo de datos. Cada componente funciona a la frecuencia a la que le llegan los datos, de forma que estarán ociosos si no existe información en alguno de sus puertos de entrada.

4.3. Requisitos del sistema

En esta sección se enumeran los requisitos del sistema. Primeramente se presenta el modelo de casos de uso. Para continuar con la especificación de requisitos identificados a partir de dicho modelo.

4.3.1. Modelo de casos de uso

En esta sección se expone el modelo de casos de uso desarrollado para la identificación de requisitos del sistema. Para ello se presenta una primera identificación de los tipos de usuarios (actores) y por otro de las acciones que pueden realizar con el software (casos de uso).

Actores

Podemos identificar a tres grupos de usuarios del sistema:

- **Operador de sistemas.**

  En este grupo se identifican los operadores de sistemas róbóticos que hayan sido desarrollados sobre *CoolBOT*. Se trata de los usuarios que operen de forma remota cualquier tipo de sistema que use *CoolBOT* como base.

- **Desarrollador de sistemas robóticos.**

  En este grupo se engloba a todos los posibles usuarios programadores de sistemas robóticos sobre *CoolBOT*. Por un lado existen los *desarrolladores de componentes CoolBOT*, encargados de la creación de componentes individuales. Además existen los usuarios *integradores de componentes CoolBOT* para la construcción de sistemas robóticos a partir de una red de componentes.

- **Desarrollador del *framework CoolBOT*.**

  Como último grupo de usuarios aparecen los desarrolladores del propio *framework*, es decir cualquier usuario que modifique *CoolBOT*, ampliando funcionalidades, modificando las ya existentes, corrigiendo posibles errores, o añadiendo nuevas facilidades, abstacciones y funcionalidades.

Casos de uso

En los casos de uso que a continuación se exponen se reflejan exclusivamene las operaciones que involucran a las funcionalidades que se pretenden desarrollar en este proyecto.

- **Usuario Operador (figura 4.7)**.

  El usuario operador realiza las acciones sobre el sistema utilizando una interfaz de teleoperación con un sistema robótico determinado. Haciendo uso de dicha interfaz el operador controla el sistema mediante el envío de comandos. Además el operador monitoriza la actividad del sistema robótico viendo una presentación determinada de los datos según el sistema en cuestión.

- **Desarrollador de sistemas robóticos (figura 4.8).**

  Un usuario desarrollador de sistemas robóticos utilizará *CoolBOT* como herramienta. Este grupo de usuarios realiza seis operaciones usando el *framework*.

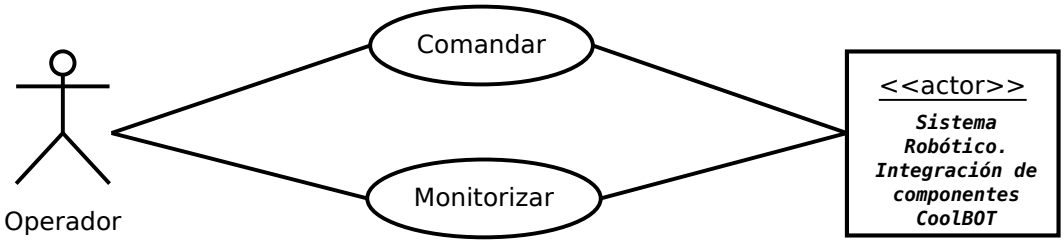

Figura 4.7: Usuario Operador de sistemas robóticos.

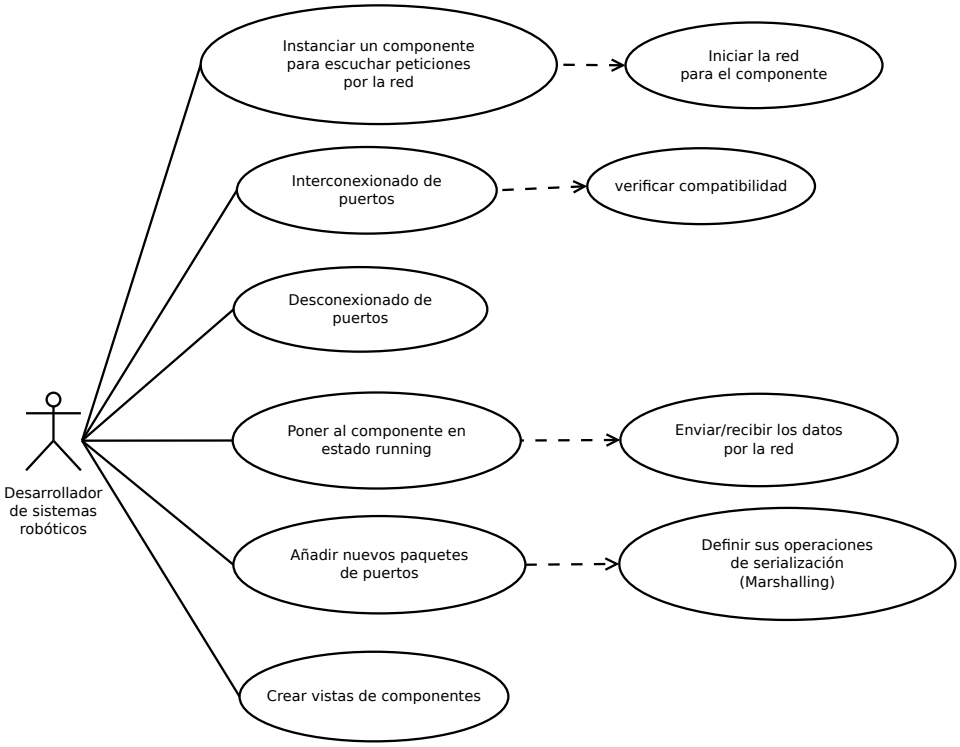

Figura 4.8: Usuario Desarrollador de sistemas robóticos.

1. Instancia un componente por red.

   Tras la creación de un componente *CoolBOT*, una vez definida su funcionalidad y sus entradas y salidas, este puede ser instanciado de forma que haga uso de la capa de red que *CoolBOT* le porporciona, es decir, enviar datos hacia componentes remotos y recibirlos. Para instanciar el componente y activar sus operaciones de red, el desarrollador de sistemas robóticos sólo tendrá que indicar un puerto *TCP* por el que el componente escuche peticiones recibidas por la red *TCP/IP*.

2. Interconexionado de puertos.

Este usuario realiza el interconexionado de puertos entre los componentes que integran el software de un sistema robótico dado. Este interconexionado debe ser tanto remoto como local, esto debe ser transparente para el usuario. Este usuario requiere poder conectar determinados puertos entre dos componentes ya instanciados para así ir construyendo el sistema al completo. La conexión siempre se realiza para un puerto de salida con un puerto de entrada y las operaciones que la capa de red de *CoolBOT* lleve a cabo serán transparentes para este usuario. Sólo se requiere que se aporten dos datos por cada componente remoto a conectar, estos serían la dirección de internet o *IP* y el puerto *TCP* de escucha. Esta operación involucra siempre una comprobación de la compatibilidad de la conexión a realizar (tipos de puertos y tipos de los paquetes de puerto que aceptan, detallados en 3.6). Dicha comprobación se realiza de manera dinámica cuando la integración se pone en ejecución.

3. Deconexionado de conexiones de puertos.

   En un momento dado un desarrollador puede querer desconectar un determinado puerto porque ya no se requieren más envíos hacia otro componente, bien por finalizar la aplicación o bien por alguna situación concreta durante la ejecución del sistema. Al igual que la creación de conexiones, las desconexiones se realizan desde un puerto de salida a un puerto de entrada, sin requerirse ningún tipo de chequeo determinado. Si la conexión no se hubiera realizado entre los puertos que se pretende desconectar, simplemente la operación no tendrá ningún efecto. De nuevo el usuario debe indicar la dirección *IP* donde reside el componente remoto y puerto *TCP* de escucha del componente del que se pretende desconectar. Al igual que el conexionado debe ser completamente transparente, ya sea un desconexionado remoto o local.

4. Añadir nuevos paquetes de puertos.

   El usuario puede crear paquetes de puerto diferentes a los ya existentes en *CoolBOT* y así adaptarlos a las necesidades de su sistema. Estos paquetes almacenan los datos que intercambian los componentes en su funcionamiento. Un ejemplo puede ser un paquete para enviar datos tomados por una cámara de abordo en un robot, que puede ser nombrado como (*CameraPacket*).

   Además, un desarrollador de sistemas puede necesitar tipos determinados de datos que viajarán en los paquetes de puertos, bien en los ya existentes o bien nuevos. Por ejemplo, para un tipo de paquete de puerto ya creado como *CameraPacket*, las estructuras que dentro almacena presentan los datos de una cámara de una forma determinada, por ejemplo una imágen *RGB*, donde cada pixel se representa por sus tres componentes de color (*red, green, blue*). Un desarrollador de sistemas robóticos puede no estar interesado en esa representación de los datos y sólo necesitar una parte marcada dentro de una imágen, podría ser aquella donde se detecte una cara. Por tanto, el paquete *CameraPacket*, además de envíar la imágen en su representación *RGB*, requiere indicar una región dentro de la misma, lo que puede marcarse indicando el pixel superior izquierdo, el ancho y alto de la zona a marcar. Por tanto los tipos de datos que viajan en el paquete necesitan ser cambiados o crear un tipo nuevo, dando de esta forma libertad al desarollador de sistemas para la definición de los mismos. Esta operación obliga a que el

usuario defina qué datos en concreto viajarán por la red, indicando también el orden en que se serializarán. Con esto se concretan cuáles son los datos necesarios a intecambiar y se logra una correcta recepción, además de un envío eficiente. Este paso es crucial, ya que determinará cómo se transformarán los datos para una adecuada interpretación de los mismos entre diferentes máquinas.

5. Crear vistas *CoolBOT*

   El desarrollador de sistemas robóticos, necesita en ocasiones de una interfaz gráfica donde poder mostrar los datos provenientes de componentes o desde la que enviar ciertos comandos. En esta interfaz la presentación de los datos varía en función de la naturaleza de los mismos en busca de una representación de fácil interpretación. Estos usuarios tienen libertad para decidir cuál es la mejor presentación de datos según sus necesidades, a esto es a lo que se denomina *vista*.

- **Desarrollador de CoolBOT (figura 4.9).**

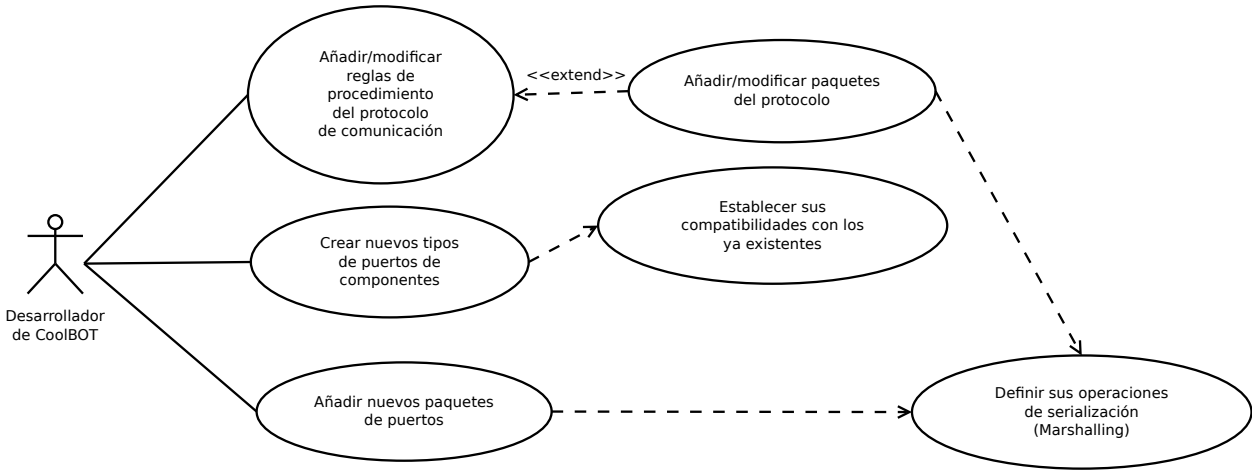

Figura 4.9: Usuario Desarrollador de CoolBOT.

En cuanto a los desarrolladores de *CoolBOT*, estos realizan tres operaciones.

1. Añadir/modificar reglas de procedimiento al protocolo de comunicación.

   Para permitir la escalabilidad de *CoolBOT* frente a requerimientos futuros entre componentes, los desarrolladores del *framework* disponen de la posibilidad de añadir o modificar reglas de procedimiento del protocolo de comunicación entre componentes. Añadir una nueva regla de procedimiento consiste en formalizar una serie de intercambios de mensajes del protocolo para obtener una nueva funcionalidad en las comunicaciones. Para tal fin, es posible que fuera necesario añadir o modificar tipos de mensajes (paquetes de datos), al conjunto ya existente en el protocolo.

2. Crear nuevos tipos de puertos de componentes.

   Ante requerimientos futuros, puede aparecer la necesidad de crear nuevos tipos de puertos de componentes y añadirlos a los ya existentes (detallados en 3.6). En tal caso los desarrolladores del *framework CoolBOT* deben indicar las relaciones de compatibilidad con los ya existentes, relaciones que la infraestructura de red debe conocer para permitir o rechazar conexiones entre componentes distribuidos.

3. Añadir nuevos paquetes de puertos. Un desarrollador del *framework CoolBOT* tiene la posibilidad de crear nuevos paquetes de puerto de componentes, que serán intercambiados por las conexiones de puertos.

   Además esta operación incluye añadir nuevas clases de datos que se envíen en paquetes de puertos de componentes. En cualquier caso el desarrollador debe indicar de nuevo cómo será la serialización de esos datos: qué datos y el orden de *marshalling* de los mismos.

En todas aquellas operaciones en las que se crean datos que pueden ser enviados por la red (operaciones 1, 3 y 4) el desarrollador establecería cuales son los datos concretos a intercambiar con otro componente en el nuevo paquete. Para todo paquete de datos o estructura de datos almacenada en los mismos como paquetes de puertos, es necesario que se indiquen los datos y el orden en que serán serializados.

4.3.2. Especificación de requisitos

Partiendo de los casos de uso se identifican los siguientes requisistos del sistema:

- Operador.

  El operador debe disponer de una **interfaz de teleoperación** con el sistema a manejar. En esta interfaz se presentarán los datos considerados necesarios para un correcto seguimiento del sistema robótico en general y de cada componentes del mismo. Así mismo, ofrecerá los mecanismos necesarios para que el usuario operador pueda enviar mensajes de control, bien a un componente en particular, o bien comandar una tarea a realizar por todo el sistema en conjunto. Dicha interfaz será creada por un usuario desarrollador de sistemás robóticos y estará adaptada al sistema en particular utilizando diferentes **vistas**, de forma que muestre los datos referentes a los componentes concretos que integren el sistema.

- Desarrollador de sistemas robóticos.

  Un usuario de *CoolBOT* (integrador de componentes, desarrollador de componentes) que quiera diseñar sistemas robóticos capaces de funcionar de forma distribuida, debe disponer de **operaciones que permitan la comunicación vía red** de los componentes *CoolBOT*. Las **comunicaciones se realizarían de la forma más transparente posible** para estos usuarios, aunque es necesario que aporten ciertos datos a la hora de realizar estas operaciones, estos se reducen a una dirección de internet o *IP* y un puerto *TCP*.

- Desarrollador del *framework CoolBOT*.

  En cuanto a este grupo de usuarios, uno de sus requerimientos importantes es la **facilidad de mantenimiento** del software. Es importante obtener un **software escalable**, donde las futuras funcionalidades de *CoolBOT* se puedan cubrir con modificaciones modulares.

Requisitos de comunicación entre componentes *CoolBOT*

En este apartado se pretende describir el nuevo modelo de comunicaciones entre componentes *CoolBOT* y sus requerimientos. Este modelo se obtiene de la extensión del modelo actual de conexionado de puertos de entrada y salida entre componentes con el requisito de que dicho conexionado se pueda realizar de manera transparente entre componentes que residan en distintas máquinas en una red de ordenadores.

Portar el modelo de intercomunicación de componentes a un modelo distribuido, debe contemplar cómo es la información que los componentes *CoolBOT* intercambian en la version operativa existente del *framework*. En particular, el diseño del *framework CoolBOT* proporciona amplia libertad al programador en la creación de componentes. Por tanto, la variedad de datos que pueden manejarse está determinada por las necesidades propias del diseño de componentes realizado por el desarrollador de cada uno de ellos.

Sin embargo, sí existen ciertas comprobaciones sobre los puertos que los componentes interconectan, principalmente en cuanto a sus tipos y compatibilidades (sección 3.5). Si la conexión entre un puerto de salida de un componente y el de entrada de un componente es compatible, se permitirá que se establezca una conexión entre ellos, a través de la que los componentes se intercambiarán paquetes de puerto de tipos determinados.

En base a los tipos de puertos descritos en 3.6 estos se pueden diferenciar en dos grupos:

- *SinglePacket*: aceptan un único tipo de paquete de puerto.
- *MultiPacket*: aceptan varios tipos de paquetes de puerto, para lo que se subdividen en un *Slot* por cada tipo aceptado.

De esta forma y siempre sujeto a las compatibilidades de conexiones presentadas en las tablas 3.3 y 3.8 es posible que un usuario programador de sistemas robóticos conecte puertos de cuatro formas, indicadas en la tabla 4.1.

Puerto de Salida		**Puerto de Entrada**	
SinglePacket		SinglePacket	
SinglePacket		MultiPacket	Slot
MultiPacket		MultiPacket	
MultiPacket	Slot	MultiPacket	Slot
MultiPacket	Slot	SinglePacket	

Tabla 4.1: Variedad conexionados de puertos.

Un puerto de salida *SinglePacket* puede ser conectado a un puerto de entrada *SinglePacket* o a un puerto de entrada *MultiPacket* en un *Slot* determinado. Un puerto de salida *MultiPacket* puede conectarse a un puerto de entrada *MultiPacket*, indicando *Slots* independientes a conectar o sin indicarlo especificamente en cuyo caso se conectan todos los *Slots* entre ambos puertos *MultiPacket* (lo que supone que deben ser iguales en número de *Slots* además de compatibles). Además un *Slot* concreto de un puerto *MultiPacket* puede conectarse a un puerto *SinglePacket*.

Además de las comprobaciones en cuanto al interconexionado de componentes, el hecho de pasar a un modelo en el que los componentes *CoolBOT* puedan ubicarse en máquinas remotas, supone la aparición de posibles retardos en los envíos de datos o la imposibilidad de comprobar *in situ* que, un componente que se comunica con otro está efectivamente *online* y ejecutándose normalmente, y no ha quedado bloqueado por algún motivo, como por ejemplo, que haya *caído* la máquina en la que se ejecutaba.

Todas los puntos analizados son consideraciones importantes que han motivado el diseño del modelo de comunicaciones que se presenta como solución para extender el *framework CoolBOT* a un modelo de componentes distribuido. Para tal efecto se ha desarrollado un protocolo de comunicaciones (*DC3P*, del inglés Distributed CoolBOT Components Communication Protocol), descrito en detalle en el capítulo 5.2.

Representación intermedia de datos

La interacción entre componentes software distribuidos debe ser eficiente y transparente para el programador de los mismos (usuario desarrollador de sistemas robóticos). La importancia de estas características se maximiza si nos referimos a procesos que forman parte de sistemas, en los que es necesario acortar los tiempos de respuesta, bien por su uso en entornos críticos con necesidades de tiempo real; bien por una interacción hombre-máquina como es el caso de sistemas teleoperados, o incluso por combinaciones de ambos requisitos. Otra cualidad deseable es la interoperabilidad entre elementos distribuidos, dándole al programador la flexibilidad de elegir entre diferentes máquinas y sistemas operativos donde ejecutar los componentes *CoolBOT* que integren un sistema dado.

El problema surge del hecho de que en el software se utilizan estructuras de datos que para su transmisión por la red que deben convertirse a paquetes de secuencias de bytes. A esto se suma el hecho de que distintas máquinas almacenan los datos de distinta forma, con distintos tamaños para un mismo tipo de datos o con distintos tipos de codificaciones para las cadenas de texto.

Como solución se plantean diferentes representaciones intermedias de datos, así como lenguajes de definición de interfaces que encapsulan el manejo de los mismos. Definiendo una representación de datos externa a la máquina el problema se reduce al uso de operaciones que transformen a dicha representación y desde la misma a los formatos nativos de cada máquina involucrada en la comunicación, lo que se denomina *marshalling* o *serialización* y *demarshalling* o *deserialización*, respectivamente.

Existen varios tipos de representaciones para el empaquetamiento de estructuras de datos para su envío a través de un entorno distribuido en redes de ordenadores, entre los que destacamos por su amplio uso *XDR* (del inglés *External Data Representation*) [RFCs XDR], y *CDR* (del inglés *Common Data Representation*) [OMG,2002]. El proyecto software *middleware ACE: Adaptive Communication Environment*, descrito en detalle en la sección 4.4.3, ofrece un conjunto de opera-

ciones para realizar *marshalling/demarshalling* sobre tipos de datos primitivos y manejar datos en el formato *CDR*.

Creación de Vistas *CoolBOT*

Un usuario operador necesita un interfaz para llevar a acabo sus funciones. En dicha interfaz deben aparecer ciertos datos que un componente o varios emiten como resultados de su ejecución. Dichos datos son dependientes de la funcionalidad propia de cada componente, de forma que no son conocidos a priori ni existe ninguna restricción con respecto a los mismos en el contexto de *CoolBOT*. Por ello la tarea final de dotar a un sistema robótico determinado de una interfaz gráfica de teleoperación queda en manos del desarrollador de sistemas robóticos. Sin embargo, sería de gran utilidad facilitar dicha tarea y que *CoolBOT* ofrezca un modelo de creación de lo que se denominan *vistas CoolBOT* o *vistas* . Una *vista* no es más que una presentación concreta de ciertos datos, que podrán ser enviados desde uno o varios componentes. Puede darse el caso de que una *vista* permita el envío de algún tipo de dato desde la interfaz hacia un componente, ofreciendo así la posibilidad de que un operador controle un sistema robótico.

4.4. Selección de herramientas para el desarrollo

Con el fin de satisfacer los requisitos presentados en la sección 4.3 se han seleccionado una serie de herramientas de desarrollo. El objetivo de las siguientes secciones es argumentar los principios de diseño y funcionalidades que han motivado la elección de las diferentes herramientas software utilizadas.

A continuación se analiza la representación intermedia de datos seleccionada para los datos que se intercambiarán en las comunicaciones (*CDR: Common Data Representation*), para posteriormente presentar varios *middlewares* y librerías de desarrollo de sistemas distribuidos, entre ellos el que ha sido utilizado en este proyecto, *ACE: Adaptive Communication Environment*. Finalmente se describe la librería gráfica con la que se han implementado las *vistas CoolBOT* del sistema robótico semiautónomo teleoperado desarrollado en este proyecto.

4.4.1. CDR: Common Data Representation

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.

Orden de representación

El orden de representación de los datos de más de un byte en memoria (*byte order*) puede ser de dos tipos: *big-endian* o *little-endian*. En máquinas *Big-endian* los datos son insertados en

la memoria colocando el byte más significativo en la posición de memoria más alta, mientras que en máquinas *little-endian* el byte menos significativo es insertado en la posición más alta de la memoria. Las figuras 4.10 y 4.11 muestran un ejemplo de un dato concreto almacenado usando los dos órdenes posibles.

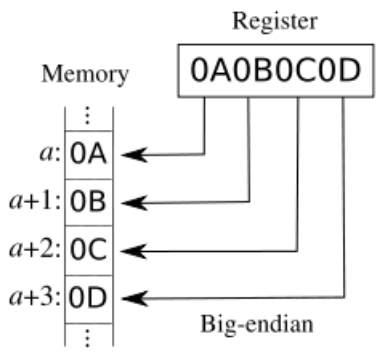

Figura 4.10: Big-endian.

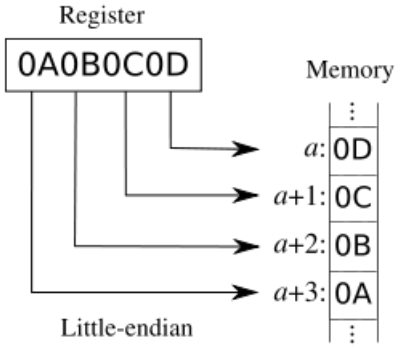

Figura 4.11: Little-endian.

En el caso de *CDR* los datos se insertan en el *stream* de bytes en el orden nativo de la máquina emisora. Esto supone que a la hora de interpretar los datos, enviados a través de la red como una ristra de bytes, se debe tener presente cual es el *byte order* de la máquina emisora, de forma que se sepa como interpretar cada tipo de dato correctamente.

Tamaño de datos primitivos

Los tamaños usados para la representación de tipos primitivos en *CDR* son los mostrados en la tabla 4.2.

Tipo	**Tamaño (bits)**	**Tipo**	**Tamaño (bits)**
Short	16	*Unsigned Short*	16
Long	32	*Unsigned Long*	32
Float	32	*Double*	64
Long Double	128	*Long Long*	64
Unsigned Long Long	64	*Char*	8
Wide Char	16	*Boolean*	8
Octect	8		

Tabla 4.2: Tamaños en bits de tipos de datos primitivos.

En el caso de tipos de datos construidos los tamaños son calculados según los datos primitivos de los que se compongan, añadiendo información necesaria para su representación en el caso de ristras de caracteres, que se representan con un tipo *unsigned long* que indica la longitud en bytes, seguido del conjunto de caracteres en orden (estos caracteres pueden ser del tipo *wide char*).

Alineamiento de datos en memoria

Con el fin de permitir que los distintos tipos de datos se inserten y extraigan en un *stream* de bytes, en *CDR* todos los datos primitivos deben alinearse a sus límites naturales (longitud en bytes del tipo de dato). Es decir, todo dato de longitud n debe empezar en un índice del *stream* de bytes que sea múltiplo de n. En concreto para *CDR*, los valores de n son 1, 2, 4 u 8.

Tipo	Alineamiento	Tipo	Alineamiento
Char	1	Long	4
Octeto	1	Unsigned Long	4
Boolean	1	Long Long	8
Short	2	Unsigned Long Long	8
Unsigned Short	2	Double	8
Float	4	Long Double	8

Tabla 4.3: Alineamientos de tipos 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 4.3 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.

4.4.2. Alternativas de middleware para comunicaciones

El crecimiento y la popularidad de las aplicaciones distribuidas ha provocado que los desarrolladores demanden mecanismos para programar de forma fácil, eficiente y portable, aplicaciones de este tipo. Existen una serie de tareas comunes en cualquier aplicación distribuida como pueden ser:

- Gestión de las conexiones de red, direcciones e identificación de servicios.
- Comunicación entre procesos residentes en diferentes plataformas, sistemas operativos y/o tipos de redes.
- Manejo de diferentes mecanismos de intercomunicación de procesos: *sockets, RPC (Remote Procedure Call)*, etc.

A la hora de desarrollar una aplicación distribuida se debe tomar una decisión crucial a la hora de programar: utilizar mecanismos de bajo nivel o utilizar herramientas que posibiliten mayor nivel de abstracción como un *middleware*. Por supuesto la programación de aplicaciones distribuidas se facilita enormemente al usar herramientas de alto nivel, sin embargo restan la flexibilidad y en ocasiones se pierde la eficiencia que permiten los mecanismos a un nivel inferior como pueden

ser los *sockets*. Los *sockets* no son más que un nivel de abstracción bajo el que subyacen las capas de transporte de red como *TCP* o *UDP*, de forma que se logra abstraer al programador de las operaciones en las capas de red inferiores, pero sin perder flexibilidad en la programación. Sin embargo, los mecanismos en este nivel de abstracción tienen limitaciones en la portabilidad. Códigos de error, estrategias en la inicialización o los tipos de datos con los que se representan estos mecanismos difieren entre sistemas operativos, con lo que se añade la imposibilidad de chequear ciertos tipos u operaciones validas en tiempo de compilación (un ejemplo es la representación de un *socket* como un descriptor de fichero en *UNIX* o como un puntero en *Win32*). Debido a estas limitaciones en las *APIs* existentes para programar con mecanismos de intercomunicación de procesos (*IPC*) como *sockets* surgen herramientas orientadas a objetos. Estas librerias encapsulan en clases mecanismos de intercomunicación de procesos de bajo nivel. Mediante dicho encapsulado se logra:

- Abstraer al programador de la heterogeneidad en las implementaciones de mecanismos de *IPC*.
- facilitar la programación.
- Reusabilidad de código.
- Robustez de las aplicaciones y facilidad de depuración.

A continuación se comentan algunos ejemplos de *middleware* para el desarrollo de aplicaciones distribuidas en C++ y sus características distintivas.

Net.h++

Net.h++ [NET++] ha sido desarrollado utilizando la librería de propóstico general *Tools.h++*. Esta librería ofrece independencia de diferentes sistemas operativos y protocolos de red.

Arquitectura

La arquitectura de *Net.h++* está dividida en cuatro capas mostradas en la figura 4.12, donde se indican en gris los servicios que aún no están implementados en la versión 1.0 actual.

La capa inferior proporciona los servicios básicos de *Tools.h++*. La siguiente es la capa de adaptadores de comunicación. Esta capa encapsula los mecanismos básicos de comunicación entre procesos, concretamente de *sockets*, direcciones *TCP/IP*, *TCL: Tool Command Language*, *pipes*, y mecanismos de *tunneling*. Sobre estas funciones aparece la capa portal, que proporciona un nivel superior de abstracción en la *API*, unificando la interfaz para todos los mecanismos de comunicación existentes. Además añade métodos de envío de datos no proporcionados en la capa inferior. La capa de más alto nivel es la capa de servicios de comunicación. A este nivel los programadores manejan los diferentes mecanismos de comunicación como canales de entrada-salida de datos y disponen de operaciones que para convertir datos complejos en una serie de bytes y viceversa.

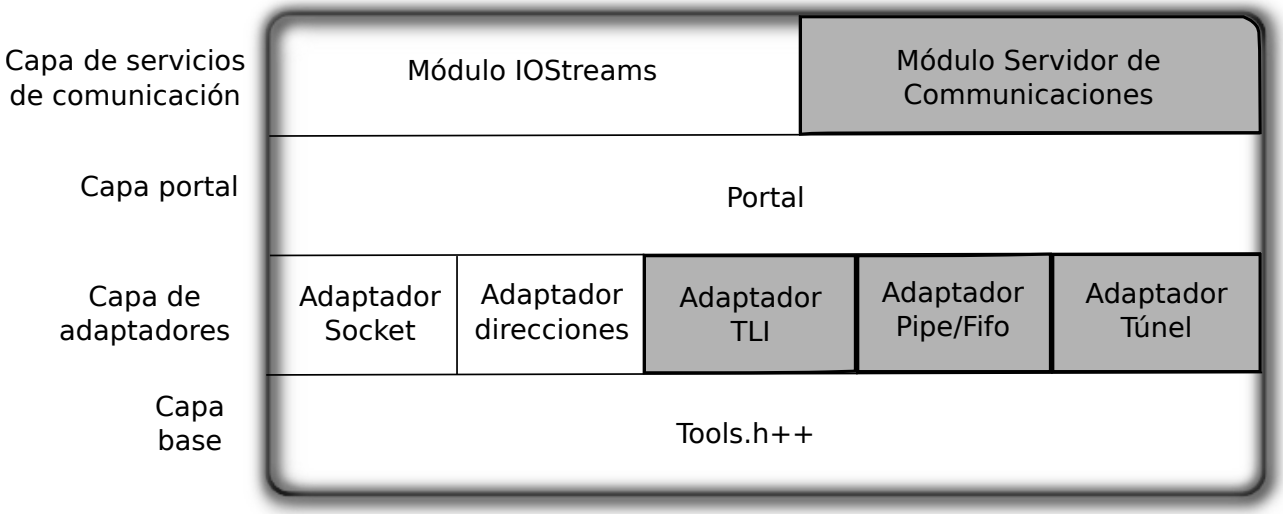

Figura 4.12: Arquitectura en capas de *Net.h++*.

Desventajas

Existen ciertas características de este *middleware* que han declinado su uso para la implementación del nivel de red que nos ocupa en este desarrollo. La primera es que se trata de un software privativo y con una licencia de pago, además de la dependencia de la librería Tools.h++ también de pago. En cuanto a características técnicas, aún no presenta un desarrollo muy maduro y, aunque planificados para ser añadidos en un futuro, carece de muchos mecanismos de comunicación en su versión actual.

Socket++

La librería *Socket++* [**?**] proporciona un conjunto de clases que pueden utilizarse de forma más segura y efectiva que haciendo llamadas directas al sistema. Estas clases presentan una interfaz similar a la librería *IOStream* de C++, con lo que puede hacerse uso de los operadores << y >> para la entrada y salida de datos por la red, lo que simplifica el aprendizaje de usuarios ya familiarizados con esta interfaz.

Jerarquía de clases

En la figura 4.13 se ilustra la jerarquía de las clases básicas de la librería *Socket++*. La clase *sockbuf* es una derivada de la clase *streambuf* de la librería *IOStream* de C++. Esta clase es utilizada para almacenar la entrada-salida de un *socket*, es decir la lectura y escritura de datos en un *socket* tiene lugar en esta clase. Las clases *sockunixbuf* y *sockinetbuf* son especializaciones de *sockbuf* para las comunicaciones *Unix* o *Inet*. La clase *iosockstream* ha sido derivada de *iostream*, con lo

que proporciona el manejo de datos en un *socket* como un *stream* de bytes. Las clases *isockstream* y *osockstream* proporcionan acceso a los datos de sólo lectura o sólo escritura respectivamente. Las clases derivadas de la clase abstracta *sockAddr* ofrecen encapsulados de las diferentes familias de direcciones de red. Como añadido existen clases que ofrecen una implementación de diferentes protocolos del nivel de aplicación como *Echo*, *FTP* y *SMTP*. *Socket++* es un proyecto de software libre y se actualmente se encuentra en su versión 1.11.

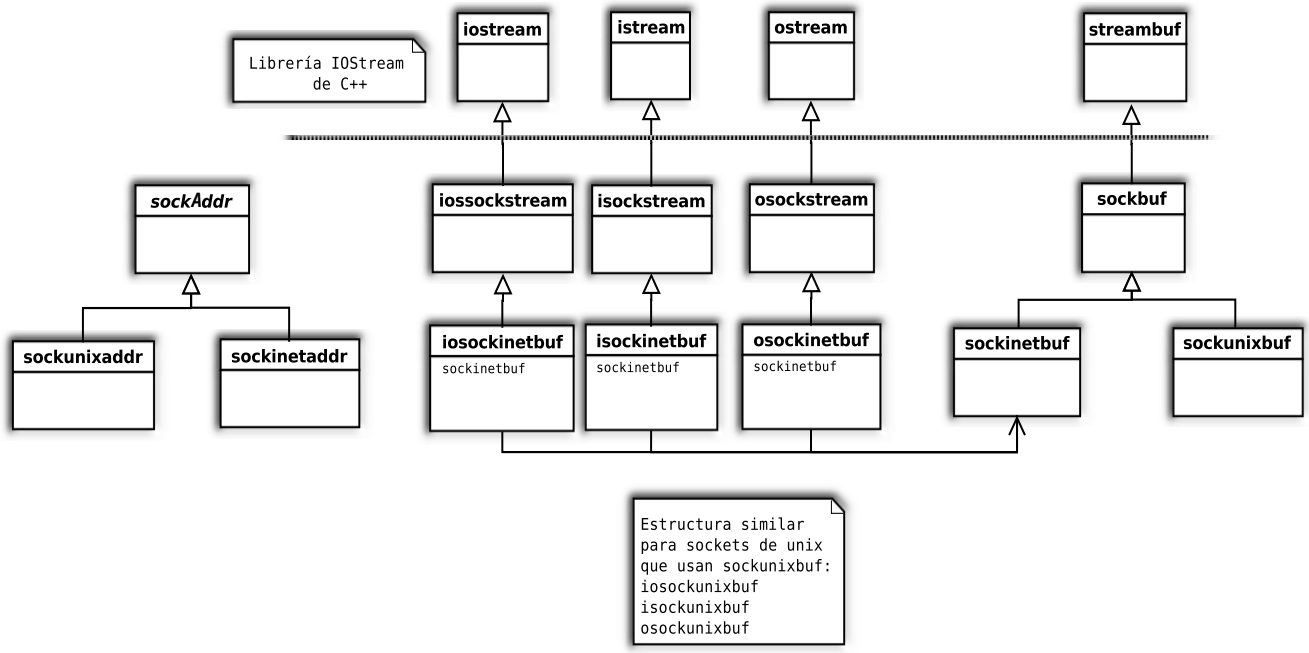

Figura 4.13: Jerarquía de clases de *Socket++*.

Desventajas

Como desventaja principal al uso de esta librería se presenta el hecho de que carece de ciertas estructuras de comunicación entre procesos y no ofrece mecanismos de sincronización. Aún es un proyecto en sus primeras versiones y sólo ofrece encapsulados de las clases más básicas para comunicaciones además del hecho de que su portabilidad está limitada a sistemas *Unix* sin que aún existan encapsulados de sockets de *win32*.

4.4.3. ACE: Adaptive Communication Environment

El *framework ACE* es un ejemplo ampliamente utilizado de *middleware* orientado a objetos. *ACE* está disponible como software de código abierto y gratuitamente, sus distintas versiones pueden ser descargadas desde su web oficial *http://www.cs.wustl.edu/ schmidt/ACE.html*.

Las características que ofrece *ACE* están básicamente relacionadas con tres aspectos principales en relación a la programación distribuida: la portabilidad del software que se desarrolle, la facilidad de programación y la reusabilidad de código.

ACE está soportado para un gran número de sistemas operativos como por ejemplo: Windows(32/64 bits), Redhat, Debian, Suse Linux y Macintosh OS X, además de las mayoría de versiones de UNIX y sistemas de tiempo real; asimismo ofrece componentes comunes en el desarrollo de aplicaciones distribuidas.

Principales características de ACE

ACE está desarrollado sobre una arquitectura en capas (Figura 4.14). La base es una capa de adaptación al Sistema Operativo (*SO*): *adaptation layer* y una capa denominada *C++ wrapper facades*. Estas capas encapsulan el núcleo del *SO* y los mecanismos de programación concurrente. Las capas superiores proporcinan el *framework* de componentes reusables, componentes de servicios de red y un *middleware* de *CORBA* para la invocación de operaciones sobre objetos distribuidos. En definitiva pone a nuestra disposición muchas de las herramientas y patrones utilizados para el desarrollo de software de comunicaciones.

Como podemos ver en la figura 4.14, las capas de la arquitectura de *ACE* permiten separar las áreas y abstraerlas entre sí, reduciendo la complejidad, desde las capas más bajas, cercanas al *SO*, hasta las superiores de mayor nivel de abstracción.

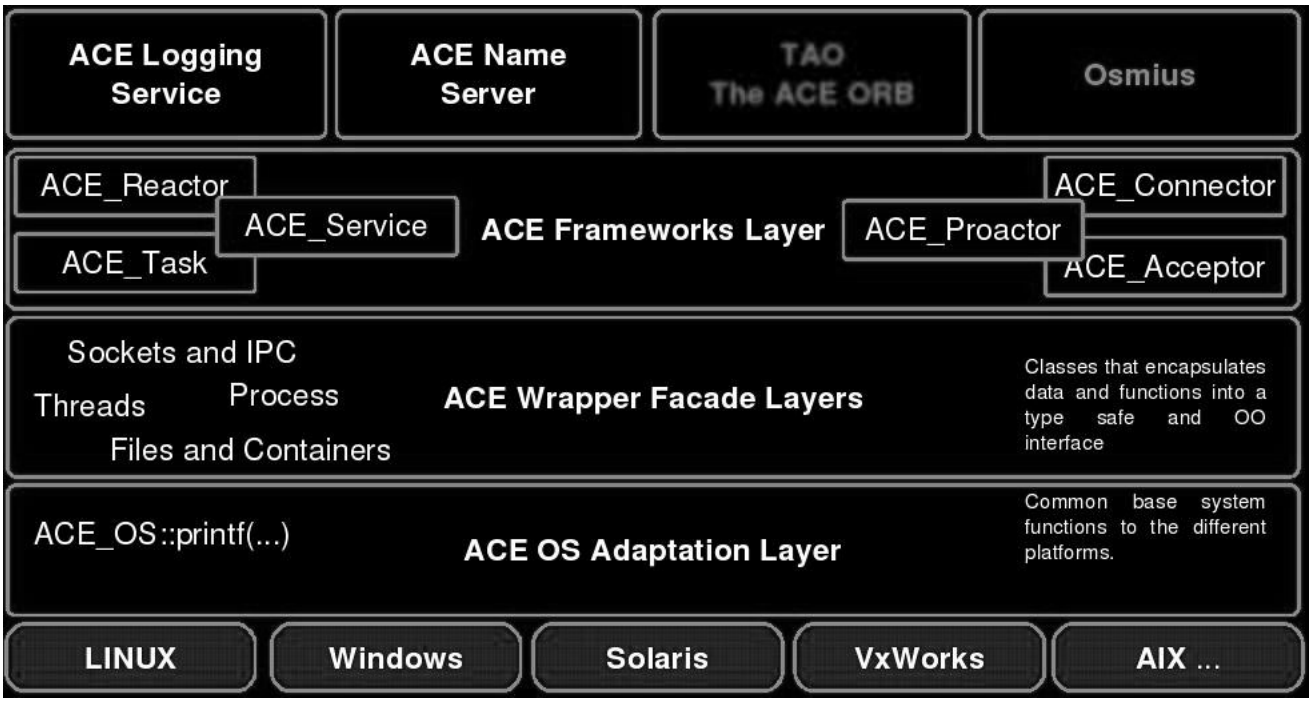

Figura 4.14: Arquitectura de capas de ACE

- **Capa de adaptación al SO (*Adaptation Layer*).**

  Esta capa proporciona funciones para realizar las llamadas al sistema. Se trata en concreto de una clase, *ACE_OS*, que encapsula todas las llamadas al sistema proporcionando una interfaz común e independiente de la plataforma. Esta clase es de gran importancia, ya que supone el enlace entre las distintas plataformas que soportan ACE y las capas superiores del *framework*. Es, por tanto, esta capa la que proporciona el alto grado de portabilidad de *ACE*, haciendo transparente los detalles del sistema sobre el que se ejecutará la aplicación desarrollada. Añadir soporte para *ACE* para nuevos sistemas operativos se basa, exclusivamente, en modificaciones sobre la capa de adaptación al *SO*.

- **Capa de envoltorios de C++ (*C++ Wrapper Facades*).**

  Esta capa simplemente añade un nivel de abstracción para lograr simplificar el desarrollo de aplicaciones. Utiliza clases C++ para proporcionar una interfaz a la capa de abstracción del SO, es decir, no se añaden nuevas funcionalidades sobre la capa anterior, simplemente ofrece un empaquetado que facilita el uso al programador.

  Los criterios base en el diseño de esta capa fueron, principalmente la mejora de la seguridad de los tipos de datos, uso de jerarquías y ocultar detalles de cada plataforma, todo esto en busca de transparencia y simplicidad para el programador además de eficiencia. Es a partir de esta capa donde comienzan a hacerse visibles los beneficios que aporta *ACE* frente proyectos para programación distribuida desarrollados en C. Mientras que para una *API* de C serían necesarias varias funciones para ciertas tareas comunes como escuchar en un puerto *TCP*, *ACE* proporciona clases que realizan esta acción con un único método. Esto favorece la reutilización de código y la eliminación de errores, por ejemplo, por olvido de iniciar una estructura antes de alguna de las múltiples llamadas que serían necesarias en C, siendo detectados en tiempo de compilación y no durante la ejecución. Evidentemente todo ello viene dado como consecuencia de utilizar C++, un lenguaje orientado a Objetos, como lenguaje de implementacion del propio framework, así como de las aplicaciones que utilizándolo se pueden desarrollar.

  El conjunto de clases de la capa de envoltorios de C++ se encuentra dividida en distintas áreas:

  - Concurrencia y sincronización multihilo.
    Clases para el manejo de programación multihilo: semáforos, variables condición, etc.
  - Comunicación entre procesos: mecanismos *IPC*.
    Clases que encapsulan el paso de mensajes, manejo de sockets, *named pipes*, etc.
  - Timers.
    Clases para programar temporizadores y alarmas.
  - Contenedores.
    Clases que implementan contenedores como mapas hash, listas, etc.
  - Manejo de señales.
    Clases que encapsulan los diferentes tipos de señales del sistema operativo.

- Sistema de ficheros.

  Clases para el manejo de la entrada/salida.

- Hilos.

  Clases que encapsulan el manejo de hilos independientemente del sistema operativo.

- Manejo de memoria.

  Clases para el manejo de memoria de forma dinámica, memoria que es manejada localmente en *ACE*. La gestión de la memoria es un punto muy importante para el sistema que se pretende desarrollar en este proyecto. En qué sentido permite *ACE* manejar la memoria dinámica se encuentra descrito con mayor detalle en los próximos párrafos.

- **Capa de *Frameworks*.**

  Se trata de la capa de más alto nivel proporcionada en *ACE*. En esta capa se aportan componentes basados en patrones de diseño de software distribuido. Estos componentes facilitan las labores de desarrollo de aplicaciones distribuidas gracias a la combinación de patrones de diseño ampliamente utilizados en este tipo de aplicaciones, con las posibilidades del lenguaje C++ como pueden ser la herencia o la sobrecarga de métodos y operadores. Estas características permiten la escalabilidad del software desarrollado utilizando ACE, posibilitando añadir, modificar y actualizar funcionalidades sin necesidad de recompilar el mismo.

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 sólo 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. Téngase en cuenta que se trata de evitar un manejo de la memoria dinámica por parte del sistema operativo, puesto que ello conllevaría un cambio de contexto entre espacio de usuario y espacio de kernel, en *ACE* se intenta evitar esto mediante la implementación de gestores de bloques de memoria para la utilización de memoría de dinámica en espacio de usuario.

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* o sobrecarga en el 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,2003], 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 segmentados en varias secciones, como cabecera, cuerpo, etc.
- 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 4.15.
- **Compuesto**: contiene múltiples *ACE_Message_Block* enlazados, figura 4.16. Esto proporciona una estructura que permite agregar elementos de forma recursiva.

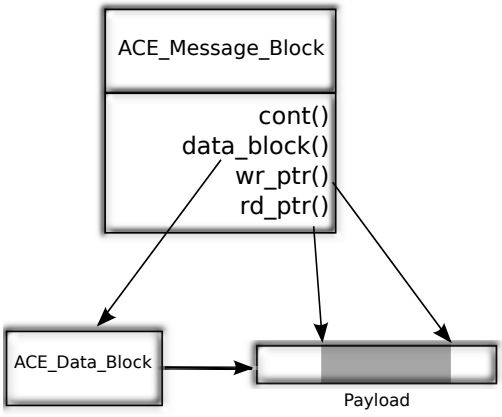

Figura 4.15: Mensaje simple.

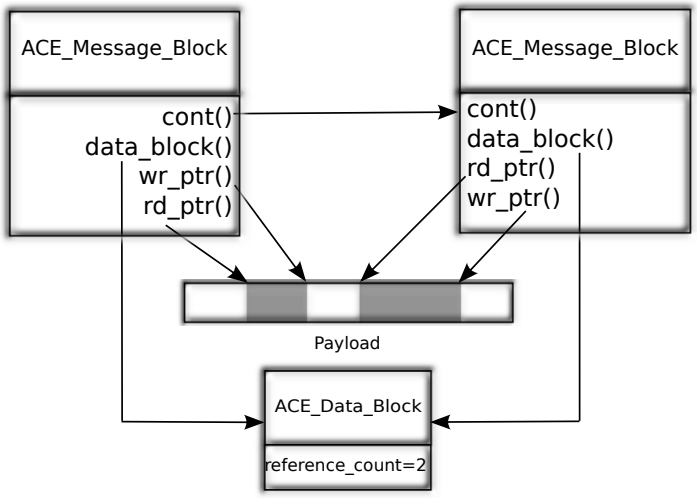

Figura 4.16: Mensaje compuesto.

CDR Streams en *ACE*

Las aplicaciones de red suelen requerir serializado (*marshalling*) 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* ya introducida en la sección 4.4.1, 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* de 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).

Ventajas de *ACE*

Las características descritas con anterioridad hacen de ACE un *framework* genérico de programación distribuida 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 el desarrollo de este proyecto. *ACE* está soportado por una variedad creciente 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.

Una razón importante de la elección de este software es que está desarrollado en C++, con lo que es fácilmente integrable en *CoolBOT* que también está desarrollado en este lenguaje.

4.4.4. Librería gráfica GTK+

GTK+, o *The GIMP Toolkit*, es un conjunto de librerías escritas en C para la creación de interfaces gráficas de usuario (*GUI: Graphical User Interface*). *GTK+* es un proyecto se software libre. A pesar de estar escrita en C, en *GTK+* existen una serie de encapsulador para dar soporte clases y punteros a funciones.

Las librerías que componen *GTK+* son:

- GLib

  Es la librería de más bajo nivel, básica en *GTK+*. Proporciona el nivel de portabilidad, ofreciendo gestión de hilos, errores y una base de objetos para el desarrollo.

- GTK

  Contiene los objetos dedicados a la construcción de interfaces: ventanas, botones, menús, etc.

- GDK

  Librería intermedia entre gráficos de bajo y alto nivel de abstracción.

- ATK

  Librería que ofrece una serie de objetos para la construcción de interfaces accesibles para personas con deficiencias visuales u otras discapacidades.

- Pango

  Librería para el manejo de cadenas de texto.

- Cairo

  Librería para los controles de la aplicación.

Tipos básicos de datos

GTK+ define su propio conjunto de datos básicos, proporcionados por la librería GLib. A excepción de los punteros, se llaman igual que los tipos básicos de C pero con una g delante:

Tipos básicos	
Tipo GLib	**Tipo C**
gchar	char
gshort	short
glong	long
gint	int
gboolean	boolean
gpointer	void*

Tabla 4.4: Tipos básicos de GTK+

Widgets y herencia

Un concepto fundamental en *GTK+* son los *widgets*. Se trata de una porción de la interfaz gráfica con la que el usuario puede interactuar, es decir, son cada uno de los elementos que conforman la interfaz. Un *widget* posee propiedades y puede ser programado para reaccionar ante eventos durante la ejecución de un programa. A partir de la combinación de *widgets* se realiza la construcción de interfaces.

Figura 4.17: Widgets en GTK+

En *GTK+* es posible derivar de cada widget para crear otros nuevos. La clase básica para tal fin es *GtkObject* derivada de *GObject*. Sin embargo en el uso habitual se utiliza como base la clase *GtkWidget* que hereda de *GtkObject*.

Cualquier función de creación de elementos para una interfaz retorna un *widget*.

Funciones principales

En un ambiente gráfico la iteracción con el usuario no se realiza mediante acciones secuenciales, es decir el usuario dispone de cierta libertad para manejar la interfaz y sus elementos. GTK+ proporciona funciones controlar las acciones sobre la interfaz. La función *gtk_init ()* debe ser llamada antes que cualquier otra función de *GTK+*, esto es porque es la encargada de inicializar todo lo necesario para trabajar con la librería gráfica. Otra función importante de control es *gtk_main ()*, que ejecuta un ciclo infinito y captura todos los posibles eventos que se den en la interfaz. Cuando tenga lugar un evento esta función emite una señal.

Control de eventos

Una señal proporciona la posibilidad de asociar funciones que se ejecutarán cuando se emita dicha señal. A estas funciones se las denomina *callbacks*. De esta forma cuando tenga lugar un evento *gtk_main ()* emite una determinada señal en función del mismo, en ese momento se llama al *callback* asociado con dicha señal. Por tanto para controlar los eventos que se producen en una interfaz bastará con definir los *callbacks* necesarios para la señal emitida ante determinado evento.

Contenedores

Los contenedores son una clase de *widgets* con la característica especial de que son capaces de alojar otros *widgets* en su interior. El objetivo es facilitar que los elementos de una interfaz se ajusten a diferentes tamaños y tipografías. Los *widgets* contenedores ocupan todo el espacio disponible que encuentren, de este forma si disponemos de una ventana y un *widget* contenedor en ella, si se cambia el tamaño de la ventana el *widget* que se encuentre en su contenedor también adaptará su tamaño.

El uso de contenedores es una característica distintiva de la programación de *GUIs* con *GTK+*, similar a la creación de interfaces con *Java*.

En este proyecto se ha utilizado *GTK+* para el desarrollo de *vistas CoolBOT* y la creación de una interfaz de teleoperación a partir de diferentes *vistas*.

La razón principal de la elección de esta librería gráfica es que es software libre licenciado como *LGPL*, además de estar bien soportada tanto en GNU/Linux y múltiples variedades de UNIX, así como en Windows. [GTK+ Project]

Capítulo 5

Diseño

Este capítulo se centra en exponer las actividades y resultados de la fase de diseño de este proyecto. Inicialemente se describe el diseño de las operaciones de *marshalling/demarshalling* 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 para componentes distribuidos. En una siguiente sección se presenta el modelo de componentes *CoolBOT* distribuidos y a continuación se describe el diseño genérico de *Vistas CoolBOT*, así como de sondas de componentes. Para finalizar el capítulo se ilustra la estructura de la interfaz de teleoperación desarrollada haciendo uso de *Vistas GTK*.

Finalmente este capítulo describe la estructura interna del nuevo modelo de componente distribuido de *CoolBOT* y la estructura genérica de una *vista* de componente.

5.1. 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 diseñado y construido por el usuario). 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) de manera que tales aspectos sean completamente transparentes al desarrollador de componentes o integrador de sistemas *CoolBOT*.

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* o plantilla. 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 esta interfaz se describe en detalle en el apéndice C.1.

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 C.3.5). 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*).

5.2. Especificación del protocolo *DC3P*

El diseño de protocolos en sí mismo puede ser un gran desafío. Existen protocolos amplios y sofisticados que intentan ofrecer una mayor funcionalidad y fiabilidad, pero como resultado, se ven incrementados en tamaño y complejidad. El problema fundamental reside en crear un conjunto de reglas consistentes y completas que a su vez minimice los tamaños de los mensajes y por ende, la complejidad de las comunicaciones.

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*, y sobre el que se implementará el modelo de interconexionado de componentes mediante conexiones de puertos de entrada y salida existente en el *framework*. 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 línea 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].

5.2.1. 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 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 direcciones *IP*, o nombres *DNS*, y puertos *TCP* de los componentes que pretende interconectar, actuando la capa del servicio *DC3P* integrada en *CoolBOT* como una caja negra, figura 5.1

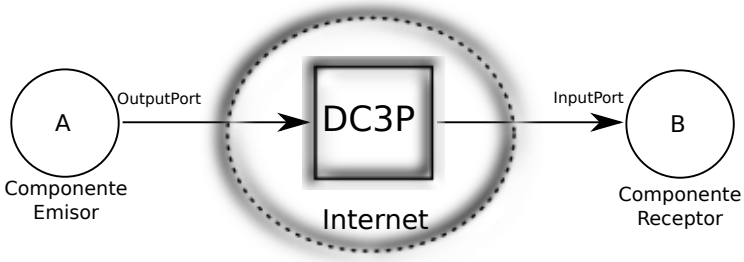

Figura 5.1: Visión a alto nivel del servicio DC3P

Con mayor concreción, las funcionalidades proporcionadas a los componentes *CoolBOT* por el protocolo 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.

- Multiplexado de conexiones de puerto entre componentes sobre conexiones *TCP*.

  Se ofrece la información sobre puertos e identificación de los extremos (componentes) en la conexión de forma que sea factible multiplexar un conjunto de conexiones de puertos sobre una conexión *TCP*.

5.2.2. 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 *TCP*. Se crearán conexiones *TCP*, concretamente se establece una conexión *TCP/IP* entre dos componentes que residan en distintos procesos o máquinas, y sobre ella se multiplexan todas las conexiones entre puertos de entrada y salida con el mismo sentido que se establezcan entre dichos componentes. Por tanto, cada conexión *TCP* creada se utilizará para el intercambio de datos entre una serie de pares de puertos *CoolBOT* de componente (*InputPort-OutputPort*). La figura 5.2 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.

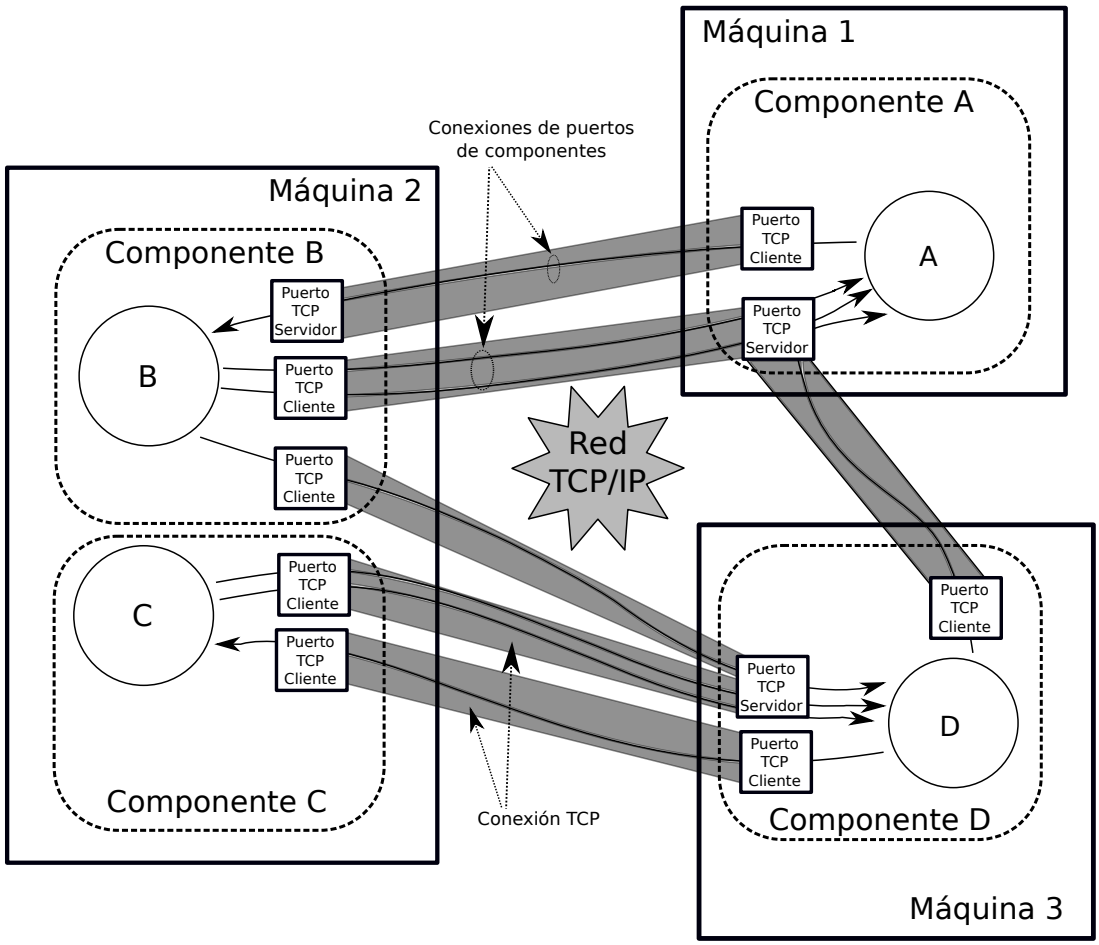

Figura 5.2: Multiplexación de conexiones de puertos entre componentes distribuidos sobre conexiones TCP

El componente que inicia la conexión con otro, es decir el que juega un papel activo en el inicio de la comunicación, es 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 por el *emisor* y usados por el componente *receptor*. Como se comentó en la sección 5.2.1, a pesar de que el flujo de datos entre componentes *CoolBOT* es de un solo sentido, de puerto de salida a puerto de entrada, ciertas reglas de procedimiento del protocolo (descritas en detalle en la sección 5.2.5), contemplan el intercambio de mensajes de control del protocolo en ambos sentidos, para así proporcionar correctamente todas las funcionalidades descritas para el servicio.

5.2.3. 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 5.2.1.

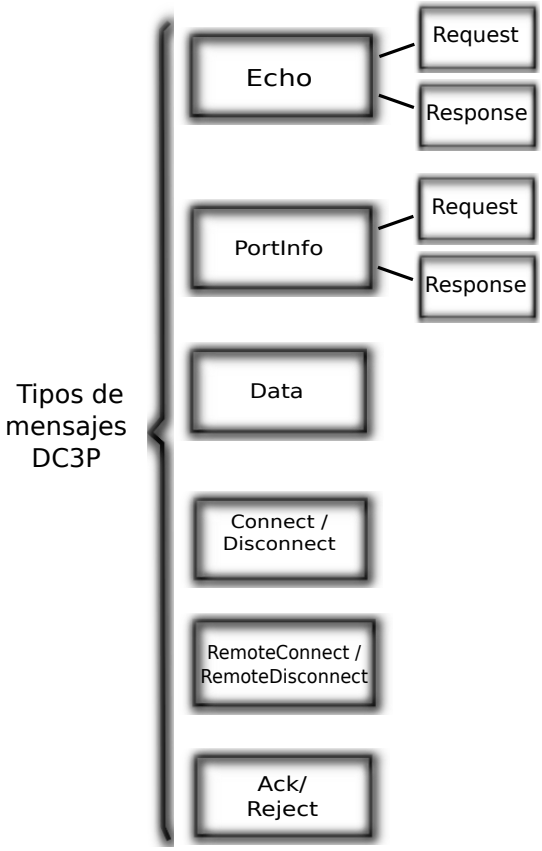

Figura 5.3: Tipos de mensajes *DC3P*

El vocabulario del protocolo, como se muestra gráficamente en la figura 5.3, 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.

5.2.4. Formato de los mensajes

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.

La cabecera, mostrada en la figura 5.4, 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. El *ByteOrder* toma valores 1 ó 0, indicando *little-endian* o *big-endian* respectivamente. El segundo byte de la cabecera indica el tipo de mensaje del protocolo que viene a continuación de la cabecera. Este campo permite identificar cómo interpretar adecuadamente el resto del mensaje.

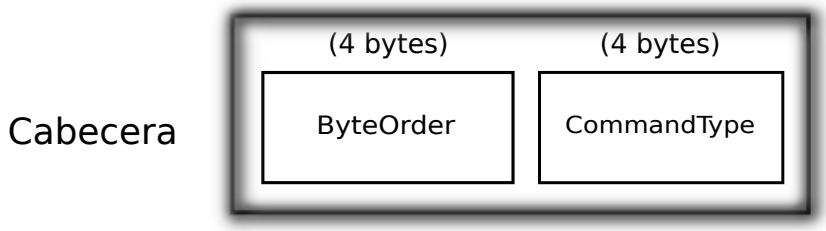

Figura 5.4: Formato de la cabecera *DC3P*

Los valores que puede tomar el campo *CommandType* para los distintos tipos de mensajes del protocolo son los indicados en la tabla 5.1.

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

Tabla 5.1: Valores del campo CommandType.

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 mensaje indicados en la tablas 5.1.

Connect/Disconnect

Como se indica en los requisitos de comunicación de componentes 4.3.2, en *CoolBOT* la tipología de conexiones de puertos que pueden establecerse se pueden clasificar en dos grupos a tenor de los tipos de paquetes de puertos que éstas puedan transportar: aquellos con soporte para un único tipo de paquetes de puerto (*SinglePacket*) y aquellos que soportan varios tipos de paquetes de puerto (*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 tipo de paquete de puerto. En caso de tratarse de una conexión que involucre a algún puerto *SinglePacket* bastará con indicar con valores enteros los identificadores de puertos de salida y entrada de los componetes que se pretenden conectar o desconectar. En caso de conexiones/desconexiones que involucren a un puerto *MultiPacket*, además de estos identificadores de puertos de salida y entrada, habrá que aportar un identificador que concreta el *Slot* para ese puerto determinado. mismos. Antes de establecer una conexión es preciso comprobar la compatibilidad entre los tipos de puertos a conectar, mediante el trasiego de mensajes del protocolo PortInfo que se detallará más adelante.

Como se detalla a continuación, no existen diferencias en los formatos de *Connect* con respecto al *Disconnect*. Las figuras 5.5, 5.6, 5.7, 5.8 ilustran los diferentes formatos de mensajes connect y disconnect.

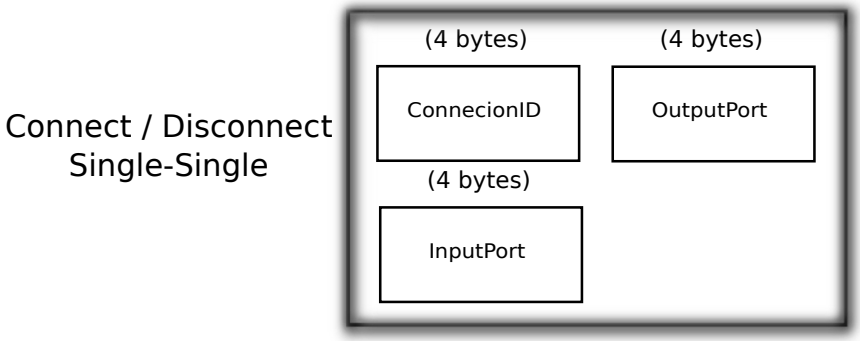

Figura 5.5: Formato de mensajes *Connect/Disconnect Single-Single*.

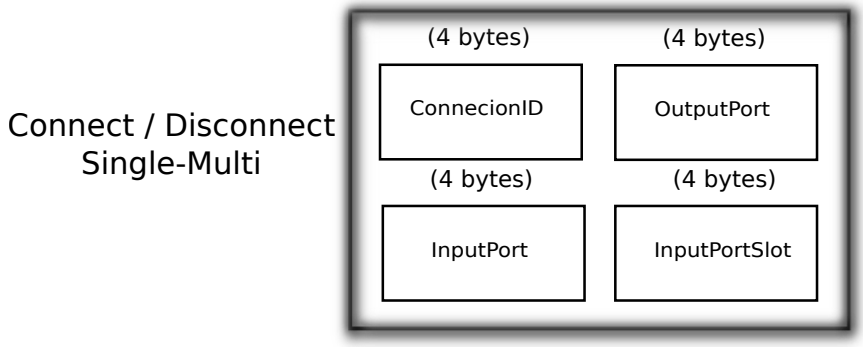

Figura 5.6: Formato de mensajes *Connect/Disconnect Single-Multi*.

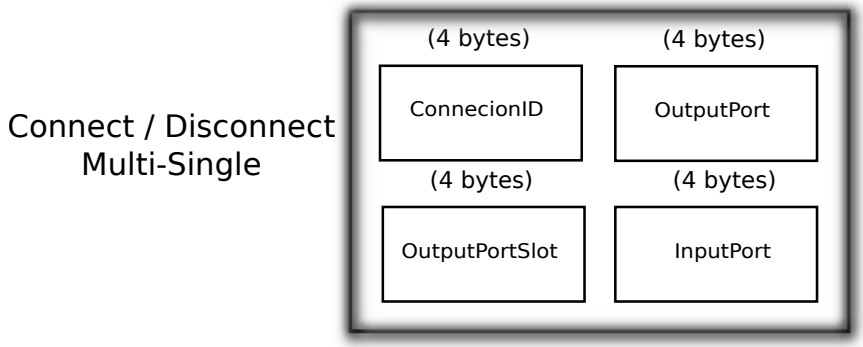

Figura 5.7: Formato de mensajes *Connect/Disconnect Multi-Single*.

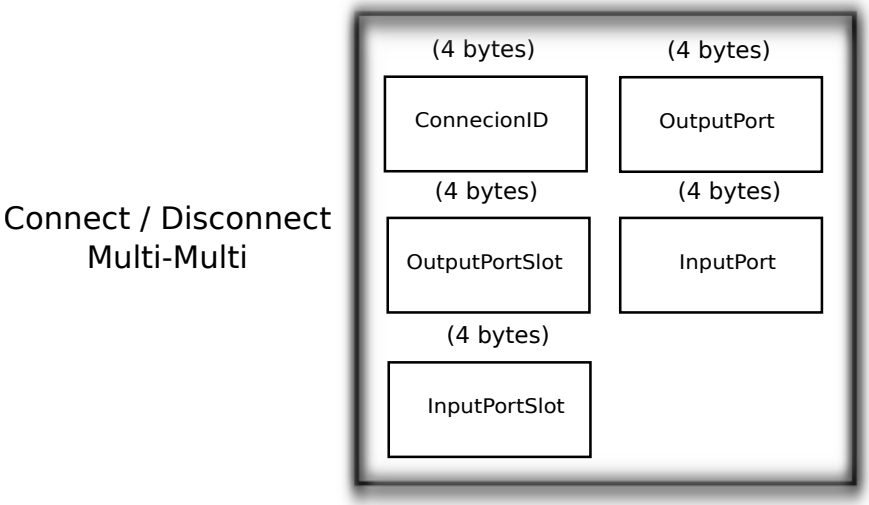

Figura 5.8: Formato de mensajes *Connect/Disconnect Multi-Multi*.

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*.

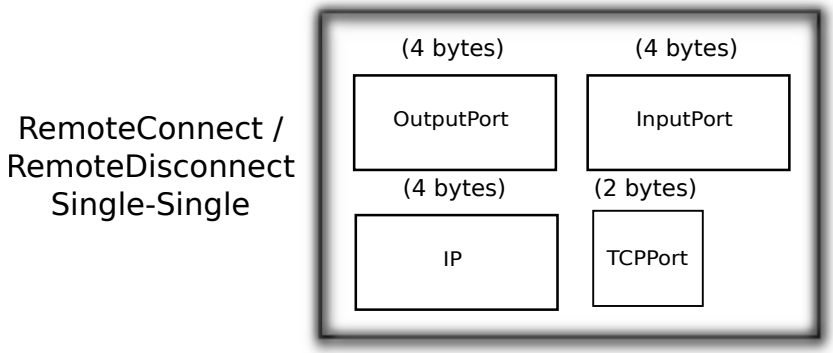

Figura 5.9: Formato de mensajes *RemoteConnect/RemoteDisconnect Single-Single*.

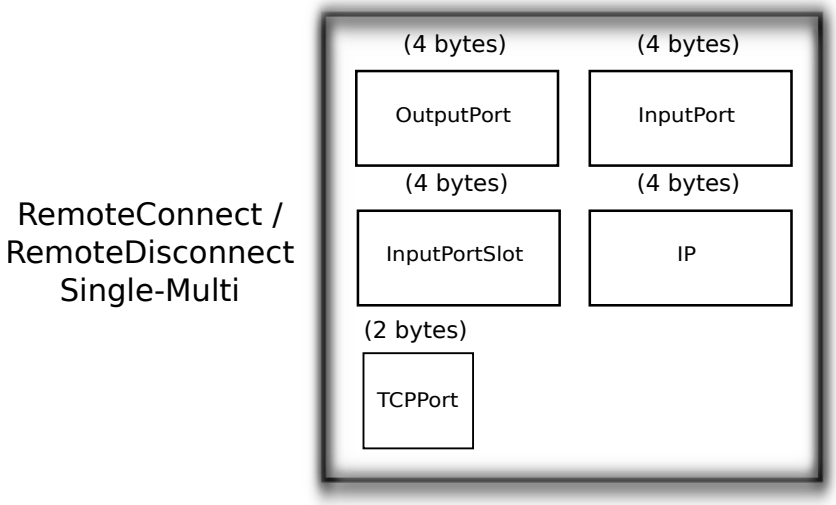

Figura 5.10: Formato de mensajes *RemoteConnect/RemoteDisconnect Single-Multi*.

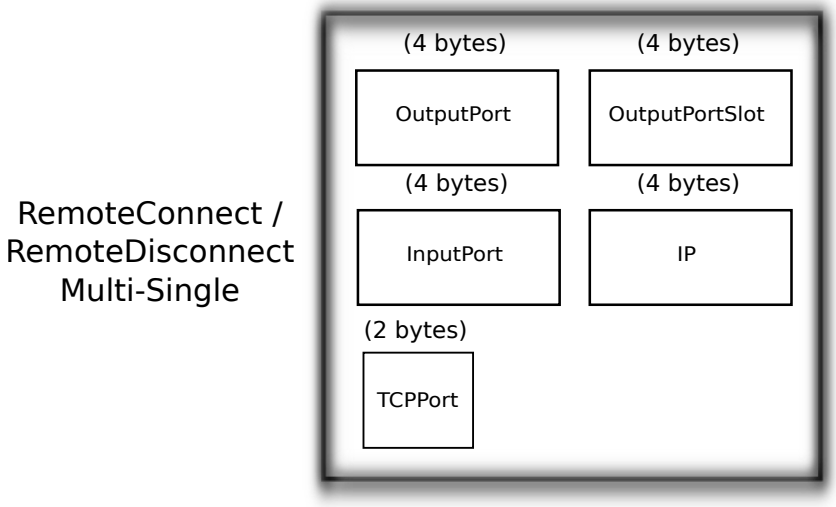

Figura 5.11: Formato de mensajes *RemoteConnect/RemoteDisconnect Multi-Single*.

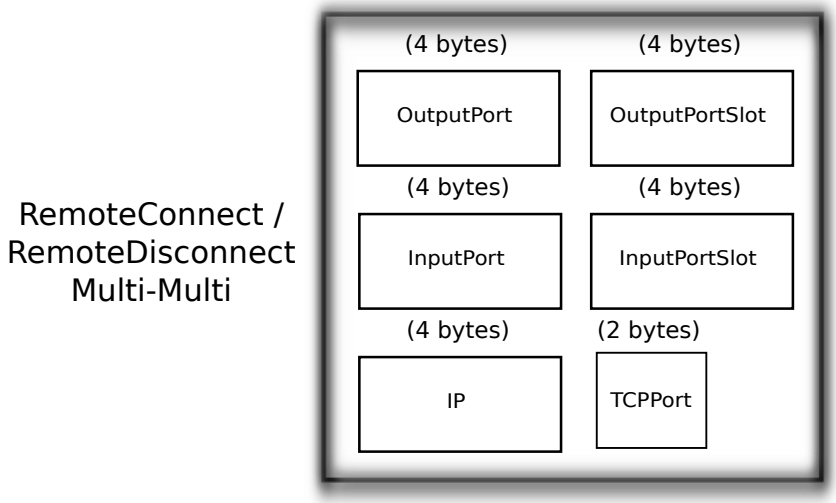

Figura 5.12: Formato de mensajes *RemoteConnect/RemoteDisconnect Multi-Multi*.

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 difrencia radica, de nuevo, en que para los tipos de puertos de entrada involucrados en conexiones Multipacket además del identificador de puerto de entrada es preciso indicar identificador de tipo de paquete de puerto mediante su *Slot*.

SinglePacket

En el caso de puertos *SinglePacket* los formatos de mensajes son los indicados en la figura 5.13, para el caso de un *Request* y en la figura 5.14, 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.

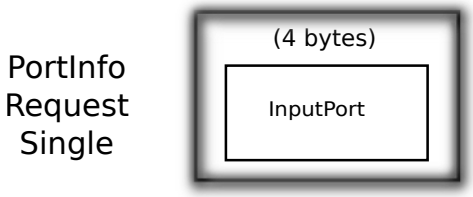

Figura 5.13: Formato de mensajes *SinglePacket PortInfo Request*

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*.

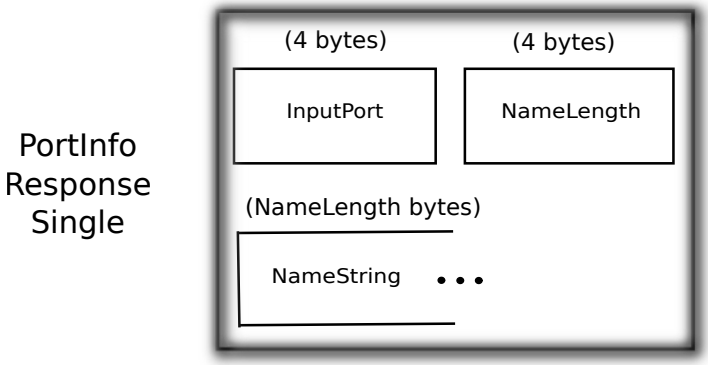

Figura 5.14: Formato de mensajes *SinglePacket PortInfo Response*

MultiPacket

Para puertos *Multipacket* los formatos de mensajes son los indicados en la figura 5.15, para el caso de un *Request* y en la figura 5.16, 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*.

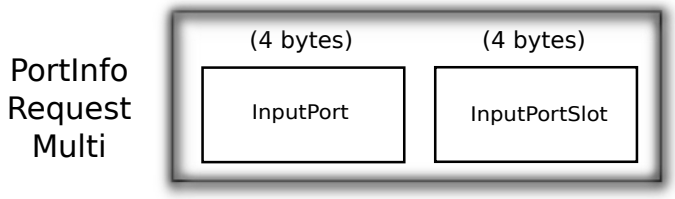

Figura 5.15: Formato de mensajes *PortInfo Request Multi*

La respuesta a 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.

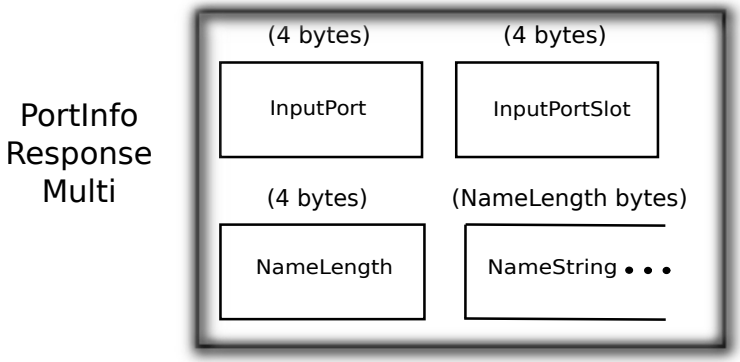

Figura 5.16: Formato de mensajes *PortInfo Response Multi*

Echo

Se trata de un tipo de mensaje destinado a verificar si un componente se encuentra en línea y activo. Existen dos variantes: *Request* y *Response*. La particularidad de este tipo de mensaje es que consiste exclusivamente en un envío de la cabecera (figura 5.4), sin ser añadido ningún campo adicional. La figura 5.17 detalla el formato de este tipo de mensaje. El uso de estos mensajes de *Echo* de manera periódica permite la determinación de si un componente remoto no responde, o se encuentra inaccesible durante la ejecución del sistema, de manera que se pueden detectar fallos bien en un componente remoto, o en la red a través de la que se realiza la comunicación.

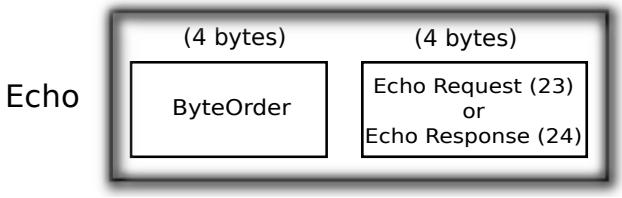

Figura 5.17: Formato de mensajes de *Echo*

Data

Este formato de mensaje está destinado al envío de datos mediante el envío de paquetes de puertos que se intercambian entre componentes *CoolBOT* que integren un sistema dado. De nuevo, en función del tipo de puerto de entrada del componente remoto, existe un formato adecuado a cada tipo.

SinglePacket

El formato para el envío de datos en el caso de puertos de entrada *SinglePacket* se refleja en la figura 5.18.

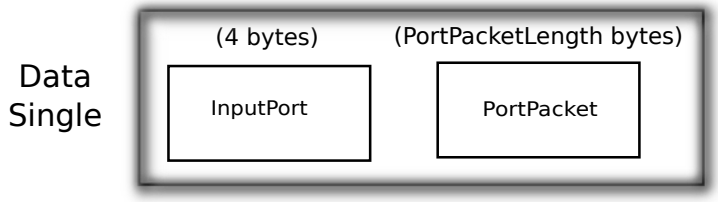

Figura 5.18: Formato de mensajes de *Data Single*

MultiPacket En caso de envío de datos hacia puertos de entrada de tipo *MultiPacket* el formato es el representado en la figura 5.20.

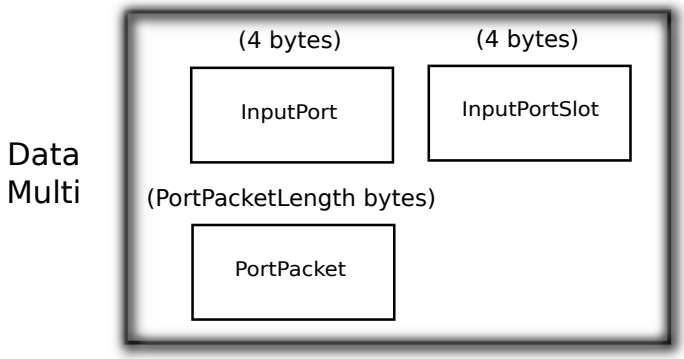

Figura 5.19: Formato de mensajes de *Data Multi*

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.

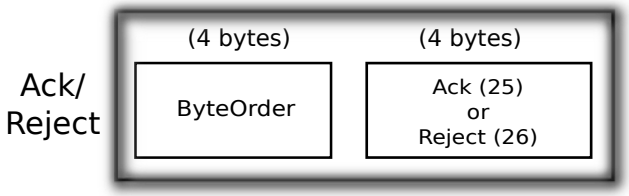

Figura 5.20: Formato de mensajes *Ack/Reject*

5.2.5. Reglas de procedimiento

Las reglas de procedimiento 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.

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.

- **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 5.21 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.

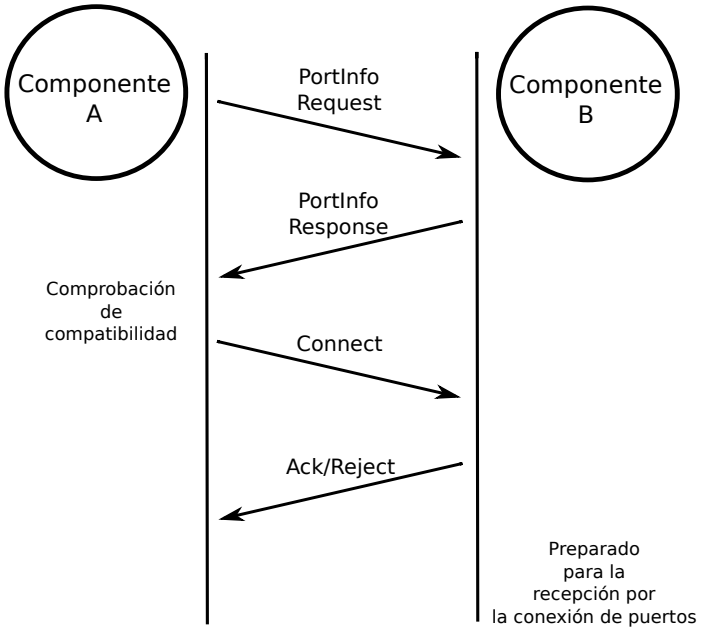

Figura 5.21: Secuencia de conexión

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, aunque existen casos en los que el extremo emisor se desconecte de forma inesperada por ejemplo por que la máquina en la que resida se apague. Esto no supone un problema para el extremo receptor ya que dispone del mecanismo de *Echo* para detectar el fallo (descrito más adelante).

  Como se observa en la imagen 5.22 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* 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.

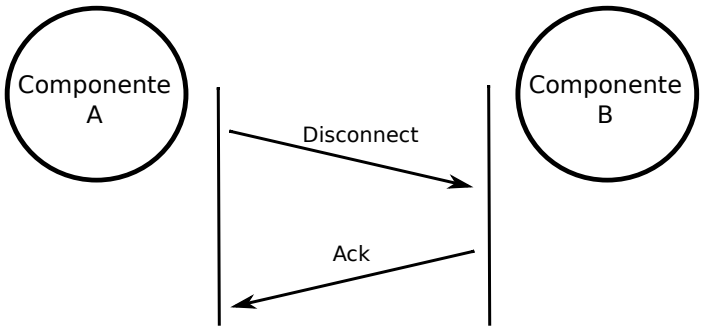

Figura 5.22: Secuencia de desconexión

- **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 5.23 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.

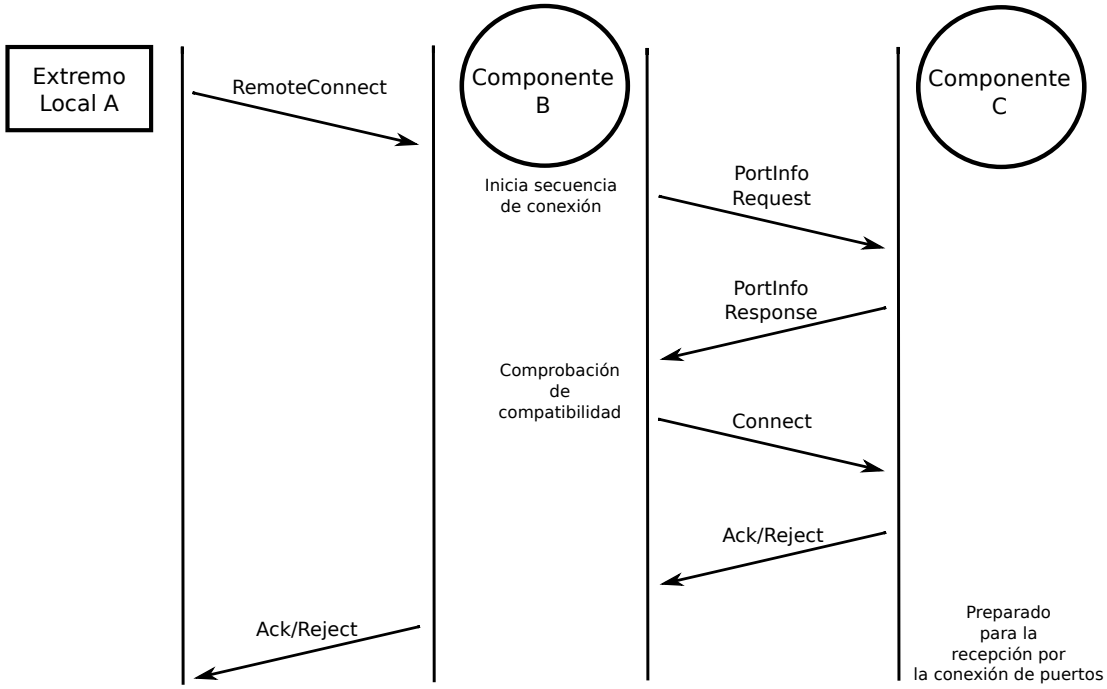

Figura 5.23: Secuencia de conexión remota

- **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 5.24, 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 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.

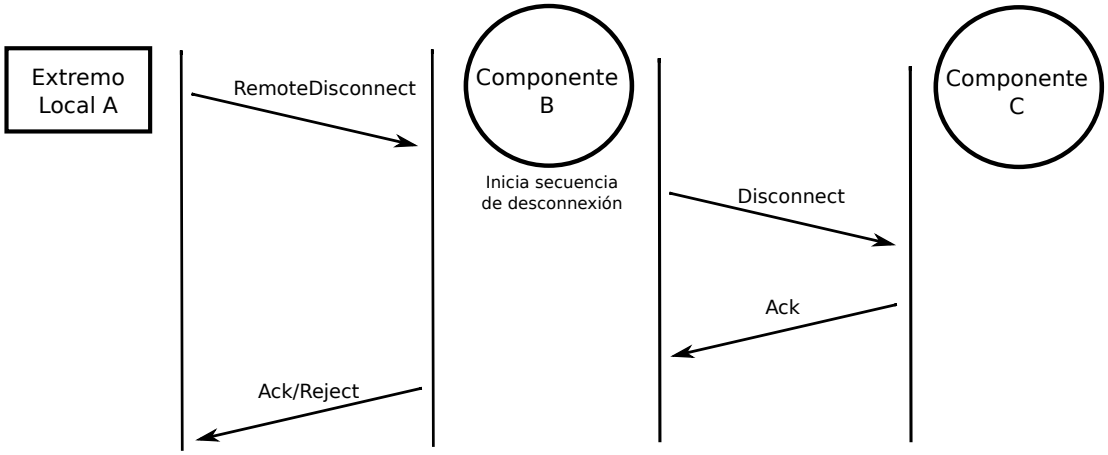

Figura 5.24: Secuencia de desconexión remota

- **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 5.25 refleja el envío de varios paquetes de puertos, cada uno encapsulado en el paquete *Data* correspondiente.

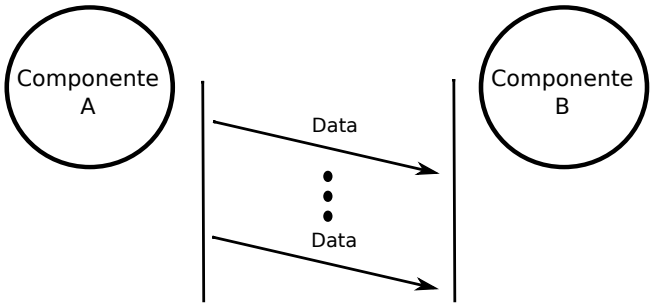

Figura 5.25: Secuencia de envío de datos

- **Envío de Echos.**

  El mecanismo de *Echo* permite interrogar a un componente remoto. Con esta secuencia de mensajes se permite que un componente verifique si un componente remoto se encuentra a la escucha y activo. La figura 5.26 clarifica la secuencia de mensajes del mecanismo de *Echo* proporcionado por *DC3P*.

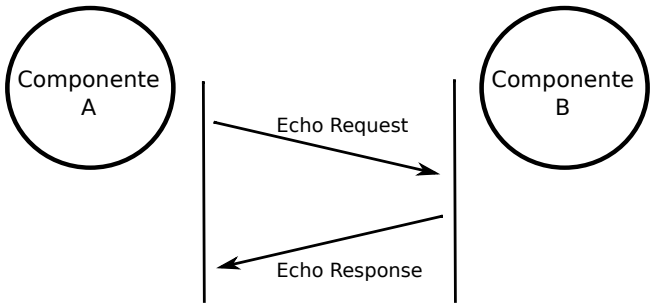

Figura 5.26: Secuencia de *Echo*

5.3. Componentes *Coolbot* con soporte de red

Una vez disponible un protocolo que recoge 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.

5.3.1. Modelo de componente sin red

Como se describe en el capítulo 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 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 5.27 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*,

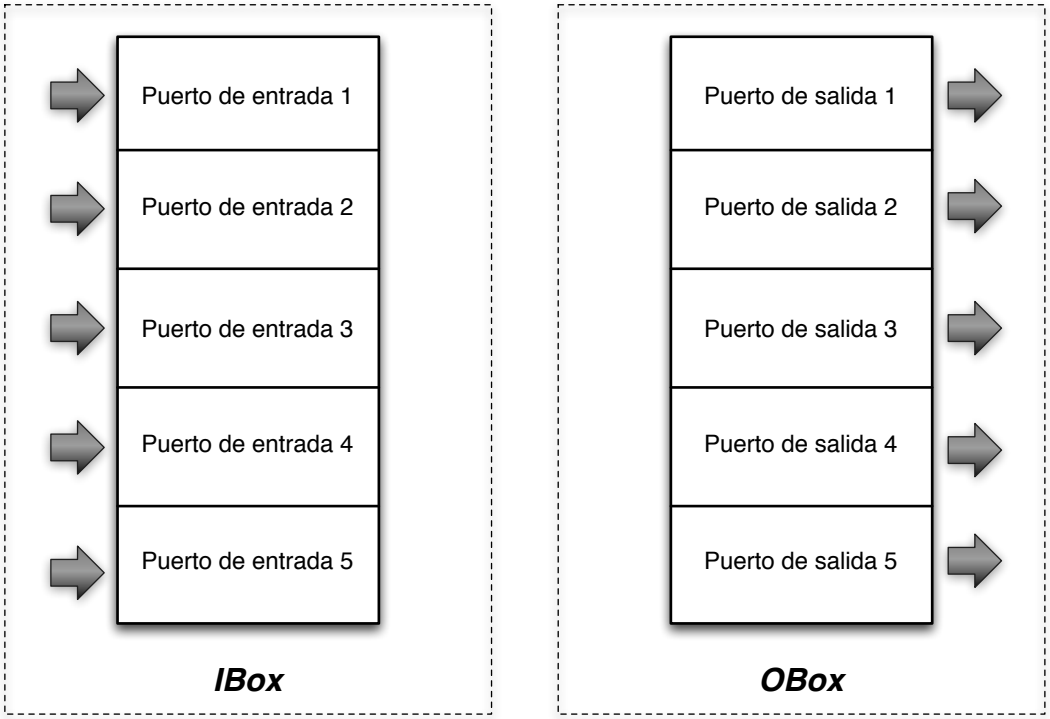

Figura 5.27: Ejemplo de IBox y OBox

cada componente dispone de un hilo *timer*. Este hilo se ejecuta en segundo plano y está destinado a realizar tareas que requieran procesado cada vez que transcurra un cierto periodo de tiempo. La figura 5.28 aclara la organización interna de un componente.

5.3.2. 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

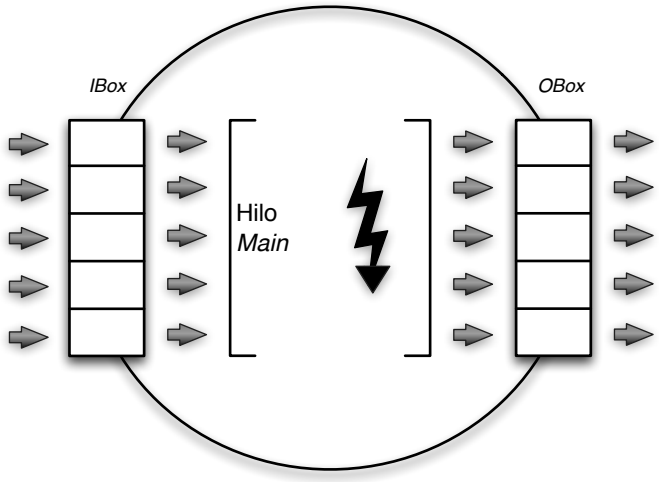

Figura 5.28: Hilo *main*, IBox y OBox de un componente

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 (conexiónes entrantes), 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)*. El *INT* es un hilo de puertos, es decir reacciona ante los eventos que se produzcan el los puertos que maneja. Este tipo de hilos se ejecutan en segundo plano de forma transparente para el usuario del componente.

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 5.2.2.

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 5.29 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.

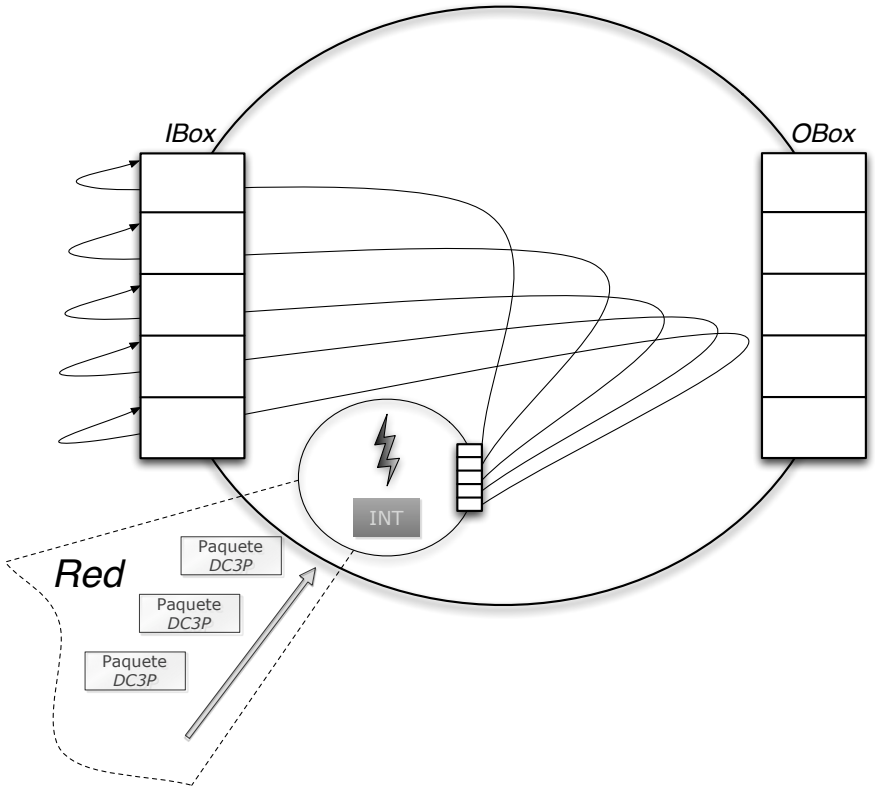

Figura 5.29: Hilo *INT* de un componente

Los envíos por la red que realiza un componente se resuelven de manera similar a las recepciones. Existe un hilo de puertos *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 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 5.30 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*.

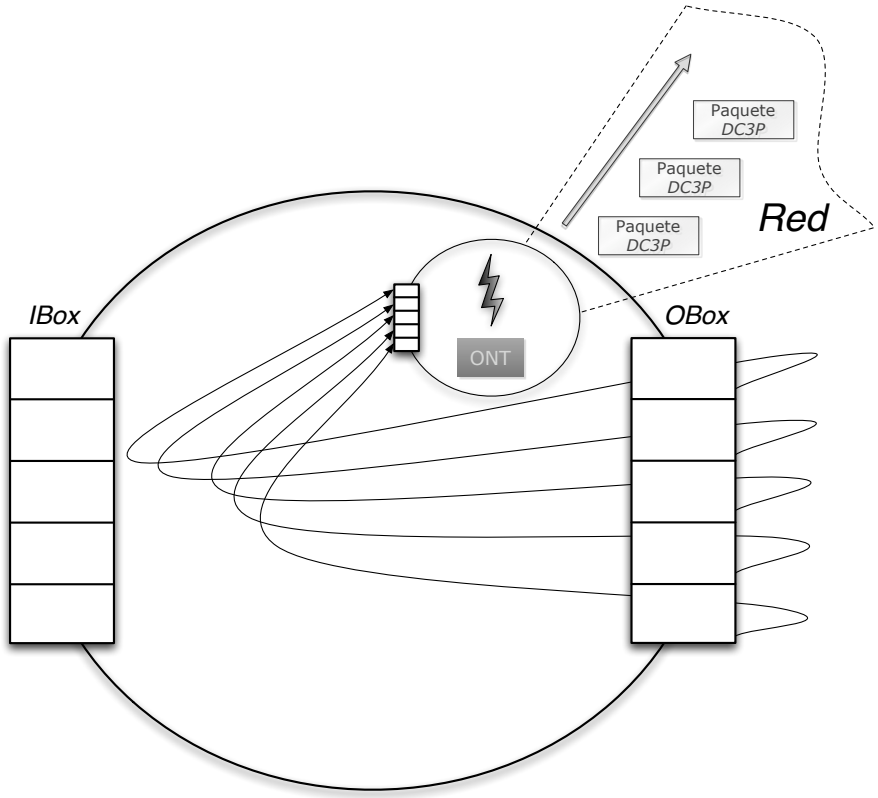

Figura 5.30: Hilo *ONT* de un componente

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 4.4.1. 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*.

Un componente con soporte de red puede ser instanciado sin que haga uso de la misma si no se le proporciona un puerto *TCP* de escucha. En este caso la infraestructura de red no se inicia y por lo tanto los hilos *INT* y *ONT* no son lanzados.

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.

5.4. Vistas *CoolBOT*

Una *vista* CoolBOT se puede definir como una vista gráfica determinada de algún aspecto de un sistema integrado por componentes *CoolBOT*. En una interfaz de teleoperación se requiere que aparezcan diferentes datos del sistema utilizando una representación apropiada. Por tanto el desarrollo de una interfaz de teleoperación con un sistema de componentes *CoolBOT* requiere la creación de *vistas* del sistema, que aparecerán anidadas en una interfaz. Para lograr el objetivo de facilitar y sistematizar la creación de *vistas* se propone el siguiente modelo, inspirado en el conjunto de patrones de diseño *Modelo Vista Controlador (MVC)*.

5.4.1. Modelo de Vista *CoolBOT*

Estas *vistas* ofrecen un diseño en el que la propia *vista* se puede ver como un componente simplificado. Dispone de una serie de puertos de salida y puertos de entrada organizados en un *IBox* y un *OBox* respectivamente y procesa al igual que un componente, es decir reacciona ante la llegada de datos a sus puertos de entrada. La función que ejecuta este tipo de componentes no es otra que la de representar los datos para su visualización gráfica, así como capturar eventos que se produzcan en la interfaz gráfica por la interacción con un usuario operador. Este diseño de *vista* utiliza los puertos de salida para el envío de comandos que un operador indique mediante un evento, como por ejemplo pulsar un botón. La estructura de una *vista*, representada por un cuadrado en la figura 5.31, es como la de un componente con infraestructura de red, pero en el que no existen estados (autómata del componente) y tampoco puertos de control ni monitorización. Con esta estructura una *vista* puede construirse de forma que tenga los puertos de entrada necesarios para capturar datos que resulten útiles para una determinada presentación gráfica, incluso proviniendo de distintos componentes. Así pues una *vista* pude tener puertos de entrada y salida para conectarse a los diferentes componentes de un sistema dado. Por ejemplo una *vista* puede tener dos puertos de entrada. Esta *vista* recibe datos de un componente que construye un mapa del entorno de un robot representándolo en una imagen. Además recibe los datos de un sensor láser desde un componente que lee los sensores de un robot real. Dicha *vista* puede combinar esos datos dibujando la imágen del mapa y sobre ella, a partir del punto en el que se encuentre el robot, un haz de líneas representando los datos del sensor láser. Esta misma *vista* puede permitir al usuario operador del sistema que al hacer *clic* sobre una zona del mapa se comande al robot para que se mueva hacia ese punto. Por tanto, la vista tendría un puerto de salida por el que se enviarían comandos al componente que a su vez los envía hacia el robot real.

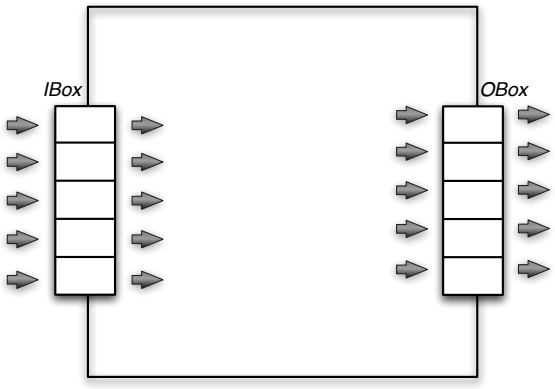

Figura 5.31: Estructura de una *Vista*

5.4.2. Sondas de componentes

Una sonda es una entidad que permite sondear, medir y analizar aspectos de algún experimento, entorno, etc. Una sonda de componente es una estructura de datos que permite acceder a la interfaz externa de un componente *CoolBOT* ya sea local o remoto. Para cada componente se define su sonda como la imagen especular de su interfaz de puertos, es decir los puertos de entrada del componente son puertos de salida en su sonda y los puertos de salida del componente son puertos de entrada de su sonda. Con este diseño una sonda puede enviar cualquier paquete de puerto a cualquiera de los puertos de entrada de un componente y recibir los paquetes de puerto que un componente emite por cualquiera de sus puertos de salida. El diagrama de la figura 5.32 refleja un componente y su sonda, con todos sus puertos de entrada y de salida interconectados.

Las sondas de componentes permiten que los datos recibidos desde un componente se presenten sin necesidad de utilizar una interfaz gráfica. Las sondas pueden ser utilizadas por ejemplo en una interfaz a través de consola o simplemente para realizar un historial de datos emitidos almacenados en ficheros. Además la correctitud en la programación de componentes puede ser verificada usando una sonda, permitiendo el intercambio de datos con cualquier puerto del componente y monitorizarlos.

La figura 5.33 representa un ejemplo de uso de una sonda. El elemento etiquetado con un **uno** es un componente *CoolBOT* que representa a un robot móvil. Este componente tiene un puerto de entrada por el que recibe un paquete de puerto que almacena coordenadas cartesianas X e Y, para comandar al robot a moverse a ese punto. También dispone de dos puertos de salida, uno emite paquetes de puerto iguales a los del puerto de entrada, indicando las coordenadas de la posición actual del robot, el segundo puerto de salida emite un paquete de puerto que almacena una imagen de una cámara de abordo, representada por sus pixeles en *RGB*. El elemento **dos** en la figura es la sonda del componente, presenta por tanto los mismos puertos pero como la imagen especular de la interfaz de puertos, de forma que las entradas son: un puerto para los datos de la posición actual del robot y otro para una imagen en *RGB*. El puerto de salida emite paquetes almacenando coordenadas para indicar al robot a donde debe moverse. En el ejemplo de la figura la sonda conecta por la red su puerto de salida al de entrada del componente y el puerto de salida de

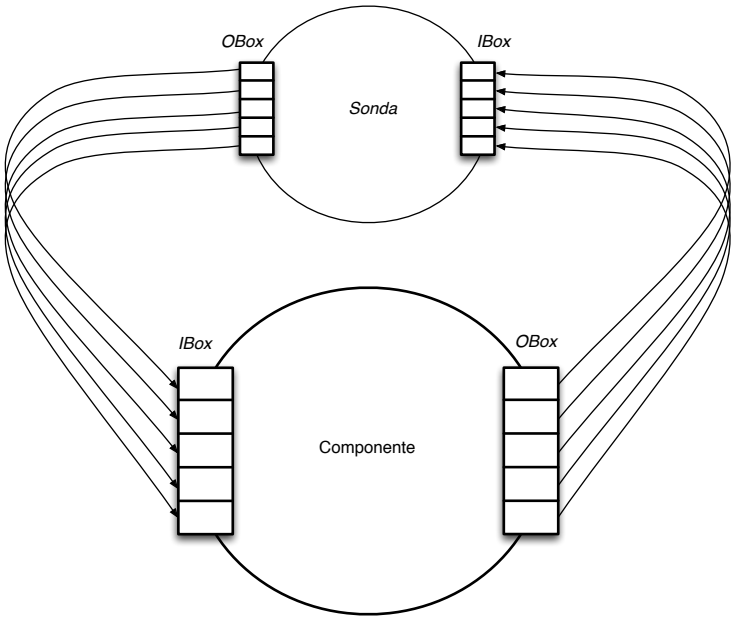

Figura 5.32: Componente interconectado con su *ComponentProbe*

coordenadas actuales del robot se conecta por la red al de entrada de coordenadas de la sonda. La aplicación qu utiliza la sonda se encarga de presentar en consola las coordenadas actuales del robot que le lleguen desde el componente. Asimismo permite que desde consola se inserten coordenadas X e Y para comandar un movimiento al robot. El número **tres** en la imagen etiqueta lo que se considera como una *vista* simple de un componente a partir del uso de una sonda.

5.5. Interfaz de teleoperación en GTK

El uso de *vistas gráficas* facilita el desarrollo de aplicaciones de teleoperación en tanto en cuanto permite obtener de forma remota y desacoplada datos recibidos desde el sistema robótico (uno o más componentes) y presentarlos de la manera más apropiada de cara al operador del sistema. Además posibilita la captura de eventos en una interfaz gráfica que son emitidos hacia componentes del sistema robótico como comandos de control.

La interfaz de teleoperación para un robot semiautónomo desarrollada en este proyecto se compone de un conjunto de *vistas gráficas* desarrolladas en *GTK* (librería presentada en la sección 4.4.4). La interfaz se estructura como aparece en la figura 5.34.

Etiquetado con un **uno** aparece una zona común de la interfaz para que las *vistas* puedan insertar elementos gráficos comunes a toda la interfaz. Un ejemplo es el conjunto de botones que se observan (*motion control*). Estos botones permiten que operador controle directamente los

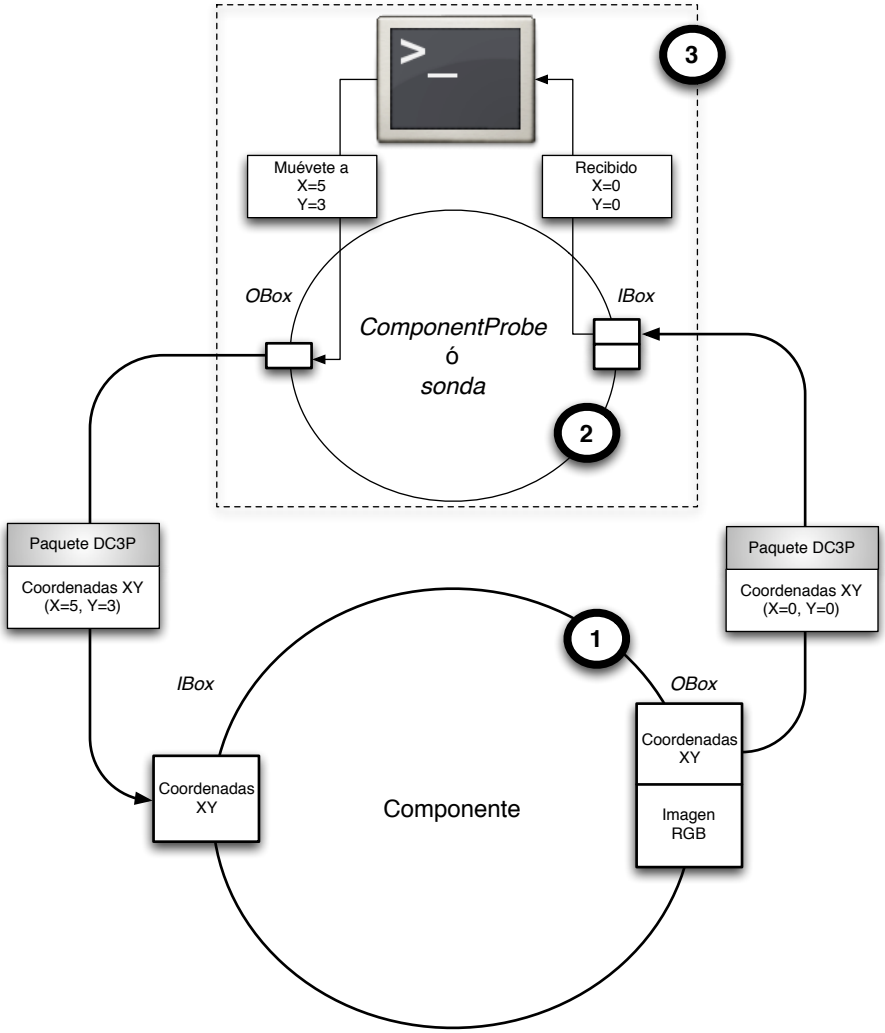

Figura 5.33: Vista de componente por consola usando una sonda

movimientos del robot, el botón central para al robot, el resto de botones realizan las siguientes acciones:

⇑: Avanzar hacia el frente.

⇓: Retroceder.

⇐: Girar sobre sí mismo hacia la izquierda.

⇒: Girar sobre sí mismo hacia la derecha.

La zona etiquetada con un **dos** muestra un conjunto de pestañas, cada una de las cuales se corresponde con una *vista CoolBOT*. La zona etiquetada con un *tres* es la zona de presentación

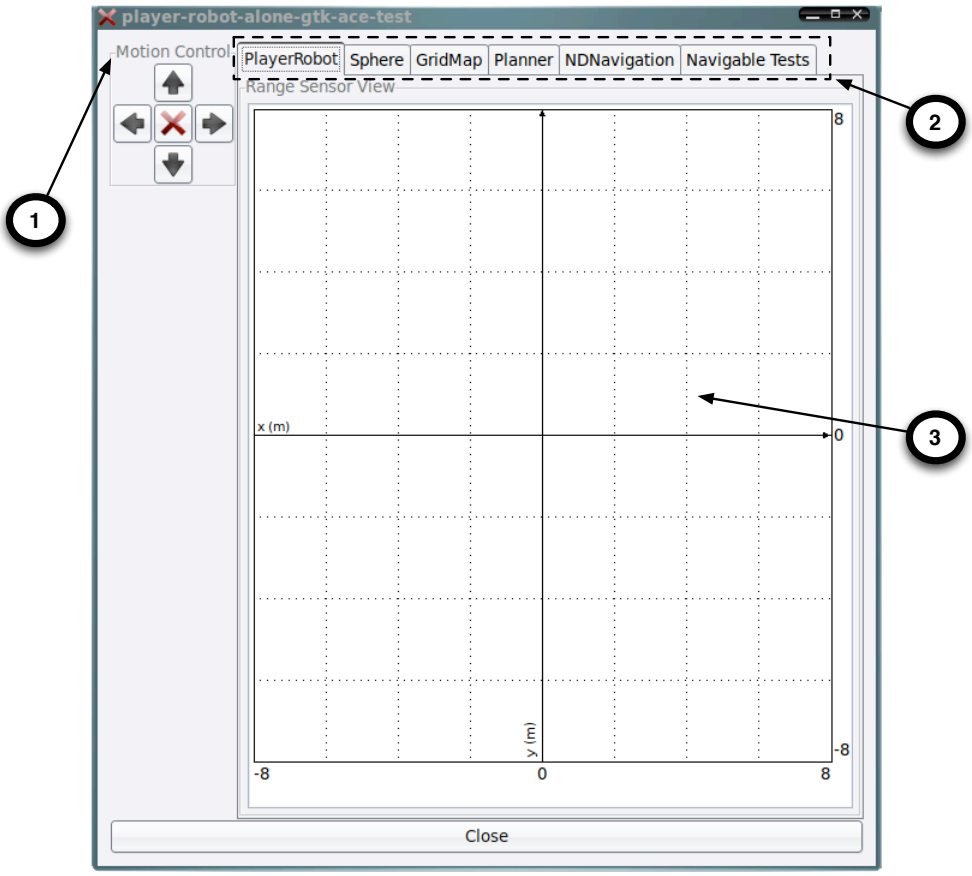

Figura 5.34: Interfaz gráfica en GTK

de datos y cada pestaña dispone de esta zona de dibujo organizada de forma diferente según la representación de datos que realice la *vista CoolBOT* correspondiente.

Cada *vista* gráfica es un objeto *GtkElement*, con lo que se dispone del método *widgetCreation* para crear el *widget GTK* que muestra en cada pestaña un área de dibujo sobre la que se representan los datos recibidos desde los componentes *CoolBOT* correspondientes en esa *vista*. Además en esta función se indica cuáles son los eventos que un operador puede realizar sobre el *widget* y las acciones que la *vista* realizará.

Cada vista dispone de un temporizador (*timer*) encargado del refresco de los datos. Estos temporizadores se ejecutan en segundo plano en un hilo transparente al usuario. Cada vez que alguno de los temporizadores se disparan se realiza un redibujado de la *vista* que corresponda. El valor de este temporizador depende del refresco de datos que la vista requiera. Si los datos se corresponden con imágenes de una cámara es probable que se requiera un refresco más rápido que cuando se trate de valores de lectura de sensores como un láser. En cualquier caso, este valor dependerá de los requerimientos concretos de cada sistema teleoperado. Principalmente el *timer* es un hilo específico en cada *vista* gráfica en *GTK*. Cuando el *timer* se dispara se ejecuta una función (*expose*) encargada de comprobar si existen datos nuevos en los puertos de entrada de la *vista*. Si

así fuera se irán recogiendo los paquetes de puertos recibidos y redibujando su representación en la zona de dibujo del *widget GTK*. Cuando la vista permite al operador actuar sobre el sistema robótico, por ejemplo para comandar un movimiento al robot, la vista asocia a los eventos (señales) que se quiera atender un *callback*. Ese *callback* lleva a cabo la acción requerida, por ejemplo el envío de un comando de movimiento hacia el puerto de entrada concreto del robot.

Esta estructura para la interfaz de teleoperación se describe en detalle en el capítulo 6.

Capítulo 6

Pruebas y resultados

Con objeto de realizar la fase de pruebas de este proyecto se ha partido de un conjunto de vistas y componentes *CoolBOT* ya desarrollados y plenamente operativos bajo la versión previa de *CoolBOT*. Cada uno de estos componentes implementa una funcionalidad específica, y a cada uno se le ha añadido el soporte de red DC3P/*CoolBOT*, descrito en el apartado 5.3.2, así como las operaciones de conexionado y desconexionado de puertos con componentes remotos. Por otro lado se ha añadido también el soporte de red a un conjunto de *vistas CoolBOT* en *GTK*, como se verá estas vistas constituyen interfaces de teleoperación para sistemas integrados por estos componentes.

Como veremos los mencionados componentes y vistas *CoolBOT* conforman un sistema de navegación segura para un robot móvil que se encuentra actualmente operativo en el Laboratorio de Robótica e Interación del Instituto Universitario SIANI. En el resto de este cápitulo pasaremos a explicar pormenorizadamente cada uno de de estos componentes y vistas. Por último se describiran las pruebas realizadas con objeto de demostrar la operatividad del desarrollo realizado en este proyecto.

6.1. Componente *PlayerRobot*

Este componente funciona internamente como un cliente de un servidor *Player* (ver sección 2.1.1) vinculado a un robot móvil real, o bien a un simulador *Stage*. En el sistema que nos ocupa robot móvil dispone de sensores de ultrasonidos o sonars, de un sensor frontral de rango láser, de *bumpers* frontales y traseras, de una cámara *Pan Tilt* de abordo, además del propio hardware de la plataforma robótica (odometría, baterías, motores, etc.). La figura 6.1 muestra la interfaz externa de este componente. Como se observa este componente abstrae el hardware proporcionado por el servidor *Player*, tanto sensores como actuadores, y proporciona dicha abstracción a otros componentes CoolBOT a través de su interfaz externo de puertos de entrada y salida.

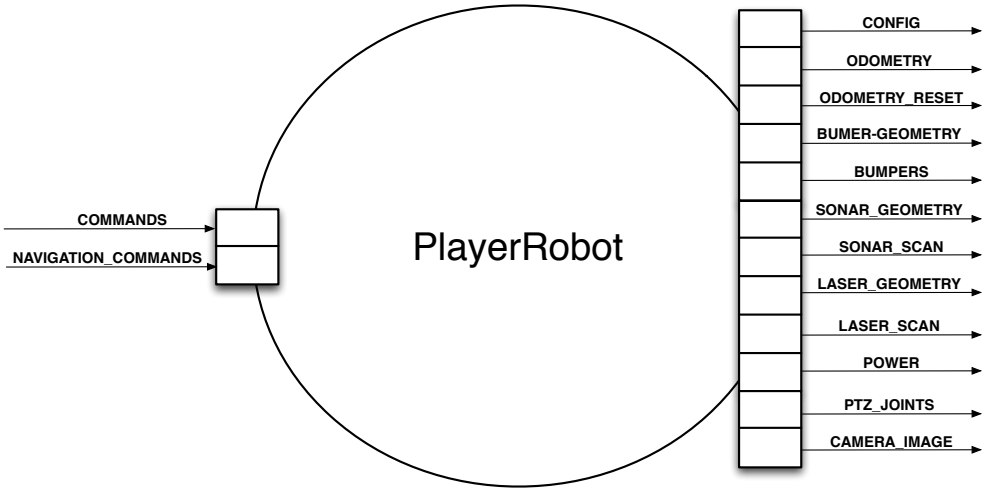

Figura 6.1: Componente *PlayerRobot*

6.2. Componente *GridMap*

Este componente ha sido diseñado para la construcción de un mapa de rejilla [ref] del entorno del robot mientras éste evoluciona en el mismo, de manera que 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, principalmente bárridos del sensor de rango láser frontal y odometría. Este comoponente adicionalmente periódicamente ejecuta un algoritmo de "ray tracing" con objeto de generar un barrido de 360^o de sensor de rango "virtual" sobre dicho mapa. Tanto el mapa de rejilla como este barrido se publican a través de uno de sus puertos de salida de manera periódica. La figura 6.2 ilustra con detalle los puertos de entrada y salida de este componente.

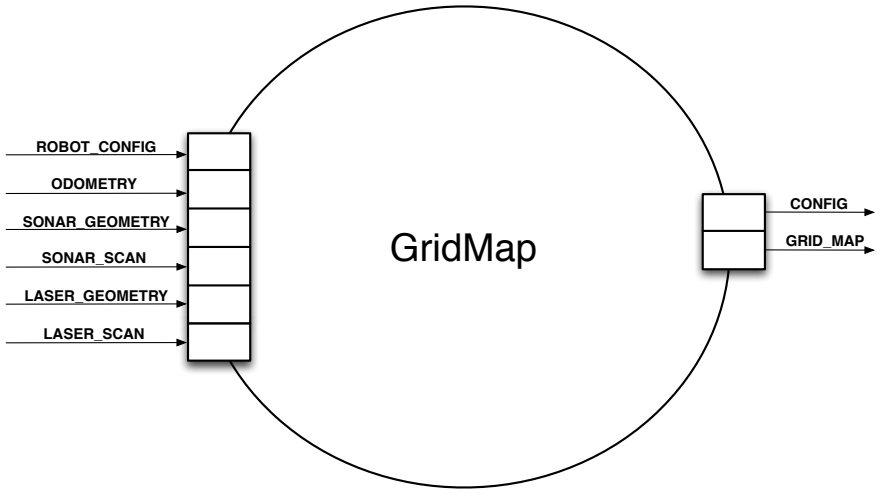

Figura 6.2: Componente *GridMap*

6.3. Componente *NDNavigation*

NDnavigation es un componente que implementa el algoritmo reactivo de evitación de obstáculos *Nearness Diagram navigation* o algoritmo (*ND*) [Minguez,2004]. Este algoritmo permite seleccionar las zonas a las que se puede mover un robot de forma segura, evitando colisionar con los obstáculos existentes en el entorno para alcanzar una zona objetivo, para ello precisa de un barrido 360^o de un sensor de rango, así como un mapa de rejilla del entorno. Este algoritmo es capaz de adaptarse a entornos dinámicos. Como veremos en los apartados finales de este capítulo, junto con los componentes *PlayerRobot* y *Grid*, conforma el bucle de control de navegación de más bajo nivel en el sistema de navegación segura que integraremos con objeto de realizar la fase de pruebas de este proyecto. Este componente hace uso del mapa y del barrido 360^o proporcionados por el componente *GridMap* para determinar el camino a seguir en cada paso del bucle de control del algoritmo de envitación de obstáculos, para ello también en dicho bucle se comanda al robot a través del componente *PlayerRobot*. Adicionalmente este componente también proporciona una serie de datos para la depuración y seguimiento del algoritmo durante su ejecución, con objeto de que los detalles de su ejecución se puedan visualizar en alguna vista. La figura 6.3 aclara la interfaz de este componente.

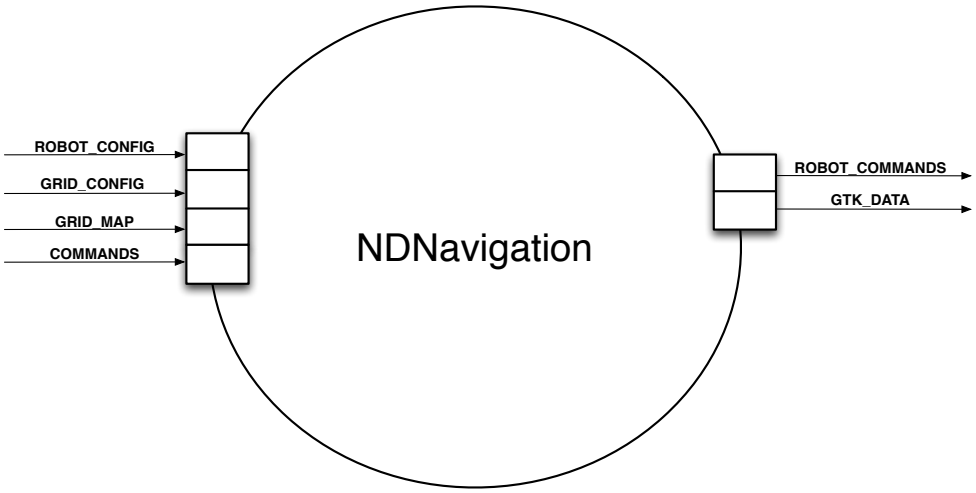

Figura 6.3: Componente *NDNavigation*

6.4. Componente *ShortTermPlanner*

Este componente *CoolBOT* implementa un planificador a corto plazo basado en una modificación del algorimto NF2 [ref] para mapas de rejilla. 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. Dicha ruta se determina sobre un mapa de rejilla que representa al entorno de planificación. 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 del punto objetivo o *goal* que se pretende alcanzar. Como resultados *ShortTermPlanner* emite una ruta a seguir como un conjunto de subobjetivos, pensados para que sean seguidos por un algoritmo reactivo de evitación de obstáculos, 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), con objeto de que se pudea visualizar su funcionamiento interno. Debido a que el mapa sobre el que el sistema robótico es dinámico y varía a medida que el sistema evoluciona y se mueve en el entorno, este componente cuando se encuentra en funcionamiento, replanifica periódicamente con objeto de encontrar la mejor ruta posible al punto objetivo. Cuando este algoritmo se encuentra activo envía comandos de navegación a los subobjetivos del camino planificado al componente *NDNavigation*, también es responsable de determinar la transición entre subobjetivos a lo largo del camino, además de la determinación de cuando el sistema llega al punto de destino final. La figura 6.4 describe las entradas y salidas de este componente.

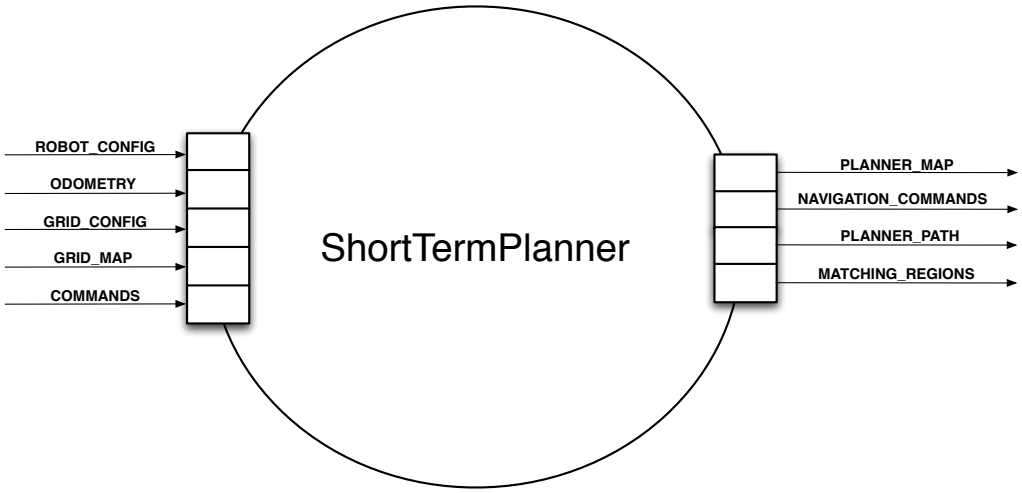

Figura 6.4: Componente *ShortTermPlanner*

6.5. Vista PlayerRobotGtk

Se trata de una *vista CoolBOT* 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 permite controlar directamente al sistema robótico móvil al añadir botones de control de movimiento al zona común a todas las vistas de la interfaz del sistema. La figura 6.5 muestra la interfaz de puertos de la *vista PlayerRobotGtk*. La imagen de la figura 6.6 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, y en la parte inferior de la vista se muestra la odometría del robot, así como el nivel de baterías y otra información de interés.

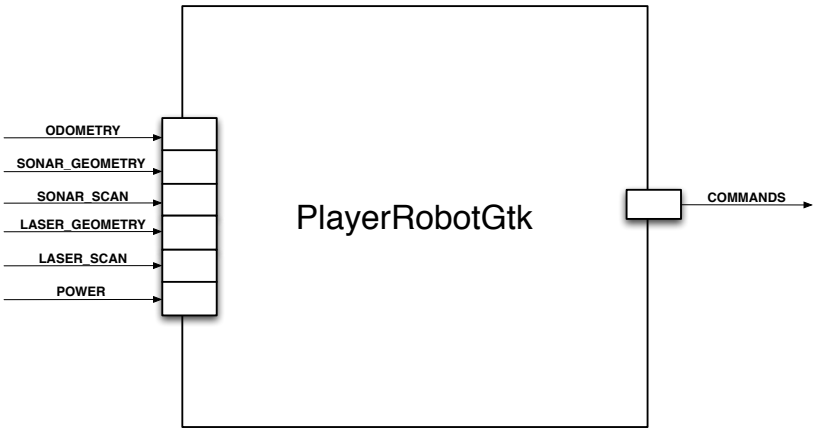

Figura 6.5: Interfaz de puertos de *PlayerRobotGtk*

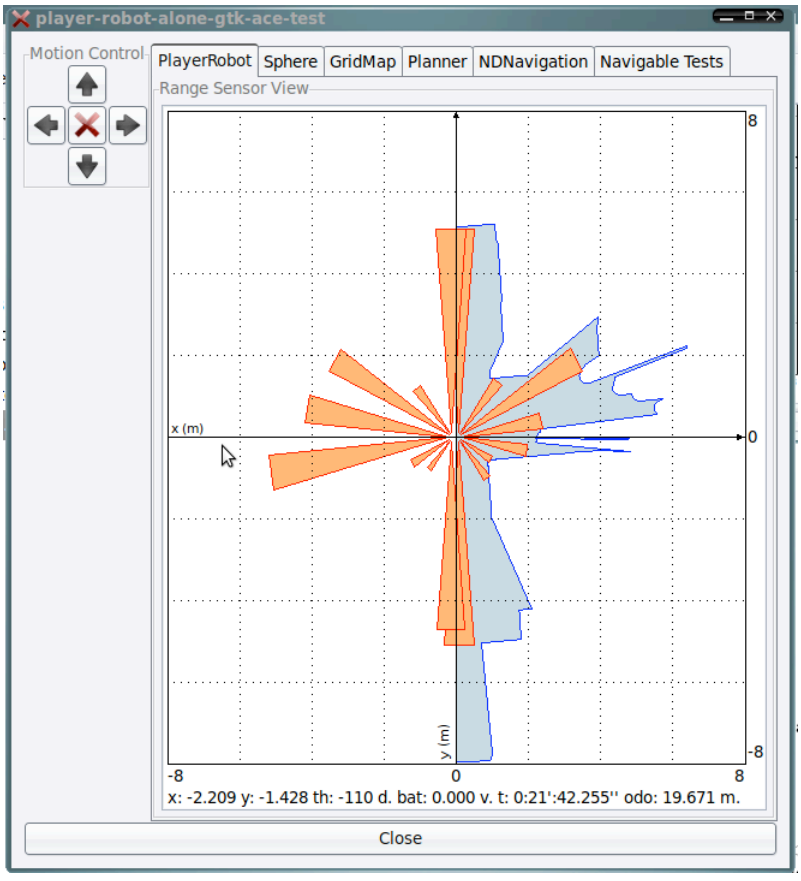

Figura 6.6: Vista *PlayerRobotGtk*

6.6. Vista SphereGtk

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 y horizontal así como inicializarla con objeto de que se situa en una posición conocida de inicialización o (*home*). Para ello, la *vista* ofrece dos barras deslizables (*sliders*) en vertical y horizontal, además de un botón *home*. La imagen de la figura 6.8 refleja una recepción de imágenes en esta vista. En la figura 6.7 se observa la interfaz de puertos de la *vista*.

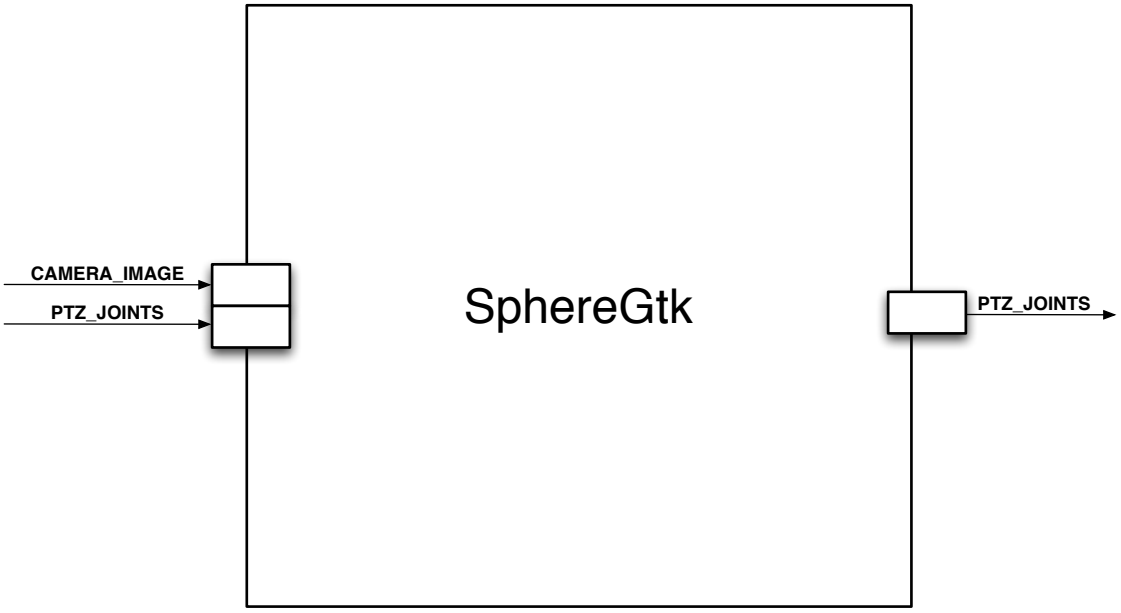

Figura 6.7: Interfaz de puertos de *SphereGtk*

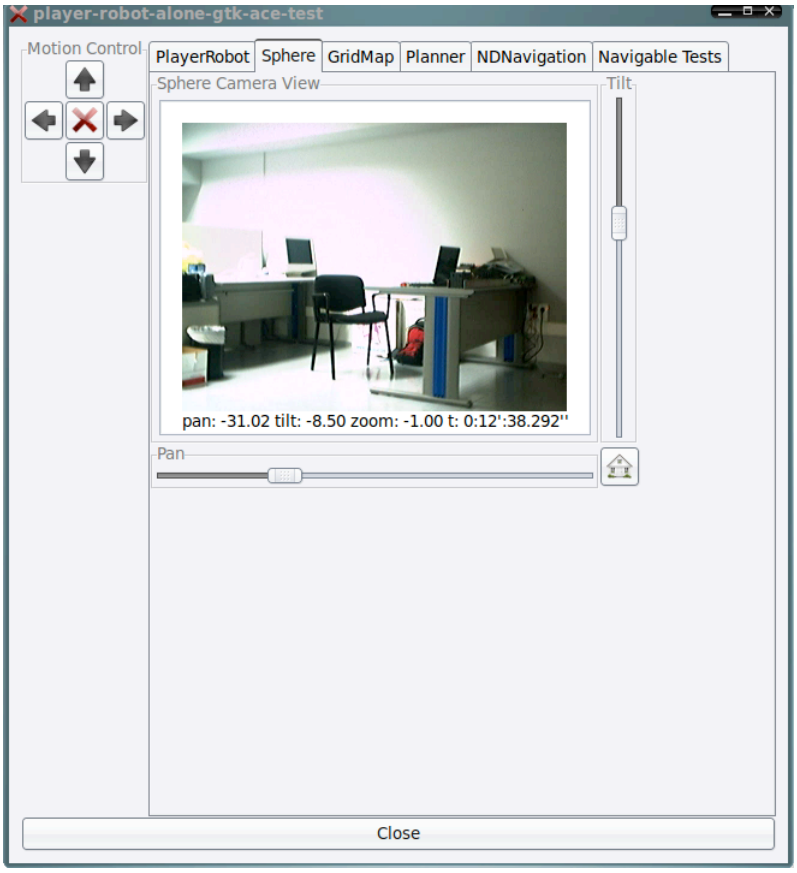

Figura 6.8: Vista *SphereGtk*

6.7. Vista GridGtk

En la figura 6.10 se observa la interfaz de teleoperación con la *vista GridGtk*. 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 *ray tracing* en color cian, es decir, dibuja segmentos con origen el la posición del robot hasta las zonas de obstáculos en el mapa, se trata este de el barrido de sensor de rango virtual generado por el comoponente *GridMap* comentado previamente. La vista presenta la ruta planificada por el componente *ShortTermPlanner* indicando con un círculo en *goal* final y con cuadrados los subobjetivos. La *vista GridGtk* ofrece al teleoperador 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 evitando obstáculos de forma autónoma hasta el objetivo, 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 sin planificación a corto plazo, es decir, sin que el componente *ShortTermPlanner* calcule una ruta

hasta el objetivo final, con objeto de depurar aisladamente el comportamiento del algoritmo reactivo de evitación de obstáculos. 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 6.9 ilustra los puertos de *GridGtk*.

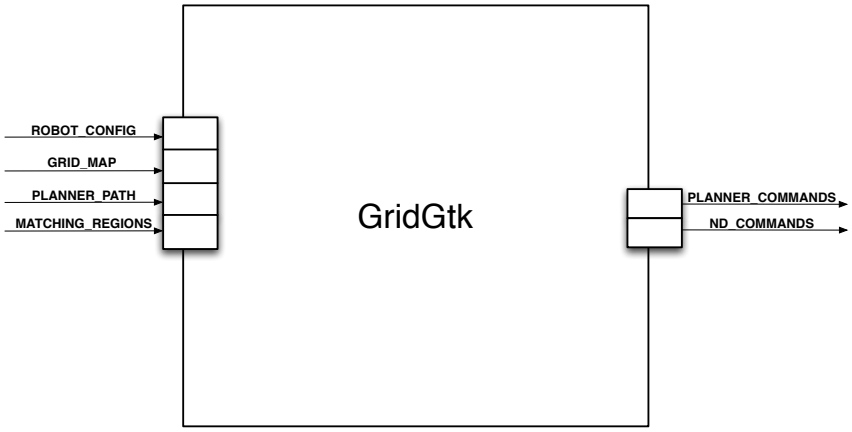

Figura 6.9: Interfaz de puertos de *GridGtk*

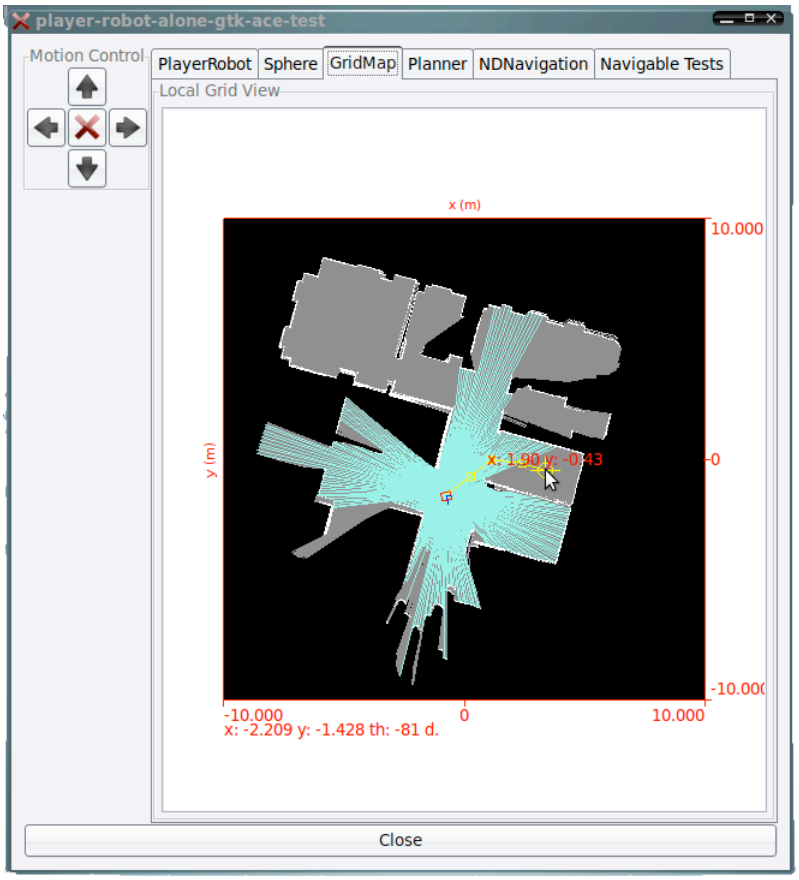

Figura 6.10: Vista *GridGtk*

6.8. Vista PlannerGtk

Esta vista muestra datos generados por por el componente *ShortTermPlanner*, concetamente la planificación que realiza el mismo sobre el mapa del entorno del sistema. Este mapa está basado en el generado por el componente *GridMap*, con la salvedad de que presenta diferentes zonas coloreadas: obstáculos en amarillo, ruta planificada en azul, objetivo final y subobjetivos en cuadros verdes, líneas rojas para el esqueleto del mapa (división del mapa en regiones) y en gris las zonas restantes. En la imagen 6.12 se observa la *vista PlannerGtk* mostrando un mapa generado por el componente planificador. En la figura 6.11 se reflejan los puertos necesarios para esta *vista*.

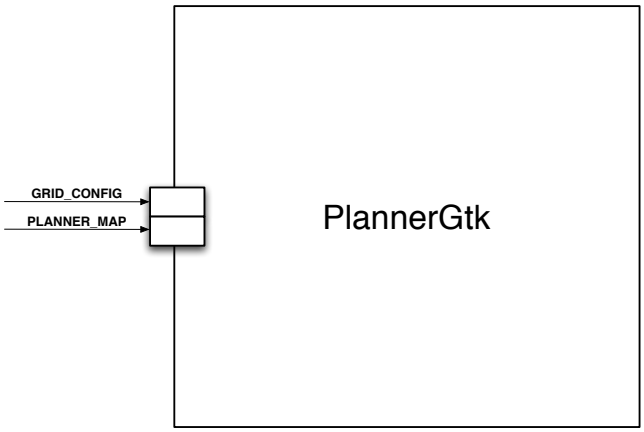

Figura 6.11: Interfaz de puertos de *GridGtk*

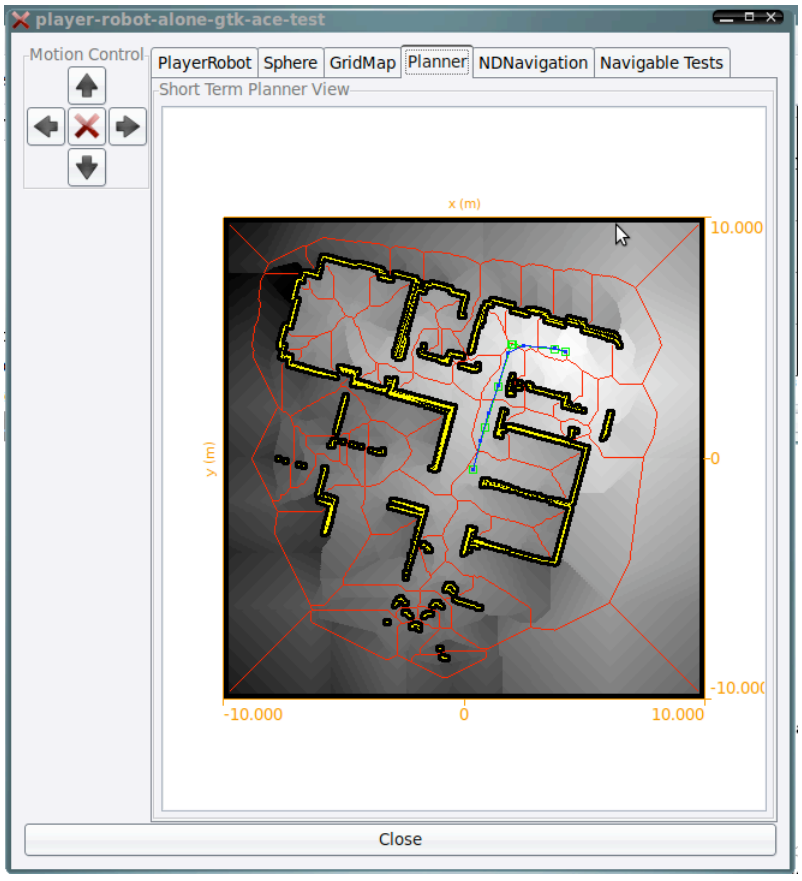

Figura 6.12: Vista *PlannerGtk*

6.9. Vista NDNavigationGtk

La *vista NDNavigation* tiene un carácter de depuración y seguimiento de la ejecución del algoritmo de navegación *ND* que implementa el componente *NDNavigation*. Está orientada principalmente hacia desarrolladores de sistemas robóticos que usen este componente *CoolBOT* y deseen monitorizar en detalle la actividad del mismo. Principalmente se reflejan los valles (posibles zonas de paso) que el algoritmo ND maneja en su ejecución. Como se observa en la imagen de la figura 6.14, éstos se presentan dibujados en un eje de coordenadas centrado en el robot (zona superior de la *vista*) y en dos gráficas donde se marca en verde el valle seleccionado y por tanto indicativo de la zona a la que se moverá el robot en cada una de las ejecuciones periódicas del bucle de control del algoritmo de evitación de obstáculos. La figura 6.13 refleja la interfaz de puertos de esta *vista*.

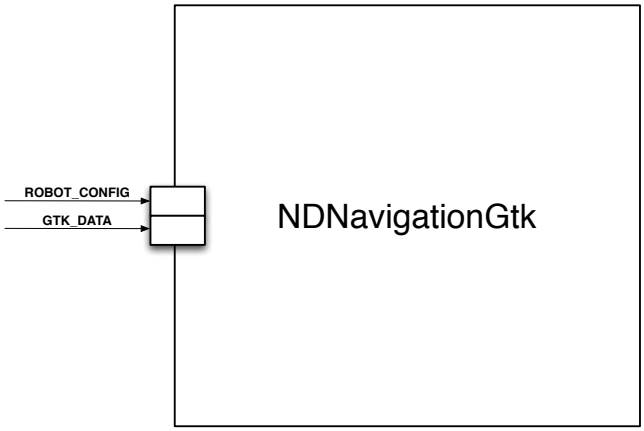

Figura 6.13: Interfaz de puertos de *NDNavigationGtk*

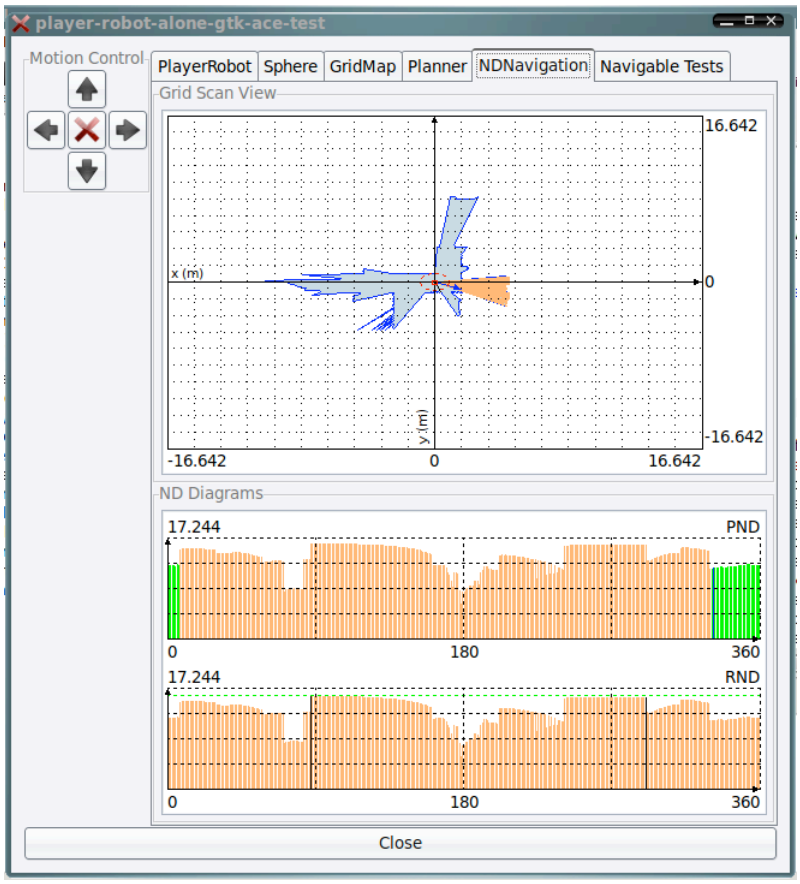

Figura 6.14: Vista *NDNavigationGtk*

6.10. Conexionado del sistema completo

La figura 6.15 muestra las conexiones de puertos entre todos los componentes del sistema semiautónomo desarrollado, no se muestran las conexiones con las *vistas*.

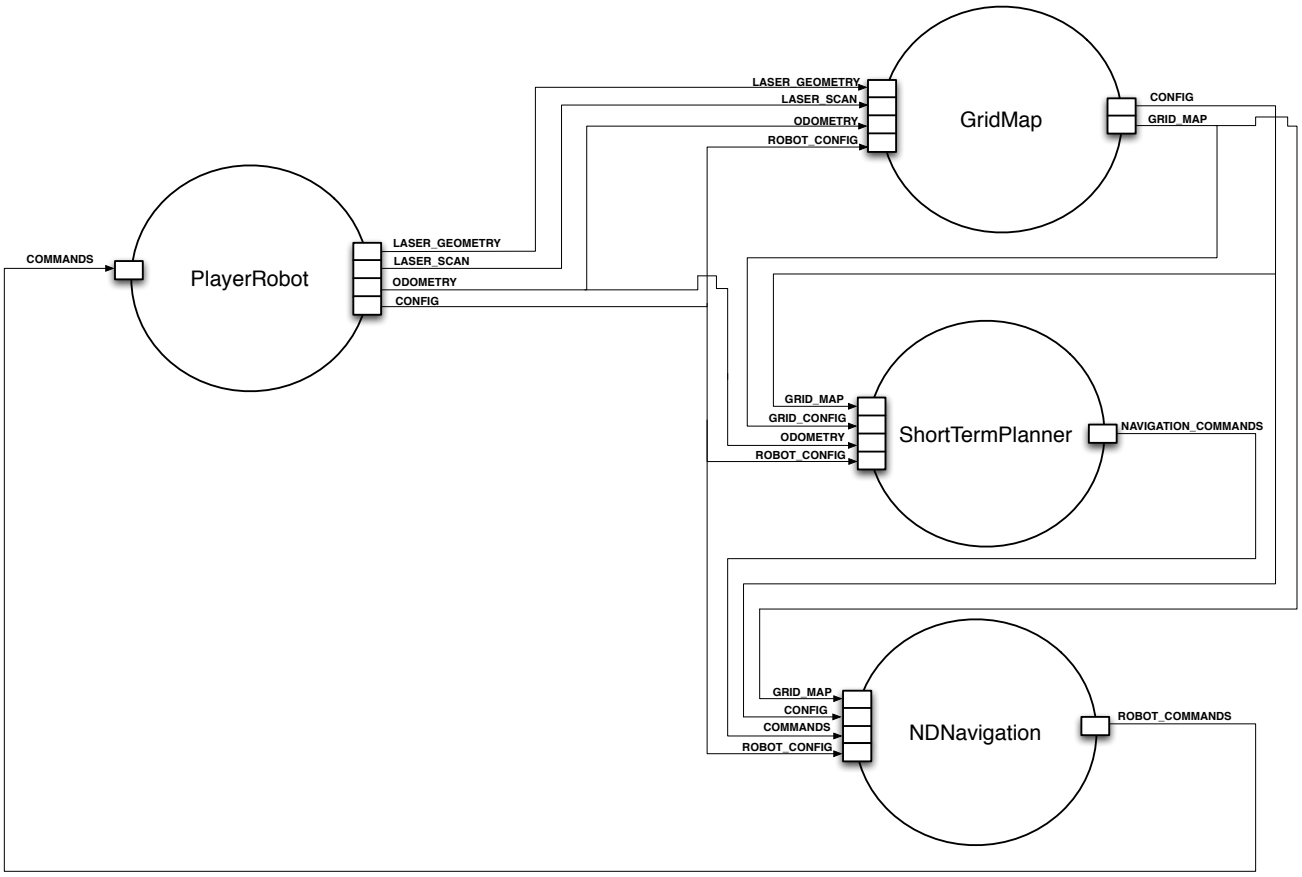

Figura 6.15: Conexionado de componentes del sistema

[Debería de haber una gráfica con el conexionado de las vistas con los componentes]

6.11. Pruebas realizadas

Con el sistema robótico semiautónomo 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 través de la *vista GridMapGtk* 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.

6.11.1. 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 6.16 ilustra la configuración realizada.

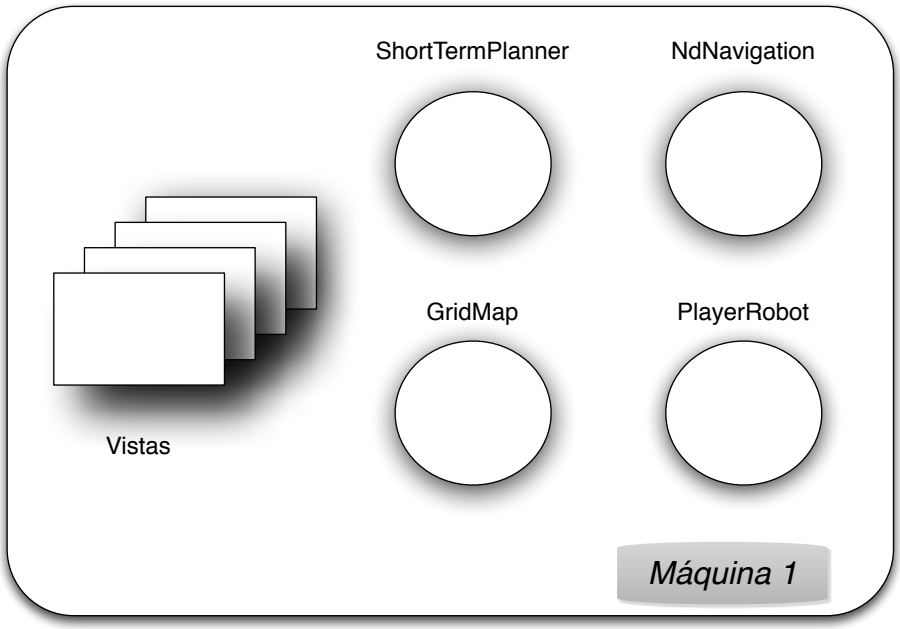

Figura 6.16: Configuración Test1

6.11.2. Test2: Sistema teleoperado

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

En una primera máquina se localizan todos los componenes del sistema: *PlayerRobot, NDNavigation, GridMap, ShortTermPlanner*. En esta máquina las conexiones entre componentes se 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 6.17 refleja la distribución realizada para este test.

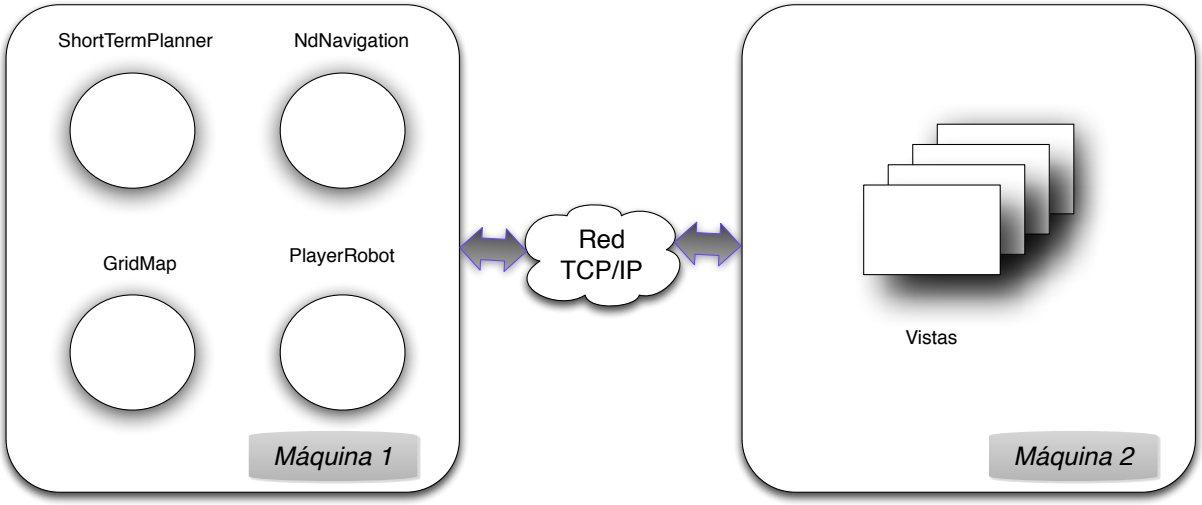

Figura 6.17: Configuración Test2

Capítulo 7

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.

7.1. 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 que se desenvuelve en entornos de interiores.

7.1.1. Conclusiones generales

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

El mecanismo de diseño aplicado basado en el framework *CoolBOT* 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 robot móvil semiautónomo con telecontrol y capacidad para la inspección de interiores, se han descompuesto en componentes siguiendo la filosofía de diseño y desarrollo *it*. 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 delegar tareas para ser ejecutadas de forma semiautónoma (permitiendo la intervención del operador humano en cualquier momento).

Con esto se ha logrado satisfacer el objetivo final de este proyecto: obtener un sistema robótico móvil semiautónomo teleoperado que, con los sensores manejados y las conductas integradas, está preparado para realizar labores de acceso a zonas de interior.

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 *CoolBOT*

Para la creación de *vistas CoolBOT*se ha creado un esquema inspirado en el patrón de diseño *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, puede sistematizarse en cualquiera de sus dos diseños.

7.1.2. Conclusiones experimentales

Mediante la integración de un sistema de navegación segura para un robot móvil que se desenvuelve en un entorno de interiores, 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 un sistema integrado 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 cualquier 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*.

7.2. Trabajo futuro

En este proyecto se describe el desarrollado 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 integrado por componentes y vistas que permite la teleoperación de un sistema robótico que se desenvuelve en un entorno de interiores. 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, las sondas de componentes ni las *vistas CoolBOT*. Para incrementar la facilidad de creación de estos elementos se plantea la posibilidad de que esta herramienta automatice también la construcción de sondas y *vistas*, 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 direcciones 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 computació basado en *grids*.

Apéndice A

Manuales de usuario

A.1. Guía de instaliónde de *CoolBOT* y componentes del sistema de evitación de obstáculos

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.1.1. *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.1.2. 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.1.3. 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.1.4. 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.1.5. 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*.

A.2. Guía de creación de paquetes de puerto

Esta sección presenta una guía sobre cómo definir las estructuras de datos, bien clases o structs, de paquetes de puertos de componentes. Se explica paso a paso cómo definir dichos datos para lo que se presenta una clase de ejemplo: *Packet*. Esta clase irá tomando forma y creciendo hasta estar lista para ser usada por los componentes *CoolBOT*.

Cuando se definen los datos que algún componente necesita como entrada, o que emite como salida, por alguno de sus puertos, se debe dotar a las clases de una serie de operaciones que permiten a los componentes *CoolBOT* el manejo de dichos paquetes de puertos. La interfaz de esas operaciones se encuentra en una serie de clases abstractas que, con el mecanismo de herencia de C++, imponen su definición de las operaciones en las clases hijas. Todas las interfaces de las que deben heredar las clases de paquetes de puertos: *PackingInterface, Cloninginterface, DeepCopyingInterface, DebuggingInterface, NamingInterface*, se encuentan unificadas en la clase abstracta *PortPacket*. Para más detalle sobre cada una de estas clases ver el apéndice C.3.

Para la clase *Packet* que se usará de ejemplo, se supondrá una serie de atributos de diferentes tipos: un entero, un booleano y un objeto de una clase *ExampleClass*. La clase *ExampleClass* debe heredar de forma pública la interfaz *PortPacket*:

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

class Packet:  public PortPacket
{
        public:
                Packet(){}

                //inherited member functions
        private:
                //atributes
                int _integer_;
                bool _boolean_;
                ExampleClass _exampleClass_;
};
```

A.2.1. PackingInterface

Las clases de paquetes de puertos deben definir las operaciones *toBytes* y *fromBytes*, que transforman los datos a la representación intermedia de red *CDR*(sección 4.4.1), o los extraen de la misma. Así mismo deben definir la operación *packingMaxLength*, que calculará el tamaño máximo que los datos ocupan en el formato *CDR*. La interfaz de estas operaciones está definida en la clase abstracta *PackingInterface*, (apéndice C.3.5). Estas operaciones están destinadas principalmente a permitir que los datos viajen por la red *TCP/IP*.

```
#include "ports/coolbot_portpacket.h"
#include "network/coolbot_packinginterfacewrapper.h"

class Packet:  public PortPacket
{
        public:
                Packet(){}

                //member functions from PackingInterface
                bool toBytes(ACE_OutputCDR &oCDR); //marshalling data Packet
                bool fromBytes(ACE_InputCDR &iCDR); //demarshalling data Packet
                int packingMaxLength(); //maximun length of data in CDR format.

        private:
```

```
                //atributes
                int _integer_;
                bool _boolean_;
                ExampleClass _exampleClassObj_;
};
```

Para definir las funciones *toBytes* y *fromBytes* existen una serie de funciones ya suministradas, también denominadas *toBytes* y *fromBytes* pero con distinto prototipado (proporcionadas en la librería *packinginterfacewrapper.h*). Estas funciones facilitan la tarea a la hora de definir las funciones para las nuevas clases y abstraen de detalles sobre el manejo de la librería de comunicaciones *ACE*. Estas funciones se encuentran sobrecargadas para realizar *(de)marshalling* de cualquier dato primitivo y vectores de datos primitivos. También pueden usarse con datos construidos que ya tengan definidas sus funciones miembro *toBytes* y *fromBytes*. Para detalles sobre la implementación de estas funciones básicas ver el apéndice C.1.

Una definición concreta de las funciones *toBytes* y *fromBytes* para la clase *Packet* es la siguiente:

```
bool toBytes(ACE_OutputCDR &oCDR) //marshalling data Packet
{
        CoolBOT::toBytes(_integer_,oCDR);
        CoolBOT::toBytes(_boolean_,oCDR);
        CoolBOT::toBytes(_exampleClassObj_,oCDR);
        //an alternative could be _exampleClassObj_.toBytes(oCDR)
        return oCDR.good_bit();
}

bool fromBytes(ACE_InputCDR &iCDR) //demarshalling data Packet
{
        CoolBOT::fromBytes(_integer_,iCDR);
        CoolBOT::fromBytes(_boolean_,iCDR);
        CoolBOT::fromBytes(_exampleClassObj_,iCDR);
        //an alternative could be _exampleClassObj_.fromBytes(iCDR)
        return iCDR.good_bit();
}
```

En el ejemplo concreto de *Packet*, la clase *ExampleClass* implementa la interfaz *packingInterface*, ya que son datos que viajarán por la red como atributo de un paquete de puerto. Por tanto las instancias de *ExampleClass* cuentan con sus funciones *toBytes* y *fromBytes*. Como se observa en el código superior existen dos posibilidades a la hora de empaquetar el objeto *_exampleClassObj_*, bien llamando directamente a sus funciones miembro, o bien usando las funciones básicas de empaquetado de la librería *packinginterfacewrapper.h* que, en definitiva, hará una llamada a la función miembro que corresponda de la instancia que se le pase como parámetro.

Como se observa en el código anterior, el **orden de *marshalling*** de los atributos de la clase es exactamente igual al orden en que se hace el *demarshalling*. Esto es absolutamente necesario por las características de manejo de los *CDR Streams* que la librería *ACE* presenta y por tanto, para que los datos se interpreten adecuadamente.

Todas las funciones de *(de)marshalling* retornan un valor boolano indicando si la operación se ha efectuado con éxito. Como se observa el valor concreto retornado es el devuelto por la función *good_bit*, miembro de las clases *ACE_OutputCDR* y *ACE_InputCDR*.

La última función a definir relacionada con el paso de los datos a un *CDR Stream* es *packingMaxLength*. Se trata de una función que devuelve un valor entero que indica el tamaño en bytes que ocupan los datos en formato *CDR*. Esta función es utilizada para determinar los tamaños necesarios de los streams *ACE_OutputCDR* y *ACE_InputCDR* que se pasan como parámetros en las funciones *toBytes* y *fromBytes*. De nuevo, de cara a facilitar la definición y abstracción de detalles de la librería *ACE* y la representación *CDR*, se suministran en *packinginterfacewrapper.h* una serie de funciones básicas para el cálculo de longitudes. Estas funciones devuelven, para cada tipo de dato primitivo o vectores de los mismos, el valor en bytes que ocupa dicho dato sumándole los bytes de relleno máximo, que pueden ser necesarios para que el dato se alinee en el *CDRstream*. Para detalles sobre la implementación de estas funciones básicas ver el apéndice C.2.

```
int packingMaxLength()
{
        return  CoolBOT::packingMaxLength(_integer_) +
                CoolBOT::packingMaxLength(_boolean_) +
                // same as: _exampleClassObj_.packingMaxLength()
                CoolBOT::packingMaxLength(_exampleClassObj_);
}
```

Como en el caso de las funciones *toBytes* y *fromBytes* existen dos alternativas para calcular el tamaño de un tipo de dato construido. Una posibilidad es usar la función miembro de la instancia del tipo de dato y otra es usando las funciones básicas suministradas por *packinginterfacewrapping.h*.

A.2.2. ClonningInterface

Los paquetes de puertos de *CoolBOT* se han diseñado usando el patrón *prototype* para su manejo, por lo que heredan de la clase abstracta *CloningInterface*, (apéndice C.3.4). Este patrón permite que los paquetes del protocolo de red *DC3P*, encargado de las comunicaciones de red entre componentes, manejen indistintamente cualquier variedad de paquetes de puerto. Por tanto las clases que definen los paquetes de puerto heredan la interfaz *CloningInterface*, para implementar el patrón *prototype*. Además la clase debe tener un atributo que será un puntero estático a su tipo, que define un prototipo para la clase. Así mismo contará con una función miembro estática que retorne dicho prototipo en la forma de la clase padre *PortPacket*. Para realizar adecuadamente la inicilización de las variables estáticas se hace uso de la clase template *StaticInit*. Por tanto, para inicializar el puntero *_pPrototype_* se añaden en la sección privada de la clase las funciones *_staticInitialization_*,

*_staticFinalization_* y la clase *friend StaticInit*. La función *StaticInit* será llamada cuando se haga un *include* del fichero de cabecera de la clase que define el paquete de puerto.

```
#include "ports/coolbot_portpacket.h"
#include "network/coolbot_packinginterfacewrapper.h"

class Packet:  public PortPacket
{
        public:
                Packet(){}

                //copy constructor
                Packet(const Packet& packet);

                //member functions from PackingInterface
                bool toBytes(ACE_OutputCDR &oCDR); //marshalling data Packet
                bool fromBytes(ACE_InputCDR &iCDR); //demarshalling data Packet
                int packingMaxLength(); //maximun length of data in CDR format.

                //member functions from CloningInterface and prototype pattern
                CloningInterface* clone() const; //clone an object
                static PortPacket* prototype()  // return a prototype
                {
                        return _pPrototype_;
                }

        private:
                //atributes
                int _integer_;
                bool _boolean_;
                ExampleClass _exampleClassObj_;

                //static atributes for prototype pattern
                static Packet *_pPrototype_;

                //functions for static initializations
                static void _staticInitialization_()
                {
                        _pPrototype_=new Packet();
                }

                static void _staticFinalization_()
                {
```

```
                        if(_pPrototype_)
                                delete _pPrototype_;
                }

                // StaticInit<Packet> carries out
                //initialization of static variables
                friend class StaticInit<Packet>;
};
```

La definición de la función *clone* devuelve un nuevo objeto de tipo *Packet* igual que el objeto actual, para lo que, en este ejemplo, se apoya en un constructor de copia:

```
//copy constructor
Packet(const Packet& packet)
{
        _integer_=packet._integer_;
        _boolean_=packet._boolean_;
        //supposing ExampleClass has overloaded operator =
        _exampleClassObj_=packet._exampleClassObj_;
}


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

A.2.3. DeepCopyingInterface

Para permitir que un paquete de puerto pueda copiarse elemento a elemento sin que compartan bloques de memoria, debe definir una función a tal efecto, para ello implementa la intefaz *DeepCopyingInterface*, (apéndice C.3.3). Esto permite copiar un objeto en otro ya creado previamente y por tanto reutilizándlo sin crear un objeto nuevo con constructores de copia.

```
#include "ports/coolbot_portpacket.h"
#include "network/coolbot_packinginterfacewrapper.h"

class Packet:  public PortPacket
{
        public:
                Packet(){}

                //copy constructor
```

```
                Packet(const Packet& packet);

                //member functions from PackingInterface
                bool toBytes(ACE_OutputCDR &oCDR); //marshalling data Packet
                bool fromBytes(ACE_InputCDR &iCDR); //demarshalling data Packet
                int packingMaxLength(); //maximun length of data in CDR format.

                //member functions from CloningInterface and prototype pattern
                CloningInterface* clone() const; //clone an object
                static PortPacket* prototype() // return a prototype
                {
                        return _pPrototype_;
                }

                //member functions from DeepCopyingInterface
                bool deepCopy(DeepCopyingInterface* pThing);

        private:
                //atributes
                int _integer_;
                bool _boolean_;
                ExampleClass _exampleClassObj_;

                //static atributes for prototype pattern
                static Packet *_pPrototype_;

                //functions for static initializations
                static void _staticInitialization_()
                {
                        _pPrototype_=new Packet();
                }

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

                // StaticInit<Packet> carries out
                // initialization of static variables
                friend class StaticInit<Packet>;
};
```

Una definición concreta para la función *deepCopy* podría ser la siguiente:

```
bool deepCopy(DeepCopyingInterface* pThing)
{
        if(!pThing) return false;

        Packet* pPacket=static_cast<Packet*>(pThing);
        if(this!=pPacket)
        {
                 _assign_(*pPacket);
                _integer_=pPacket->_integer_;
                _boolean_=pPacket->_boolean_;
                //supposing ExampleClass has overloaded operator =
                _exampleClassObj_=pPacket->_exampleClassObj_;
        }

        return true;
}
```

A.2.4. DebuggingInterface y NamingInterface

De cara a obtener una fácil depuración y seguimiento de los paquetes, se definen las funciones *debug* (con las sobrecargas de los operadores << y >>) y *name*. Para ello la clase hereda las interfaces *DebuggingInterface* y *NamingInterface*,(apéndice C.3.2 y C.3.1).

```
#include "ports/coolbot_portpacket.h"
#include "network/coolbot_packinginterfacewrapper.h"

class Packet:  public PortPacket
{
        public:
                Packet(){}

                //copy constructor
                Packet(const Packet& packet);

                //member functions from PackingInterface
                bool toBytes(ACE_OutputCDR &oCDR); //marshalling data Packet
                bool fromBytes(ACE_InputCDR &iCDR); //demarshalling data Packet
                int packingMaxLength(); //maximun length of data in CDR format.

                //member functions from CloningInterface and prototype pattern
                CloningInterface* clone() const; //clone an object
                static PortPacket* prototype() // return a prototype
```

```
                {
                        return _pPrototype_;
                }

                //member function from DeepCopyingInterface
                bool deepCopy(DeepCopyingInterface* pThing);

                //member function from DebuggingInterface
                ostream& debug(ostream& os) const;

                //member function from NamingInterface
                const char* name()
                {
                        return "CoolBOT::OwnNameSpace::Packet";
                }

        private:
                //atributes
                int _integer_;
                bool _boolean_;
                ExampleClass _exampleClassObj_;

                //static atributes for prototype pattern
                static Packet *_pPrototype_;

                //functions for static initializations
                static void _staticInitialization_()
                {
                        _pPrototype_=new Packet();
                }

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

                // StaticInit<Packet> carries out
                // initialization of static variables
                friend class StaticInit<Packet>;
};
```

Una posible definición de la función *debug*, encargada de imprimir los atributos del paquete, es la siguiente:

```
ostream& debug(ostream& os) const  //printing packet for debugging
{
        os << "CoolBOT::OwnNameSpace::Packet" << endl;
        os << "Integer = "<< _integer_ <<endl;
        os << "Boolean = "<< _boolean_ <<endl;
        //supposing ExampleClass has overloaded operator "<<"
        //or inherits from Debugginginterface
        os << "ExampleClassObject = "<< _exampleClassObj_ <<endl;
        return os;
}
```

Bastará definir la función *debug* para imprimir la clase creada, ya que los operadores << y >> ya se encuentran sobrecargados y llaman a la función *debug* correspondiente en cada caso (ver apéndice C.3.2).

A.3. Guía de creación de nuevos mensajes DC3P

En esta guía se presentan los pasos a seguir y las modificaciones de código necesarias para ampliar los mensajes del protocolo *DC3P*. Se expondrán los cambios a realizar desde alto nivel, para ir descendiendo en la jerarquía de clases que componen los formatos de mensajes.

Los ficheros que contienen las clases que implementan los paquetes se ubican en el directorio *network* de *CoolBOT*. Estos ficheros y las clases que contienen son los siguientes:

- **coolbot_dc3pcommand(.h y .cpp)**

  Clases:

  PacketHeader

  Dc3pCommand.

- **coolbot_commandbody(.h y .cpp)**

  Clases:

  PacketBody

- **coolbot_portinfo(.h y .cpp)**

  Clases:

  PortInfoRequestSingle

  PortInfoRequestMulti

  PortInfoResponseSingle

  PortInfoResponseMulti

- **coolbot_connectdisconnect(.h y .cpp)**

  Clases:

  ConnectDisconnectSingleSingle

  ConnectDisconnectSingleMulti

  ConnectDisconnectMultiSingle

  ConnectDisconnectMultiMulti

- **coolbot_data(.h y .cpp)**

  Clases:

  DataSingle

  DataMulti

En el capítulo 5.2 se presenta la especificación del protocolo y en el capítulo B.2 una descripción detallada del diseño del mismo.

A.3.1. Modificaciones en la cabecera

La cabecera de los paquetes del protocolo *DC3P* está implementada con la clase *PacketHeader*. Dicha clase está definida dentro de la clase *Dc3pCommand* que implementa toda la variedad de paquetes del protocolo.

La clase *PacketHeader* tiene un enumerado que define un valor para el campo *CommandType* de la cabecera. Este campo identifica el tipo de mensaje que contiene el paquete *Dc3pCommand*. La creación de un nuevo mensaje supondrá identificarlo apropiadamente en la cabecera de los comandos, creando para ellos una nueva macro en el enumerado. este enumerado se encuentra estructurado en dos partes, un conjunto de valores para aquellos paquetes del protocolo que tienen definido un cuerpo de mensaje, y otro conjunto para aquellos que sólo se componen de una cabecera. Los límites de estos grupos están marcados por los valores *MAX_COMMANDS_WITH_BODY* y *MAX_COMMANDS_DEFAULT*. Según la estructura que tenga el nuevo paquete que se desea crear se insertará la macro identificativa en un grupo u otro.

```
 enum
{
                                  // Packets with body fields.
        CONNECT_SINGLE_SINGLE,
        CONNECT_SINGLE_MULTI,
        CONNECT_MULTI_SINGLE,
        CONNECT_MULTI_MULTI,
        DISCONNECT_SINGLE_SINGLE,
        DISCONNECT_SINGLE_MULTI,
        DISCONNECT_MULTI_SINGLE,
        DISCONNECT_MULTI_MULTI,
```

```
        PORTINFO_REQUEST_SINGLE,
        PORTINFO_REQUEST_MULTI,
        PORTINFO_RESPONSE_SINGLE,
        PORTINFO_RESPONSE_MULTI,
        DATA_SINGLE,
        DATA_MULTI,

        MAX_COMMANDS_WITH_BODY,

        ECHO_REQUEST,           // Packets with no body fields.
        ECHO_RESPONSE,
        ACK,
        REJECT,
        MAX_COMMANDS_DEFAULT
};
```

En el caso de que el nuevo paquete tenga un cuerpo de mensaje se creará una clase concreta para el mismo. Las características que debe tener dicha clase se detalla en la siguiente sección.

A.3.2. Creación del cuerpo del mensaje

Añadir un paquete nuevo al protocolo que tenga en su estructura campos en el cuerpo, supone definir una nueva clase para tal fin. Esta clase heredará de forma pública de la clase *PacketBody*, lo que permitirá manejar y crear adecuadamente el cuerpo desde la clase *Dc3pCommand*, ya que se dispondrá de las interfaces necesarias para tal fin: *CloningInterface, DeepCopyingInterface, PackingInterface, NamingInterface, DebuggingInterface* y de la implementación del patrón de diseño *prototype*. Por tanto, el esqueleto inicial de toda clase que define un cuerpo concreto es el siguiente:

Fichero coolbot_newbody.h:

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

namespace CoolBOT
{
        class NewBody: public PacketBody
        {
                public:
                        NewBody();
                        ~NewBody();

                        //CloningInterface
                        CloningInterface* clone() const;

                        //DeepCopyingInterface
```

```
                        bool deepCopy(DeepCopyingInterface* pThing);

                        //PackingInterface
                        bool toBytes(ACE_OutputCDR &oCDR);
                        bool fromBytes(ACE_InputCDR &iCDR);
                        int packingMaxLength();

                        //DebuggingInterface
                        ostream& debug(ostream& os) const;

                        //NamingInterface
                        const char* name();

                        //prototype pattern
                        static PacketBody* prototype() { return _pPrototype_; }

                        private:
                                static NewBody *_pPrototype_;

        };
}
```

Fichero coolbot_newbody.cpp:

```
#include "network/coolbot_newbody.h"

namespace CoolBOT
{
        NewBody* NewBody::_pPrototype_=NULL;
}
```

A partir de este esqueleto se definirán las funciones miembro heredadas de cada interfaz (ver apéndice C.3) y las funciones miembro propias de la clase así como sus atributos.

A.3.3. Añadir el nuevo cuerpo al conjunto de paquetes *DC3P*

Finalmente se procederá a añadir el nuevo cuerpo creado al conjunto de paquetes del protocolo. Para que la clase *Dc3pCommand* incluya el nuevo mensaje bastará con que disponga del prototipo del nuevo cuerpo. Para ello será necesario que se añada en el vector *ppPrototypes* declarado en la clase *Dc3pCommand*. esto se llevará a cabo en la inicialización del atributo *ppPrototypes*, código que se encuentra en el fichero *coolbot_dc3pcommand.cpp*. El prototipo del nuevo cuerpo definido se ubicará en el vector en la posición indicada por la macro correspondiente del enumerado de la clase *PacketHeader*, que actúa como índice de acceso al vector. En dicha posición sólo será necesario hacer una llamada a la función *prototype* de la clase que define el cuerpo del nuevo mensaje:

```
PacketBody* Dc3pCommand::ppPrototypes[PacketHeader::MAX_COMMANDS_WITH_BODY]={
                        ConnectDisconnectSingleSingle::prototype(),
                        ConnectDisconnectSingleMulti::prototype(),
                        ...
                        NewBody::prototype(),
                        ...
                        };
```

Apéndice B

Detalles de diseño

B.1. 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.

B.1.1. 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 algunos pocos.

  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 B.1.

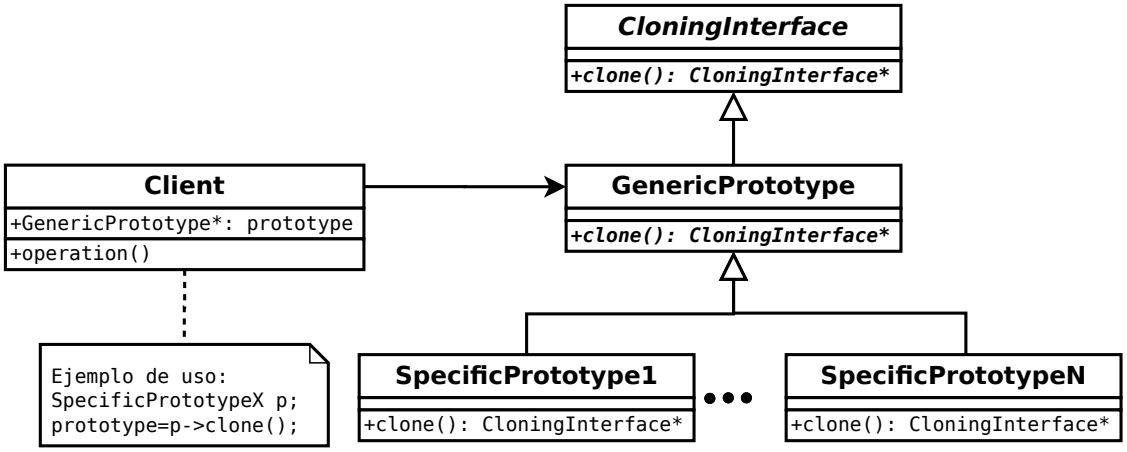

Figura B.1: Diagrama de clases del patrón *prototype*

B.1.2. 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 B.2. 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*.

B.2. Diseño del protocolo *DC3P*

En esta sección se presenta el diseño para la implementación del protocolo *DC3P* que se ha realizado en este proyecto.

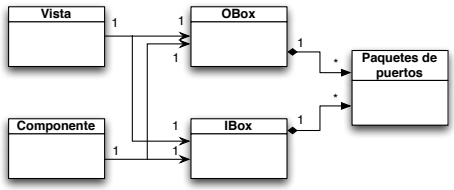

Figura B.2: Diagrama de clases del patrón *MVC*

Como se detalla en el apartado 5.2.4, los formatos de paquetes *DC3P* se componen de una cabecera de longitud fija (8 bytes) y un cuerpo. El cuerpo varía entre los distintos tipos de paquetes. En el diseño de la estructura de paquetes del protocolo se ha aplicado el patrón *prototype* [Gamma,2003], que permite manejar de forma polimórfica los distintos tipos de cuerpo de los paquetes del protocolo a través de una instancia prototípica (patrón explicado en mayor detalle en el apéndice B.1).

El diagrama de clases de la figuraB.3 muestra en detalle el diseño realizado para la implementación de los cuerpos de los paquetes *DC3P*. Como se observa, los paquetes tipo *Echo*, tanto *Request* como *Response* y los paquetes *Ack/Reject*, no tienen una clase definiendo su cuerpo ya que se trata de paquetes que envían exclusivamente una cabecera.

El diagrama de la figura B.4 refleja la clase que define los mensajes *DC3P* que se componen de cabecera y cuerpo. Cabe destacar que la clase *Dc3pCommand* maneja un vector de los prototipos de cuerpos posibles, de forma que, en base al valor del atributo *packetType* de la cabecera (que internamente se maneja como índice del vector *ppPrototypes*), el atributo *body* toma la forma adecuada de la clase derivada que corresponda.

En ambos diagramas (B.3 y B.4) aparecen una serie de clases abstractas de las que heredan tanto las clases *PacketBody* y *PacketHeader*, como la clase *Dc3pCommand*. Estas clases definen una serie de interfaces que deben definir sus clases hijas. Para más detalle ver el apéndice C.3

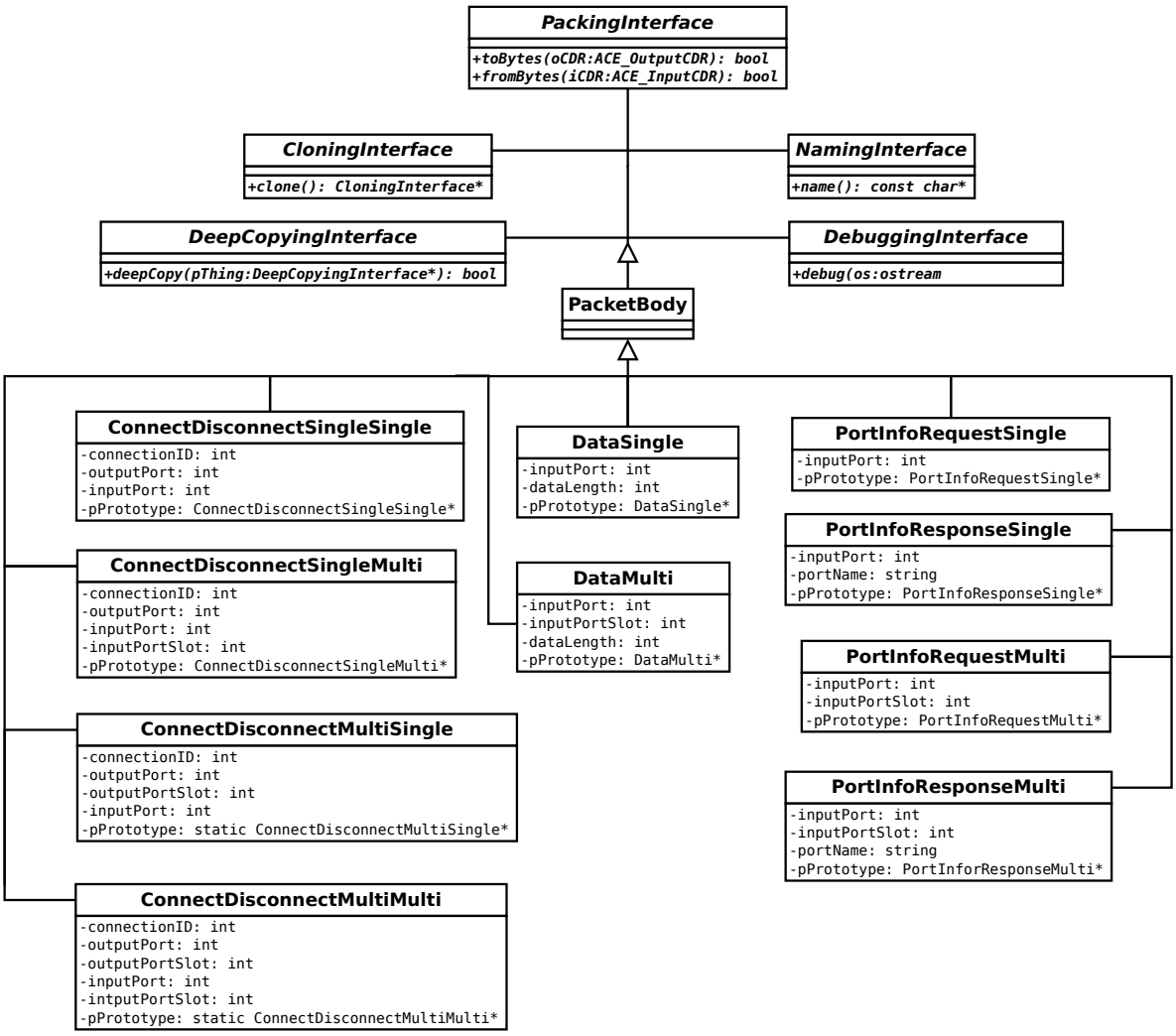

Figura B.3: Diagrama de clases de paquetes DC3P

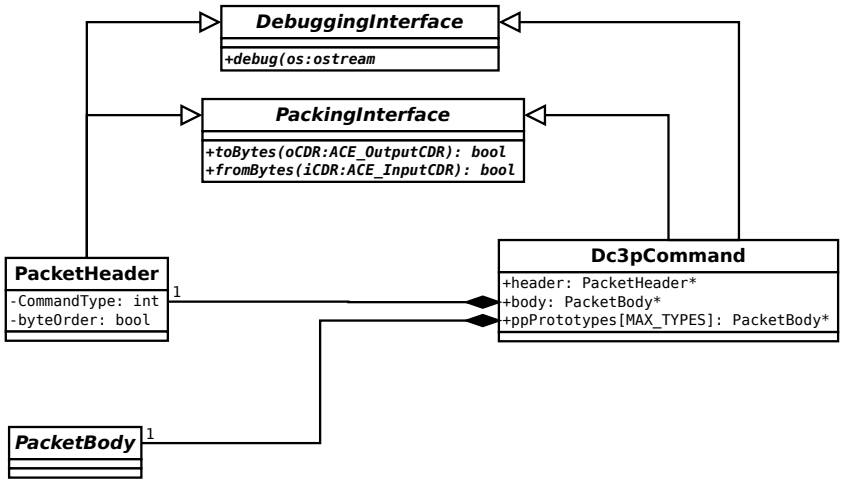

Figura B.4: Diagrama de clases de paquetes *DC3P*

Apéndice C

Detalles técnicos sobre la implementación del proyecto

C.1. Encapsulado de operaciones de *Marshalling: toBytes y fromBytes*

Las operaciones específicas suministradas por la librería *ACE* para realizar el *marshall* y *unmarshall* de datos a un *CDR Stream* (sección 4.4.3), se han suministrado al usuario de *CoolBOT* encapsuladas en las funciones *toBytes* (*marshalling*) y *fromBytes* (*demarshalling*). Estas funciones, suministradas en el espacio de nombres *CoolBOT*, se han implementado con funciones *template*, sobrecargándose para los tipos de datos primitivos y llamando a la correspondiente función, suministrada en la librería *ACE*, para cada tipo concreto. Es esta última llamada la que realiza el *marshalling* a un *ACE_OutputCDR* y el *demarshalling* desde un *ACE_InputCDR*. La programación genérica, facilitada en C++ con los *template* [Vandevoorde,2003], nos permite crear familias de funciones, funciones que además no son generadas por el compilador hasta que no exista un uso de la misma (instanciación del *template*), con lo que los tiempos de compilación de componentes *CoolBOT* se optimizan.

Las funciones template *toBytes* y *fromBytes* son las mostradas a continuación:

```
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 podemos ver, para cualquier tipo generico T se llama a las funciones *toBytes* y *fromBytes* propias del tipo. Estas dos versiones genéricas sólo se instanciarán cuando el usuario pretenda empaquetar un tipo construido, para el que se habrán especificado sus funciones concretas.

Para los tipos básicos se suministran una serie de sobrecargas de estas funciones template, especializándolas para cada tipo primitivo. Algunos ejemplos de estas sobrecargas son los mostrados a continuación:

- **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();
}
```

- **Int**

```
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();
}
```

- **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();
}
```

Se observa en este último caso el uso de una variable intermedia a la que se asigna el parámetro de entrada antes de llamar a la función de empaquetado *write_char(c)*, o que se pasa a la función de desempaquetado *read_char(c)*. Esto es debido a que al llamar a dichas funciones no se produce una conversión implícita entre el tipo *unsigned char* y el tipo *ACE_CDR::Char*. Esto sucede también para los tipos primitivos *long*, *unsigned long* y *long double*, resolviéndose de forma similar en cada una de las especializaciones.

Este mecanismo de sobrecarga se ha relizado con todos los modificadores y tipos primitivos, de forma que el usuario dispone de estas especializaciones para implementar el empaquetado de sus tipos construidos (más detalle en C.3.5).

Con el objetivo de proporcionar suficientes mecanismos que faciliten la labor de programación al usuario, existen dos funciones template genéricas para el empaquetado de vectores:

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

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

Estas funciones están sobrecargadas igualmente para vectores de tipos primitivos. A continuación se presentan algunos ejemplos:

- **Unsigned Short**

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

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

- **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();
}
```

Para la sobrecarga del empaquetado con vectores de *long*, se ha utilizado una llamada al *toBytes/fromBytes* de un *long*, por cada elemento del vector. Podría, sin embargo, haberse realizado en una única llamada a la función correspondiente de *ACE*: *oCDR.write_long_array(array,arrayLength)* y *iCDR.read_long_array(array,arrayLength)*, de forma similar a la especialización de empaquetado de vectores de *Unsigned Short*. La explicación a esta decisión de implementación es que no se proporciona una conversión explícita para tipos tanto *long* como *unsigned long* o *long double* al tipo *ACE_CDR::Long*, que recibe por parámetro las funciones de la librería *ACE*.

C.2. 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 4.4.1). *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 tribial, 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 C.1), 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 función concreta A.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);
}
```

C.3. Definición de Interfaces

En esta sección se detallan las clases abstractas que se han diseñado definiendo una serie de interfaces. A continuación se describe la estructura de cada una de ellas, así como los objetivos y la motivación del diseño de las mismas.

C.3.1. *NamingInterface*

El objetivo de esta clase es poder asignar un nombre para cualquier clase. Esto es muy útil para poder identificar tipos en tiempo de ejecución, ya que obtner un nombre de un tipo puede depender de los detalles de implementación de cada compilador. Particularmente esta interfaz se

usará para verificar tipos de paquetes de puertos cuando se establecen conexiones de red entre componentes.

A continuación se presenta la definición de la interfaz:

```
class NamingInterface
{
     public:
          virtual ~NamingInterface() {}
          virtual const char* name() = 0;
};
```

Se trata de una clase abstracta que define una operación virtual pura *name()*. Cada clase hija definirá esta operación adecuadamente de forma que retorne un nombre que identifique el tipo de dicha clase, por ejemplo de la siguiente forma:

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

C.3.2. *DebuggingInterface*

Esta interfaz permite que cualquier clase pueda imprimir sus atributos. Es de gran utilidad de cara a depurar los datos que se transforman a la representación *CDR*, verificando en cada momento si se ha hecho el *(de)marshalling* de forma correcta.

La definición de la clase es la siguiente:

```
class DebuggingInterface
{
     public:
          virtual ~DebuggingInterface() {}
          virtual ostream& debug(ostream& os) const = 0;
};

inline ostream& operator<<(ostream& os,const DebuggingInterface& debuggingObject)
{
     return debuggingObject.debug(os);
}

inline ostream& operator<<(ostream& os,const DebuggingInterface* pDebuggingObject)
{
     return ( (pDebuggingObject)?
               pDebuggingObject->debug(os):
               os << "CoolBOT::DebuggingInterface: unknown (null pointer)" );
}
```

Como podemos ver cualquier clase que herede esta interfaz tendrá sobrecargados los operadores << y >>. Esto permite la impresión de datos de la siguiente forma:

```
cout << objeto << endl;
```

C.3.3. *DeepCopyingInterface*

Se trata de una interfaz para permitir a cualquier clase ser copiada adecuadamente. esto facilita que toda clase qy herede esta interfaz tenga una función definida que permita duplicar objetos.

Las definición de la clase tiene la siguiente forma:

```
class DeepCopyingInterface
{
     public:
          virtual ~DeepCopyingInterface() {}
          virtual bool deepCopy(DeepCopyingInterface* pThing) = 0;
};
```

Una definición concreta de la operación *deepCopy* podría ser la siguiente en una clase hipotética *ExampleClass*:

```
class ExampleClass: public DeepCopyingInterface
{
     public:
          ExampleClass() {}
          ExampleClass(ExampleClass& objt) { deepCopy(&objt); }

          bool deepCopy(DeepCopyingInterface* pThing)
          {
               if(!pThing) return false;
               ExampleClass *pClass=static_cast<ExampleClass*>(pThing);
               if(this!=pClass) _i_=pClass->_i_;
               return true;
          }

     private:
          int _i_;
};
```

Como se observa la clase *ExampleClass* utiliza la función *deepCopy* en la sobrecarga del constructor que crea un objeto a partir de otro.

C.3.4. *CloningInterface*

Esta clase abstracta permite a una clase ser clonada. El uso de esta interfaz otorga a una clase la posibilidad de ser un protocipo, siguiendo el patrón de diseño *prototype*.

La clase es la mostrad a continuación:

```
class CloningInterface
{
     public:
          virtual ~CloningInterface() {}
          virtual CloningInterface* clone() const = 0;
};
```

C.3.5. *PackingInterface*

La interfaz *PackingInterface* es una clase abstracta que suministra una serie de métodos prototipo para el empaquetado de datos construidos. Cualquier clase definida por el usuario, cuyas instancias (objetos) sean susceptibles de ser enviadas por la red, heredarán esta interfaz, obligando así a definir las funciones de empaquetado: *toBytes* y *fromBytes*; y la función de cálculo de longitud en bytes del *Stream* de empaquetado: *packingMaxLength*. Cabe aclarar que los atributos de una clase, cuyas instancias quieran enviarse a través de la red, que sean a su vez tipos construidos, deberán disponer igualmente de esta interfaz de empaquetado.

Con tipos construidos como los *struct* del lenguaje *C* no es aplicable la herencia, con lo que el usuario debe definir obligatoriamente las funciones *toBytes*, *fromBytes* y *packingMaxLength* para estos tipos, respetando el mismo prototipado que el suministrado por la clase *PackingInterface*.

El siguiente código muestra la implementación de la clase abstracta *PackingInterface*:

```
class PackingInterface
{
     public:
          virtual ~PackingInterface() {}
          virtual bool toBytes(ACE_OutputCDR &oCDR) = 0;
          virtual bool fromBytes(ACE_InputCDR &iCDR) = 0;
          virtual int packingMaxLength()=0;
};
```

C.3.6. *PortPacket*

Se trata de una interfaz que unifica e impone a sus clase hijas una serie de interfaces necesarias en los paquetes de puertos de componentes, de ahí su nombre.

La interfaz es la siguiente:

```
class PortPacket: public CloningInterface,
                  public DeepCopyingInterface,
```

```
                public PackingInterface,
                public NamingInterface,
                public DebuggingInterface
{
              // Nothing
};
```

Bibliografía

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

[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

[Coulouris,2001] Coulouris, G. *Sistemas Distribuidos*. Tercera Edición. Addison Wesley. 2001.

[CVS] CVS documentation *http://www.nongnu.org/cvs/*.

[DIA] DIA *http://live.gnome.org/Dia*.

[Domínguez,2003] Domínguez-Brito, A. C. *Tesis doctoral, CoolBOT: un marco de programación orientado a componentes para robótica*. Dpto. Informática y Sistemas, Universidad de Las Palmas de Gran Canaria, 2003.

[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.

[Gamma,2003] Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John *Patrones de diseño: elementos de software orientado a objetos reutilizable*. Addison Wesley, 2003.

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

[GIT] Git, the fast version control system *http://git-scm.com/*.

[Goldberg,2002] Goldberg, K.; Siegwart, R. *Beyond webcams: an introduction to online robots*. The MIT Press, 2002.

[GTK+ Project] The GTK+ project *http://www.gtk.org/*.

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

[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.

[InkScape] InkScape *http://www.inkscape.org/?lang=es*.

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

[Koren,1991] Koren, Y.; Borenstein, J. *Potential field methods and their inherent limitations for mobilerobot navigation*. Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on. pp. 1398–1404.

[Larman,2003] Larman, Craig. *UML y patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado*. Pearson, Prentice Hall, 2003.

[Latex] Latex, a document preparation system. *http://www.latex-project.org/*.

[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.

[Murphy,2000] Murphy R. Introduction to AI Robotics MIT Press, 2000.

[NET++] Middleware Net.h++ *http://www.quickerwit.com/links/113514.htm*.

[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).

[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.

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

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

[Steenstrup83-jcss,1983] Steenstrup, M. and Arbib, M. A. and Manes, E. G. *Port Automata and the Algebra of Concurrent Processes*. Journal of Computer and System Sciences, 1983, vol 27, pag 29-50

[Stewart97-ieee-tse,1997] Stewart, D. B. and Volpe, R. A. and Khosla, P. *Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects*. IEEE Transactions on Software Engineering, December, 1997, vol 23,num 12, pag 759-776

[Szyperski,1999] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1999.

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

[SOCK++] Librería Socket++ *http://www.cs.utexas.edu/users/lavender/courses/socket++/*.

[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.

[RFCs XDR] *XDR: External Data Representation Standard RFC 4506*.

[Domínguez,2007] Antonio C. Domínguez-Brito, Daniel Hernández-Sosa, José Isern-González, Jorge Cabrera-Gámez *CoolBOT: a Component Model and Software Infrastructure for Robotics*. Springer Tracts in Advanced Robotics Series, Vol. 30, pp. 143-168, Brugali, Davide (Ed.), 490 p., ISBN: 978-3-540-68949-2. April, 2007.