Universidad
Carlos III de Madrid

DEPARTAMENTO DE INGENIERÍA DE SISTEMAS Y AUTOMÁTICA

TESIS DE MÁSTER

COORDINADOR DE TAREAS GENÉRICO BASADO EN LA ARQUITECTURA MIPS PARA ROBÓTICA

Autor: Miguel Angel Maldonado Agramonte
Director: Miguel González-Fierro Palacios
Director: Juan González Víctores

MÁSTER OFICIAL EN
ROBÓTICA Y AUTOMATIZACIÓN

LEGANÉS, MADRID
JULIO 2011

UNIVERSIDAD CARLOS III DE MADRID
MASTER OFICIAL EN ROBÓTICA Y AUTOMATIZACIÓN

El tribunal aprueba la tesis de Master titulada "**COORDINADOR DE TAREAS GENÉRICO BASADO EN LA ARQUITECTURA MIPS PARA ROBÓTICA** " realizado por **Miguel Angel Maldonado Agramonte**.

Fecha: Julio 2011

Tribunal: _____

_____

_____

For where your treasure is, there your heart will

Porque donde esté vuestro tesoro, allí estará vuestro corazón

A mis padres

Índice general

Índice de figuras

Agradecimientos

Quisiera agradecer a toda la gente que me ha ayudado y apoyado en todo este tiempo.

En primer lugar quiero dar las gracias a mi padre por apoyarme. A Paola por su paciencia a la hora de leer la tesis, ¡¡¡Gracias por tu ayuda!!!. A mi hermano que me ha apoyado incondicionalmente en todo aunque se ha perdido ya en lo que estoy haciendo.

A Miguel y a Juan, por haberme orientado y ayudado a la hora de realizar este proyecto. Gracias por vuestro tiempo y conocimientos. Y en especial, vuestra paciencia.

A Justyna, que ha conseguido darme ánimos para acabarlo, aun estando tan lejos de mi hogar. Gracias por estar a mi lado y ayudar a encontrarme.

A mis amigos, simplemente por estar ahí cuando los necesitaba sin importarles la hora y ni el lugar.

Gracias a todos. Un abrazo.

Miguel

Resumen

Este documento propone utilizar el esquema de funcionamiento de la arquitectura MIPS como base para el desarrollo de un coordinador de tareas genérico para robótica.

La idea de este proyecto es crear un componente software multiplataforma, adaptable a cualquier arquitectura hardware, capaz de gestionar los recursos (hardware y software) y las tareas (donde llamaremos tarea a todo procedimiento ejecutable y planificable) abordando la problemática de integrar, coordinar y planificar distintas aplicaciones que se ejecuten dentro del robot.

Además se propone un planificador de tareas de bajo nivel o *scheduler* encargado de gestionar las sub-tareas generadas por el planificador de alto nivel. Este planificador gestionará los recursos hardware (sensores, actuadores, etc) y software (generador de trayectorias, rutinas de visión, etc.) para conseguir maximizar el uso de los recursos del robot sin que se desestabilice el sistema, impidiendo, por ejemplo, la ejecución de rutinas contradictorias sobre el mismo motor.

El planificador de tareas del robot tiene la labor de dar una solución a las situaciones con que se encuentre en cada momento; para ello, debe decidir qué tarea se ejecuta en cada instante de tiempo, teniendo en cuenta la prioridad de ejecución y los recursos disponibles (motores, sensores, etc.).

Abstract

This document proposes to use the scheme of operation of the MIPS architecture as the basis for the development of a generic task manager for robotics.

The idea of this project is to create a multi-platform software component, adaptable to any hardware architecture, capable of managing resources (hardware and software) and tasks (considering a task to any procedure that can be executed and planned) to address the problem of integrating, coordinating and planning various applications running inside the robot.

Additionally, a low-level task planner (a.k.a. scheduler) in charge of managing the sub-tasks generated by high-level planner is proposed. This planner will manage the hardware (sensors, actuators, etc.) and software resources (trajectory generator routines, vision, etc.) in order to maximize the use of the resources of the robot without destabilizing the system, preventing, for example, the execution of contradictory routines on a same engine.

The scheduler of the robot has the task of giving a solution to the situations that are in every moment, for it must decide which task is executed at each instant of time, taking into account the priority of execution and the resources available (motors, sensors, etc.).

Capítulo 1

Introducción

El acelerado desarrollo de la robótica de los últimos años ha traído consigo nuevos robots con comportamientos más inteligentes y mayor capacidad de reacción y planificación, actuadores y sensores más avanzados y procesadores más rápidos. Todo ello contribuye a aumentar la complejidad y el rango de aplicación de este tipo de sistemas.

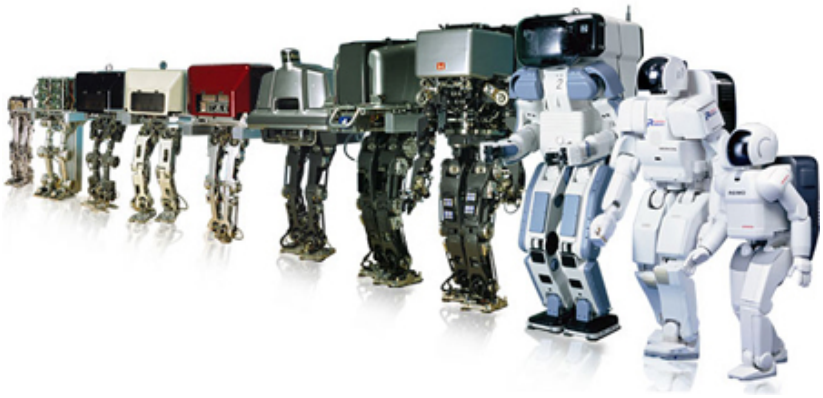


Figura 1.1: *Evolución del robot ASIMO de la compañia Japonesa HONDA*

Por otro lado, muchos de los robots actuales están diseñados para ser un sistema completo que combine muchas ramas de la robótica. Algunos ejemplos se

pueden observar al analizar algunos de los más avanzados humanoides, como los japoneses ASIMO (fig. 1.1) y HRP-2 o el español RH2. Estos robots combinan manipulación, navegación, visión, interacción, generación de trayectorias, control de estabilidad, etc.

Tener capacidad para desarrollar estas plataformas implica dos cosas; en primer lugar es necesario un equipo técnico especializado, experto en las diferentes ramas de la robótica. En segundo lugar, es necesario personal especializado en integración de sistemas software y coordinación de tareas.

1.1. Integración de software en robótica

El software en robótica engloba el sistema operativo, los framework de desarrollo y las aplicaciones. El conjunto de estos componentes, más las medidas de los sensores, determinan el comportamiento del robot, por lo que hay que encontrar el equilibrio entre las dos; no vale de nada tener unos sensores muy eficientes o unas aplicaciones muy precisas, si no hay un sistema que los coordine y los integre. En la actualidad, es uno de los elementos más importantes, y cada vez toma mayor relevancia en el desarrollo del robot por su equipo de investigación.

Se espera que un robot inteligente sea capaz de lograr los objetivos programados por el usuario en entornos dinámicos y complejos. Para lograr esos objetivos, éste debe de llevar a cabo una serie de tareas.

Por lo general, los robot industriales trabajan en entornos controlados y conocidos, con lo que se puede planificar sus objetivos de forma secuencial, por ejemplo: coger pieza, soldar pieza, dejar pieza.

Sin embargo, los robot móviles o los humanoides deben enfrentarse a un mundo, por lo general, no controlado y dinámico: hay personas que se cruzan en su camino, los objetivos no son siempre los mismos, etc. Por lo que deben de ser capaces de amoldarse a cada situación y saber cual es el objetivo mas prioritario en cada momento.

1.2. Evolución de los sistemas software en robótica

El software de robótica ha sufrido una constante evolución. Se comenzó con el desarrollo de aplicaciones específicas para cada robot debido a que no se producían en serie. En estos casos el desarrollo se realizaba a bajo nivel, comunicándose directamente con los drivers del robot (fig. 1.2a). Según fueron creciendo en popularidad e implantándose en la industria, cada compañía desarrollaba su propio lenguaje (RAPID de ABB ó V+ de Adept Technology)(Antonio Barrientos, 1997).

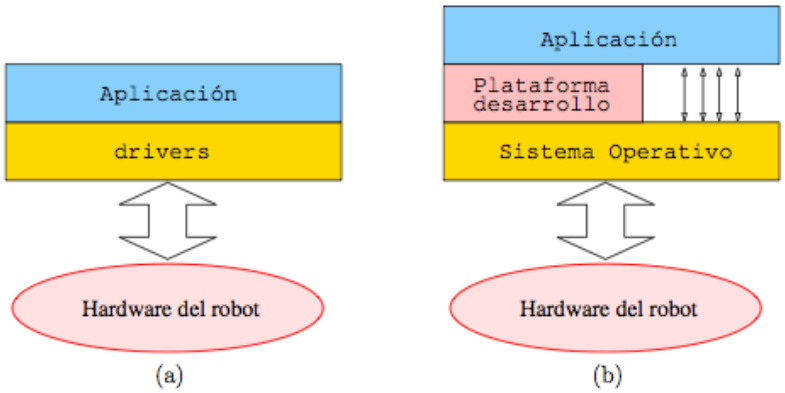


Figura 1.2: *Programación de robots: (a) sobre drivers específicos de sensores y actuadores. (b) sobre una plataforma de desarrollo*

Con el asentamiento de los fabricantes y el trabajo de muchos grupos de investigación, han ido apareciendo plataformas de desarrollo que simplifican la programación de aplicaciones robóticas. Estas plataformas ofrecen acceso más sencillo a sensores y actuadores, suelen incluir un modelo de programación que establece una determinada organización del software y permite manejar la creciente complejidad del código cuando se incrementa la funcionalidad del robot. El diseñador programa sus aplicaciones robóticas finales sobre esa plataforma de desarrollo (fig. 1.2b).

Pero aun así, es difícil la portabilidad de aplicaciones de una plataforma a otra, debido a la falta de estándares, tanto en las plataformas de desarrollo como en los lenguajes de programación utilizados (Canas, Matellán, y Montúfar, 2006).

1.3. Motivaciones

Partimos de la premisa de que el hardware a utilizar cuenta con una cantidad limitada de recursos, por lo que una buena administración de ellos permite ampliar las posibilidades de nuestro robot, además de limitar errores y esperas innecesarias. Dichos recursos no tienen por qué ser sólo hardware, ya que se pueden encontrar rutinas de funcionamiento en un robot que consumen gran cantidad de recursos, tales como memoria o tiempo de procesador.

Es necesario contar con una capa que controle las tareas en ejecución y los recursos disponibles en cada momento. Con ello se pretende limitar la sobrecarga del procesador con tareas contradictorias o innecesarias al mismo tiempo, ya que dichas tareas se podrían realizar de forma secuencial, no obligando a las tareas con mayor relevancia a tener que luchar por tiempos de ciclo.

1.4. Objetivos

La idea de este proyecto es crear un componente software multiplataforma, adaptable a cualquier arquitectura hardware, capaz de gestionar los recursos (hardware y software) y las tareas (donde llamaremos tarea a todo procedimiento ejecutable y planificable) abordando la problemática de integrar, coordinar y planificar distintas aplicaciones que se ejecuten dentro del robot.

Además se propone un planificador de tareas de bajo nivel o *scheduler* encargado de gestionar las sub-tareas generadas por el planificador de alto nivel. Este planificador gestionará los recursos hardware (sensores, actuadores, etc) y

software (generador de trayectorias, rutinas de visión, etc.) para conseguir maximizar el uso de los recursos del robot sin que se desestabilice el sistema, impidiendo, por ejemplo, la ejecución de rutinas contradictorias sobre el mismo motor.

El planificador de tareas del robot tiene la labor de dar una solución a las situaciones con que se encuentre en cada momento; para ello, debe decidir qué tarea se ejecuta en cada instante de tiempo, teniendo en cuenta la prioridad de ejecución y los recursos disponibles (motores, sensores, etc.).

Para ello se han tenido en cuenta las siguientes premisas:

- Independencia de la Morfología.
- Estrategias del planificador modificables.
- Posibilidad de cambiar el planificador fácilmente.
- Facilidad de integración con el resto del software.
- Facilidad de cambiar de S.O.
- Desarrollo modular
- Facilidad a la hora de cambiar/integrar nuevos módulos.

Estos puntos se irán comentando a lo largo del proyecto poniendo énfasis en la manera de solucionarlo y los problemas encontrados.

Finalmente se va a utilizar como plataforma de pruebas el robot humanoide HOAP3. Se van a realizar diferentes ejemplos de funcionamiento del planificador y una demo, en donde se muestra la fiabilidad del sistema.

1.5. Estructura del documento

El documento se va a organizar de la siguiente manera:

- **Capítulo 2: Estado del arte**
  Estado del Arte de los siguientes temas abordados por el proyecto: arquitecturas software, infraestructuras de comunicaciones, planificación y scheduling.

- **Capítulo 3: MIPS**
  Introducción a la arquitectura MIPS y al algoritmo de scheduling usado por éste.

- **Capítulo 4: Coordinador de tareas**
  En este capítulo se va a describir el Coordinador de tareas y librerias propuestas por este proyecto.

- **Capítulo 5: Pruebas**
  Se van a realizar diferentes ejemplos de funcionamiento del planificador y una demo, en donde se muestra la fiabilidad del sistema.

- **Capítulo 6: Conclusiones y futuras mejoras**
  Conclusiones llegadas tras realizar las pruebas y propuesta de futuras mejoras y usos para la libreria presentada.

Capítulo 2

Estado del arte

En este capítulo, se intentará explorar y dar una perspectiva lo más amplia posible sobre el software desarrollado orientado a la robótica y a la planificación. Debido a la complejidad a la hora de desarrollar aplicaciones en este campo, se relatarán los últimos avances en comunicaciones, arquitecturas software y planificación.

2.1. Infraestructuras de comunicación

En todo software desarrollado, sea orientado a robótica o no, es recomendable utilizar una infraestructura de comunicaciones bien conocida, probada y que de ciertas garantías de uso. De esta forma, podemos conectar componentes software con funcionalidades opuestas para que puedan interactuar en un marco de trabajo común, de una forma rápida y sencilla. Esto se puede realizar de dos formas: *Llamada a Procedimiento Remoto* (En ingles RPC ó Remote Procedure Calls) y *flujo de datos* mediante un protocolo de comunicaciones. RPC es un término genérico para comunicación entre procesos, que normalmente comparten otro espacio de direcciones: en local, en otro ordenador perteneciente a

nuestra red o fuera de ella. Para ello permite que un programa ejecute una subrutina o procedimiento en otro equipo distinto al nuestro, de forma transparente para el usuario dando la apariencia de que se está ejecutando en local. El flujo de datos hace referencia al paso de información entre aplicaciones o capas de la aplicación. Para ello se usan protocolos de comunicaciones, donde un protocolo de comunicaciones es un conjunto de reglas normalizadas para la representación, señalización, autenticación y detección de errores necesarios para enviar información a través de un canal de comunicaciones.

Existen muchas infraestructuras de comunicaciones; vamos a ver algunas de las más importantes y usadas.

2.1.1. ACE

ACE (Schmidt, 1993) (Adaptive Communication Enviroment - Entorno Adaptativo de Comunicaciones). ACE es un *framework* orientado a objetos que implementa una gran cantidad de *patrones* para el uso de comunicaciones concurrentes. Éste provee un gran número de clases envoltorio (wrapper class en inglés) que realizan tareas comunes de comunicación en una gran variedad de sistemas operativos; la clase envoltorio refuerza el uso de la programación orientada a objetos y facilita que las aplicaciones desarrolladas sean multiplataforma. ACE incluye wrapper con funciones para eventos de multiplexación y control de eventos, manejo de señales, inicialización del servicio, comunicación entre procesos, gestión de memoria compartida, enrutamiento de mensajes, configuración y reconfiguración dinámica de los servicios de distribución, ejecución concurrente y sincronización. La versión actual es la 6.0.0.

2.1.2. CORBA

CORBA (Common Object Request Broker Architecture - Arquitectura Común Intermediaria en Peticiones a Objetos); es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. CORBA fue definido y está controlado por el Object Management Group (OMG), formado por Hewlett-Packard, Telefónica, Sun, IBM, American Airlines... .Define las APIs, el protocolo de comunicaciones y los mecanismos necesarios para permitir la interoperabilidad entre diferentes aplicaciones escritas en diferentes lenguajes y ejecutadas en diferentes plataformas, lo que es fundamental en computación distribuida. También incluye seguridad y transacciones.

En un sentido general, CORBA "envuelve" el código escrito en otro lenguaje, en un paquete que contiene información adicional sobre las capacidades del código que contiene y sobre cómo llamar a sus métodos. Los objetos que resultan, pueden entonces ser invocados desde otro programa (u objeto CORBA) desde la red. En este sentido CORBA se puede considerar como un formato de documentación legible por la máquina, similar a un archivo de cabeceras, pero con más información (Bastide, Sy, y Palanque, 1999). CORBA utiliza un lenguaje de definición de interfaces (IDL) para especificar las interfaces con los servicios que los objetos ofrecerán. CORBA puede especificar a partir de este IDL, la interfaz a un lenguaje determinado, describiendo cómo los tipos de dato CORBA deben ser utilizados en las implementaciones del cliente y del servidor. Existen implementaciones para la mayoría de lenguajes actuales. La implementación más popular de C++ incluye The ACE ORB (TAO - una implementación Open Source del protocolo ORB (Object Request Broker) de CORBA en tiempo real basada en ACE) (Schmidt, Levine, y Mungee, 1998). La versión actual es la 3.0

2.1.3. ICE

ICE (Henning, 2004), Internet Communications Engine - Motor de comunicaciones por internet. ICE es un moderno "toolkit" orientado a objetos que permite desarrollar aplicaciones distribuidas con el mínimo esfuerzo. Desarrollado por un coautor de CORBA. ICE fue diseñado con la intención de desarrollar un *middleware* ligero con las mismas cualidades de servicio que CORBA, pero en el contexto de los MMORPG (Massively Multiplayer Online Role-Playing Games - Juego de Rol Online Multijudor Masivo). En la práctica, usa muchos conceptos de CORBA, es multi-lenguaje y multiplataforma. Actualmente está en la versión 3.4.2.

2.1.4. XML-RPC

XML-RPC (Extensible Markup Lenguage - Remote Procedure Call) XML basado en RPC usa HTTP como capa de transporte y XML como codificación. Está diseñado para ser tan simple como sea posible, permite la transmisión de complejas estructuras, procesarlas y retornarlas. Hay más de 80 librerías que implementan XML-RPC sobre diferentes lenguajes de programación y distintos sistemas operativos (Winer y cols., 1999).

2.2. Arquitectura Software

Una Arquitectura Software es un conjunto de terminologías que describen componentes e interacciones que fuerza la interacción entre componentes, mediante la infraestructura de comunicación elegida, y usualmente un subconjunto de herramientas y utilidades (V1'ctores, 2010).

A continuación se presentara una selección de las Arquitecturas Software con más aceptación en robótica.

2.2.1. OpenRTM-aist

OpenRTM-aist (Ando, Suehiro, y Kotoku, 2008) es una implementación basada en CORBA de RTM (Robot Techology Middleware) Standard definida por AIST. Tanto RTM como OpenRTM-aist fueron parcialmente financiadas como parte del programa NEDO "Development of the key enabling technologies for Robotics". Al ser un middleware basado en CORBA, OpenRTM-aist depende de OmniORB, pero también de ACE. Tiene soporte para múltiples plataformas; Linux, FreeBSD and Windows, y tiene implementaciones nativas en C++, Java y Python.

OpenRTM-aist trata cada componente como una *máquina finita de estados* donde RPC, streaming y servicios adicionales son proveídos, como puede verse en la figura2.1.

El desarrollo principal es en Japonés pero hay muchos documentos traducidos al Ingles. OpenRTM-aist es usado en la familia de humanoides HRP. La versión actual es la 1.0.0.

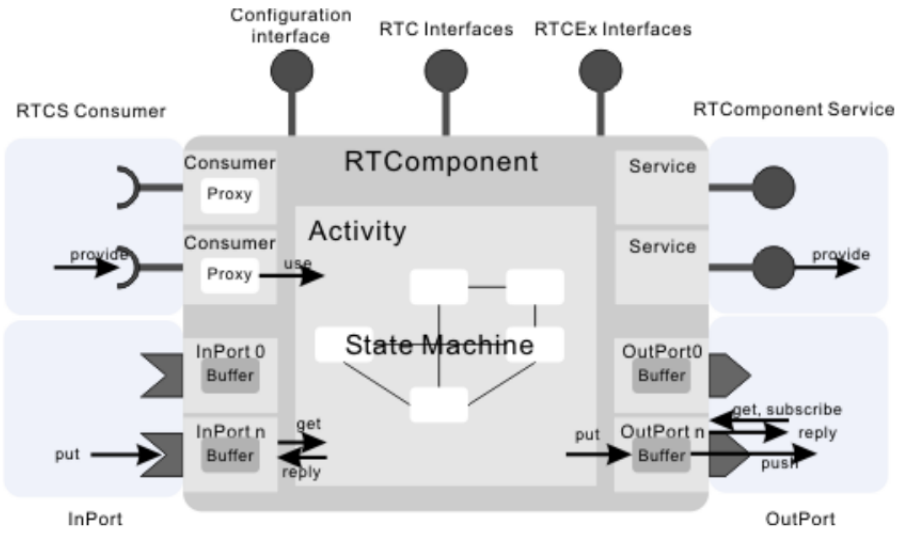


Figura 2.1: *Vista conceptual de un componente OpenRTM-aist*

2.2.2. OROCOS-RTT

OROCOS-RTT (Bruyninckx, 2001) Open Robot Control Software - Real Time Toolkit. OROCOS-RTT se ha desarrollado principalmente en la ACM KU Leuven, y se deriva del Proyecto EURON Orocos. En la actualidad también es apoyado por el FMTC, SourceWorks, y PAL Robotics. Su enfoque principal es el tiempo real duro y la dificultad de la comunicación en tiempo real entre componentes. Se ha utilizado ampliamente en robots industriales. OROCOS-RTT ha logrado una gran integración con Xenomai, una derivación del proyecto Linux en tiempo real. Al igual que en OpenRTM-AIST, cada componente funciona como una máquina de estados finitos, y posee propiedades y métodos. Esto se puede ver en la figura 2.2.

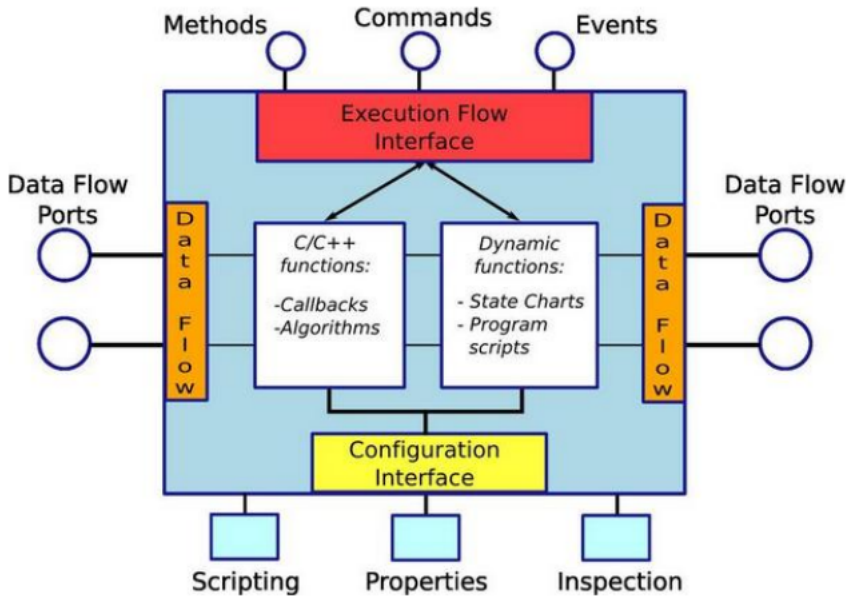


Figura 2.2: *Vista conceptual de un componente OROCOS-RTT*

El estado y propiedades de los componentes puede ser examinado externamente mediante una interfaz de linea de comandos empotrada en cada componente. Las comunicaciones de la version 1.0 estaba basada en la implementación TAO de CORBA. Actualmente la versión 2.4.0 soporta comunicaciones sobre

ROS y YARP. El robot REEM-B de PAL Robotics usa componentes realizados con OROCOS.

2.2.3. ROS

ROS (Quigley y cols., 2009) (Robot Operating System - Sistema operativo para robot). ROS esta basado en el proyecto Switchyard de Stanford, que forma parte del proyecto STAIR (apoyado por DARPA, Google, Intel y Honda entre otros). Actualmente ROS es soportado por la empresa Americana Willow Garage. ROS se describe como un "meta-sistema operativo" por sus creadores, ya que ofrece los servicios propios de un sistema operativo, abstracción del hardware de bajo nivel del dispositivo, aplicación que proveen funcionalidades de uso común, paso de mensaje entre procesos, y de gestión de paquetes. Fácil de usar, ROS tiene dos interfaces nativas en C + + y Python, y una gran cantidad de herramientas de línea de comandos y utilidades gráficas. El apoyo esta limitado sólo a Ubuntu, pero existen versiones experimentales o parcialmente funcionales para otros sistemas operativos. Actualmente sigue en la versión 1.0, ROS tiene un enorme crecimiento de usuarios y de comunidad de desarrolladores. La variedad de robots que se utiliza en la actualidad es amplia, aunque una gran parte de los desarrolladores centra su trabajo en el manipulador móvil PR2 de Willow Garage.

2.2.4. YARP

YARP (Metta, Fitzpatrick, y Natale, 2006) (Yet Another Robot Platform - Otra Plataforma Para Robot). YARP fue inicialmente desarrollado por el MIT y LiraLab (Universidad de Génova) para robótica humanoide, actualmente es apoyada también por el Italian Institute of Technology (IIT).

Caracteristicas:

- YARP es ligero y fácil de usar para los principiantes y con experiencia se puede crear código complejo y eficaz.

- YARP es compatible con Linux, Windows y Mac OS X. Su única dependencia es la librería ACE, que pronto será eliminada.

- La interfaz nativa de YARP fue escrita en C++, aunque hay envolturas creadas para distintos lenguajes (SWIG wrappers), entre los que se encuentran: Python, Java, Tcl, Lisp, Ruby y muchos más.

- Es compatible con una amplia familia de protocolos de conexión: TCP/IP, UDP, multicast, memoria compartida (local), MPI, mjpg-over-HTTP, XML-RPC, TCPROS (un nuevo tipo de conexión nativo desarrollado por ROS) ... que se configuran en tiempo de ejecución, por lo que no es necesario modificar el código para modificar el tipo de conexión.

- YARP incluye herramientas de lineas de comando, utilidades gráficas y basadas en Web.

Se encuentra actualmente en la versión 2.3.5 y tiene un crecimiento constante tanto de usuarios como de desarrolladores. YARP se utiliza en la actualidad en el robot iCub entre otros.

2.2.5. Interoperabilidad

Para facilitar la vida a los grupos de investigación la mayoría de Arquitecturas Software ponen a disposición de los desarrolladores métodos para poder comunicarse con otra Arquitectura Software, en busca de la reutilización de funcionalidades y poder elegir la que mejor se adapte en cada situación. En la figura 2.3 podemos observar un diagrama de interoperabilidad entre las distintas Arquitecturas Software(V1'ctores, 2010).

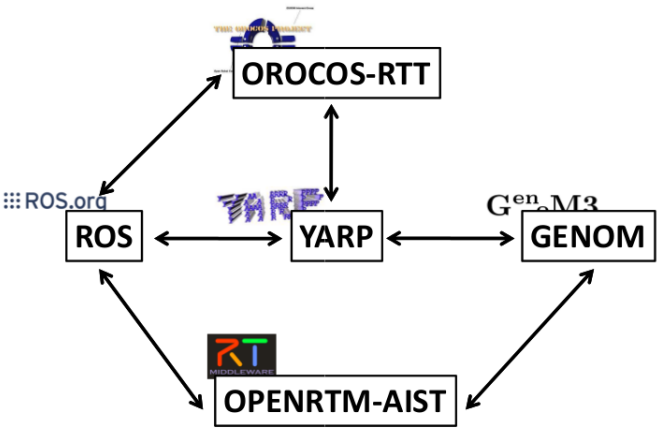


Figura 2.3: *Diagrama de interoperabilidad entre las distintas Arquitecturas Software*

2.3. Planificación

La planificación en IA tiene el objetivo de, dado cualquier problema expresado como una situación o estado inicial y unas metas a conseguir, obtener un conjunto de acciones, denominado plan, que mediante su ejecución permitan llegar desde el estado inicial a otro estado en que todas las metas del problema se satisfacen (Arregui, 2006). Un problema básico en robótica es la planificación de movimiento para resolver alguna tarea, y luego controlar al robot cuando ejecuta las órdenes necesarias para conseguir esas acciones. Por lo que planificación significa decidir un curso de acción antes de actuar. Esta parte de síntesis de acción del problema del robot se puede lograr mediante un sistema de resolución de problemas que logrará algún objetivo marcado, dada alguna situación inicial. Para ello se tiene que apoyar en la información que tiene el robot de su propio estado y de su entorno, aportado a partir de los sensores disponibles, donde esta información es facilitada normalmente de una forma comprensible para el robot gracias a una de las Arquitecturas Software presentadas en la sección anterior.

A la hora de realizar planificación de tareas, hay que enfrentarse a varios problemas que no tienen una única solución: suposiciones sobre la naturaleza

de las acciones y problemas, la presencia de sensores que realimentan los sistemas con sus mediciones,... han dado lugar a varias formas de planificación. La planificación clásica se centra en problemas cuya solución es un conjunto de acciones deterministas y se tiene una información completa del problema. La planificación con incertidumbre resuelve problemas en donde no se tiene una información completa del problema y las acciones no son deterministas. Dentro de ella, se distingue la denominada planificación conformante (Smith y Weld, 1998) en que no es posible utilizar sensores para medir el estado actual. La planificación con tiempo y recursos o "scheduling" que trata problemas en los que las acciones tienen duración y utilizan recursos concurrentemente (Arregui, 2006).

- **Planificador clásico:**
  Los modelos matemáticos que subyacen a la planificación clásica son los llamados "modelos de estados" en que las acciones, de forma determinista, transforman un estado en otro y la tarea de planificación consiste en encontrar la secuencia de acciones que pasen del estado inicial a un estado en que están presentes las metas del problema. Como ejemplo de planificadores clásicos tenemos algoritmo STRIPS (fig. 2.4) (Fikes y Nilsson, 1971), planificación de orden parcial (Barrettand Daniel y cols., 1994), planificación basada en grafos de planes (Blum y Furst, 1997) y planificación con búsqueda heurística (Bonet y Geffner, 2001).

  Otra línea de investigación importante en planificación clásica la constituyen los planificadores jerárquicos. Con la definición de un modelo de planificación jerárquica se pretende conseguir dos objetivos: adaptar el proceso de planificación a la manera en que los humanos resolvemos la mayoría de los problemas complejos y reales, y reducir la complejidad computacional. En esta línea destacan los trabajos sobre Redes de Tareas Jerárquicas (HTN) (Erol, Hendler, y Nau, 1995).

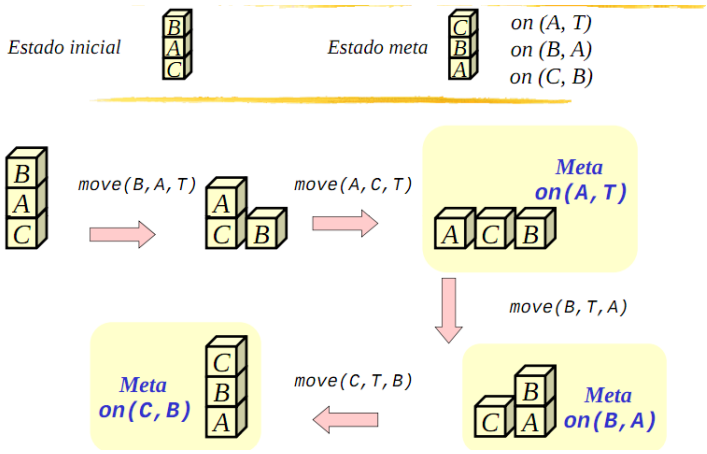


Figura 2.4: *Ejemplo de planificación STRIPS*

- **Planificación conformante:**
  Las acciones transforman un estado en un conjunto de posibles estados sucesores y la tarea de planificación consiste en encontrar la secuencia de acciones que lleve a conseguir las metas por una cualquiera de las posibles transiciones, desde el estado inicial (Palacios, Bonet, Darwiche, y Geffner, 2005).

- **Planificación con incertidumbre y realimentación:**
  Cuando hay incertidumbre en la transición de estados o en el propio estado inicial, la realimentación pasa a ser un factor importante, afectando drásticamente la forma de los planes. En ausencia de realimentación, como en planificación clásica y conformante, un plan es una secuencia fija de acciones; sin embargo, en presencia de realimentación, las acciones dependen del estado observado y los planes se convierten en funciones (políticas) que transforman estados en acciones. La planificación con incertidumbre se aplica para resolver problemas de control en los que se tienen unas metas que se quieren cumplir y unos agentes capaces de realizar acciones. Son problemas en los que no se tiene un conocimiento determinista "a priori" de las acciones de los agentes; una acción no siempre produce el

mismo efecto en el sistema y hasta que no se ejecuta la acción no se sabe el efecto que producirá. Cuando, para conocer el estado del sistema, hay sensores y las acciones ya no sólo dependen del estado actual, sino también de las medidas de dichos sensores, se habla de planificación con realimentación. Un ejemplo de este tipo de planificador seria Procesos de decisión de Markov (MDP) (Howard, 1960).

2.4. Scheduling

La planificación con tiempo y recursos se aplica para resolver problemas cuyas acciones pueden tener duración y recursos. Se pueden tratar como problemas STRIPS, añadiendo a las acciones a una duración D(a) y unos recursos R(a); por lo que un recurso puede estar en uso o puede estar libre. Dos acciones a y a' son mutex si necesitan un recurso común o interactúan destructivamente (los efectos de una borran una de las precondiciones de la otra). Un schedule es un conjunto de acciones junto con los instantes de tiempo en que empiezan y terminan cada una. Por scheduling se entiende la asignación de recursos y tiempos de inicio de las actividades, obedeciendo a las restricciones temporales de las actividades y las limitaciones de capacidad de los recursos compartidos. Scheduling es también una tarea de optimización donde recursos limitados se disponen a lo largo del tiempo entre actividades que se pueden ejecutar en serie o en paralelo (plan paralelo) de acuerdo con el objetivo de, por ejemplo, minimizar el tiempo de ejecución de todas las actividades (makespan).

Muchas decisiones que se toman a la hora de resolver diversos problemas cotidianos están sujetas a restricciones. Decisiones tan cotidianas como fijar una cita, planificar un viaje, comprar un coche o preparar un plato de cocina pueden depender de muchos aspectos interdependientes e incluso conflictivos, cada uno de los cuales está sujeto a un conjunto de restricciones que se deben satisfacer para que la decisión sea válida. Además, cuando se encuentra una solución que satisface plenamente unos criterios, puede que no sea tan apropiada para otros,

por lo que, para obtener una solución optimizada, no suele ser suficiente con obtener una única solución. Los primeros trabajos relacionados con la programación de restricciones datan de los años 60 y 70 en el campo de la Inteligencia Artificial. Durante los últimos años, la programación de restricciones ha generado una creciente expectación entre expertos de muchas áreas debido a su potencial para la resolución de grandes y complejos problemas reales. Sin embargo, al mismo tiempo, se considera como una de las tecnologías menos conocida y comprendida. La programación de restricciones se define como el estudio de sistemas computacionales basados en restricciones. La idea de la programación de restricciones es resolver problemas mediante la declaración de restricciones sobre el dominio del problema y consecuentemente encontrar soluciones a instancias de los problemas de dicho dominio que satisfagan todas las restricciones y, en su caso, optimicen unos criterios determinados.

La programación por restricciones es una metodología software utilizada para la descripción y posterior resolución efectiva de cierto tipo de problemas, típicamente combinatorios y de optimización. Estos problemas aparecen en muy diversas áreas, incluyendo inteligencia artificial, investigación operativa, bases de datos y sistemas de recuperación de la información, etc., con aplicaciones en scheduling, planificación, razonamiento temporal, diseño en la ingeniería, problemas de empaquetamiento, criptografía, diagnóstico, toma de decisiones, etc. Estos problemas pueden modelarse como problemas de satisfacción de restricciones (Constraint Satisfaction Problems - CSP) y resolverse usando técnicas de satisfacción de restricciones. En general, se trata de grandes y complejos problemas, típicamente de complejidad NP. Las etapas básicas para la resolución de un problema CSP son su modelización y su posterior resolución mediante la aplicación de técnicas CSP específicas, que incluyen procesos de búsqueda apoyados con métodos heurísticos y procesos inferenciales. En este capítulo se presentan los conceptos, algoritmos y técnicas más relevantes en el área de los CSP, junto con diversos ejemplos y ejercicios.

Los modelos matemáticos más usados son:

- **Problema de satisfacción de restricciones (CSP)**
  Un problema de satisfacción de restricciones (Constraint Satisfaction Problem, CSP)(Eiben y Ruttkay, 1997) es una tripleta (X, D, R).

  - Donde X, es un conjunto de variables. X=$\{x_1$,x_2,...,$x_n\}$.
  - $D : X \rightarrow V$ es una función total que asigna un *dominio* (conjunto de valores de V_i) a cada variable. Frecuentemente se escribe D_i en vez de $D(x_i)$ para referirse al dominio de la variable x_i.
  - R = R_1,..., R_k es un conjunto de restricciones tal que cada R_i es un predicado sobre un subconjunto de los variables de X. Formalmente: R_i(x_i,..., x_n) $\subseteq$ D_i x... x D_n.

  Una solución es la asignación de valores del dominio a las variables, de manera que satisfagan todas las restricciones. Un CSP es consistente si tiene una o varias soluciones.

- **Problema de satisfacción de restricciones dinámico (DCSP)**
  Problema de satisfacción de restricciones dinámico (DCSP) (Mittal y Falkenhainer, 1990). Generalización de los CSP en los que las variables tienen etiquetas de actividad. Inicialmente sólo están activas un subconjunto de variables y el objetivo es encontrar una asignación válida para las variables activas. También contiene un conjunto de restricciones de actividad de la forma: si una variable x toma el valor v_x, entonces las variables y, z, w . . . pasan a estar activas.

- **Problema de optimización de restricciones (COPs)**
  Problema de optimización de restricciones (COPs) (Dechter, 2003). Iguales que los anteriores pero con una función de optimización.

Otros métodos para hacer *scheduling* son: heurística (Caramia y Giordani, 2009), algoritmos genéticos híbridos (Gen y Yoo, 2008), inteligencia artificial mediante

el planificador PRODIGY (Rodríguez-Moreno, Borrajo, y Meziat, 2004), redes de Petri (Asfour, Ly, Regenstein, y Dillmann, 2006), ...

Capítulo 3

MIPS

Este capítulo pretende ser una breve introducción para el lector en la arquitectura MIPS.

3.1. Introducción

Con el nombre de MIPS (Microprocessor without Interlocked Pipeline Stages) se conoce a toda una familia de microprocesadores de arquitectura RISC desarrollados por MIPS Technologies.

Los diseños del MIPS son utilizados en la línea de productos informáticos de SGI; en muchos sistemas embebidos; en dispositivos para Windows CE; routers Cisco; y videoconsolas como la Nintendo 64 o las Sony PlayStation, PlayStation 2 y PlayStation Portable.

Las primeras arquitecturas MIPS fueron implementadas en 32 bits (generalmente rutas de datos y registros de 32 bits de ancho), si bien versiones posteriores fueron implementadas en 64 bits. Existen cinco revisiones compatibles hacia atrás del conjunto de instrucciones del MIPS, llamadas MIPS I, MIPS II, MIPS III, MIPS IV y MIPS 32/64. En la última de ellas, la MIPS 32/64 Release 2, se

define a mayores un conjunto de control de registros. Así mismo están disponibles varias extensiones, tales como la MIPS-3D consistente en un simple conjunto de instrucciones SIMD en coma flotante dedicadas a tareas 3D comunes, la MDMX(MaDMaX) compuesta por un conjunto más extenso de instrucciones SIMD enteras que utilizan los registros de coma flotante de 64 bits, la MIPS16 que añade compresión al flujo de instrucciones para hacer que los programas ocupen menos espacio (presuntamente como respuesta a la tecnología de compresión Thumb de la arquitectura ARM) o la reciente MIPS MT que añade funcionalidades multithreading similares a la tecnología HyperThreading de los procesadores Intel Pentium 4.

3.2. Diseño

Debido a que los diseñadores crearon un conjunto de instrucciones tan claro, los cursos sobre arquitectura de computadores en universidades y escuelas técnicas a menudo se basan en la arquitectura MIPS. Por lo que lo convierte en una arquitectura muy conocida y estudiada. Sobre ella hay mucha información disponible.

3.2.1. Caracteristicas

- 64 bits
- Segmentación. Pipeline bien conocido (5 etapas).
- Ejecución fuera de orden
- Registros de instrucciones y datos separados (Modelo de memoria Harvard, muy utilizado en arquitectura RISC).

3.2.2. Unidades Funcionales

La ALU (Arithmetic Logic Unit - Unidad Aritmético Lógica) es el circuito encargado de realizar las operaciones aritméticas (sumar, restar,...) y lógicas (and, or,...) entre dos números, en el caso del MIPS se pueden utilizar registros de datos, registros de dirección y operandos inmediatos (no están guardados en un registro). Además, ayuda al procesador en guiar el control del programa; evalúa las condiciones de salto y en el caso de que se produzca, calcula los desplazamientos aplicables al contador del programa (PC)[1].

La ALU del MIPS esta dividida en 5 unidades funcionales, estas unidades son independientes una de otra para funcionar, pero pueden producirse interbloqueos por dependencia de registros.

- Enteros (sumas, restas, saltos y operaciones lógicas)
- Multiplicador 1 (Coma flotante)
- Multiplicador2 (Coma flotante)
- Sumador (Coma flotante)
- Divisor (Coma flotante)

3.3. Segmentación

El procesador MIPS utiliza segmentación. La *segmentación* es una técnica que persigue aumentar las prestaciones de los procesadores intentando maximizar el número de instrucciones ejecutadas por ciclo de reloj (CPI). Para ello la arquitectura debe cumplir las siguientes caracteristicas:

- Divide la ejecución de una instrucción en etapas (camino de datos).

[1] Lleva el registro de la próxima instrucción a ejecutar

- Cada etapa se especializa en una tarea determinada, para ello utiliza diferentes recursos hardware propios de cada etapa. La arquitectura MIPS tiene las siguientes etapas:

  1. IF: búsqueda de instrucciones (instruction fetch).
  2. ID: decodificación de instrucciones y lectura del banco de registros (instruction decodification).
  3. EX: Ejecución o cálculo de direcciones (execution).
  4. MEM: Acceso a la memoria de datos.
  5. WB: Escritura de resultados (write back)

- Permite comenzar una instrucción en cada ciclo de reloj.

- Permite solapar la ejecución de distintas instrucciones en diferentes etapas. En la fig. 3.1 se observa el cauce de ejecución de un conjunto de instrucciones dentro del procesador MIPS, se están ejecutando todas a la vez, pero cada una esta en una etapa del procesador en cada ciclo de reloj. Si todo va bien, se consigue terminar una instrucción por ciclo de reloj (CPI = 1).

N° Ciclo	1	2	3	4	5	6	7	8	9	10
Inst. 1	IF	ID	EX	MEM	WB	-	-	-	-	-
Inst. 2	-	IF	ID	EX	MEM	WB	-	-	-	-
Inst. 3	-	-	IF	ID	EX	MEM	WB	-	-	-
Inst. 4	-	-	-	IF	ID	EX	MEM	WB	-	-
Inst. 5	-	-	-	-	IF	ID	EX	MEM	WB	-
Inst. 6	-	-	-	-	-	IF	ID	EX	MEM	WB

Figura 3.1: *Cauce de ejecución de un conjunto de instrucciones dentro de un procesador segmentado*

En la figura 3.2 se puede observar como se dividen los recursos entre las 5 etapas del pipeline.

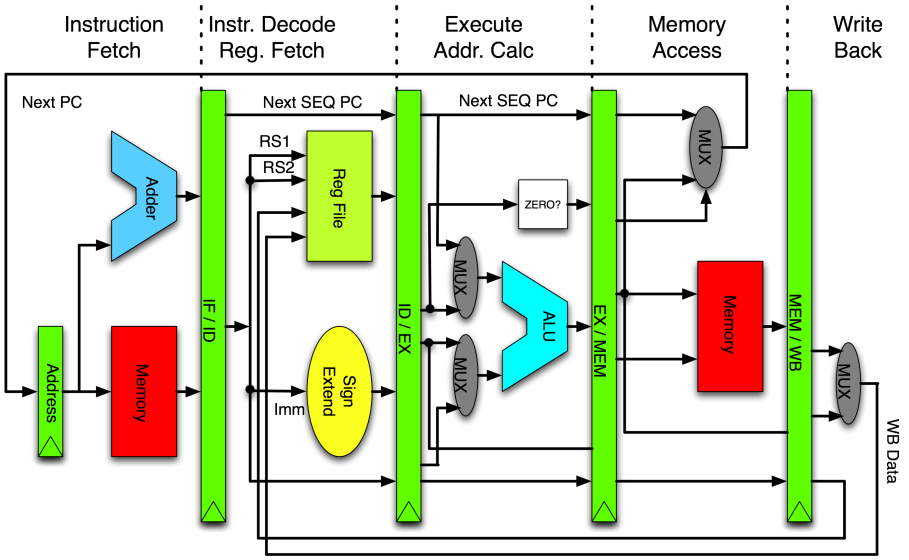


Figura 3.2: *Esquema del camino de datos del procesador MIPS*

A partir del esquema se observa cual sería el recorrido de las instrucciones dentro del procesador (conocido normalmente como camino de datos).

3.4. Gestión dinámica de instrucciones del procesador MIPS

El hardware aumenta el ILP (Instruction Level Parallelism - Paralelismo a Nivel de Instrucción) reordenando las instrucciones en tiempo de ejecución.

1. Las instrucciones independientes se ejecutan simultáneamente en la unidad segmentada.

2. Las instrucciones dependientes se ejecutan secuencialmente.

Hasta ahora, si una instrucción i se queda parada, ninguna instrucción j posterior puede continuar, incluso si j es independiente de las que están en ejecución, y el operador que j necesita está libre. El hardware debe poder lanzar a ejecución instrucciones posteriores a la que se ha parado $\rightarrow$ se altera dinámicamente el orden de la ejecución, evitando que el hecho de parar una instrucción afecte a

las que le siguen.

Ventajas:

1. Simplifica el diseño del compilador.
2. Soluciona eficientemente dependencias desconocidas en tiempo de compilación (por ejemplo, originadas entre instrucciones sólo cuando cierta condición de salto se produce).
3. Permite ejecutar eficientemente cualquier código, independientemente de si ha sido optimizado para otra unidad de instrucción segmentada $\rightarrow$ compatibilidad binaria eficiente.

Inconvenientes:

1. Complica el diseño del hardware.
2. Hay que tener cuidado con las excepciones (interrupciones).

La planificación dinámica intenta reducir al máximo el número de paradas del cauce.

En el cauce del MIPS los riesgos estructurales y las dependencias de datos se comprueban durante la etapa de ID.

Esta etapa se puede dividir en dos fases:

- Emisión: Decodificación de la instrucción y comprobación de riesgos estructurales.
- Lectura de operandos: Se espera hasta que no haya riesgos datos y se leen los operandos.

3.5. Algoritmo de Scheduling - Método de Pizarra

El algoritmo de Pizarra o Marcador es un método centralizado, utilizado en el CDC 6600 para planificar de manera dinámica la segmentación, de forma

que las instrucciones pueden ser ejecutadas fuera de orden cuando no existen conflictos y el hardware está disponible. En un marcador se registran las dependencias de datos de cada instrucción. Las instrucciones son emitidas solamente cuando el marcador determina que ya no hay conflictos con las instrucciones previamente ejecutadas o en ejecución. Si una instrucción sufre la inserción de una burbuja por considerarse insegura su ejecución, el marcador vigila el flujo de ejecución de las instrucciones hasta que todas las dependencias hayan sido resueltas, pudiendo así ser relanzada la instrucción detenida. El método de la pizarra garantiza la terminación de una instrucción por ciclo (CPI=1) siempre y cuando no existan riesgos estructurales.

- Permite que las instrucciones se ejecuten fuera de orden siempre que tengan sus operandos disponibles y recursos suficientes para su ejecución.
- La ejecución de instrucciones fuera de orden conlleva la aparición de riesgos WAR (Write After Read - Escribir Después de Leer) y WAW (Write After Write - Escribir Después de Escribir). Los riesgos WAR y WAW, son riesgos asociados a los datos, se pueden producir cuando una instrucción dentro del cauce escribe un registro, que necesita leer otra instrucción que entra después al procesador, produciendo que lea datos erróneos, debido a que se guardan los resultados obtenidos en la ultima etapa del cauce. En el cauce sin planificación dinámica no existían.
- Para conseguir este objetivo, puede que haya varias instrucciones al mismo tiempo en la etapa EX. En el cauce que vamos a estudiar, para que esto sea posible se incluyen dos multiplicadores, un divisor, un sumador y una unidad de enteros para la ejecución de saltos y accesos a memoria.

El módulo denominado Pizarra se encarga de controlar todo el proceso y centraliza la resolución de todos los posibles riesgos:

- Comprueba cuándo una instrucción está en disposición de leer sus operandos.

- Comprueba cuándo una instrucción puede pasar a ejecutarse porque tiene los recursos necesarios disponibles.
- Comprueba cuándo una instrucción puede escribir sus resultados.
- Ahora las fases por las que pasa cualquier instrucción del cauce son cuatro (la fase de búsqueda ya no se cuenta porque la hace una unidad de búsqueda externa al cauce).

Al introducir la Pizarra en el camino de datos (fig. 3.3) se obtienen las siguientes ventajas:

- No hay cortocircuitos.
- Emisión en orden.
- Ejecución fuera de orden.
- Finalización fuera de orden.

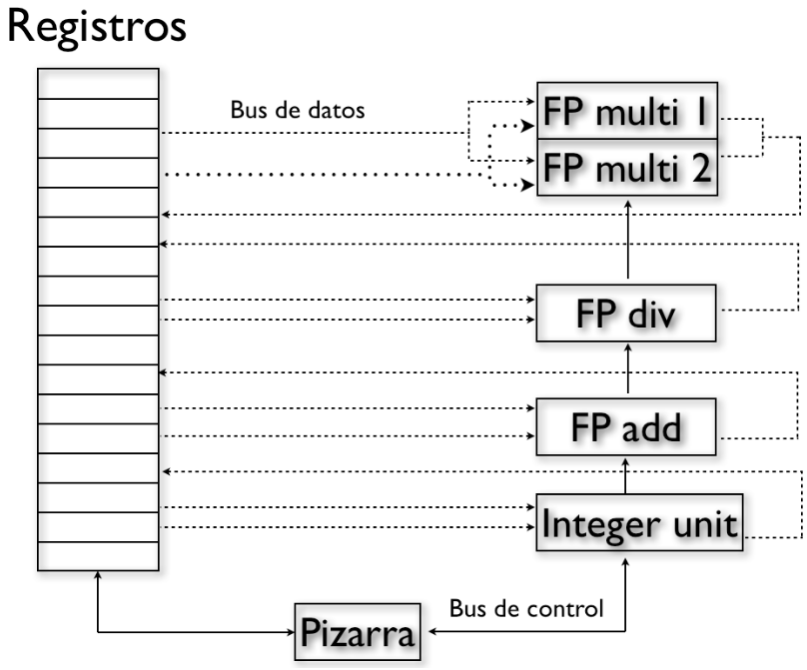


Figura 3.3: *Esquema parcial del procesador MIPS añadiendo la Pizarra*

3.5.1. Etapas del algoritmo

1. **Emisión:**
   a) Si la unidad funcional que necesita una instrucción está libre y ninguna instrucción activa tiene como registro destino el mismo que ella, la pizarra emite la instrucción.
   b) Queda reservada la unidad funcional que va a utilizar y se actualiza la información interna de la pizarra.
2. **Lectura de operandos:**
   a) La pizarra comprueba la disponibilidad de los operandos.
   b) Un operando está disponible si ninguna instrucción que se haya emitido anteriormente tiene que escribir este operando.
   c) Si todos los operandos están disponibles, la pizarra ordena a la unidad funcional que lea los operandos y que comience la ejecución. Los riesgos RAW se resuelven dinámicamente en esta fase.
3. **Ejecución:**
   a) La unidad funcional avisa a la pizarra cuando tiene el resultado listo.
4. **Escritura de resultados:**
   a) La pizarra comprueba si existen riesgos WAR.
   b) Si no existen, escribe directamente el resultado que le pasa la unidad funcional en el registro adecuado.
   c) Si existe un riesgo, para el cauce hasta que se resuelva.

3.5.2. Estructuras de control

La pizarra mantiene tres estructuras internas de datos para controlar estas etapas. Una mantiene información sobre el estado de las instrucciones, otra sobre el estado de las unidades funcionales y la última, sobre el banco de registros.

- **Estado de las instrucciones:**
  - Cada instrucción mantiene una entrada actualizada en esta estructura.
  - Se anota en qué etapa se encuentra actualmente cada instrucción.

Instrucción	Emitida	Leer Operando	Fin de ejecución	Escribo Resultado

Figura 3.4: *Estructura que guarda el estado de las instrucciones.*

- **Estado de las unidades funcionales:**
  - Cada unidad funcional mantiene una entrada actualizada.
  - Estas entradas indican si la UF está ocupada, la operación que se está realizando, cuáles son los registros fuente y destino,...
  - Si se está esperando por alguno de los operandos fuente, también se anota qué UF es la que va a producir el resultado que se necesita.

Tiempo	FU	Ocup.	Oper.	Dest. Fi	Dest. Fj	$1 Fj	%2 Fk	Qj	Qk	Fj? Rj	Fk ?Rk
	Entera										
	Mult1										
	Mult2										
	Add										
	Div										

Figura 3.5: *Estructura que guarda el estado de las unidades funcionales.*

- **Estado del banco de registros:**
  - Cada registro mantiene una entrada actualizada.
  - Esta entrada almacena la UF que tiene que escribir su resultado en cada registro. Si no se especifica ninguna UF significa que el valor

almacenado en ese registro está actualizado y se puede leer sin problema.

	F0	F2	F4	F6	F8	F10	F12
FU							

Figura 3.6: *Estructura que guarda el estado de las registros.*

El método de la Pizarra está limitado en su objetivo de minimizar el número de paradas del cauce por varios factores:

- El paralelismo intrínseco de la secuencia de instrucciones que se están ejecutando. Si todas las instrucciones dependen de las anteriores, no hay ningún tipo de planificación dinámica que pueda evitar las paradas.
- Limita la presencia de riesgos WAR y WAW.
- El número de entradas en la Pizarra. Es lo que se denomina tamaño de la ventana, y determina cuántas instrucciones examina la pizarra para encontrar instrucciones independientes.
- El número y el tipo de unidades funcionales disponibles en el cauce.

Capítulo 4

Coordinador de tareas

La planificación de tareas (a bajo nivel) es una de las piezas más importantes en la arquitectura de control del robot. Ya que es la encargada de reservar los recursos, ejecutar tareas en orden y determinar la viabilidad de las tareas pendientes. El Coordinador de tareas va ha ser la capa de software encargada de realizar estas tareas.

En este capítulo se detalla el planteamiento seguido para desarrollar este componente software basado en el *Algoritmo de Pizarra* de la arquitectura MIPS.

4.1. Planteamiento del problema

Partimos de la premisa de que el hardware a utilizar cuenta por lo general con una cantidad limitada de recursos, por lo que una buena administración de ellos nos permite ampliar las posibilidades de nuestro robot, además de limitar errores y esperas innecesarias.

La idea básica es conseguir separar el planificador de tareas de alto nivel de los recursos limitados de la máquina, dichos recursos no tienen por qué ser sólo hardware, ya que por lo general se pueden encontrar rutinas de funcionamiento en un robot que consumen una gran cantidad de recursos tales como memoria o

tiempo de procesador, al introducir una capa que controle las tareas en ejecución y los recursos disponibles en cada momento, se limita la sobrecarga del procesador con tareas innecesarias al mismo tiempo, ya que dichas tareas se podrían realizar de forma secuencial, no obligando a las tareas con mayor relevancia a tener que luchar por tiempos de ciclo. En el caso de recursos hardware, se puede encontrar un ejemplo en la ejecución de rutinas de trayectorias, además del control de estabilidad. Ambos procesos comparten los mismos recursos: motores, sensores, etc . Están mandando órdenes sobre los mismos recursos sin un orden aparente alguno. Con un coordinador de tareas se puede dar preferencia a las órdenes del estabilizador sobre las rutinas de trayectorias, por ejemplo, en los brazos a la hora de coger objetos.

4.1.1. Restricciones

A la hora de plantear una solución, se tuvieron en cuenta las siguientes premisas:

- Independencia de la Morfología.
- Estrategias del planificador modificables.
- Posibilidad de cambiar el planificador fácilmente.
- Facilidad de integración con el resto del software.
- Facilidad de cambiar de S.O.
- Desarrollo modular
- Facilidad a la hora de cambiar/integrar nuevos módulos.

Estos puntos se irán comentando a lo largo del capítulo, poniendo énfasis en la manera de solucionarlo y los problemas encontrados.

4.1.2. Coordinador de tareas

La idea básica es integrar en el robot una capa más de software encargada gestionar la ejecución de las tareas según su importancia y los recursos disponibles (scheduling). Con el añadido de conseguir un seguimiento en tiempo real del estado del robot.

En la figura 4.1 se encuentra el esquema final del Coordinador, en él se ven las distintas partes que lo componen y da una idea de su funcionamiento.

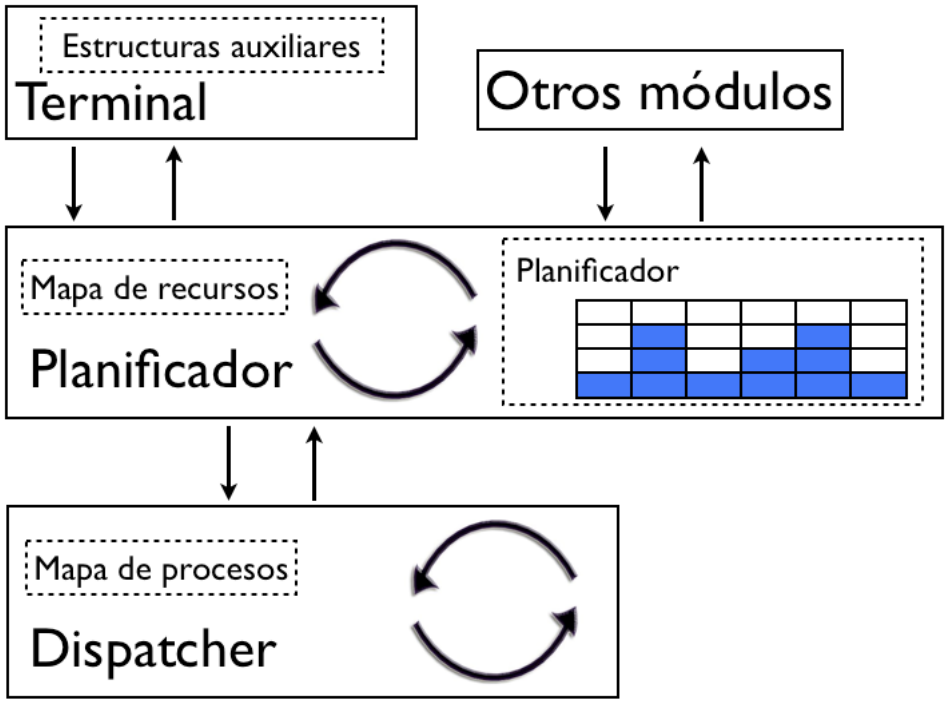


Figura 4.1: *Esquema del Coordinador de tareas*

Como se observa en la fig. 4.1 el Coordinador de tareas está dividido en dos módulos principales[1] , el planificador y el dispatcher, más uno auxiliar, la terminal. Estos módulos contienen toda la lógica de la aplicación con el apoyo

[1] **nota:** Cuando se cita *la aplicación* se corresponde con todo el coordinador de tareas, módulos incluidos

de la *librería* desarrollada para simplificar el desarrollo y el uso de estructuras de datos internas. La idea general de funcionamiento es la siguiente:

1. **Enviar de tareas al Coordinador:**
   El *planificador* de tareas recibe las tareas enviadas por el usuario u otros módulos del robot, y las organiza en colas FIFO (First in, first out) según su prioridad.

2. **Buscar siguiente tarea a ejecutar:**
   Cuando salta el temporizador examina la siguiente tarea a ejecutar (la más prioritaria) y comprueba si hay recursos disponibles.

3. **Ejecutar tarea seleccionada:**
   Una vez elegida la tarea, se reservan los recursos necesarios para su ejecución y se envía al *dispatcher*.

4. **Preparar el entorno de ejecución:**
   Cada vez que el *dispatcher* recibe una tarea prepara un entorno de ejecución para poder llevar el control en todo momento del resultado de la ejecución.

5. **Ejecutar la tarea:**
   Se lanza la tarea en su propio hilo de ejecución.

6. **Finalizar:**
   Una vez ha finalizado la tarea, se liberan los recursos utilizados por está y se notifica al planificador el resultado de la ejecución.

Este esquema es una versión simplificada, a lo largo del capítulo se ira detallando el funcionamiento exacto del *coordinador de tareas*. Para facilitar las pruebas y su posterior uso, se ha desarrollado una aplicación *terminal* para poder comunicarse con el *coordinador de tareas*, será descrito brevemente en la parte final del capítulo.

Todo el proyecto se ha desarrollado con la idea de poder ser reutilizado en otros proyectos, de está forma se divide en dos partes:

- **librería:**
  La librería es la parte de más bajo nivel de la aplicación. Simplifica entre otras cosas, la entrada y salida de datos (E/S), el formato de los ficheros y facilita la gestión de tareas y recursos. Además contiene estructuras de datos encargadas de gestionar las tareas y los recursos.

- **módulos:**
  Los módulos componen la parte de más alto nivel de la aplicación, hacen uso de la librería para desarrollar la lógica de la aplicación. El *coordinador de tareas* está compuesto de 2 módulos principales (*planificador* Y *dispatcher*) y uno auxiliar (*terminal*).

4.2. Librerías

Durante el desarrollo de las librerías, se tuvo siempre presente la posibilidad de poder crear una gran variedad de tipos de planificador, dejando la libertad de poder integrarlo en cualquier proyecto de una manera fácil y con pocos cambios en el código. Permitiendo crear nuevos módulos de alto nivel específicos para cada aplicación. De está forma se pueden crear módulos de *scheduling* y planificación acordes a las necesidades de cualquier proyecto de una forma fácil y rápida, ya que sólo habría que ocuparse de las reglas del planificador y la máquina de estados.

Para entender completamente el funcionamiento y el diseño interno del *coordinador de tareas* se describirá a continuación las clases más importantes de la librería.

4.2.1. Comunicaciones

Las comunicaciones entre los módulos (*planificador, el dispatcher y la terminal*) se realizan mediante la librería de comunicaciones *YARP* [2]. *YARP* proporciona

[2] http://eris.liralab.it/yarp/

una API de desarrollo que abstrae las comunicaciones entre software. Para ello, sigue un modelo *Publicador/Suscriptor*. Este modelo tiene 2 roles: el Publicador es el que genera el contenido y el Suscriptor es el que se registra a uno o varios Publicadores y recibe el contenido generado por ellos. Además, estos roles no tienen por qué ser excluyentes; un Publicador puede ser Suscriptor de otros y viceversa. En nuestro caso cada módulo declara un puerto *YARP* con el que se va a comunicar al exterior (Suscriptor) y se conecta al módulo adecuado en cada caso (Publicador) (ver fig. 4.2).

Figura 4.2: *Esquema simplificado de comunicaciones*

Al utilizar este modelo de comunicaciones conseguimos una conexión débil entre los componentes, con lo que para reemplazar/añadir un módulo sólo sería necesario declarar correctamente los puertos y suscribirlo al componente adecuado.

Cabe destacar que cada componente se comunica con los demás mediante el paso de mensajes. Al utilizar *YARP* como capa de abstracción para las comunicaciones se puede cambiar el lugar de ejecución de cada componente obteniendo un grado de libertad mayor si contamos con hardware muy limitado. Podríamos situar un componente en el robot, y el resto en un computador conectando ambos por wifi; otra solución sería incluir todos los componentes en el robot.

4.2.2. Protocolo de aplicación

Sobre *YARP* se ha establecido un protocolo de comunicaciones en el *nivel de aplicación*, pero aprovechando las facilidades que nos proporciona la API de

YARP. Para ello se utiliza la clase *Bottle*[3](disponible en *YARP*) para encapsular los mensajes utilizados en el protocolo de comunicaciones.

Para ello se utilizan 3 tipos de mensajes:

- Mensaje de tarea
- Mensaje de protocolo
- Mensaje de sincronización

Todas ellas se encapsulan en un *Bottle*. Cabe destacar que realmente el único mensaje que contiene datos de *usuario* (En este software serían las tareas) es únicamente el *mensaje de tareas*. *Mensaje de protocolo y mensajes de sincronización* son cabeceras que se añaden a los *Bottle* para intercambiar información de estado o facilitar la sincronización. Para facilitar la comprensión se asemeja al protocolo de comunicación TCP/IP.

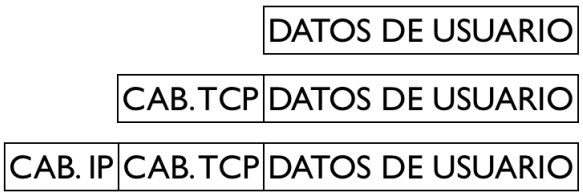


Figura 4.3: *Esquema simplificado protocolo TCP/IP*

En la fig. 4.3 se puede ver cómo se van añadiendo cabeceras según van pasando por cada fase de la pila de protocolos TCP/IP.

El protocolo de aplicación propuesto sería más bien el siguiente (fig. 4.4), donde a los datos de la aplicación se les puede añadir cabeceras si fuera necesario, o usar las cabeceras por si solas. Esto, aunque parezca contradictorio, reduce la cantidad de información transmitida por la aplicación, ya que muchas veces no es necesario reenviar la misma información entre los distintos componentes. Con mandar el identificador del recurso a utilizar y la información asociada es suficiente.

[3]Clase envoltorio utilizada en las librerías YARP para las comunicaciones.

Figura 4.4: *Esquema simplificado del protocolo de aplicación*

4.2.2.1. Mensaje de tarea

En este mensaje (fig. 4.5) se encapsulan las tareas, para mandarlas entre los distintos componentes de la aplicación.

Figura 4.5: *Composición de un mensaje tarea*

Cuenta con los siguientes campos:

- **TIPO:** Identifica el contenido del mensaje, en este caso que es una tarea. [4]
- **ID TAREA:** Identificador único para el mensaje, que enumera el número de tareas enviadas; este campo permite hacer un historial de tareas enviadas.
- **TAREA:** Contiene la tarea a enviar.

4.2.2.2. Mensaje de protocolo

Esta cabecera se utiliza para mandar datos de aplicación entre los módulos, de está forma se consigue separar los datos de la aplicación del protocolo utilizado para establecer una comunicación clara.

[4] Este campo tiene relevancia por el echo de que deja las bases preparadas para en una futura revisión de la aplicación poder añadir nuevos tipos de recursos a compartir, no solamente tareas, sin tener que realizar grandes cambios en la aplicación.

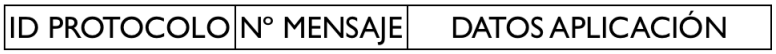


Figura 4.6: *Composición de una cabecera de protocolo*

Los campos de está cabecera son los siguientes:

- **ID PROTOCOLO:** Identificador que indica a que acción (nueva tarea, ejecutar tarea,...) está asociada los datos de la aplicación (ver Apéndice A).
- **NUMERO MENSAJE:** Número que marca el mensaje con un identificador único facilitando la realización de un historial de mensajes.
- **DATOS DE APLICACIÓN:** Información original a la que se le ha añadido una cabecera, en éste caso *tareas*.

4.2.2.3. Mensaje de sincronización

Esta cabecera/mensaje se utiliza por lo general como respuesta a un *mensaje de protocolo*. Por lo que se pueden encontrar mensajes únicamente formados por está cabecera, en donde se indica la acción que genera la respuesta y el identificador de estado/tarea asociado. De está forma se reducen la cantidad de información enviada. En algunas ocasiones puede actuar como cabecera acompañando una tarea y un estado asociado.

Figura 4.7: *Composición de una cabecera/mensaje de sincronización*

Los campos de está cabecera son los siguientes:

- **ID PROTOCOLO:** Identificador que indica a qué acción está asociada los datos de la aplicación (ver Apendice A).
- **ID TAREA/ESTADO:** Indica el resultado de la acción indicada por el protocolo o la tarea asociada. El significado de este campo está establecido en la acción asociada (nueva tarea, tarea finalizada, ...).

- **DATOS DE APLICACIÓN:** Información original a la que se le ha añadido una cabecera, en nuestro caso *tareas*.

4.2.3. Mapa de recursos

En el *mapa de recursos* se almacenan los recursos (hardware y software) disponibles para la plataforma destino. Para poder englobar el mayor número de plataformas posible, los recursos se declaran de forma independientemente en un fichero XML y son leídos en el momento de inicialización del módulo que gestiona el *mapa de recursos*.

4.2.3.1. Recurso

Un recurso es todo componente hardware y software, que se puede reservar de forma independiente para una tarea y puede resultar crítico para la estabilidad del sistema. Por ejemplo, no se puede reservar partes del sistema operativo base, pero sí reservar módulos software críticos para la estabilidad, como por ejemplo el planificador de trayectorias. La definición de un recurso se encuentra en los ficheros *Resource.h* y *Resource.cpp*.

De un recurso se almacena la siguiente información:

- **id**
  Es el identificador de recurso, debe de ser único y sirve para identificar cada recurso desde otras partes del software.

- **name**
  Nombre del recurso, sirve como descripción para saber rápidamente con qué estamos trabando en cada momento.

- **state**
  Estado actual del recurso, inicialmente un recurso está libre, pero a lo largo del tiempo, cada recurso puede pasar por los siguientes estados:

1. *free*: El recurso está libre, y puede utilizarse.
2. *working*: El recurso está en uso.
3. *sleep*: El recurso está reservado, pero el proceso al que pertenece está dormido.
4. *wait*: El recurso está reservado, pero el proceso al que pertenece está esperando fuera del procesador.

Como se puede apreciar, el estado de los recursos, se corresponde al estado del proceso que lo ha reservado[5].

- **idTask**
  Identificador de la tarea, a la que está asociado el recurso.

- **task**
  Nombre de la tarea a la que está asociado el recurso.

- **priority**
  Prioridad de la tarea a la que está asociado el recurso.

4.2.3.2. Mapa de recursos

El *mapa de recursos* es un tipo de datos creado (*clase c++*)[6] para manejar todos los recursos con los que cuenta la arquitectura hardware destino. La clase utiliza un *hashmap* para almacenar todos los recursos de una manera fácil y poder recuperarlos rápidamente.

Para una mejor comprensión, se puede asemejar a una tabla como la de la figura 4.8.

[5] El estado *sleep y wait* no se utilizan en la versión actual del programa, están declarados para futuros revisiones.

[6] En la programación orientada a objetos, una clase es una construcción que se utiliza como un modelo (o plantilla) para crear objetos de ese tipo. El modelo describe el estado y el comportamiento que todos los objetos de la clase comparten.

id	name	state	idTask	task	priority
0	EncodersPiernas	working	5	andar	3
1	MotoresPiernas	working	5	andar	3
2	EncodersBrazos	free	0	null	99
3	MotoresBrazos	free	0	null	99
4	Camara	working	6	buscar pelota	3

Figura 4.8: *Ejemplo de mapa de recursos para un robot humanoide*

Esta clase, nos permite conocer de un vistazo rápido el estado actual de la arquitectura objetivo. Además, para minimizar los errores, solamente los métodos de la clase pueden modificar el estado del procesador, limitando la aparición de inconsistencias.

4.2.3.3. Fichero de inicialización

Ya se ha indicado que el fichero de recursos es un fichero XML, para limitar el número de errores tiene un fichero *DTD* [7] asociado, de está forma se minimizan los errores de sintaxis.

En la fig. 4.9 se observa el contenido del fichero ResourceFiles.dtd:

```
1  <?xml version="1.0" encoding="ISO-8859-1" ?>
2
3  <!-- Este es el DTD de ResourceFile.dtd -->
4
5  <! DOCTYPE ResourceFile[
6          <!ELEMENT ResourceFile (resource+)>
7          <!ELEMENT resource (id, name,state,task,idTask,priority)>
8          <!ELEMENT id (#PCDATA)>
9          <!ELEMENT name (#PCDATA)>
10         <!ELEMENT state (#PCDATA)>
11         <!ELEMENT task (#PCDATA)>
12         <!ELEMENT idTask (#PCDATA)>
```

[7] Describe la estructura y sintaxis de los documentos XML.

```
13          <!ELEMENT priority (#PCDATA)>
14  ]>
```

Figura 4.9: *Fichero DTD asociado a un fichero XML de recursos*

En la fig. 4.10 se observa un fichero de ejemplo con recursos:

```
1   <?xml version="1.0"?>
2   <!DOCTYPE ResourceFile SYSTEM "ResourceFile.dtd">
3   <ResourceFile>
4           <resource>
5                   <id>0</id>
6                   <name>EncoderPiernas</name>
7                   <state>0</state>
8                   <task>null</task>
9                   <idTask>0</idTask>
10                  <priority>99</priority>
11          </resource>
12          <resource>
13                  <id>1</id>
14                  <name>ComandarPiernas</name>
15                  <state>0</state>
16                  <task>null</task>
17                  <idTask>0</idTask>
18                  <priority>99</priority>
19          </resource>
20  </ResourceFile>
```

Figura 4.10: *Ejemplo de fichero XML con recursos*

4.2.4. Tareas

Una *tarea*[8] es el objeto encargado de gestionar la información correspondiente de una acción (aplicación, comando, script,...) llegada del exterior, a través de otro módulo o de un operador humano.

Una tarea tiene los siguientes campos:

[8]Correspondiente a la clase *Task.h y Task.cpp*.

- **id**

  Es el identificador único de la tarea.

- **task**

  Es el comando u aplicación a ejecutar.

- **argumentList**

  Lista de argumentos necesarios para poder ejecutar correctamente la tarea.

- **resourceList**

  Recursos necesarios para una correcta ejecución de la tarea.

- **priority**

  Prioridad de ejecución de la tarea siendo 0 la más alta y 99 la más baja.

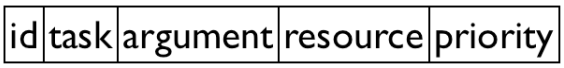


Figura 4.11: *Diagrama de una tarea*

En la fig. 4.12 hay un ejemplo de una *tarea* sencilla, en la que se distinguen los distintos campos descritos en el diagrama de la fig. 4.11.

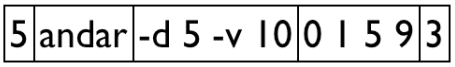


Figura 4.12: *Ejemplo de una tarea*

4.2.4.1. Paquete de tareas

Para simplificar el entendimiento del problema, se ha establecido un estándar para un fichero de tareas, lo que permite cargar desde un mismo archivo tareas relacionadas, con lo que se pueden tener varios ficheros con tareas específicas para realizar una acción. De está forma se pueden eliminar errores rápidamente y facilitar la realización de las pruebas. El fichero de tareas (fig. 4.14) es un fichero *XML* con un fichero *DTD* asociado (fig. 4.13).

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Este es el DTD de TaskFile.dtd -->
<! DOCTYPE TaskFile[
<!ELEMENT TaskFile (task+)>
<!ELEMENT task (name, argument,resources,priority)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT argument (arg*)>
<!ELEMENT arg (#PCDATA)>
<!ELEMENT resources (rec+)>
<!ELEMENT rec (#PCDATA)>
<!ELEMENT priority (#PCDATA)>
]>
```

Figura 4.13: *Fichero DTD asociado a un fichero de tareas*

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
        <task>
                <name>run</name>
                <argument>
                        <arg>rapido</arg>
                        <arg>speed</arg>
                        <arg>5</arg>
                </argument>
                <resources>
                        <rec>1</rec>
                        <rec>2</rec>
                        <rec>3</rec>
                </resources>
                <priority>1</priority>
        </task>
</TaskFile>
```

Figura 4.14: *Ejemplo de fichero de tareas*

4.2.5. Cola de tareas

La *cola de tareas*[9] es la clase más básica de todas las existentes en la librería que se encargan de gestionar las *tareas* que van llegando a la aplicación final.

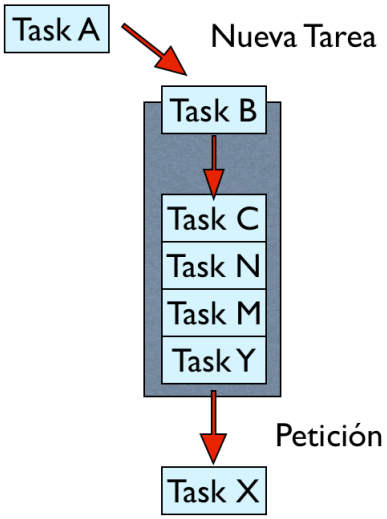


Figura 4.15: *Esquema simplificado de cola de tareas*

Esta clase sólo admite en la estructura objetos de la clase *tarea* y está optimizada para poder trabajar con ella de una forma fácil (fig. 4.15). Entre los métodos añadidos está la posibilidad de devolver una copia de la cola ordenada por la prioridad de las tareas y poder exportarla a una lista de tareas.

4.2.6. Planificador

El planificador es una clase genérica[10] para la construcción de todo tipo de planificadores; también se puede utilizar sólo la interfaz y crear otro planificador usando las mismas cabeceras.

[9] Correspondiente a las clases *TailTask.h y TailTask.cpp*.

[10] Para más información consultar la documentación de las clases planificador.h y planificador.cpp

El planificador básicamente es un vector dinámico de colas de tareas, que además aprovecha la funcionalidad de éstas para poder escoger varias estrategias a la hora de elegir cual es la siguiente tarea a ejecutar.

En la fig. 4.16 muestra un esquema básico de la clase planificador; se puede observar que está compuesto por varias *colas de tareas* cada una se corresponde con un nivel de prioridad.

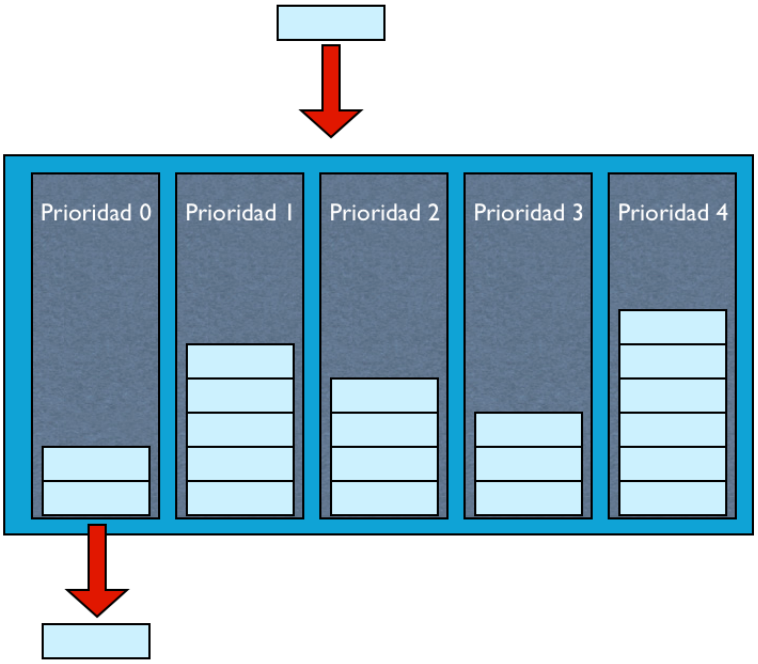


Figura 4.16: *Esquema simplificado del planificador*

4.2.6.1. Estrategias de planificación

El *planificador* acepta varias estrategias de planificación debido al diseño utilizado. Para ello se puede configurar mediante parámetros a la hora de inicializarlo y cabe destacar que algunos se pueden modificar en tiempo de ejecución.

Veamos las estrategias básicas con las que cuenta:

- **FIFO (Primero en entrar primero en salir):**
  Esta estrategia es la más básica de todas (fig. 4.17); para ello sólo utiliza

una cola de tareas y ordena las tareas por orden de llegada sin tener en cuenta las prioridades.

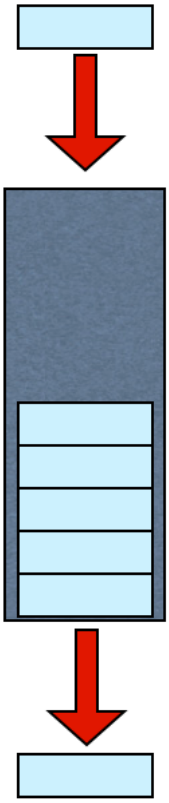


Figura 4.17: *Esquema de planificador FIFO*

- **Cola simple de prioridades:**
  Una única cola de tareas pero ordena las tareas por prioridades (siempre se ejecutan primero las más prioritarias) sin importar el orden de llegada (fig. 4.18).

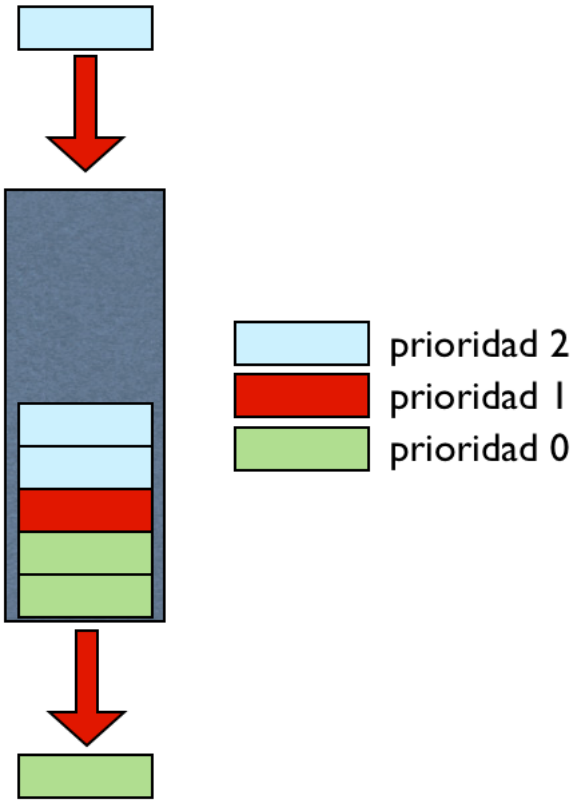


Figura 4.18: *Esquema de cola prioritaria*

- **FIFO múltiple orden simple:**
  Múltiples colas de tareas, a cada una le corresponde un orden de prioridad; si hay más niveles de prioridad que colas de tareas, se almacenan en la cola de prioridad más baja por orden de llegada (fig. 4.19).

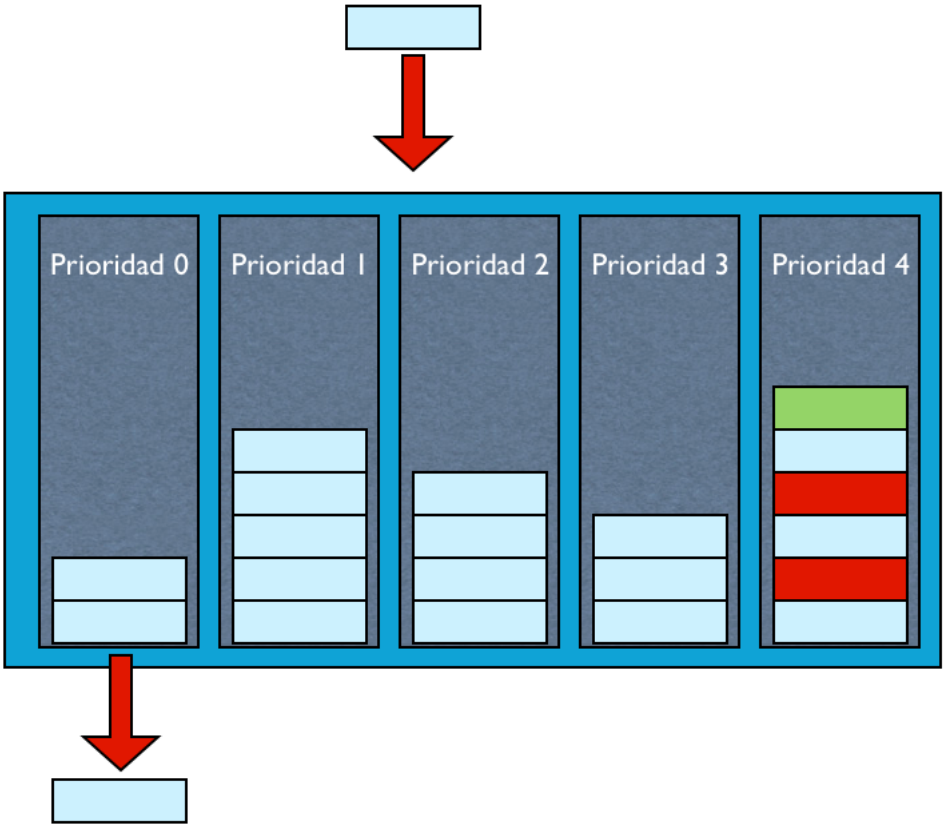


Figura 4.19: *Esquema de FIFO múltiple orden simple*

- **FIFO múltiple orden prioritario:**
  Múltiples colas de tareas, a cada una le corresponde un orden de prioridad; si hay más niveles de prioridad que colas de tareas, se almacenan en la cola de prioridad más baja por prioridad de tarea (fig. 4.20).

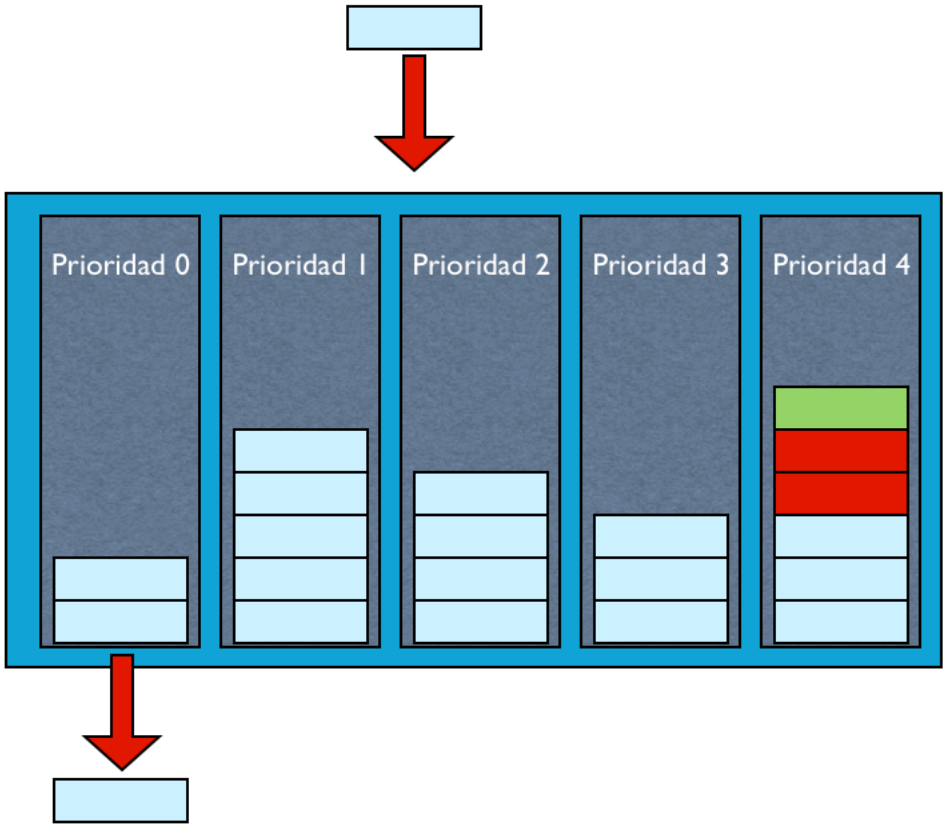


Figura 4.20: *Esquema de FIFO multiple orden prioritario*

4.2.6.2. Parametrización

El funcionamiento de la aplicación se puede modificar mediante el uso de parámetros para poder crear varias estrategias de planificación de forma fácil y rápida. No hay limite en el número de colas, aparte de la memoria física disponible [11]

[11] Por simplificar el diseño la prioridad más baja es 99 que se utiliza para dar una prioridad por defecto a las tareas que no se les indica la prioridad, lo que no llega ha ser ninguna limitación practica, ya que un planificador de 99 colas, puede llegar a ser una tarea imposible de comprobar, pero aun así sólo habría que cambiar un parámetro del código para saltar está limitación.

4.2.7. Mapa de procesos

El *mapa de procesos*[12] es una estructura de datos creada para gestionar los procesos en ejecución. Se ha implementado como un *HashMap* con lo que conseguimos un acceso directo a los procesos almacenados.

4.2.7.1. Proceso

La clase *proceso*[13] almacena el estado de cada tarea en ejecución, permite ejecutar y detener cada tarea generando un entorno de ejecución seguro para evitar que un fallo en un proceso detenga la aplicación.

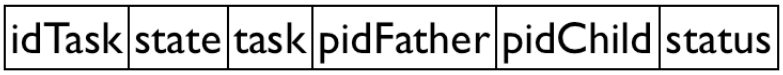


Figura 4.21: *Esquema de un proceso*

La información que se almacena es la siguiente:

- **idTask:**
  Identificador de la tarea en ejecución, al ser identificadores únicos para cada tarea, se puede utilizar como identificador del proceso.

- **state:**
  Estado actual de la tarea (en ejecución, finalización correcta, finalización forzada,...).

- **task:**
  Tarea a ejecutar (es un objeto de la clase tarea).

- **pidFather:**
  Pid del proceso padre que gestiona la tarea (ver secc. 4.2.7.2).

[12] Para más información mirar la documentación de *ProcessMap.h* y *ProcessMap.cpp*
[13] Para más información mirar la información relativa a *Process.h* y *Process.cpp*

- **pidChild:**
  Pid del proceso hijo que gestiona la tarea (ver secc. 4.2.7.2).

- **status:**
  Información devuelta por el proceso cuando termina.

4.2.7.2. Entorno de ejecución

Para conseguir más estabilidad en la aplicación, cada tarea se ejecuta en un proceso independiente, y éste es el encargado de generar dos procesos, un proceso hijo que realiza la ejecución de la tarea y un proceso padre que espera a que termine el hijo. De está forma, aunque falle la ejecución, no afecta a la aplicación ya que está en procesos independientes. La comunicación de los resultados se realiza a través de YARP, ya que al ser procesos independientes, no hay posibilidad de compartir recursos (se podría usar memoria compartida, pero de está forma se consigue independencia de la plataforma y poder poner la aplicación en una máquina y que ejecute en otra)[14].

En la fig. 4.22 se encuentra un diagrama del funcionamiento de la ejecución de una tarea. Cuando se recibe la petición de ejecución de tarea, se crea un proceso asociado a la tarea (**punto 0**) y se almacena en el mapa de procesos. Una vez creado y comprobado que se puede ejecutar, comienza el proceso de ejecución que pasa por los siguientes puntos:

1. Ejecutamos el proceso. En este punto se prepara el entorno de ejecución, se crean variables comunes a los dos futuros hijos.

2. Mediante la llamada al sistema *fork()* se crea el proceso padre.

3. El programa principal continua con la ejecución del programa.

[14] **Nota:** Esta es la unica parte dependiente del sistema operativo, para la creación de los procesos y la ejecución de las tareas se han utilizado las *llamadas al sistema* (Mecanismo usado por una aplicación para solicitar un servicio al sistema operativo) disponibles en GNU/Linux.

4. El proceso padre realiza la llamada al sistema *fork()* y crea el proceso hijo. El padre realiza comprobaciones referentes al proceso hijo.

5. El proceso hijo prepara la ejecución de la tarea asociada al proceso y realiza la llamada al sistema *exec()* con los parámetros de la tarea, reemplazando el proceso hijo con el contexto de la tarea ejecutada.

6. El proceso padre espera la terminación de la tarea en ejecución.

7. Cuando el proceso padre recibe la señal de que la tarea en ejecución ha terminado, le notifica la terminación de la ejecución al programa principal y el resultado devuelto por el proceso hijo.

8. El proceso padre libera recursos y termina.

9. El programa principal notifica la terminación de la ejecución de la tarea y notifica los errores si los hubiera.

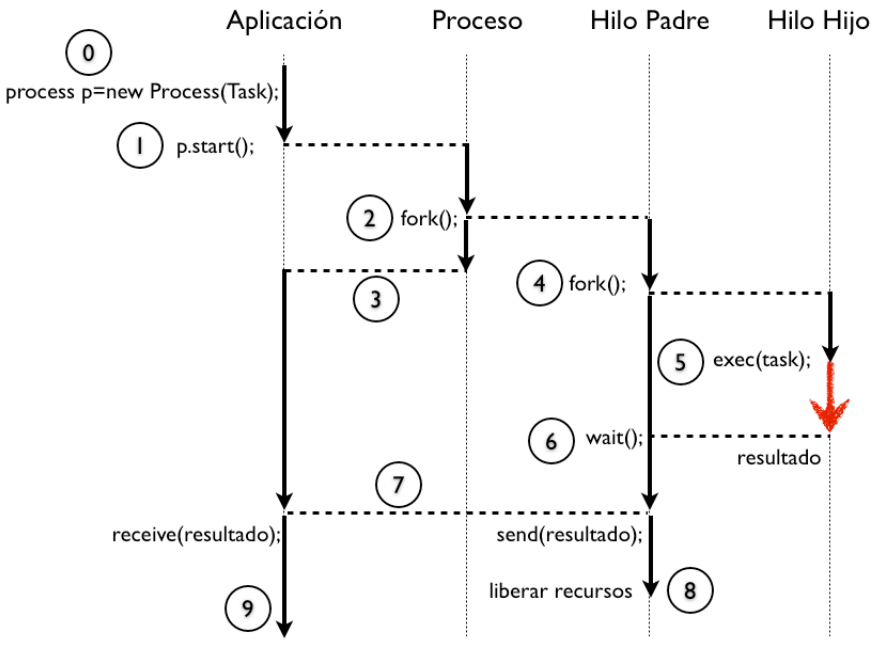


Figura 4.22: *Esquema de ejecución de una tarea*

4.3. Módulos

En el punto anterior se ha visto las librerías desarrolladas para poder crear coordinadores de tareas a medida de una forma fácil. En este punto se muestran las soluciones propuestas por este proyecto fin de máster.

La solución propuesta se compone de 3 módulos: *Planificador*, *Dispatcher* y *Terminal*. El planificador y la terminal se pueden situar en el robot o en un computador en la misma red. El *dispatcher* en su versión actual, debe residir en el robot.

Todos los módulos utilizan *YARP* como capa de abstracción en las comunicaciones (ver sección 4.2.1).

4.3.1. Visión general del Coordinador de tareas

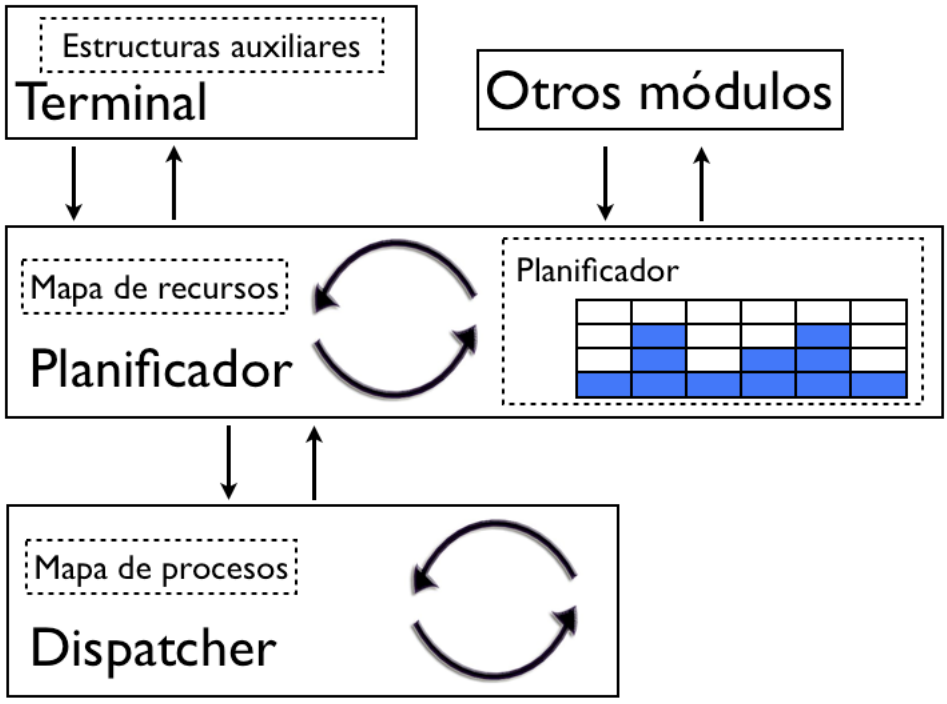


Figura 4.23: *Esquema del Coordinador de tareas*

En la fig. 4.23 se ve el esquema de alto nivel del *coordinador de tareas*. En él se observan los dos módulos principales y la terminal, y la conexión que hay entre ellos. Se aprecia como los módulos están conectados en cadena, es decir para comunicarse los extremos deben de pasar los mensajes por el *planificador* y el se encargara de enviárselos a quien corresponda. De está forma eliminamos los efectos secundarios y aumentamos la seguridad. Para comprender mejor está idea, la *terminal* manda una tarea al *planificador*, cuando la tarea tenga los recursos necesarios para ejecutarse y sea prioritaria, sera mandada al *dispatcher* y se ejecutara. Cuando termine, le notificara el resultado al *planificador*, este a su vez, liberara los recursos reservados para esa tarea y le notificara la terminación de la tarea a la *terminal* además del resultado de la ejecución.

En las siguientes secciones se vera una breve descripción de los módulos y cual es su cometido dentro del *coordinador de tareas*, además de su relación con el resto de módulos.

4.3.2. Planificador

El *planificador* tiene encomendados los siguientes cometidos:

- **Recibir las peticiones:** Recibirá todas las peticiones del exterior, descartando las peticiones defectuosas.

- **Gestionar las tareas y recursos:** Se encarga de gestionar el orden de las tareas que le van llegando y buscar la tarea más prioritaria. Para ello reserva los recursos necesarios, pone a la espera las tareas que no dispongan de ellos y en el caso de una tarea prioritaria que no cuente con los recursos suficientes, desaloja otras tareas menos prioritarias para liberar los recursos para la nueva tarea. Se encarga de liberar los recursos según van acabando las tareas.

- **Gestionar las peticiones:** Traducir las peticiones del exterior en acciones: Añadir nuevas tareas, mandar a ejecutar la tarea prioritaria en cada momento y notificar los resultados de cada ejecución.

4.3.2.1. Funcionamiento

El planificador cuenta con dos modos de funcionamiento: manual y automático. El manual requiere la interacción del usuario para solicitar tareas al planificador. El automático cuenta con un temporizador regulable, que solicita tareas cada X milisegundos, está cantidad de tiempo[15] es un parámetro de la aplicación.

Este planificador utiliza la clase *planificador.c* [16] para almacenar las tareas que va recibiendo y decidir cual es la siguiente tarea a ejecutar, para ello utiliza el temporizador para disparar la ejecución del siguiente algoritmo:

1. Busca la tarea más prioritaria dentro del planificador, para ello busca por las colas de prioridades. Si la encuentra y hay recursos disponibles la manda al *dispatcher*.

2. Busca la tarea más prioritaria dentro del planificador, para ello busca por las colas de prioridades. Si la encuentra y no hay recursos disponibles pueden pasar dos cosas:

   - Si los recursos necesarios para la ejecución de la tarea seleccionada, están reservados por tareas de menor prioridad, se manda una petición de terminación al *dispatcher* de cada tarea, se liberan los recursos y se lanza la ejecución de la tarea seleccionada.
   - Si los recursos necesarios para la ejecución de la tarea seleccionada

[15] Para facilitar la depuración por defecto son 5 segundos.
[16] Para más información ver la sección 4.2.6.

están reservados por tareas de mayor prioridad, se deja la tarea seleccionada en espera hasta que se liberen los recursos.

3. Se busca la siguiente tarea con mayor prioridad en la siguiente cola del planificador, se repiten los pasos 2 y 3 hasta que se consiga ejecutar una tarea o se acaben las colas de ejecución.

4. Si no consigue encontrar ninguna tarea con recursos suficientes para ejecutarse, se espera a la terminación del temporizador para volver a ejecutar el algoritmo.

4.3.3. Dispatcher

El *dispatcher* tiene encomendados los siguientes cometidos:

- Preparar un entorno seguro de ejecución [17]
- Gestionar las tareas en ejecución: Almacenar la tarea, los datos del proceso (PID del padre y del hijo de los procesos creados).
- Esperar la terminación de las tareas lanzadas y notificar los resultados al *planificador*.
- Forzar la terminación de tareas cuando el *planificador* lo estipule.

4.3.4. Terminal

La *terminal* es un módulo auxiliar, fuera del *coordinador de tareas*, aun así tiene bastante importancia, ya que permite probar el correcto funcionamiento y el protocolo de comunicaciones de la aplicación.

En la fig. 4.24 se ven las opciones disponibles en el módulo.

[17] Para más información ver la sección 4.2.7.2.

```
Bienvenido al menu principal:
1. Nueva Tarea
2. Cargar Lista de tareas
3. Eliminar tarea en ejecucion (Simulacion de terminacion de tarea)
4. Mostrar el historial de tareas
5. Mostrar el historial de tareas en ejecucion
6. Solicitar tarea
7. Salir
```

Figura 4.24: *Menú de opciones del terminal*

- **Nueva tarea:** Permite introducir una tarea de forma manual y mandarla al *planificador*.
- **Cargar lista de tareas:** Permite cargar bloques de tareas con el formato especificado en la sección 4.2.4.1. Esta solución permite automatizar la carga de tareas.
- **Eliminar Tarea en Ejecución:** Esta opción permite eliminar una tarea manualmente del procesador y liberar los recursos correctamente.
- **Mostrar el historial de Tareas:** Muestra todas las tareas ejecutadas o no, que han pasado por el planificador.
- **Mostrar el historial de Tareas en Ejecución:** Muestra todas las tareas en ejecución actualmente.
- **Solicitar Tarea:** Solicita la ejecución de la tarea más prioritaria.

4.3.5. Coordinador de tareas

El conjunto de los módulos forman el *coordinador de tareas*, como se puede apreciar se ha utilizado el paradigma de programación por capas; en nuestro caso está formado por 3 capas:

- **Vista:** Esta capa se encarga de la presentación de los datos, se correspondería con la terminal.
- **Lógica de la aplicación:** Esta capa contiene toda las reglas de la aplicación para que pueda funcionar correctamente, algoritmos de *scheduling* además de los datos.
- **Controlador:**. Esta capa se encarga de gestionar la ejecución de las tareas.

El funcionamiento de toda la aplicación se basa en el paso de mensajes entre capas, cada mensaje produce un cambio de estado del resto de capas y genera un mensaje de respuesta acorde al estado actual. Se podría asemejar al comportamiento de una *máquina de estados*. En la figura 4.25 se muestra el ciclo simplificado de funcionamiento[18].

El *coordinador* pasa por las siguientes fases:

1. Llega una nueva tarea al *planificador*.
2. Se añade la tarea nueva al *planificador*.
3. Cuando salta el temporizador, se selecciona la siguiente tarea a ejecutar[19].
4. Manda la tarea seleccionada al *dispatcher*.
5. El *dispatcher* recibe la tarea, prepara el entorno de *ejecución seguro*[20].
6. El *entorno de ejecución* notifica al *dispatcher* el comienzo de la ejecución de la tarea.
7. El *dispatcher* notifica al *planificador* el comienzo de la ejecución de la tarea.
8. El *planificador* notifica a la *terminal* el comienzo de la ejecución de la tarea.

[18] En este caso es la *terminal* el método de entrada al coordinador, pero el funcionamiento es el mismo para un módulo externo
[19] Ver sección 4.3.2.1
[20] Ver sección 4.2.7.2

9. Finaliza la tarea y el *entorno de ejecución* notifica la terminación y el resultado de la ejecución al *dispatcher*.

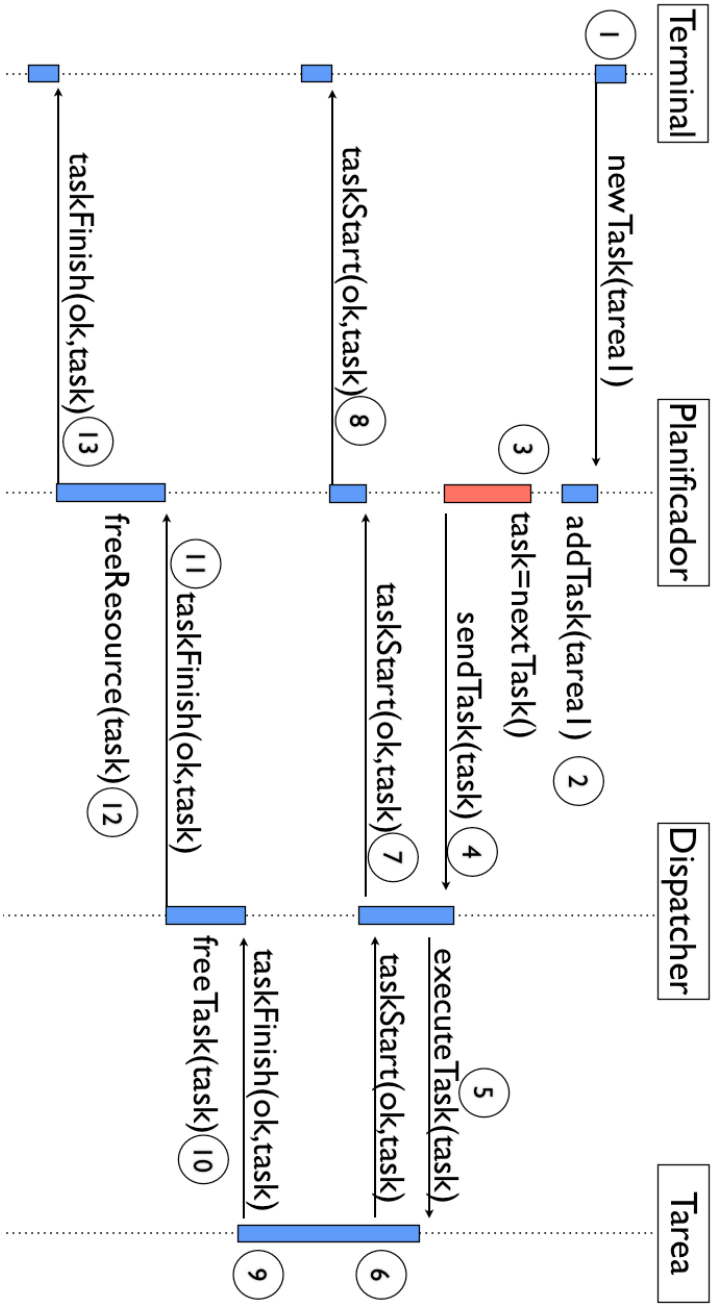


Figura 4.25: *Esquema general de funcionamiento*

10. El *dispatcher* libera los recursos reservados para el *entorno de ejecución*: mata los procesos, los puertos *YARP*,.. .
11. El *dispatcher* notifica la finalización y el resultado al *planificador*.
12. El *planificador* libera los recursos asociados a la tarea.
13. El *planificador* notifica a la terminal la terminación de la tarea y el estado de la ejecución.

En la figura 4.25 se muestra la ejecución sin errores de una tarea. En el caso de error el *coordinador* notifica el problema ocurrido a la terminal y como solucionarlo.

4.3.5.1. Tipos de ejecución

Dependiendo de la prioridad de la tarea y de los recursos asociados, se pueden producir tres situaciones distintas al ejecutar la siguiente tarea dentro del planificador:

- **Ejecución secuencial:**

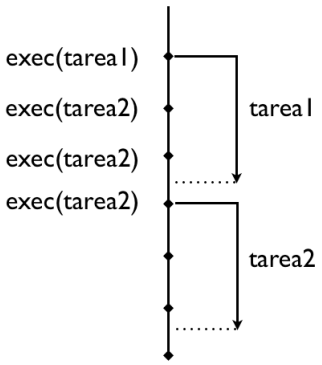


Figura 4.26: *Ejemplo de una ejecución secuencial*

Este tipo de ejecución se produce cuando hay dependencias de recursos entre tareas, y la **tarea 1** tiene mayor prioridad que la **tarea 2**(fig. 4.26).

- **Ejecución paralela:**

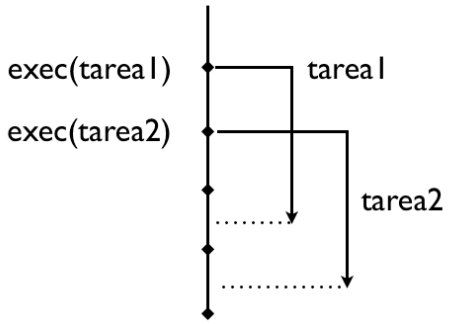


Figura 4.27: *Ejemplo de una ejecución paralela*

Este tipo de ejecución se produce cuando no hay dependencias de recursos entre tareas independientemente de la prioridad de cada una (fig. 4.27).

- **Ejecución interrumpida:**

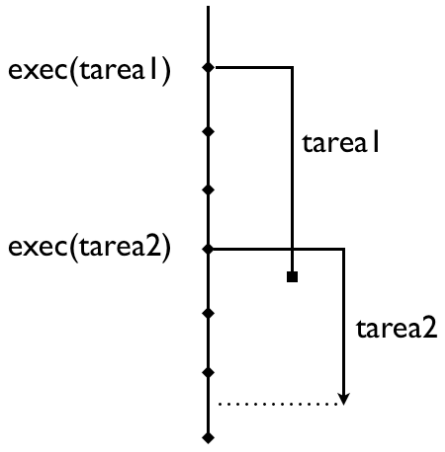


Figura 4.28: *Ejemplo de una ejecución interrumpida*

Este tipo de ejecución se produce cuando hay dependencias de recursos entre tareas, y la **tarea 2** tiene mayor prioridad que la **tarea 1**. La primera **tarea 1** es abortada, se liberan los recursos y se empieza a ejecutar la **tarea 2**.

Capítulo 5

Resultados

En este capítulo se van a presentar las pruebas realizadas para validar la aplicación. Para ello se ha realizado una batería de pruebas que verifiquen las distintas funcionalidades del *coordinador de tareas*. Las pruebas se realizarán por fases, para poder validar correctamente las funciones de *scheduling* del planificador. Finalmente, se realizará una batería de pruebas sobre la plataforma HOAP3 para ver cómo trabaja en una plataforma robótica real.

5.1. Preparación y configuración del entorno de ejecución

Para realizar las pruebas es necesario ejecutar la aplicación siguiendo estos pasos:

1. Iniciar yarpserver (fig. 5.1). En este paso se arranca el servidor de comunicaciones proporcionado por *YARP*. Para ello, sólo hay que ejecutar en una terminal el comando **yarpserver** ó **yarp server**.

```
hoap3@hoap3-dev:~/workspace$ yarp server
hoap3@hoap3-dev:~/workspace$
hoap3@hoap3-dev:~/workspace$ yarpserver
yarp: Port /root active at tcp://192.168.101.164:10000
```

```
yarp: Name server can be browsed at http://192.168.101.164:10000/
yarp: Bootstrap server listening at mcast://224.2.1.1:10001
```

Figura 5.1: *Inicio del servidor de comunicaciones YARP*

2. Iniciar el planificador. Arrancamos el planificador, que por defecto tiene la configuración mostrada en la figura 5.2 con los parámetros que creamos necesarios. Si ejecutamos la opción **-h** mostrará la ayuda, con los parámetros de configuración(fig. 5.3).

```
hoap3@hoap3-dev:~/workspace/Planificador$ ./out/linux/Planificador 2>
    planificador.log
Planificador de tareas
Autor: Miguel Maldonado
version 1.4

-----------------------------------------------------
Tamano del planificador: 5
Prioridad activada: ACTIVADO
Modo automatico activado: ACTIVADO
Peticiones cada 1 segundos
Forzar desalojo de tareas: ACTIVADO
Estadisticas activadas: ACTIVADO
Fichero de recursos: conf/resource.xml
Puerto de entrada del planificador /Planificador/in
Puerto de salida del planificador /Planificador/out
Puerto de comunicaciones del planificador /Planificador/com
-----------------------------------------------------
```

Figura 5.2: *Configuración por defecto del planificador*

```
hoap3@hoap3-dev:~/workspace/Planificador$ ./out/linux/Planificador -h
Planificador de tareas
Autor: Miguel Maldonado
version 1.4

-----------------------------------------------------
Parametros:
```

```
-h Muestra la ayuda
-p Desactiva la prioridad
-f Desactiva forzar el desalojo de tareas
-a Desactiva el modo automatico
-i Modifica el intervalo del modo automatico, ejemplo "-i 5" establece el
    timer en 5 segundos
-r Indica la localizaci n del fichero de recursos, ejemplo "-r pruebas/
    resource.xml"
-IN Indica el puerto de entrada al planificador, ejemplo "-IN /
    planificador/in "
-OUT Indica el puerto de salida del planificador, ejemplo "-OUT /
    planificador/out "
-s Desactiva las estadisticas
-t Indica el tama o del planificador, ejemplo "-t 5" establece el tama o
     del planificador en 5 colas
-----------------------------------------------------
```

Figura 5.3: *Información de ayuda del planificador*

3. Iniciar el Dispatcher (fig. 5.4).

```
hoap3@hoap3-dev:~/workspace/Dispatcher$ ./out/linux/Dispatcher 2>
    dispatcher.log
Prueba del Dispatcher
yarp: Port /Dispatcher/in active at tcp://192.168.101.164:10042
yarp: Port /Dispatcher/out active at tcp://192.168.101.164:10052
El dispatcher recibe peticiones en /Dispatcher/in
El dispatcher envia peticiones por /Dispatcher/out
yarp: Sending output from /Dispatcher/out to /Planificador/in using tcp
yarp: Receiving input from /Planificador/out to /Dispatcher/in using tcp
yarp: Port /Process active at tcp://192.168.101.164:10062
El proceso recibe peticiones en /Process
yarp: Sending output from /Process to /Dispatcher/in using tcp
Ready
Esperando peticiones
yarp: Receiving input from /Process to /Dispatcher/in using tcp
```

Figura 5.4: *Ejecución del dispatcher*

4. Iniciar la terminal (fig. 5.6).

```
hoap3@hoap3-dev:~/workspace/Terminal$ ./out/linux/Terminal 2> terminal.log
Prueba de la terminal para el planificador
Preparamos las comunicaciones
Bienvenido al menu principal:
1. Nueva Tarea
2. Cargar Lista de tareas
3. Eliminar Tarea en Ejecucion (Simulacion terminaci n de tarea)
4. Mostrar el historial de Tareas
5. Mostrar el historial de Tareas en Ejecuci n
6. Solicitar Tarea
0. Salir
Elija opci n:
```

Figura 5.5: *Ejecución de la terminal*

Cada módulo del *coordinador de tareas* incluye una interfaz de usuario que facilita la verificación de una correcta ejecución del programa:

- **Planificador:**
  Muestra la siguiente información:

  - **Mapa de recursos:** Muestra el estado de los recursos, si están en ejecución y que tarea los ha reservado (fig. 4.8).

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    1 (working)         5  xterm   3
1  ComandarPiernas   1 (working)         5  xterm   3
2  EncoderBrazoIz    1 (working)         5  xterm   3
3  EncoderBrazoDer   1 (working)        12  xterm   4
4  ComandarBrazoIzq  1 (working)        12  xterm   4
5  ComandarBrazoDer  1 (working)        12  xterm   4
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
```

```
12 Altavoz          0 (free)          0  null    99
13 Infrarrojo       0 (free)          0  null    99
```

Figura 5.6: *Información del mapa de recursos*

- **Mapa de Tareas:**
  Indica la tareas en ejecución (fig. 5.7).

```
Mapa de Tareas:
5 xterm -A -e ping -c 15 127.0.0.1 -R 0 1 2 -P 3
12 xterm -A -e ping -c 15 127.0.0.1 -R 3 4 5 -P 4
```

Figura 5.7: *Mapa de tareas: Muestra las tareas en ejecución*

- **Contenido del Planificador:**
  Muestra el contenido del planificador, el número de colas y las tareas asociadas a cada cola (fig. 5.8).

```
Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3
0 6 xterm -A -e ping -c 15 127.0.0.1 -R 0 1 2 -P 3
1 9 xterm -A -e ping -c 15 127.0.0.1 -R 10 11 12 -P 3
2 10 xterm -A -e ping -c 15 127.0.0.1 -R 0 1 2 -P 3
3 11 xterm -A -e ping -c 15 127.0.0.1 -R 0 1 2 -P 3
4 14 xterm -A -e ping -c 15 127.0.0.1 -R 10 11 12 -P 3
5 15 xterm -A -e ping -c 15 127.0.0.1 -R 0 1 2 -P 3

Cola con prioridad 4
0 17 xterm -A -e ping -c 15 127.0.0.1 -R 3 4 5 -P 4
1 22 xterm -A -e ping -c 15 127.0.0.1 -R 3 4 5 -P 4
2 27 xterm -A -e ping -c 15 127.0.0.1 -R 3 4 5 -P 4
3 3 xterm -A -e ping -c 15 127.0.0.1 -R 8 -P 6
```

```
4 8 xterm -A -e ping -c 15 127.0.0.1 -R 8 -P 6
```

Figura 5.8: *Contenido del Planificador: Muestra las tareas dentro de cada cola del planificador*

- **Estadísticas:**
  Estadísticas de la ejecución del planificador (fig. 5.9).

```
Estadisticas ejecucion
Total de tareas: 23
Total de tareas ejecutadas: 3
Total de tareas ejecutadas y no terminadas: 0
Total de fallos por falta de recursos: 11
Total de fallos por falta de prioridad: 0
Tiempo de ejecuci n 41.0166 segundos
```

Figura 5.9: *Ejemplo de las estadísticas de la terminal*

- **Dispatcher:**
  El dispatcher muestra la información de los procesos en ejecución y la información asociada a estos (fig. 5.10): tarea, PID padre, PID hijo, estado de la ejecución.

```
Mapa de Procesos (3):
0 0 Task-> 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 3 pidFather 7557
     pidChild 7558 StatusChild 0
1 0 Task-> 1 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 2 pidFather 7552
     pidChild 7553 StatusChild 0
2 0 Task-> 2 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 1 pidFather 7545
     pidChild 7546 StatusChild 0
```

Figura 5.10: *Información relativa a los procesos en ejecución.*

5.1.1. Ficheros de log

Si quiere ver la evolución del fichero del log del planificador, dispatcher o terminal en tiempo real, puede ejecutar el siguiente comando en una terminal

(fig. 5.11): **tailf -n 10 [fichero de log]**.

```
hoap3@hoap3-dev:~/workspace/Terminal$ tailf -n 10 terminal.log
yarp: Port /Terminal/in active at tcp://192.168.101.164:10072
yarp: Port /Terminal/out active at tcp://192.168.101.164:10082
yarp: Port /InternalTerminalCom active at tcp://192.168.101.164:10092
yarp: Sending output from /InternalTerminalCom to /Terminal/in using tcp
yarp: Receiving input from /InternalTerminalCom to /Terminal/in using tcp
```

Figura 5.11: *Seguimiento del fichero terminal.log*

5.2. Reglas de scheduling del planificador

En esta sección se van a probar las opciones principales del planificador y una pequeña muestra de las posibilidades.

5.2.1. Prioridad

El planificador tiene por defecto activada la función de cola de prioridad. Esta opción ordena las tareas por orden de prioridad, y tiene mayor prioridad que el orden de llegada. Se puede desactivar con la opción **-p** del planificador; al desactivarla la ordenación de las tareas es por orden de llegada. Cabe destacar, que esta opción sólo afecta a la cola de menor prioridad del planificador, ya que las tareas que no tengan una cola de prioridad asignada se guardan en la última cola. El resto de colas la prioridad es por orden de llegada.

Para realizar la prueba se ha utilizado el siguiente fichero de tareas (fig. 5.12). **NOTA IMPORTANTE:** Para realizar las pruebas de validación se ha utilizado el siguiente comando **xterm -e ping -c 50 127.0.0.1**, ya que permite ver fácilmente si se produce la ejecución de forma correcta, el número de tareas ejecutadas al mismo tiempo y cómo se van desalojando del procesador cuando entra una tarea más prioritaria.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
  <task>
    <name>xterm</name>
    <argument>
      <arg>-e</arg>
      <arg>ping</arg><arg>-c</arg><arg>50</arg>
      <arg>127.0.0.1</arg>
    </argument>
    <resources>
      <rec>0</rec><rec>1</rec><rec>2</rec>
    </resources>
    <priority>4</priority>
  </task>
  <task>
    <name>xterm</name>
    <argument>
      <arg>-e</arg>
      <arg>ping</arg><arg>-c</arg><arg>50</arg>
      <arg>127.0.0.1</arg>
    </argument>
    <resources>
      <rec>3</rec><rec>4</rec><rec>5</rec>
    </resources>
    <priority>6</priority>
  </task>
  <task>
    <name>xterm</name>
    <argument>
      <arg>-e</arg>
      <arg>ping</arg><arg>-c</arg><arg>50</arg>
      <arg>127.0.0.1</arg>
    </argument>
    <resources>
      <rec>0</rec><rec>1</rec><rec>2</rec>
    </resources>
    <priority>4</priority>
  </task>
  <task>
    <name>xterm</name>
```

```
42      <argument>
43        <arg>-e</arg>
44        <arg>ping</arg><arg>-c</arg><arg>50</arg>
45        <arg>127.0.0.1</arg>
46      </argument>
47      <resources>
48        <rec>3</rec><rec>4</rec><rec>5</rec>
49      </resources>
50      <priority>6</priority>
51    </task>
52    <task>
53      <name>xterm</name>
54      <argument>
55        <arg>-e</arg>
56        <arg>ping</arg><arg>-c</arg><arg>50</arg>
57        <arg>127.0.0.1</arg>
58      </argument>
59      <resources>
60        <rec>6</rec><rec>7</rec><rec>8</rec>
61      </resources>
62      <priority>0</priority>
63    </task>
64  </TaskFile>
```

Figura 5.12: *Fichero utilizado para las pruebas*

En la fig. 5.13 se muestra un ejemplo con la prioridad activada (por defecto).

```
Contenido de Planificador

Cola con prioridad 0
0 4 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 0
1 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
2 2 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
3 1 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
4 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
```

Figura 5.13: *Planificador con la prioridad activada*

En la fig. 5.14 se muestra un ejemplo con la prioridad desactivada (por defecto).

```
Mapa de Tareas:

Contenido de Planificador

Cola con prioridad 0
0 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
1 1 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
2 2 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
3 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
4 4 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 0
```

Figura 5.14: *Planificador con la prioridad activada*

El resultado obtenido en la fig. 5.14 se observa cómo las tareas están ordenadas por orden de llegada (el orden de envío se puede ver en la fig. 5.12). En la fig. 5.14 las tareas están ordenadas por el campo prioridad de cada tarea, siendo la tarea 0 la más prioritaria.

5.2.2. Forzar ejecución

La regla **Forzar ejecución** viene activada por defecto, esta regla fuerza el desalojo de tareas cuando los recursos que han reservado estas, son requeridas por una tarea más prioritaria. Si la regla esta desactivada (**-f**), el planificador sólo notifica que es necesario desalojar las tareas que tienen reservados los recursos necesarios para la tarea más prioritaria.

5.2.3. Tamaño del planificador

El tamaño del planificador por defecto es 5; es decir, tiene 5 colas de prioridad. Se puede modificar el tamaño del planificador mediante el parámetro **-t**. Para una misma batería de pruebas, se obtienen distintos resultados de ejecución al modificar el tamaño del planificador.

Utilizando la siguiente batería de pruebas (fig. 5.15):

```
Historial de tareas:
1) 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
2) 1 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
3) 2 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
4) 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
5) 4 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1
6) 5 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 2
7) 6 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 8
8) 7 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3
9) 8 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1
10) 9 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
11) 10 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
12) 11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
13) 12 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
14) 13 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
15) 14 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
16) 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
17) 16 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 0
```

Figura 5.15: *Bateria de pruebas 2*

El orden de ejecución de las tareas depende tanto del tamaño del planificador, cómo de prioridad asociada a cada tarea [1]. Para un tamaño N y con la batería de pruebas de la fig. 5.15 el estado del planificador sería:

- **Tamaño 1:**

```
Cola con prioridad 0
0 16 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 0
1 4 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1
2 8 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1
3 5 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 2
4 7 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3
5 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
```

[1] Se puede modificar el orden de ejecución desactivando más opciones del planificador. Para más información fig. 5.3.

```
6 2 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
7 10 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
8 12 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
9 14 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
10 1 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
11 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
12 9 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
13 11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
14 13 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
15 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
16 6 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 8
```

Figura 5.16: *Orden de ejecución planificador tamaño 1*

- **Tamaño 4:**

```
Contenido de Planificador

Cola con prioridad 0
0 16 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 0

Cola con prioridad 1
0 4 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1
1 8 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1

Cola con prioridad 2
0 5 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 2

Cola con prioridad 3
0 7 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3
1 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
2 2 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
3 10 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
4 12 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
5 14 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
6 1 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
7 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
8 9 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
9 11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
```

```
10 13 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
11 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
12 6 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 8
```

Figura 5.17: *Orden de ejecución planificador tamaño 4*

- **Tamaño 5:**

```
Contenido de Planificador

Cola con prioridad 0
0 16 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 0

Cola con prioridad 1
0 4 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1
1 8 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1

Cola con prioridad 2
0 5 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 2

Cola con prioridad 3
0 7 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3

Cola con prioridad 4
0 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
1 2 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
2 10 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
3 12 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
4 14 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
5 1 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
6 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
7 9 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
8 11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
9 13 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
10 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
11 6 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 8
```

Figura 5.18: *Orden de ejecución planificador tamaño 5*

- **Tamaño 7:**

```
Contenido de Planificador

Cola con prioridad 0
0 16 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 0

Cola con prioridad 1
0 4 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1
1 8 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 1

Cola con prioridad 2
0 5 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 2

Cola con prioridad 3
0 7 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3

Cola con prioridad 4
0 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
1 2 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
2 10 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
3 12 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
4 14 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4

Cola con prioridad 5

Cola con prioridad 6
0 1 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
1 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
2 9 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
3 11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
4 13 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
5 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
6 6 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 8
```

Figura 5.19: *Orden de ejecución planificador tamaño 7*

Como se puede apreciar en las figuras 5.17, 5.18, 5.19 respecto a la figura 5.16, el orden de ejecución no varía para instrucciones que necesitan los mismos

recursos; se empezaría buscando instrucciones a ejecutar por la cola de ejecución 0 (más prioritaria), y continuando por la $1, 2...N$, se ejecutarían en orden una a una, con lo que no tendríamos ninguna diferencia respecto a usar sólo una estructura FIFO.

Entonces, ¿Cúal es la ventaja de usar el planificador? la ventaja se encuentra en el uso combinado de recursos y prioridades. Tareas que usen los mismo recursos, se ejecutarían de forma secuencial. Pero por lo general, en el planificador habra distintos tipos de tareas: movimientos, manipulación, visión, ... por lo que tendrán asociados distintos recursos dependiendo de la naturaleza de la tarea. He aquí cuando destaca el uso del Planificador. En la figura 5.20 se ve la ejecución de tareas con un uso diferente de los recursos, en esta ejecución se podrían ejecutar desde el primer instante 6 tareas al mismo tiempo. El planificador empezaría a ejecutar las tareas de la cola 0, hasta que no pueda ejecutar más por necesidad de recursos, por lo que comenzaría a buscar tareas en la cola 1, ejecutaría todas las tareas hasta que se bloqueara por falta de recursos, continuaría con la cola 2 y así sucesivamente. En la figura 5.18, se observa que el número de tareas máximo a ejecutar de forma paralela seria 3, debido a la configuración de recursos.

```
Contenido de Planificador

Cola con prioridad 0
0 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 0
1 1 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 -P 0
2 12 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 0
3 16 xterm -A -e ping -c 50 127.0.0.1 -R 6 7 8 -P 0

Cola con prioridad 1
0 2 xterm -A -e ping -c 50 127.0.0.1 -R 3 -P 1
1 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 -P 1

Cola con prioridad 2
0 4 xterm -A -e ping -c 50 127.0.0.1 -R 5 -P 2
1 5 xterm -A -e ping -c 50 127.0.0.1 -R 6 -P 2
```

```
Cola con prioridad 3
0 6 xterm -A -e ping -c 50 127.0.0.1 -R 7 8 9 -P 3
1 7 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3

Cola con prioridad 4
0 8 xterm -A -e ping -c 50 127.0.0.1 -R 1 4 2 -P 4
1 10 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
2 14 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 4
3 9 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
4 11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
5 13 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
6 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 6
```

Figura 5.20: *Orden de ejecución planificador tamaño 5 con batería de pruebas 3*

Hay que tener en cuenta que estos resultados son experimentales, en un robot real las tareas no se cargarían desde el principio en el planificador, irían llegando según se fueran emitiendo por el planificador de alto nivel o por otros módulos; por ejemplo un control de estabilidad, que estaría emitiendo continuamente tareas para que el robot no se cayera al suelo. Por lo que habría que probar para robot cual sería la mejor configuración a elegir: tamaño del planificador y políticas de scheduling.

5.3. Simulación de ejecución

En esta sección se va a mostrar una prueba completa en el entorno de simulación. Para las pruebas vamos a utilizar la siguiente batería de pruebas (fig. 5.21).

```
Historial de tareas:
1) 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 0
2) 1 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 -P 0
3) 2 xterm -A -e ping -c 50 127.0.0.1 -R 3 -P 1
```

```
4) 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 -P 1
5) 4 xterm -A -e ping -c 50 127.0.0.1 -R 5 -P 2
6) 5 xterm -A -e ping -c 50 127.0.0.1 -R 6 -P 2
7) 6 xterm -A -e ping -c 50 127.0.0.1 -R 7 8 9 -P 3
8) 7 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3
```

Figura 5.21: *Batería de pruebas número 4*

- **Preparación del entorno de ejecución:**
  En la fig. 5.31 observamos la preparación del entorno de ejecución. En la fig. 5.22 se observan las instrucciones cargadas ya en el planificador y el orden de ejecución.

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    0 (free)            0  null    99
1  ComandarPiernas   0 (free)            0  null    99
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:

Contenido de Planificador

Cola con prioridad 0
0 8 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 0
1 9 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 -P 0

Cola con prioridad 1
```

```
0 10 xterm -A -e ping -c 50 127.0.0.1 -R 3 -P 1
1 11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 -P 1

Cola con prioridad 2
0 12 xterm -A -e ping -c 50 127.0.0.1 -R 5 -P 2
1 13 xterm -A -e ping -c 50 127.0.0.1 -R 6 -P 2

Cola con prioridad 3
0 14 xterm -A -e ping -c 50 127.0.0.1 -R 7 8 9 -P 3
1 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3

Cola con prioridad 4
```

Figura 5.22: *Tareas cargadas en el planificador*

```
Mapa de Procesos (0):
```

Figura 5.23: *Contenido del mapa de procesos del dispatcher (Preparación del entorno de ejecución).*

- **Comienzo de la ejecución:**

  En la fig. 5.32 se observa cómo el planificador ejecuta el mayor número de tareas posible hasta que se queda sin recursos suficientes para ejecutar las demás. En la fig. 5.24 se muestra el estado del planificador, el **mapa de recursos** muestra cómo se han reservado los recursos necesarios para cada tarea. En el **mapa de tareas** las tareas en ejecución y por último las tareas restantes en el planificador.

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    1 (working)         8  xterm    0
1  ComandarPiernas   1 (working)         8  xterm    0
2  EncoderBrazoIz    1 (working)         8  xterm    0
3  EncoderBrazoDer   1 (working)        10  xterm    1
4  ComandarBrazoIzq  0 (free)           0  null    99
5  ComandarBrazoDer  1 (working)        12  xterm    2
```

```
6  Camaras           1 (working)       13  xterm    2
7  Acelerometros     1 (working)       14  xterm    3
8  Giroscopios       1 (working)       14  xterm    3
9  FSR_Piernas       1 (working)       14  xterm    3
10 FSR_Brazos        0 (free)           0  null    99
11 Microfono         0 (free)           0  null    99
12 Altavoz           0 (free)           0  null    99
13 Infrarrojo        0 (free)           0  null    99


Mapa de Tareas:
8 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 0
10 xterm -A -e ping -c 50 127.0.0.1 -R 3 -P 1
12 xterm -A -e ping -c 50 127.0.0.1 -R 5 -P 2
13 xterm -A -e ping -c 50 127.0.0.1 -R 6 -P 2
14 xterm -A -e ping -c 50 127.0.0.1 -R 7 8 9 -P 3

Contenido de Planificador

Cola con prioridad 0
0 9 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 -P 0

Cola con prioridad 1
0 11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 -P 1

Cola con prioridad 2

Cola con prioridad 3
0 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3

Cola con prioridad 4
```

Figura 5.24: *El planificador emite el mayor número de tareas posible para su ejecución.*

```
Mapa de Procesos (5):
0 0 Task-> 0 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 2 -P 0 pidFather 4336
     pidChild 4337 StatusChild 0
2 0 Task-> 2 xterm -A -e ping -c 50 127.0.0.1 -R 3 -P 1 pidFather 4343
    pidChild 4344 StatusChild 0
4 0 Task-> 4 xterm -A -e ping -c 50 127.0.0.1 -R 5 -P 2 pidFather 4348
    pidChild 4349 StatusChild 0
```

```
5 0 Task-> 5 xterm -A -e ping -c 50 127.0.0.1 -R 6 -P 2 pidFather 4353
    pidChild 4354 StatusChild 0
6 0 Task-> 6 xterm -A -e ping -c 50 127.0.0.1 -R 7 8 9 -P 3 pidFather 4358
     pidChild 4359 StatusChild 0
```

Figura 5.25: *Contenido del mapa de procesos del dispatcher (Comienzo de la ejecución).*

- **Emisión de nuevas tareas:**
  Según se van liberando recursos por la terminación de tareas el planificador lanza nuevas (fig. 5.33) tareas que tengan recursos disponibles para poder ejecutarse (fig. 5.26).

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    1 (working)         9  xterm    0
1  ComandarPiernas   1 (working)         9  xterm    0
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   1 (working)        11  xterm    1
4  ComandarBrazoIzq  1 (working)        11  xterm    1
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99


Mapa de Tareas:
9 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 -P 0
11 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 -P 1

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1
```

```
Cola con prioridad 2

Cola con prioridad 3
0 15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3

Cola con prioridad 4
```

Figura 5.26: *El planificador lanza nuevas tareas según se van liberando recursos.*

```
Mapa de Procesos (2):
1 0 Task-> 1 xterm -A -e ping -c 50 127.0.0.1 -R 0 1 -P 0 pidFather 4371
    pidChild 4372 StatusChild 0
3 0 Task-> 3 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 -P 1 pidFather 4376
    pidChild 4377 StatusChild 0
```

Figura 5.27: *Contenido del mapa de procesos del dispatcher (Emisión de nuevas tareas).*

- **Liberación de recursos:**
  Una vez terminada la ejecución de las tareas en curso se va liberando los recursos asociados (fig. 5.28 y 5.34).

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    0 (free)            0  null    99
1  ComandarPiernas   0 (free)            0  null    99
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   1 (working)        15  xterm    3
4  ComandarBrazoIzq  1 (working)        15  xterm    3
5  ComandarBrazoDer  1 (working)        15  xterm    3
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99
```

```
Mapa de Tareas:
15 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3

Cola con prioridad 4
```

Figura 5.28: *Última tarea de la batería 4 y liberación de recursos.*

```
Mapa de Procesos (1):
7 0 Task-> 7 xterm -A -e ping -c 50 127.0.0.1 -R 3 4 5 -P 3 pidFather 4390
     pidChild 4391 StatusChild 0
```

Figura 5.29: *Contenido del mapa de procesos del dispatcher (Liberación de recursos).*

- **Estadísticas:**

  Finalmente se obtienen las estadísticas de la ejecución.

```
Estadisticas ejecucion
Total de tareas: 8
Total de tareas ejecutadas: 8
Total de tareas ejecutadas y no terminadas: 0
Total de fallos por falta de recursos: 18
Total de fallos por falta de prioridad: 0
Tiempo de ejecuci n 186.945 segundos
```

Figura 5.30: *Estadísticas de la ejecución de la batería de pruebas 4*

En la fig. 5.30 se observan los siguientes datos.

- **Total de tareas: 8** Se han enviado al planificador 8 tareas.
- **Total de tareas ejecutadas: 8** Se han ejecutado 8 tareas.
- **Total de tareas ejecutadas y no terminadas: 0** No se han producido desalojos durante la ejecución. Al no llegar tareas nuevas con más prioridad durante la ejecución este resultado es correcto.
- **Total de fallos por falta de recursos: 18** Es el número de fallos obtenidos al intentar lanzar una tarea. Este valor incrementa cada vez que el planificador intenta lanzar una tarea y no hay recursos para ejecutarla.
- **Total de fallos por falta de prioridad: 0** No se han producido desalojos durante la ejecución. Al no llegar tareas nuevas con más prioridad durante la ejecución este resultado es correcto.
- **Tiempo de ejecución 186.945 segundos** Tiempo que lleva el planificador en ejecución.

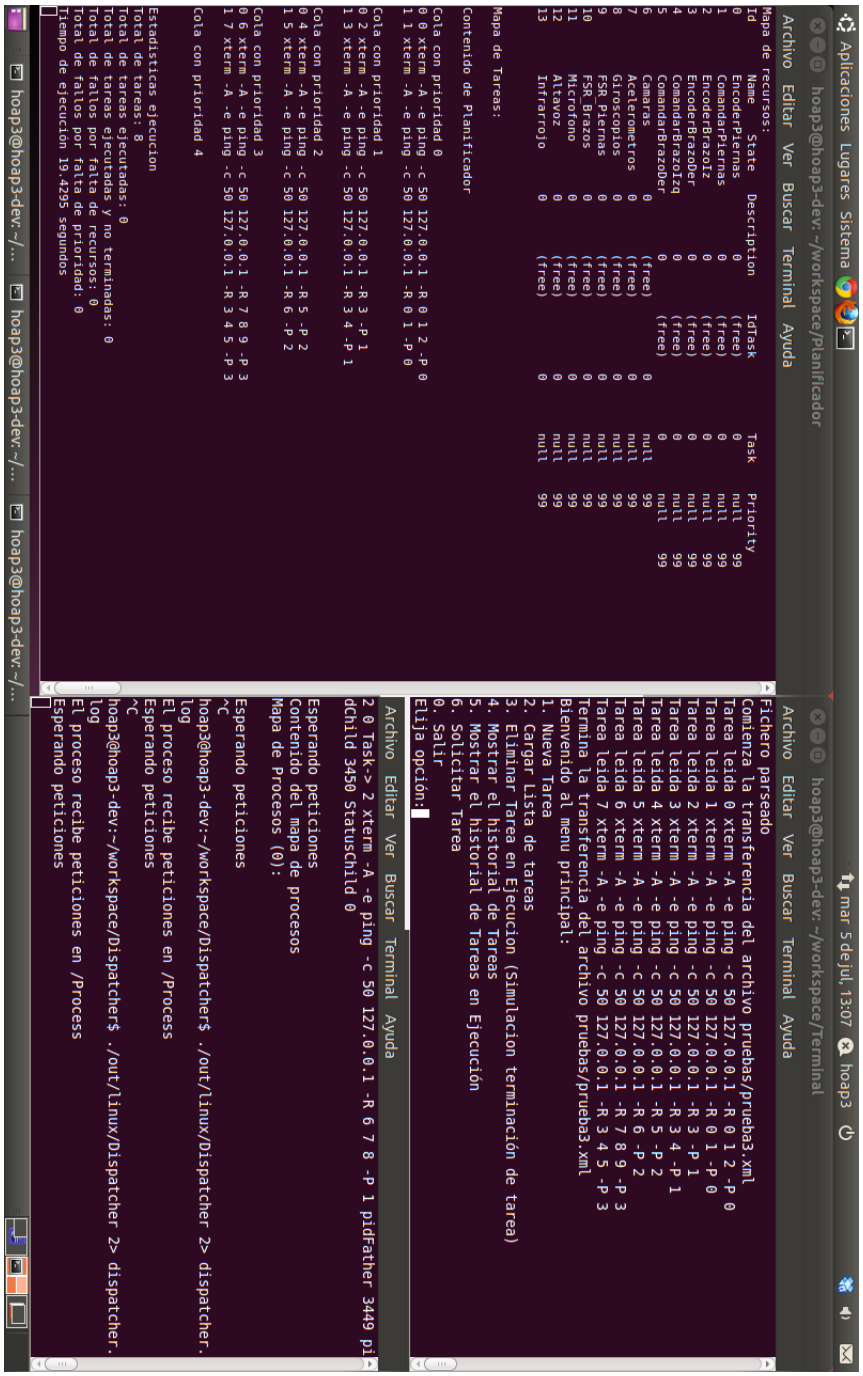


Figura 5.31: *Preparación del entorno de ejecución*

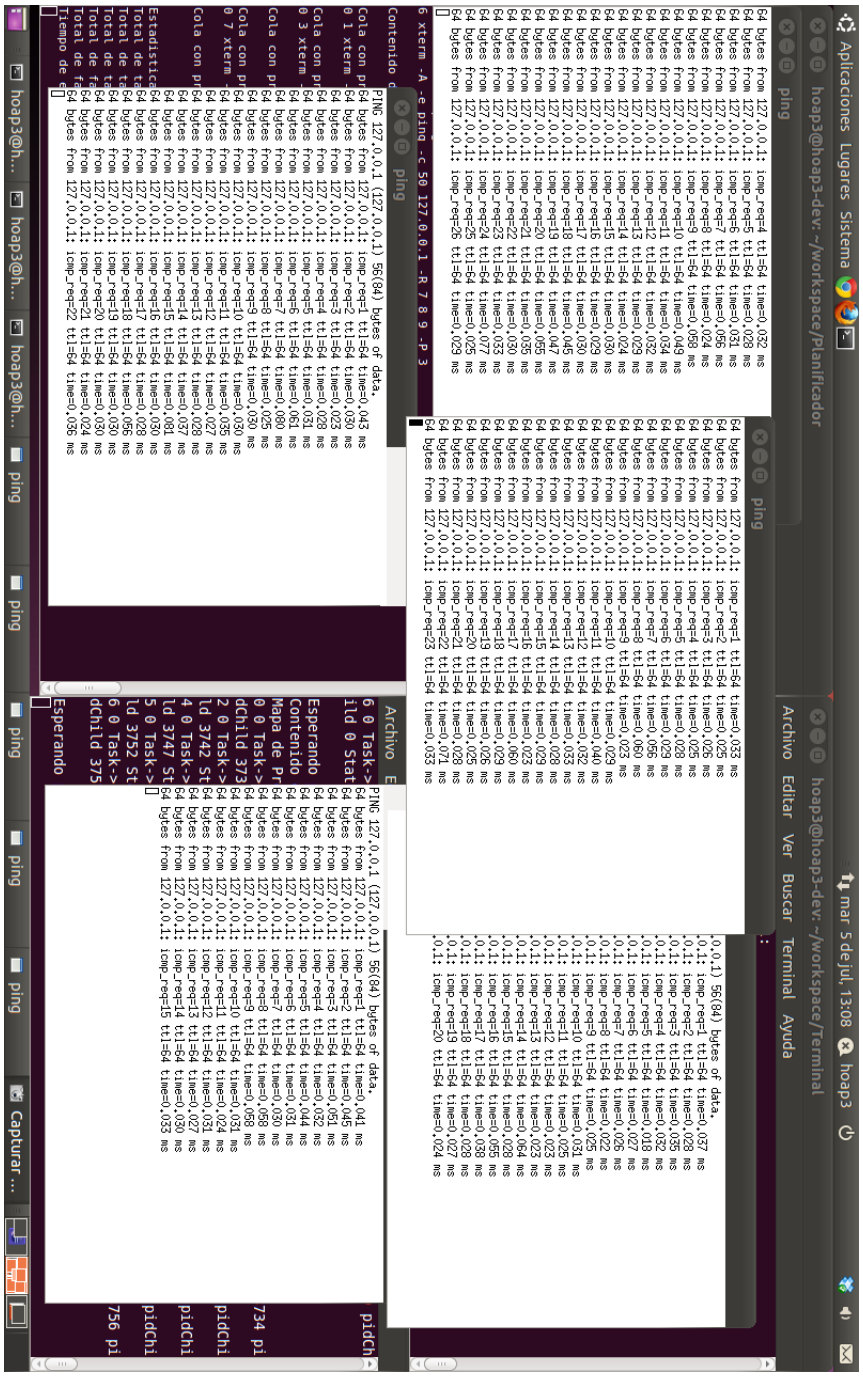


Figura 5.32: *Comienzo de la ejecución*

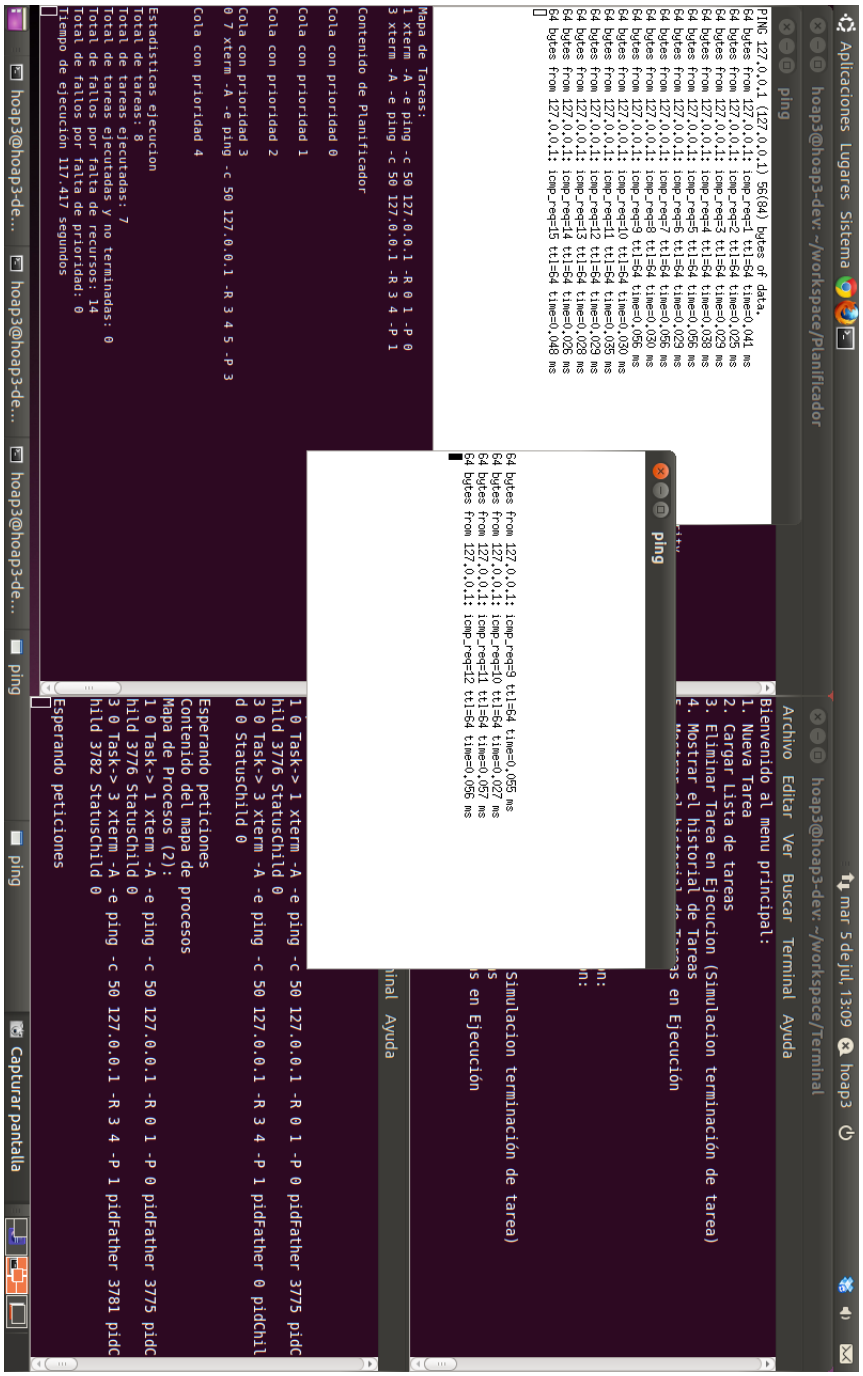


Figura 5.33: *Emisión de nuevas tareas*

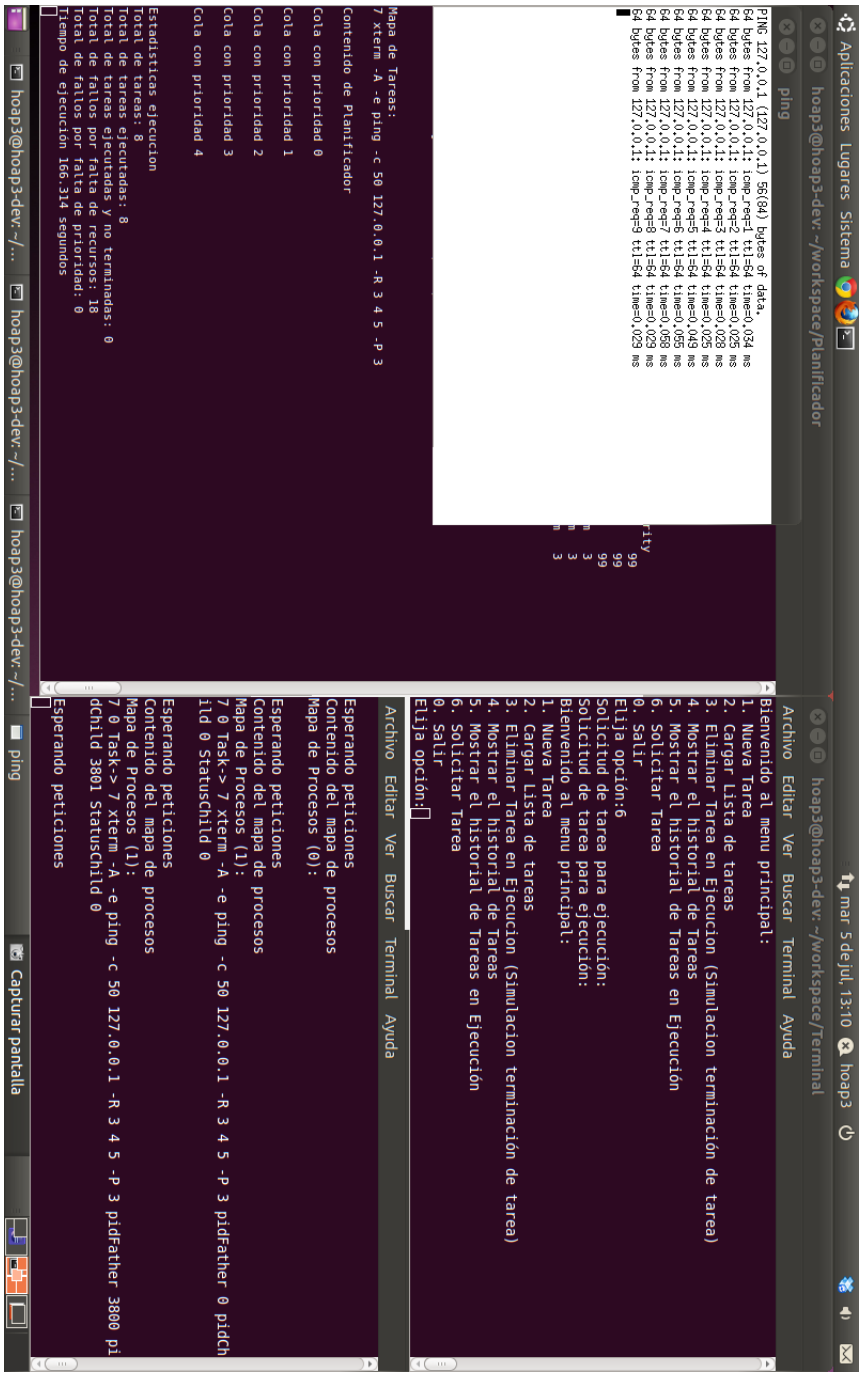


Figura 5.34: *Liberación de recursos*

5.4. Pruebas sobre la plataforma HOAP3

En esta sección se describe las pruebas realizadas sobre la plataforma robótica HOAP3.

5.4.1. El robot humanoide HOAP3

HOAP3 (Humanoid for Open Architecture Platform) (fig. 5.35) es la tercera versión de la plataforma HOAP, desarrollado por la compañía Fujitsu.

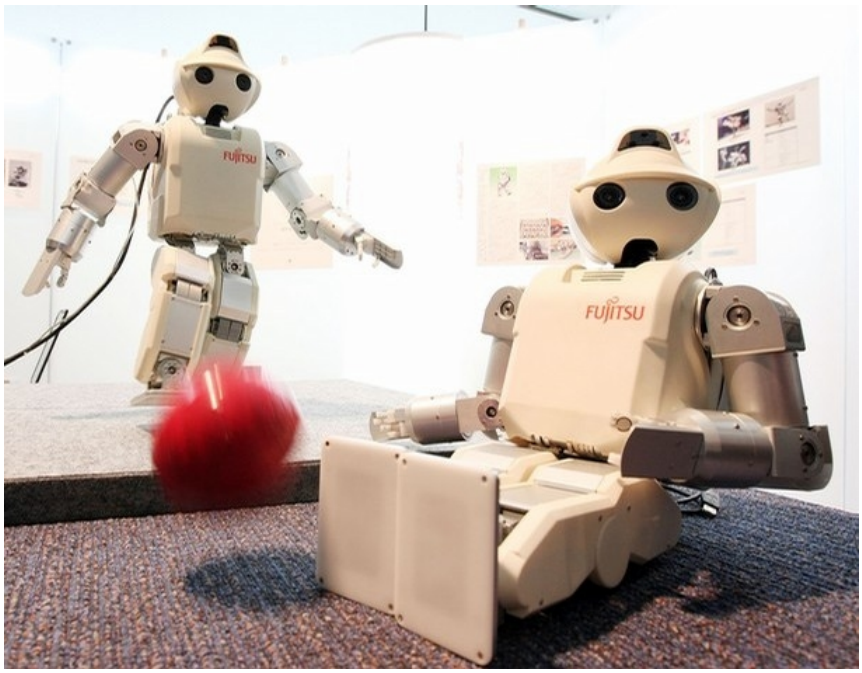


Figura 5.35: *Robot humanoide HOAP3*

5.4.1.1. Características:

- Robot humanoide.
- Tamaño y peso medio; 60 cm y 9 kg aproximadamente.
- Diseñado y fabricado por Fujitsu.

- 28 grados de libertad distribuidos cómo se puede observar en la figura 5.36. Repartidos de la siguiente forma:

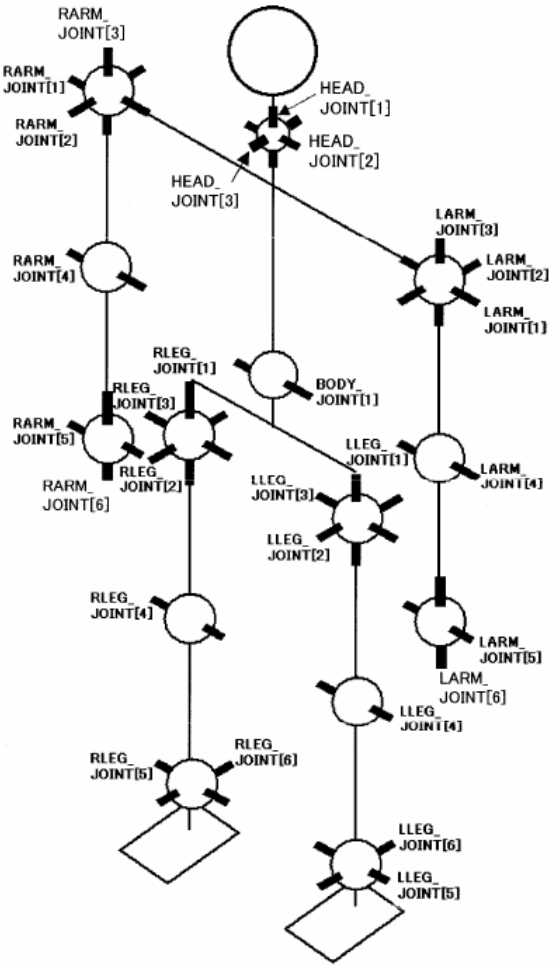


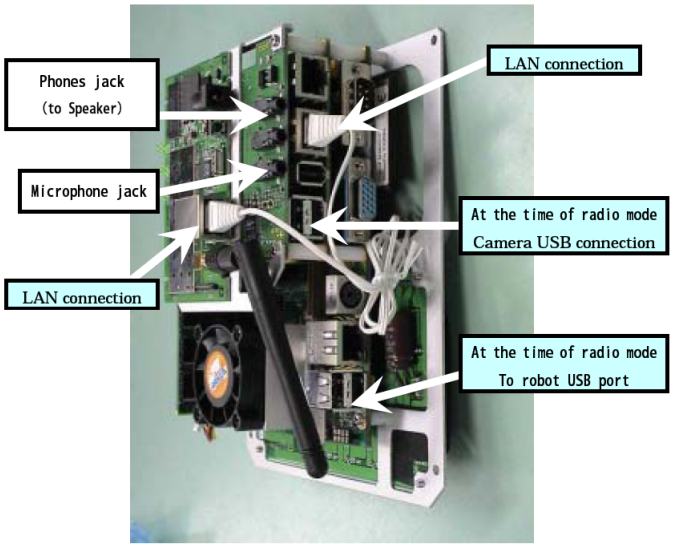
Figura 5.36: *Grados de libertad del robot HOAP3*

- 6 grados de libertad en cada pierna.
- 6 en cada brazo.
- 3 en la cabeza.
- 1 en la cadera.

Aunque posee 28 grados de libertad, dispone solamente de 23 motores. Los 21 primeros controlan piernas y brazos según se muestra en la fig. 5.36, de estos motores el fabricante no da las especificaciones técnicas aunque sí proporciona una tabla con los pares nominales y máximos que puede soportar cada motor. Estos motores disponen de encoders relativos y existe la posibilidad de controlarlos en posición y en velocidad.

Los otros dos motores, el 22 y 23, no disponen de encoder cómo el resto. El motor 22 es el encargado de controlar los 3 grados de libertad de la cabeza (pitch, roll y yaw). El motor 23 controla la rotación de las dos manos y el agarre de las manos.

- El robot lleva incorporado un PC-104 embebido en su interior, exactamente en la parte de atrás (fig. 5.37). Se trata de un procesador con una arquitectura compatible con Pentium de 1.1 Ghz con 512 Mb de RAM y una memoria Compact Flash de 1 Gb. Posee conexión Wifi IEEE802.11g y 4 puertos USB.



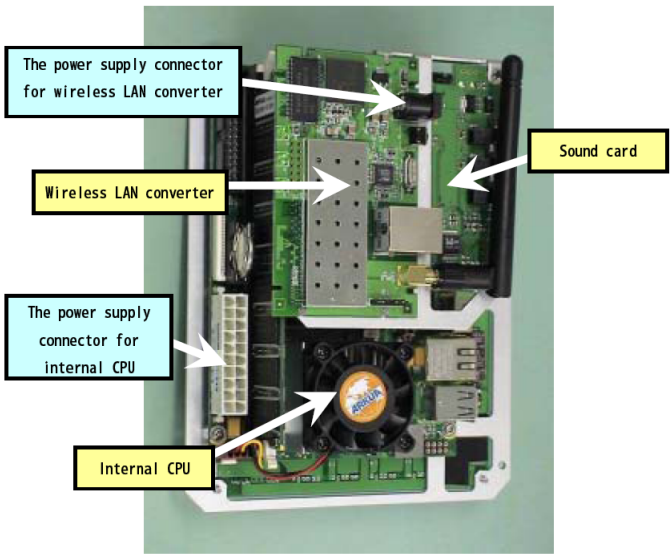


Figura 5.37: *PC embebido*

- El sistema operativo que incorpora el robot es un Linux en tiempo real, basado en Fedora Core 1 con el kernel 2.4. Es importante destacar que sin el parche de tiempo real no se pueden controlar los servos del robot.

Con el robot, el fabricante proporciona un ordenador que es un clon exacto del que lleva incorporado el HOAP, con el mismo hardware y software instalado. Para controlar al robot existen dos métodos de conexión, el primero es a través de una conexión inalámbrica, por telnet, gracias a un router que también proporciona el fabricante, la segunda opción es una conexión directa por medio de un cable usb que conecta el pc externo directamente con los motores y los drivers del robot. Además, el robot se puede alimentar directamente por cable o mediante una batería de NiMH de 24V, esto último ofrece la ventaja de una gran versatilidad en el movimiento.

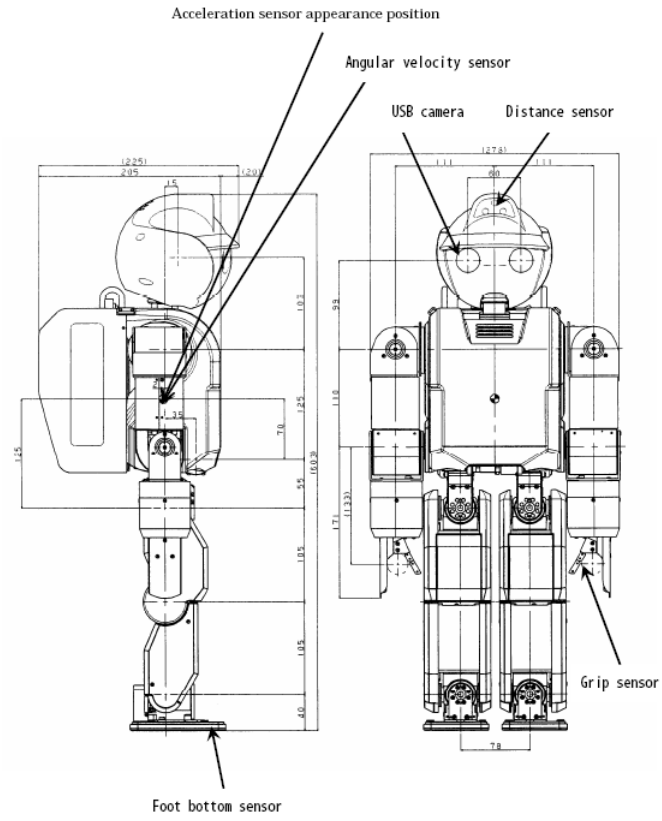


Figura 5.38: *Dimensiones y sensores del HOAP3*

Para completar las funcionalidades de este humanoide, se han añadido un conjunto de sensores que le permiten desenvolverse con naturalidad en cualquier entorno. En la Figura 5.38 se observan las dimensiones del robot en milímetros junto con algunos sensores que incorpora. El robot dispone de dos cámaras para visión estereoscópica, un micrófono, un altavoz, sensores infrarrojos de distancia, sensores de fuerza FSR en pies y manos y giróscopos y acelerómetros en los 3 ejes.

No cabe duda que el humanoide HOAP3 es una plataforma muy completa y versátil, que reúne todas las características necesarias para desarrollar cualquier tipo de investigación en robótica. De hecho, en la actualidad, existen multitud

de centros de investigación, tanto en Europa cómo en el resto del mundo, trabajando con este robot.

5.4.2. Recursos del HOAP3

Para las pruebas se ha creado un fichero de recursos especifico para el HOAP3[2]. El contenido del fichero se encuentra en el Apéndice B.

Los recursos disponibles en el HOAP3 son:

- **EncoderPiernas:**
  Sensores encargados de determinar la posición de los motores de las piernas.
- **ComandarPiernas:**
  Motores de las piernas.
- **EncoderBrazoIz:**
  Sensores encargados de determinar la posición de los motores del brazo izquierdo.
- **EncoderBrazoDer:**
  Sensores encargados de determinar la posición de los motores del brazo derecho.
- **ComandarBrazoIzq:**
  Motores del brazo izquierda.
- **ComandarBrazoDer:**
  Motores del brazo derecho.
- **Camaras:**
  Camaras situadas en la cabeza del robot.

[2]Para más información ver el capitulo 4, sección 4.2.3.3. Fichero de inicialización.

- **Acelerómetros:**
  Sensores encargados de medir las aceleraciones del robot.
- **Giroscopios:**
  Sensores encargados de medir el ángulo de giro del robot.
- **FSR Piernas:**
  FSR (Sensor de fuerza resistivo [3]) situados en las piernas.
- **FSR Brazos:**
  FSR (Sensor de fuerza resistivo) situados en los brazos.
- **Micrófono:**
  Micrófono situado en la cabeza.
- **Altavoz:**
  Altavoz situado en la mochila.
- **Infrarrojo:**
  Sensor infrarrojo situado en la cabeza

5.4.3. Batería de pruebas

La batería de pruebas consta de tres fases, que comprueban los tipos de ejecución disponibles en el coordinador (ver sección 4.3.5.1). Cada comando del robot HOAP3 utilizado en las pruebas se ha introducido dentro de una aplicación *C* muy simple (fig. 5.39), esta aplicación lo único que hace es pintar por pantalla *-nombre del comando 1-*, ejecuta el comando y pinta por pantalla *-nombre del comando 2-*. De esta forma tan simple permite seguir la ejecución de cada comando, además de permitir documentar la ejecución de una forma más ”gráfica”. Cada fase comprueba un tipo de ejecución.
NOTA: No es necesaria esta aplicación para ejecutar comandos en el robot, pero permite seguir la ejecución a la hora de realizar las pruebas.

[3] Miden los cambios de presión sobre el sensor.

```c
#include <stdio.h>
#include <unistd.h>

int main(){
  printf("comando 1\n");
  execv("comando",argumentos);
  printf("comando 2\n");
  return 0;
}
```

Figura 5.39: *Ejemplo de la aplicación C utilizada para envolver los comandos del robot*

5.4.3.1. Ejecución secuencial

La ejecución secuencial ejecuta un comando después de la terminación de otro, debido a que tienen recursos compartidos. En la fig. 5.40 se encuentran las dos tareas a ejecutar: *walk* y *great*. Al observar la lista de recursos, se ve que ambas comparten recursos y que *walk* tiene mayor prioridad (prioridad 3) que *great* (prioridad 4); lo que es de esperar es que se ejecuten secuencialmente. La ejecución y los resultados de esta comprobarán que el planificador reservará correctamente los recursos, con lo que impedirá la ejecución paralela de ambas tareas.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
        <task>
                <name>walk</name>
                <argument>
                </argument>
                <resources>
                        <rec>0</rec>
                        <rec>1</rec>
                        <rec>2</rec>
                </resources>
                <priority>3</priority>
```

```
14          </task>
15          <task>
16                  <name>great</name>
17                  <argument>
18                  </argument>
19                  <resources>
20                          <rec>0</rec>
21                          <rec>1</rec>
22                  </resources>
23                  <priority>4</priority>
24          </task>
25  </TaskFile>
```

Figura 5.40: *Batería de pruebas 1 HOAP3 (walk y great)*

La ejecución es la siguiente:

- **Carga de instrucciones:**
  Cargamos las instrucciones en el planificador.

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    0 (free)            0  null    99
1  ComandarPiernas   0 (free)            0  null    99
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:

Contenido de Planificador
```

```
Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3
0 0 walk -A -R 0 1 2 -P 3

Cola con prioridad 4
0 1 great -A -R 0 1 -P 4
```

Figura 5.41: *Tareas cargadas en el planificador (Ejecución secuencial)*

- **Ejecución del comando walk:**
  Ejecutamos el comando *walk*. El comando *great* se queda a la espera por falta de recursos.

Figura 5.42: *Hoap ejecutando comando walk (andar)*

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    1 (working)         0  walk     3
1  ComandarPiernas   1 (working)         0  walk     3
2  EncoderBrazoIz    1 (working)         0  walk     3
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:
0 walk -A -R 0 1 2 -P 3

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3

Cola con prioridad 4
0 1 great -A -R 0 1 -P 4
```

Figura 5.43: *Ejecución de la tarea walk (Ejecución secuencial)*

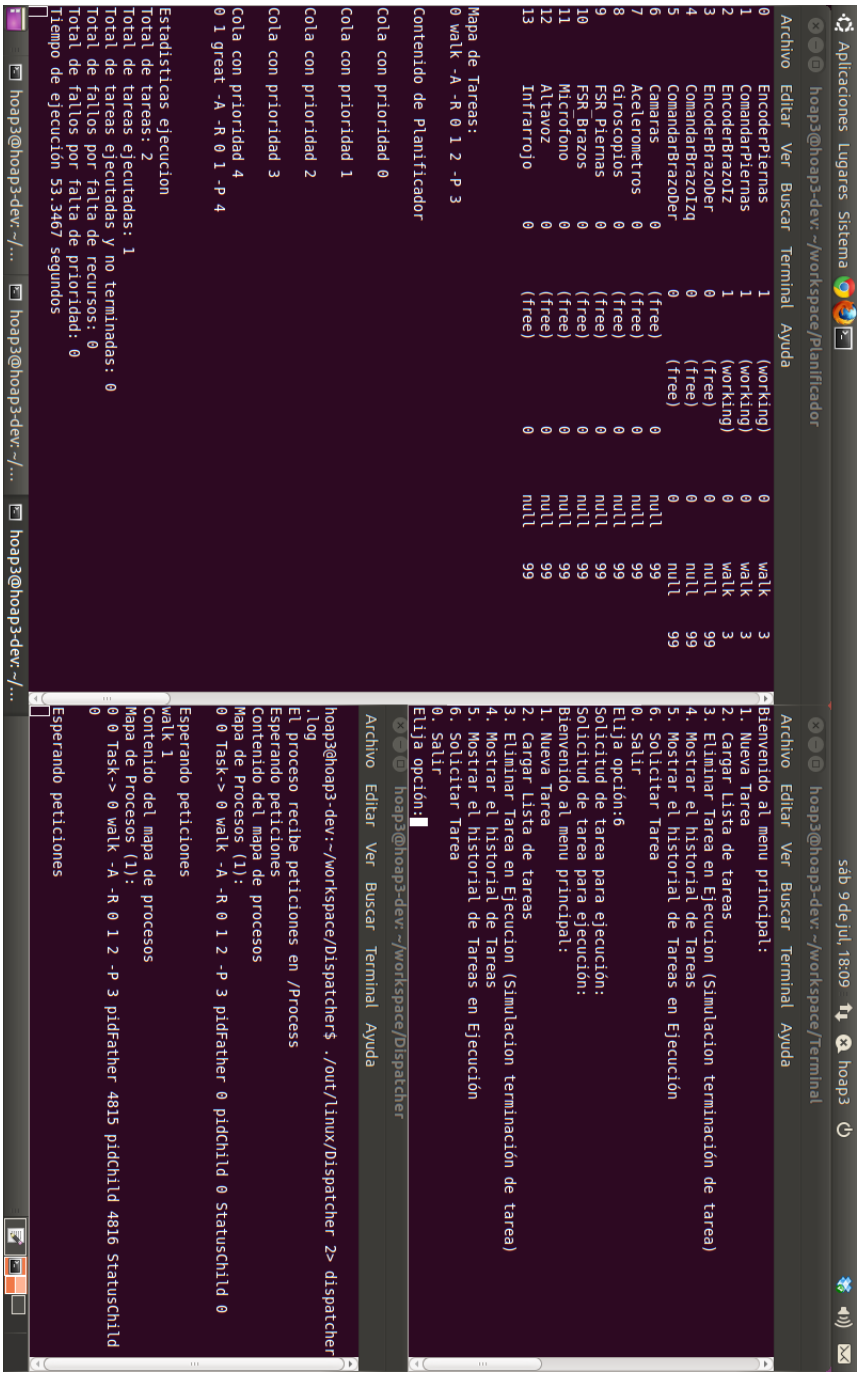


Figura 5.44: *Ejecución de la tarea walk (captura de pantalla) (Ejecución secuencial)*

- **Ejecución del comando great:**
  Cuando el comando *walk* termina libera los recursos y comienza la ejecución del comando *great*.

Figura 5.45: *Hoap ejecutando comando great (saludo)*

```
Mapa de recursos:
Id      Name          State/Description   IdTask/Task Priority
0  EncoderPiernas     1 (working)         0  great     4
1  ComandarPiernas    1 (working)         0  great     4
2  EncoderBrazoIz     0 (free)            0  null     99
3  EncoderBrazoDer    0 (free)            0  null     99
```

```
4  ComandarBrazoIzq  0 (free)          0  null    99
5  ComandarBrazoDer  0 (free)          0  null    99
6  Camaras           0 (free)          0  null    99
7  Acelerometros     0 (free)          0  null    99
8  Giroscopios       0 (free)          0  null    99
9  FSR_Piernas       0 (free)          0  null    99
10 FSR_Brazos        0 (free)          0  null    99
11 Microfono         0 (free)          0  null    99
12 Altavoz           0 (free)          0  null    99
13 Infrarrojo        0 (free)          0  null    99

Mapa de Tareas:
1 great -A -R 0 1 -P 4

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3

Cola con prioridad 4
```

Figura 5.46: *Ejecución de la tarea great (Ejecución secuencial)*

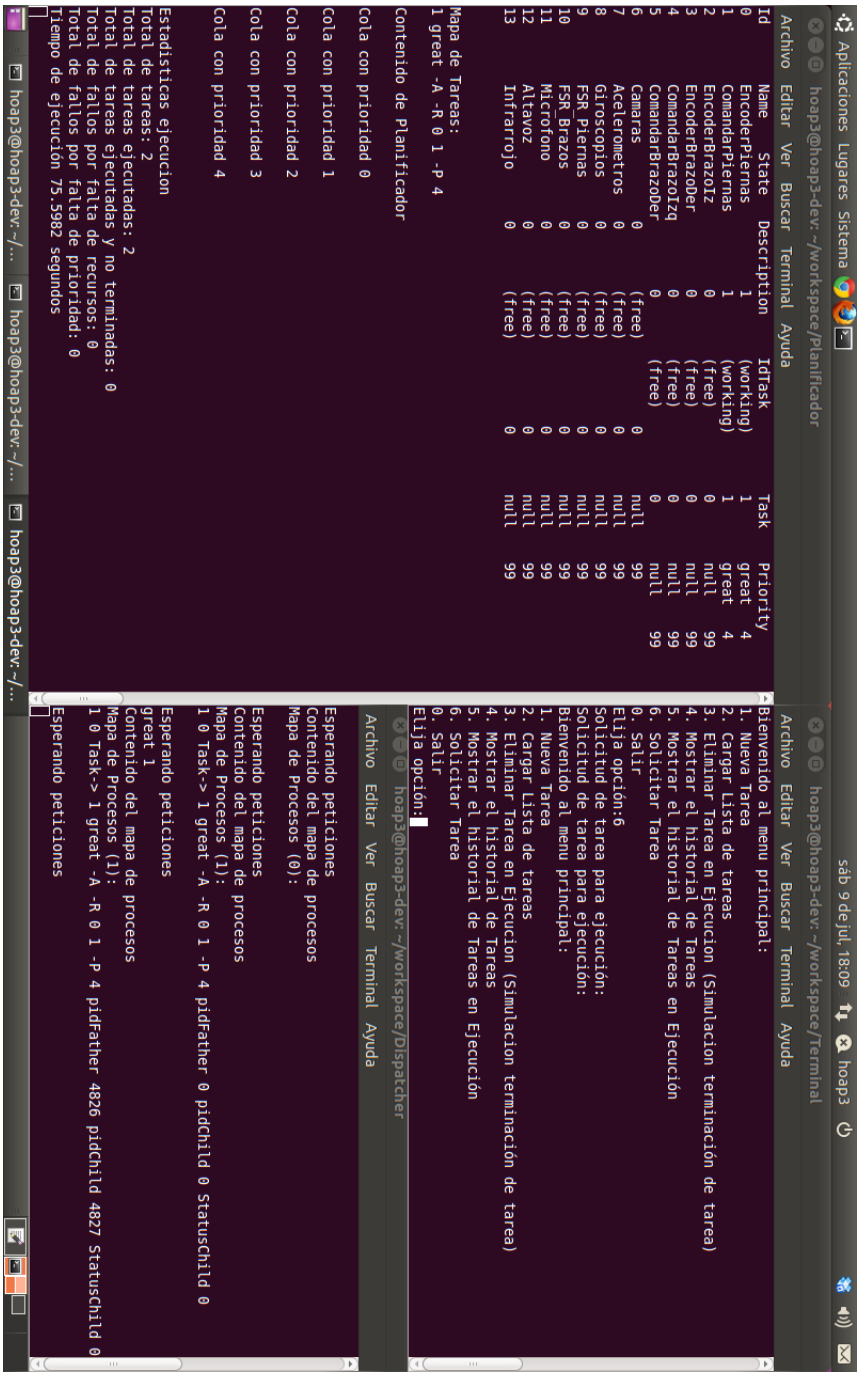


Figura 5.47: *Ejecución de la tarea great (captura de pantalla) (Ejecución secuencial)*

- **Contenido del Dispatcher:** En la fig. 5.48 se ve la salida completa de la ejecución en el módulo *Dispatcher*. Se observa cómo la aplicación envoltorio ejecuta el comando *walk 1*, cuando termina la ejecución *walk 2*. E inmediatamente comienza la ejecución de *great* donde queda comprobada la ejecución secuencial de los comandos.

```
Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 walk -A -R 0 1 2 -P 3 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
walk 1
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 walk -A -R 0 1 2 -P 3 pidFather 4815 pidChild 4816
    StatusChild 0

Esperando peticiones
walk 2
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 walk -A -R 0 1 2 -P 3 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (0):

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (1):
1 0 Task-> 1 great -A -R 0 1 -P 4 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
great 1
Contenido del mapa de procesos
Mapa de Procesos (1):
1 0 Task-> 1 great -A -R 0 1 -P 4 pidFather 4826 pidChild 4827 StatusChild
     0
```

```
Esperando peticiones
great 2
Contenido del mapa de procesos
Mapa de Procesos (1):
1 0 Task-> 1 great -A -R 0 1 -P 4 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (0):

Esperando peticiones
```

Figura 5.48: *Salida mostrada por el terminal durante toda la ejecución (Ejecución secuencial)*

5.4.3.2. Ejecución paralela

La ejecución paralela se produce cuando dos tareas se ejecutan al mismo tiempo al no compartir recursos, independientemente de la prioridad. En la fig. 5.49 se ve la batería de pruebas número 2, en ella se ve las dos tareas a ejecutar, al no compartir recursos (no hay dependencia de recursos), se pueden ejecutar las dos tareas de forma paralela.

```
1  <?xml version="1.0" encoding="UTF-8" ?>
2  <!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
3  <TaskFile>
4          <task>
5                  <name>great</name>
6                  <argument>
7                  </argument>
8                  <resources>
9                          <rec>0</rec>
10                         <rec>1</rec>
11                 </resources>
12                 <priority>3</priority>
13         </task>
14         <task>
15                 <name>speak</name>
```

```
16                  <argument>
17                  </argument>
18                  <resources>
19                          <rec>12</rec>
20                  </resources>
21                  <priority>4</priority>
22          </task>
23  </TaskFile>
```

Figura 5.49: *Batería de pruebas 2 HOAP3 (great y speak)*

La ejecución es la siguiente:

- **Carga de instrucciones:**
  Cargamos las instrucciones en el planificador (fig. 5.50).

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    0 (free)            0  null    99
1  ComandarPiernas   0 (free)            0  null    99
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1
```

```
Cola con prioridad 2

Cola con prioridad 3
0 0 great -A -R 0 1 -P 3

Cola con prioridad 4
0 1 speak -A -R 12 -P 4
```

Figura 5.50: *Tareas cargadas en el planificador (Ejecución paralela)*

- **Ejecución del comando great:**

  Ejecutamos el comando *great* (fig. 5.51).

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    1 (working)         0  great    3
1  ComandarPiernas   1 (working)         0  great    3
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:
0 great -A -R 0 1 -P 3

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1
```

```
Cola con prioridad 2

Cola con prioridad 3

Cola con prioridad 4
0 1 speak -A -R 12 -P 4
```

Figura 5.51: *Ejecución de la tarea great (Ejecución paralela)*

- **Ejecución del comando speak:**
  Cuando salta el timer del planificador, se comprueba si hay alguna tarea disponible y si es posible su ejecución. Como la tarea *speak* cumple todos los requisitos (siguiente tarea más prioritaria y recursos disponibles) comienza su ejecución a la vez que la tarea *great*.

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    1 (working)         0  great    3
1  ComandarPiernas   1 (working)         0  great    3
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           1 (working)         1  speak    4
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:
0 great -A -R 0 1 -P 3
1 speak -A -R 12 -P 4

Contenido de Planificador
```

```
Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3

Cola con prioridad 4
```

Figura 5.52: *Ejecución de la 2° tarea (Ejecución paralela)*

Figura 5.53: *Hoap ejecutando comando great (saludo) y speak (hablar) simultáneamente*

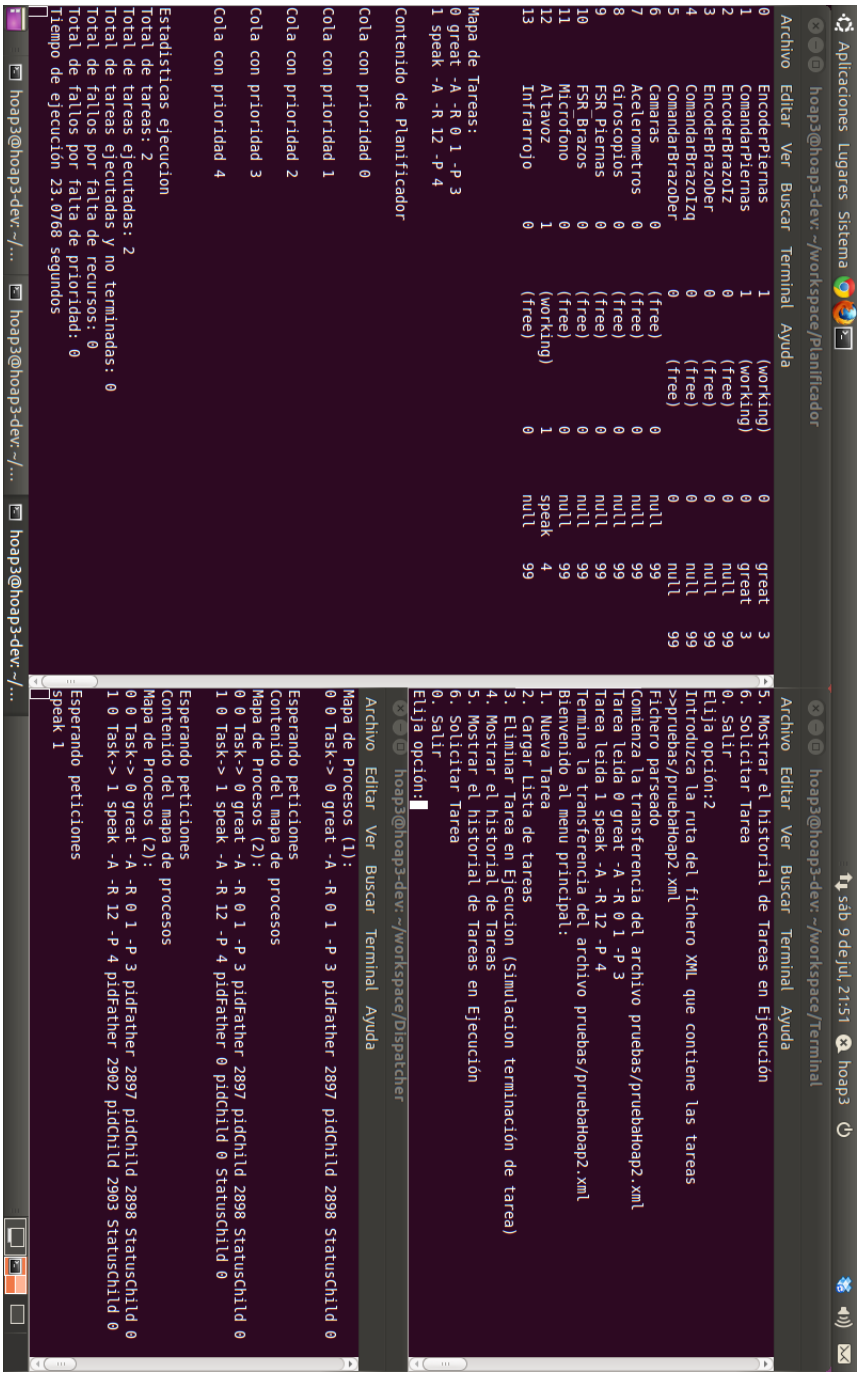


Figura 5.54: *Ejecución de las dos tareas en paralelo(Ejecución paralela)*

- **Contenido del Dispatcher:** En la fig. 5.55 se ve la salida completa de la ejecución en el modulo *Dispatcher*. Se observa cómo la aplicación envoltorio ejecuta el comando *great 1* al mismo tiempo que el comando *speak*. Se puede ver cómo finalizan casi a la vez.

```
Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 great -A -R 0 1 -P 3 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
great 1
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 great -A -R 0 1 -P 3 pidFather 3025 pidChild 3026 StatusChild
     0

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (2):
0 0 Task-> 0 great -A -R 0 1 -P 3 pidFather 3025 pidChild 3026 StatusChild
     0
1 0 Task-> 1 speak -A -R 12 -P 4 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
speak 1
Contenido del mapa de procesos
Mapa de Procesos (2):
0 0 Task-> 0 great -A -R 0 1 -P 3 pidFather 3025 pidChild 3026 StatusChild
     0
1 0 Task-> 1 speak -A -R 12 -P 4 pidFather 3030 pidChild 3031 StatusChild
    0

Esperando peticiones
great 2
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 great -A -R 0 1 -P 3 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
```

```
Contenido del mapa de procesos
Mapa de Procesos (1):
1 0 Task-> 1 speak -A -R 12 -P 4 pidFather 3030 pidChild 3031 StatusChild
    0

Esperando peticiones
speak 2
Contenido del mapa de procesos
Mapa de Procesos (2):
0 0 Task-> 0 great -A -R 0 1 -P 3 pidFather 3025 pidChild 3026 StatusChild
     0
1 0 Task-> 1 speak -A -R 12 -P 4 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (0):
```

Figura 5.55: *Salida mostrada por el terminal durante toda la ejecución (Ejecución paralela)*

5.4.3.3. Ejecución interrumpida

Una ejecución interrumpida se produce cuando una tarea en ejecución es desalojada por otra tarea con mayor prioridad que tienen recursos en común. Lo que se pretende mostrar en la ejecución interrumpida es que mientras el robot está realizando una trayectoria y utilice unos recursos determinados, se produzca una interrupción, que puede ser la detección de un obstáculo o la pérdida de la estabilidad, y que utilizando esos mismos recursos, se interrumpa la trayectoria actual y se recalcule una trayectoria nueva. El proceso de planificar una trayectoria estable ante un evento desconocido es una tarea compleja, que excede el ámbito del presente proyecto. Sin embargo, se ha hecho un ejemplo con la instrucción *speak* que se explicará a continuación.

La ejecución interrumpida se realiza en dos fases, se lanza primero la tarea *speak* y tras 5 segundos, lanzamos la tarea *great* que desalojará la tarea *speak*, liberará los recursos y comenzará la ejecución de *great*.

- **Fase 1:**

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
        <task>
                <name>speak</name>
                <argument></argument>
                <resources>
                        <rec>0</rec>
                        <rec>1</rec>
                </resources>
                <priority>3</priority>
        </task>
</TaskFile>
```

Figura 5.56: *Batería de pruebas 3 HOAP3 (speak)*

- **Fase 2:**

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
        <task>
                <name>great</name>
                <argument></argument>
                <resources>
                        <rec>0</rec>
                        <rec>1</rec>
                        <rec>2</rec>
                </resources>
                <priority>2</priority>
        </task>
</TaskFile>
```

Figura 5.57: *Batería de pruebas 3 HOAP3 (great)*

La ejecución es la siguiente:

- **Carga de instrucciones:**
  Cargamos la primera instrucción en el planificador (fig. 5.58).

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    0 (free)            0  null    99
1  ComandarPiernas   0 (free)            0  null    99
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3
0 0 speak -A -R 0 1 -P 3

Cola con prioridad 4
```

Figura 5.58: *Tareas cargada en el planificador (Ejecución interrumpida)*

- **Ejecución del comando speak:**
  Ejecutamos el comando *speak* (fig. 5.59).

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    1 (working)         0  speak    3
1  ComandarPiernas   1 (working)         0  speak    3
2  EncoderBrazoIz    0 (free)            0  null    99
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:
0 speak -A -R 0 1 -P 3

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3

Cola con prioridad 4
```

Figura 5.59: *Ejecución de la 1° tarea (Ejecución interrumpida)*

- **Ejecución del comando great:**
  Cargamos manualmente[4] el comando *great* (fig. 5.60). Cuando salta el timer del planificador, se comprueba si hay alguna tarea disponible y si es posible su ejecución. Como la tarea *great* es de mayor prioridad que la tarea *speak*, se desaloja la tarea *speak*, se liberan los recursos y comienza la ejecución de la tarea *great* (fig. 5.61).

```
Mapa de recursos:
Id      Name          State/Description   IdTask/Task Priority
0  EncoderPiernas     1 (working)         0  speak     3
1  ComandarPiernas    1 (working)         0  speak     3
2  EncoderBrazoIz     0 (free)            0  null     99
3  EncoderBrazoDer    0 (free)            0  null     99
4  ComandarBrazoIzq   0 (free)            0  null     99
5  ComandarBrazoDer   0 (free)            0  null     99
6  Camaras            0 (free)            0  null     99
7  Acelerometros      0 (free)            0  null     99
8  Giroscopios        0 (free)            0  null     99
9  FSR_Piernas        0 (free)            0  null     99
10 FSR_Brazos         0 (free)            0  null     99
11 Microfono          0 (free)            0  null     99
12 Altavoz            0 (free)            0  null     99
13 Infrarrojo         0 (free)            0  null     99

Mapa de Tareas:
0 speak -A -R 0 1 -P 3

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2
0 1 great -A -R 0 1 2 -P 2

Cola con prioridad 3
```

[4]El modo manual sirve para representar la llegada de nuevas instrucciónes en cualquier momento.

```
Cola con prioridad 4
```

Figura 5.60: *Carga de la tarea great (Ejecución interrumpida)*

```
Mapa de recursos:
Id      Name         State/Description   IdTask/Task Priority
0  EncoderPiernas    1 (working)         0  great    2
1  ComandarPiernas   1 (working)         0  great    2
2  EncoderBrazoIz    1 (working)         0  great    2
3  EncoderBrazoDer   0 (free)            0  null    99
4  ComandarBrazoIzq  0 (free)            0  null    99
5  ComandarBrazoDer  0 (free)            0  null    99
6  Camaras           0 (free)            0  null    99
7  Acelerometros     0 (free)            0  null    99
8  Giroscopios       0 (free)            0  null    99
9  FSR_Piernas       0 (free)            0  null    99
10 FSR_Brazos        0 (free)            0  null    99
11 Microfono         0 (free)            0  null    99
12 Altavoz           0 (free)            0  null    99
13 Infrarrojo        0 (free)            0  null    99

Mapa de Tareas:
1 great -A -R 0 1 2 -P 2

Contenido de Planificador

Cola con prioridad 0

Cola con prioridad 1

Cola con prioridad 2

Cola con prioridad 3

Cola con prioridad 4
```

Figura 5.61: *Desalojo de la tarea speak y ejecución de la tarea great (Ejecución interrumpida)*

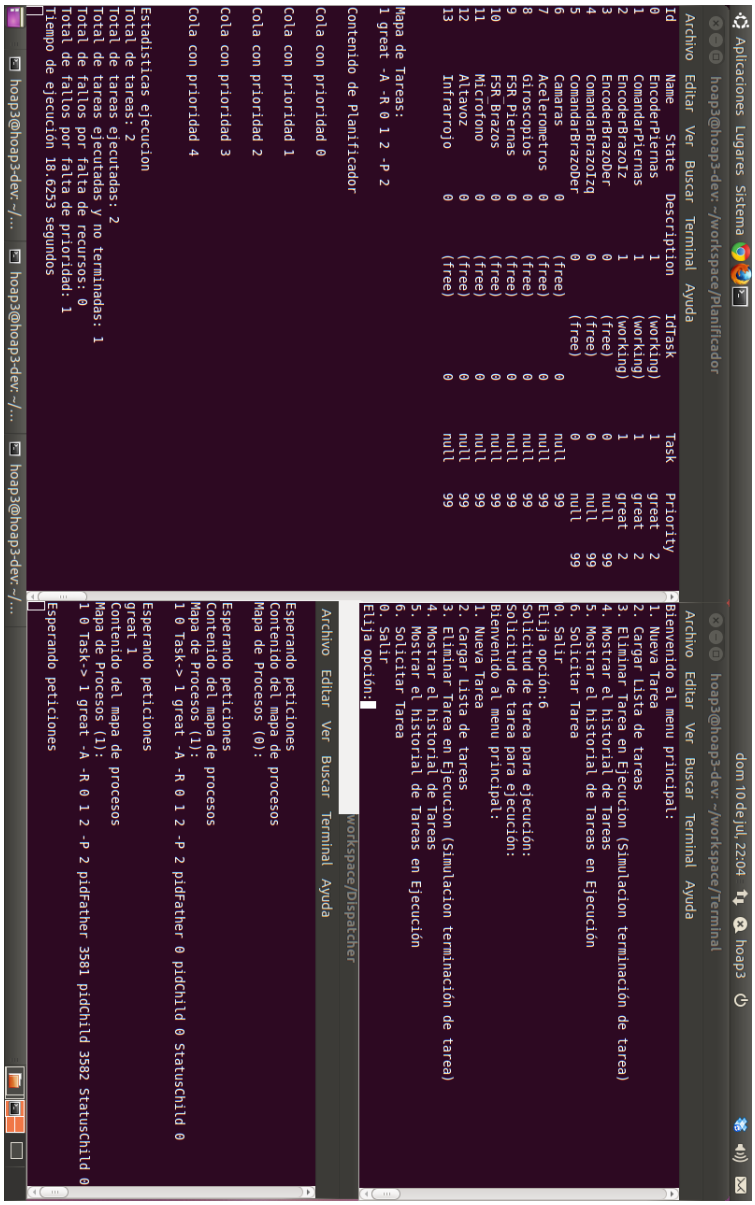


Figura 5.62: *Ejecución de la tarea great tras desalojar la tarea speak (Ejecución interrumpida)*

- **Contenido del Dispatcher:** En la fig. 5.63 se ve la salida completa de la ejecución en el modulo *Dispatcher*. Se observa cómo la aplicación envoltorio ejecuta el comando *speak 1*, no llega a terminar al no mostrar *speak 2* por lo que ha sido desalojada del planificador. Y comienza la ejecución del comando *great* que termina correctamente.

```
Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 speak -A -R 0 1 -P 3 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
speak 1
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 speak -A -R 0 1 -P 3 pidFather 3089 pidChild 3090 StatusChild
     0

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (1):
0 0 Task-> 0 speak -A -R 0 1 -P 3 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (0):

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (0):

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (1):
1 0 Task-> 1 great -A -R 0 1 2 -P 2 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
great 1
Contenido del mapa de procesos
Mapa de Procesos (1):
```

```
1 0 Task-> 1 great -A -R 0 1 2 -P 2 pidFather 3095 pidChild 3096
    StatusChild 0

Esperando peticiones
great 2
Contenido del mapa de procesos
Mapa de Procesos (1):
1 0 Task-> 1 great -A -R 0 1 2 -P 2 pidFather 0 pidChild 0 StatusChild 0

Esperando peticiones
Contenido del mapa de procesos
Mapa de Procesos (0):
```

Figura 5.63: *Salida mostrada por el terminal durante toda la ejecución (Ejecución interrumpida)*

Estadísticas de la ejecución: En la fig. 5.64 aparecen las estadísticas del terminal. Hay que destacar *Total de tareas ejecutadas y no terminadas: 1* que ratifica que la instrucción *speak* ha sido desalojada y *Total de fallos por falta de prioridad: 1*, que indica el desalojo de una tarea por una tarea más prioritaria.

```
Estadisticas ejecucion
Total de tareas: 2
Total de tareas ejecutadas: 2
Total de tareas ejecutadas y no terminadas: 1
Total de fallos por falta de recursos: 0
Total de fallos por falta de prioridad: 1
Tiempo de ejecuci n 28.6296 segundos
```

Figura 5.64: *Salida mostrada por el terminal durante toda la ejecución (Ejecución interrumpida)*

Capítulo 6

Conclusiones y trabajos futuros

6.1. Conclusiones

Las pruebas realizadas tanto en la simulación cómo en el robot HOAP3, han verificado cómo el *coordinador de tareas* cumple correctamente con los objetivos para los que fué desarrollado.

- **Protección del hardware**
  La reserva de recursos y posterior comprobación antes de ejecutar una tarea, ayudan a proteger la integridad física del hardware del robot. El coordinador impide la ejecución de más de una tarea simultanea sobre un mismo recurso; en el caso de los motores, es fundamental para alargar la vida útil de estos.

- **Prioridad de las tareas**
  Gracias a la prioridad implícita de cada tarea y a la reserva de recursos, se puede ejecutar en cada momento la tarea más prioritaria, evitando el uso de recursos de forma duplicada, dando resultados erróneos. Por ejemplo, al usar el altavoz del robot, el *coordinador de tareas* impide la superposición de mensajes de voz, evitando mensajes no comprensibles y la correcta emisión de los mensajes más importantes.

- **Ejecución en paralelo**
  Al comprobar el estado de los recursos antes la ejecución de cada tarea, el *coordinador de tareas* puede lanzar simultáneamente tareas que no tengan dependencia de recursos entre sí. La única limitación es la capacidad de computo del procesador y la memoria RAM disponible.

Además de estos objetivos, se propusieron una serie de premisas sobre el diseño del *coordinador de tareas*. A continuación se verá cómo se ha conseguido llevar a cabo cada uno de ellos:

- **Independencia de la Morfología**
  La utilización del *fichero de recursos* (ver sección 4.2.3 y apéndice B) para describir el hardware y software disponible en el robot, consigue la independencia de la morfología, permitiendo el correcto funcionamiento tanto en un robot móvil que utilice ruedas u orugas para desplazarse cómo de un robot humanoide.

- **Estrategias del planificador modificables**
  El diseño del *planificador* permite el uso de estrategias modificables mediante el uso de parámetros (ver sección 4.2.7.2 y 4.2.6.2).

- **Posibilidad de cambiar el planificador fácilmente**
  El diseño de la librería (ver sección 4.2.6) y del módulo planificador permiten cambiar el núcleo del planificador con uno más acorde a las necesidades del proyecto (CLIPS, HTN, ...).

- **Facilidad de integración con el resto del software**
  El uso de *YARP* cómo infraestructura de comunicaciones, el diseño del protocolo de comunicaciones (ver sección 4.2.1) y el uso de procesos para la ejecución (ver sección 4.2.7) permiten la integración y comunicación con la mayoría del software del robot.

- **Facilidad de cambiar de S.O.**
  El *coordinador de tareas* ha sido programado con la librería estándar de C++ y la librerías de YARP; ambas están disponibles para una amplia variedad de sistemas operativos. La única parte dependiente del sistema operativo es la encargada de ejecutar las tareas (ver sección 4.2.7.2). Para ello se han utilizado las llamadas al sistema de GNU/Linux. Sólo sería necesario modificar esta parte para la nueva plataforma destino (en el caso de que fuera necesario).

- **Desarrollo modular**
  El uso de *YARP* y el diseño del protocolo de comunicaciones, han permitido un desarrollo modular del *coordinador de tareas* permitiendo el uso de cada módulo de forma independiente y dando la posibilidad de que los módulos estén situados fuera del robot (ver sección 4.3).

- **Facilidad a la hora de cambiar/integrar nuevos módulos.**
  El uso de *YARP* y el diseño del protocolo de comunicaciones logran un débil acoplamiento entre ellos. Mediante el uso de la librería desarrollada y respetando el protocolo de comunicaciones y los mensajes entre módulos (ver sección 4.2.2 y apéndice A) se puede cambiar o añadir módulos.

6.2. Trabajos futuros

Posibles mejoras del *coordinador de tareas* serían:

- **Grupos de tareas:**
  Añadir a la clase tareas un identificador de grupo de tareas para darle mayor cohesión a tareas complejas formadas por subtareas.

- **Descartar tareas:**
  Permitir descartar tareas del planificador. Por ejemplo, si una tarea perteneciente a un grupo de tareas ha sido cancelada, poder eliminar el resto de

tareas de ese grupo.

- **Mejorar la salida por pantalla:**
  Modificar la salida por pantalla de cada módulo para mejorar la visualización de los datos y facilitar la interacción de la **terminal** con el usuario.

- **Interfaz gráfica:**
  Incluir una interfaz gráfica de usuario en el módulo **terminal** mejorando la interacción del usuario con la aplicación y la visualización de los datos.

- **Introducir nuevas políticas de planificación y planificadores por defecto:**
  Añadir más tipos de planificadores (CLIPS, HTN,...) y políticas de planificación.

Apéndices

Apéndice A

Mensajes

A.1. Mensajes

La comunicación entre módulos se realiza mediante paso de mensajes, para ello se utiliza una codificación compuesta de tres dígitos para identificar el tipo de mensaje. Cada código esta asociado a una constante que se encuentra en la clase *bottler.h*.

A.1.1. Nomenclatura

El coordinador de tareas utiliza una codificación propia para identificar el origen, destino y significado de los mensajes. Para ello utiliza la siguiente nomenclatura.

modulo	**identificador**
TerminalCom	TC
Terminal	T
Planificador	P
Dispatcher	D
DispatcherProcess	DP

Cuadro A.1: *Nomenclatura*

A.1.2. Identificador de la constante

El nombre de la constante identifica el origen del mensaje, el destinatario y una descripción breve de su significado. Para ello utiliza la nomenclatura descrita en la tabla A.1 para identificar cada una de las partes. Los nombre de las constantes tienen siempre la siguiente forma: **bottle_[origen]2[destino]_[descripción]**. Por ejemplo, el nombre de la constante bottle_TC2T_LoadListTask indica que el bottle con origen *TerminalCom* y con destino *Terminal* solicita la carga de una lista de tareas (LoadListTask).

A.1.3. Codificación

Cada mensaje tiene una constante asociada dentro de la clase *bottler.h* con un numero entero de tres dígitos asociado. Las reglas de codificación son las siguientes.

- Cada codigo esta compuesto de un numero entero de 3 digitos (XXX).
- La posición de cada dígito tiene un significado:
  - El primer dígito indica el origen del mensaje (ver tabla A.2).

modulo	**codigo**
TerminalCom	100
Terminal	200
Planificador	300
Dispatcher	400
DispatcherProcess	500

Cuadro A.2: *Codificación de los módulos*

- El segundo dígito indica la acción a realizar. Siendo $X1X$ una ejecución y $X2X$ una finalización. El resto de los valores posibles dependen del modulo origen y destino. Por ejemplo la constante **bottle_TC2T_LoadListTask** tiene asociado el codigo 102.
- El tercer dígito indica el código de respuesta a una petición o una opción de la acción a realizar.

A.1.4. Tabla de mensajes

En la tabla A.3 y A.4 aparece la lista de mensajes utilizada por la aplicación.

constante (bottle_ ...	codigo	descripción/petición
TC2T_Exit	100	Petición de salir del terminal
TC2T_NewTask	101	Creación de una nueva tarea
TC2T_LoadListTask	102	Carga de una lista de tareas
TC2T_ShowTaskList	103	Ver tareas en ejecución
TC2T_ShowExecutionTaskList	104	Ver tareas en ejecución
TC2T_TaskExecutionRequest	110	Solicita la ejecución de una tarea
TC2T_FinishTask	120	Finalizacion de una tarea
T2P_NewTask	201	Nueva tarea
T2P_TaskExecutionRequest	210	Petición de tarea
T2P_TaskTerminationRequest	220	Tarea terminada
P2D_TaskExecuteRequest	310	Solicitud de ejecución de tarea
P2T_TaskExecutionRespOk	311	Ejecución ha comenzado ok
P2T_TaskExecutionRespFail	312	Fallo en la ejecución de la tarea
P2T_TaskExecutionRespEmpty	313	Planificador vació
P2T_TaskExecutionRespPriorityFail	314	Faltan recursos ejecución prioritaria
P2T_TaskExecutionRespError	315	Ejecución errónea

Cuadro A.3: *Lista de mensajes de la aplicación*

constante (bottle_ ...	**codigo**	**descripción/petición**
P2D_TaskFinishRequest	320	Solicitud de terminación de tarea
P2T_TaskFinishOk	321	La tarea ha sido terminada
P2D_TaskFinishListRequest	322	Terminación de una lista de tareas
P2T_TaskFinishListOK	323	Terminación de una lista de tareas ok
P2T_TaskNoExist	330	Tarea no existe
D2P_ExecuteResponseOk	411	Ejecución correcta de la tarea
D2P_ExecuteResponseError	412	Ejecución incorrecta de la tarea
D2P_TaskFinishOk	421	Tarea ha sido terminada correctamente
D2P_TaskFinishListOK	422	Terminación de una lista de tareas ok
DP2D_ProcessState	500	Estado del proceso
DP2D_ProcessFinish	520	Finalización de una tarea

Cuadro A.4: *Lista de mensajes de la aplicación*

Apéndice B

Ficheros de configuración

B.1. Ficheros DTD

Los ficheros DTD describen la sintaxis a seguir por los ficheros XML.

B.1.1. Fichero DTD asociado a los recursos

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Este es el DTD de ResourceFile.dtd -->
<! DOCTYPE ResourceFile[
        <!ELEMENT ResourceFile (resource+)>
        <!ELEMENT resource (id, name,state,task,idTask,priority)>
        <!ELEMENT id (#PCDATA)>
        <!ELEMENT name (#PCDATA)>
        <!ELEMENT state (#PCDATA)>
        <!ELEMENT task (#PCDATA)>
        <!ELEMENT idTask (#PCDATA)>
        <!ELEMENT priority (#PCDATA)>
]>
```

B.1.2. Fichero DTD asociado a las tareas

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- Este es el DTD de TaskFile.dtd -->
<! DOCTYPE TaskFile[
<!ELEMENT TaskFile (task+)>
<!ELEMENT task (name, argument,resources,priority)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT argument (arg*)>
<!ELEMENT arg (#PCDATA)>
<!ELEMENT resources (rec+)>
<!ELEMENT rec (#PCDATA)>
<!ELEMENT priority (#PCDATA)>
]>
```

B.2. Fichero de recursos del robot HOAP3

Fichero de configuración del robot HOAP3, contiene los recursos de la plataforma robótica.

```
<?xml version="1.0"?>
<!DOCTYPE ResourceFile SYSTEM "ResourceFile.dtd">
<ResourceFile>
        <resource>
                <id>0</id>
                <name>EncoderPiernas</name>
                <state>0</state>
                <task>null</task>
                <idTask>0</idTask>
                <priority>99</priority>
        </resource>
        <resource>
                <id>1</id>
                <name>ComandarPiernas</name>
                <state>0</state>
                <task>null</task>
                <idTask>0</idTask>
                <priority>99</priority>
        </resource>
        <resource>
                <id>2</id>
                <name>EncoderBrazoIz</name>
                <state>0</state>
                <task>null</task>
                <idTask>0</idTask>
                <priority>99</priority>
        </resource>
        <resource>
                <id>3</id>
                <name>EncoderBrazoDer</name>
                <state>0</state>
                <task>null</task>
                <idTask>0</idTask>
                <priority>99</priority>
        </resource>
        <resource>
```

```
37                  <id>4</id>
38                  <name>ComandarBrazoIzq</name>
39                  <state>0</state>
40                  <task>null</task>
41                  <idTask>0</idTask>
42                  <priority>99</priority>
43          </resource>
44          <resource>
45                  <id>5</id>
46                  <name>ComandarBrazoDer</name>
47                  <state>0</state>
48                  <task>null</task>
49                  <idTask>0</idTask>
50                  <priority>99</priority>
51          </resource>
52          <resource>
53                  <id>6</id>
54                  <name>Camaras</name>
55                  <state>0</state>
56                  <task>null</task>
57                  <idTask>0</idTask>
58                  <priority>99</priority>
59          </resource>
60          <resource>
61                  <id>7</id>
62                  <name>Acelerometros</name>
63                  <state>0</state>
64                  <task>null</task>
65                  <idTask>0</idTask>
66                  <priority>99</priority>
67          </resource>
68          <resource>
69                  <id>8</id>
70                  <name>Giroscopios</name>
71                  <state>0</state>
72                  <task>null</task>
73                  <idTask>0</idTask>
74                  <priority>99</priority>
75          </resource>
76          <resource>
77                  <id>9</id>
78                  <name>FSR_Piernas</name>
```

```
79                  <state>0</state>
80                  <task>null</task>
81                  <idTask>0</idTask>
82                  <priority>99</priority>
83          </resource>
84          <resource>
85                  <id>10</id>
86                  <name>FSR_Brazos</name>
87                  <state>0</state>
88                  <task>null</task>
89                  <idTask>0</idTask>
90                  <priority>99</priority>
91          </resource>
92          <resource>
93                  <id>11</id>
94                  <name>Microfono</name>
95                  <state>0</state>
96                  <task>null</task>
97                  <idTask>0</idTask>
98                  <priority>99</priority>
99          </resource>
100         <resource>
101                 <id>12</id>
102                 <name>Altavoz</name>
103                 <state>0</state>
104                 <task>null</task>
105                 <idTask>0</idTask>
106                 <priority>99</priority>
107         </resource>
108         <resource>
109                 <id>13</id>
110                 <name>Infrarrojo</name>
111                 <state>0</state>
112                 <task>null</task>
113                 <idTask>0</idTask>
114                 <priority>99</priority>
115         </resource>
116 </ResourceFile>
```

B.3. Baterias de pruebas HOAP3

B.3.1. Ejecución Secuencial

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
        <task>
                <name>walk</name>
                <argument>
                </argument>
                <resources>
                        <rec>0</rec>
                        <rec>1</rec>
                        <rec>2</rec>
                </resources>
                <priority>3</priority>
        </task>
        <task>
                <name>great</name>
                <argument>
                </argument>
                <resources>
                        <rec>0</rec>
                        <rec>1</rec>
                </resources>
                <priority>4</priority>
        </task>
</TaskFile>
```

B.3.2. Ejecución Paralela

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
        <task>
                <name>walk</name>
                <argument>
                </argument>
```

```
                <resources>
                        <rec>0</rec>
                        <rec>1</rec>
                </resources>
                <priority>3</priority>
        </task>
        <task>
                <name>speak</name>
                <argument>
                </argument>
                <resources>
                        <rec>12</rec>
                </resources>
                <priority>4</priority>
        </task>
</TaskFile>
```

B.3.3. Ejecución Interrumpida

La ejecución interrumpida se realiza en dos fases, se lanza primero una tarea y tras 5 segundos, lanzamos la segunda.

B.3.3.1. Fase 1

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
        <task>
                <name>speak</name>
                <argument>
                </argument>
                <resources>
                        <rec>0</rec>
                        <rec>1</rec>
                </resources>
                <priority>3</priority>
        </task>
</TaskFile>
```

B.3.3.2. Fase 2

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE TaskFile SYSTEM "TaskFile.dtd">
<TaskFile>
        <task>
                <name>great</name>
                <argument>
                </argument>
                <resources>
                        <rec>0</rec>
                        <rec>1</rec>
                        <rec>2</rec>
                </resources>
                <priority>2</priority>
        </task>
</TaskFile>
```

Referencias

Ando, N., Suehiro, T., y Kotoku, T. (2008). A software platform for component based rt-system development: Openrtm-aist. *Simulation, Modeling, and Programming for Autonomous Robots*, 87–98.

Antonio Barrientos, C. B. R. A., Luis Felipe Peñin. (1997). Fundamentos de robótica. *Editorial McGraw-Hill*.

Arregui, S. F. (2006). *Aprendizaje de conocimiento de control para planificación de tareas*. Tesis Doctoral no publicada, uc3m.

Asfour, T., Ly, D., Regenstein, K., y Dillmann, R. (2006). Coordinated task execution for humanoid robots. *Experimental Robotics IX*, 259–267.

Barrettand Daniel, S., y cols. (1994). Partial-order planning:: Evaluating possible efficiency gains. *Artificial Intelligence*, *67*(1), 71–112.

Bastide, R., Sy, O., y Palanque, P. (1999). Formal specification and prototyping of corba systems. *ECOOP'99—Object-Oriented Programming*, 669–669.

Blum, A., y Furst, M. (1997). Fast planning through planning graph analysis* 1. *Artificial intelligence*, *90*(1-2), 281–300.

Bonet, B., y Geffner, H. (2001). Planning as heuristic search. *Artificial Intelligence*, *129*(1-2), 5–33.

Bruyninckx, H. (2001). Open robot control software: the orocos project. En *Robotics and automation, 2001. proceedings 2001 icra. ieee international conference on* (Vol. 3, pp. 2523–2528).

Canas, J., Matellán, V., y Montúfar, R. (2006). Programación de robots móviles. *Revista Iberoamericana de Automática e Informática Industrial*, *3*(2), 99–110.

Caramia, M., y Giordani, S. (2009). A new approach for scheduling independent tasks with multiple modes. *Journal of Heuristics*, *15*(4), 313–329.

Dechter, R. (2003). *Constraint processing*. Morgan Kaufmann.

Eiben, A., y Ruttkay, Z. (1997). Constraint satisfaction problems. En *Practical handbook of genetic algorithms.*

Erol, K., Hendler, J., y Nau, D. (1995). Htn planning: Complexity and expressivity. En *Proceedings of the national conference on artificial intelligence* (pp. 1123–1123).

Fikes, R., y Nilsson, N. (1971). Strips: A new approach to the application of theorem proving to problem solving. *Artificial intelligence*, *2*(3-4), 189–208.

Gen, M., y Yoo, M. (2008). Real time tasks scheduling using hybrid genetic algorithm. *Computational Intelligence in Multimedia Processing: Recent Advances*, 319–350.

Henning, M. (2004). Massively multiplayer middleware. *Queue*, *1*(10), 38–45.

Howard, R. (1960). Dynamic programming and markov process.

Metta, G., Fitzpatrick, P., y Natale, L. (2006). Yarp: yet another robot platform. *International Journal on Advanced Robotics Systems*, *3*(1), 43–48.

Mittal, S., y Falkenhainer, B. (1990). Dynamic constraint satisfaction problems. En *Proceedings of the eighth national conference on artificial intelligence* (pp. 25–32).

Palacios, H., Bonet, B., Darwiche, A., y Geffner, H. (2005). Pruning conformant plans by counting models on compiled d-dnnf representations. En *Proceedings of the 15th international conference on automated planning and scheduling (icaps)* (pp. 141–150).

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., y cols. (2009). Ros: an open-source robot operating system. En *Icra workshop on open source software.*

Rodríguez-Moreno, M., Borrajo, D., y Meziat, D. (2004). An ai planning-based tool for scheduling satellite nominal operations. *AI Magazine*, *25*(4), 9.

Schmidt, D. (1993). The adaptive communication environment: An object-oriented network programming toolkit for developing communication software.

Schmidt, D., Levine, D., y Mungee, S. (1998). The design of the tao real-time object request broker* 1. *Computer Communications*, *21*(4), 294–324.

Smith, D., y Weld, D. (1998). Conformant graphplan. En *Proceedings of the national conference on artificial intelligence* (pp. 889–896).

Vı'ctores, J. C. G. (2010). Software engineering techniques applied to assitive robotics: Guidelines & tools.

Winer, D., y cols. (1999). *Xml-rpc specification.* January.