UNIVERSITAT POLITÈCNICA DE VALÈNCIA

MÁSTER DE

AUTOMÁTICA E INFORMÁTICA INDUSTRIAL

Herramienta informática de apoyo al desarrollo de sistemas de control basados en "Núcleo de control"

Autor:

Jose Luis Beltrán Alonso

Director:

Dr. José Enrique Simó Ten

Valencia, Septiembre de 2013

Agradecimientos

A José Enrique Simó Ten por darme la oportunidad y los medios para participar en este proyecto.

A Manuel Muñoz Alcobendas y Eduardo Munera Sánchez por su constante colaboración durante el desarrollo de este trabajo.

Índice general

Índice de figuras

CAPÍTULO 1

Introducción

La constante evolución de la computación a lo largo de las últimas décadas junto a la aparición de importantes avances tecnológicos (tales como el desarrollo de microprocesadores más potentes y económicos, o el desarrollo de redes de área local de alta velocidad), ha provocado que se abandone el concepto de computación centralizada para dar paso a sistemas totalmente diferentes conocidos como "sistemas distribuidos". El profesor de ciencias de la computación de la Universidad Libre de Ámsterdam, Andrew S. Tanenbaum, define un sistema distribuido como "una colección de ordenadores independientes que se muestran a los usuarios del sistema como un solo equipo". Tan significativo ha sido este cambio en los últimos años, y tal ha sido la relevancia de estas aplicaciones distribuidas junto a los servicios en red, que hoy en día no se concibe una aplicación como un ente aislado del resto del mundo. Y es que la tendencia actual a la globalización, donde se busca que todo esté conectado entre si, junto con los avances en el campo de las redes de comunicación, exige que las prestaciones de los sistemas y aplicaciones informáticas vayan más allá de lo alcanzable por cualquier ordenador aislado. La principal razón para la descentralización de estos sistemas viene marcado por la economía, si bien también han intervenido otros factores tales como su transparencia, escalabilidad, fiabilidad y tolerancia a fallos. Por lo general, los sistemas distribuidos tienen una comparación entre coste y rendimiento mucho mayor que la proporcionada por un sistema centralizado.

En el campo de la industria, se encuentran los sistemas de control distribuidos (Distributed Control Systems (DCS) por sus siglas en inglés) que han supuesto un nuevo reto en el desarrollo de software. Esto se debe a que conforme aumentan las prestaciones y calidad de controladores, actuadores y sensores inteligentes, mayor es

la complejidad para el desarrollo de herramientas adecuadas capaces de integrar todos estos sistemas. Por lo general, los DCS se encargan de cubrir necesidades específicas, por lo que es habitual que sean ampliamente utilizados en este campo para monitorizar y controlar equipos distribuidos de control con intervención humana remota. Sin embargo, la realización de un diseño e implementación adecuados, en ocasiones supone una tarea realmente compleja de llevar a cabo. Esto se debe a que los DCS al ser sistemas de control embebidos de tiempo real, suelen tener recursos limitados (a nivel de potencia de cálculo, memoria y ancho de banda). Debido también a su condición de "sistema de tiempo real", se debe garantizar que se prestan sus servicios a las aplicaciones dentro de los estrictos plazos de tiempo. Este último aspecto cobra especial importancia sobre todo para aquellas aplicaciones que son críticas a nivel de seguridad como las utilizadas en robots móviles o en equipos médicos.

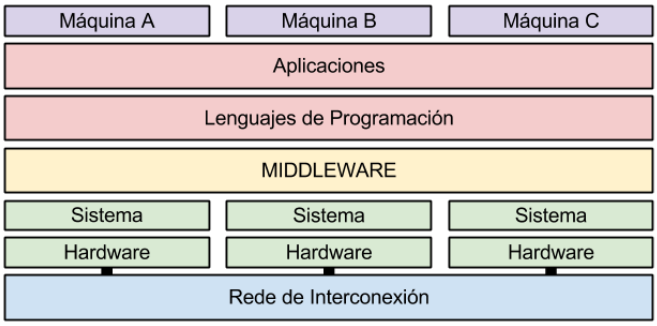

Figura 1.1: Implementación middleware sobre sistema distribuido

Por el momento, la disponibilidad de un middleware de control que te permite la abstracción de los sistemas operativos, te da el soporte de tiempo real y la infraestructura de comunicaciones, parece ser una de las claves para el éxito de este tipo de sistemas. No sólo porque permiten ocultar la complejidad de la infraestructura antes mencionada sino que además proporcionan interfaces abiertas para el desarrollo de aplicaciones a terceros.

1.1. Motivación y justificación

Los sistemas de control normalmente no ofrecen interfaces de configuración amigables para el usuario, lo que implica que posibles modificaciones en ocasiones suponga una labor costosa para quién las realiza. Los ficheros de configuración con los que se trabaja suelen tener extensiones en tamaño considerables y obliga al usuario a conocer su estructura para poder manejarlos. En este aspecto los errores humanos son habituales, bien por desconocimiento de la estructura a utilizar, bien por errores en el tipo de dato especificado o bien por conexiones inconsistentes. La disponibilidad de una herramienta informática adaptada al middleware que incorpore sus capacidades de configuración junto a las restricciones necesarias permiten abstraer al usuario de la problemática mencionada.

Teniendo presente lo que se acaba de comentar, el desarrollo del modelo de configuración del sistema permitirá conocer los parámetros y capacidades de configuración de cada uno de los elementos que lo conforman. Este modelado es necesario para establecer el diseño de la herramienta anteriormente mencionada y definir el nivel de abstracción que se pretende conseguir.

1.2. Objetivos de la tesina

Los dos objetivos principales establecidos para este trabajo consistirán en la especificación del modelo de configuración para el middleware CKM (*Control Kernel Middleware*), así como la implementación de una herramienta informática basada en dicho modelo capaz de dar soporte al diseño de aplicaciones de control.

El objetivo que se pretende conseguir con la especificación del modelo es definir los parámetros requeridos para realizar una solicitud de servicio y su configuración. Para ello se representará en el lenguaje de esquema XML Schema Definition (XSD) de modo que cualquier usuario o aplicación basada en el modelo podrá validar sus ficheros de configuración de acuerdo a las restricciones establecidas. Esto implica que los usuarios deben irremediablemente conocer las estructuras de datos utilizadas para la configuración del middleware. En este sentido, se pretende diseñar una herramienta informática que abstraiga al usuario de esta problemática.

El objetivo de la interfaz gráfica es ofrecer al usuario una forma rápida, fácil e intuitiva de definir la configuración deseada para el middleware. De este modo el escenario definido en los gráficos será trasladado a su equivalente en un fichero xml y viceversa. La interfaz deberá implementarse a partir del modelo con la finalidad de que los ficheros generados estén validados respecto al mismo.

Los requisitos mínimos con los que deberá contar son:

- Editor de recursos de comunicación
- Editor de nodos
- Editor de reguladores, sensores y actuadores
- Generación de plantillas de código
- Soporte runtime: Implica que la interfaz puede conectarse con el middleware en tiempo de ejecución para modificar la configuración del mismo dinámicamente.

1.3. Organización del documento

El presente documento se encuentra dividido en 6 capítulos que están distribuidos de la siguiente forma:

En el capítulo 2 se presenta una revisión del estado del arte referente a las diferentes implementaciones de middleware, así como una clasificación en función de su funcionalidad.

El middleware basado en núcleo de control es presentado en el capítulo 3. Este es el middleware de control utilizado y sobre el que se ha diseñado e implementado la herramienta de configuración presentada.

En el capítulo 4 se presenta el modelo realizado identificando los diferentes elementos del sistema a configurar. Las capacidades de configuración de cada uno de ellos se muestran gráficamente en la interfaz descrita en el capítulo 5.

Para finalizar, en el capítulo 6 se extraerán las principales conclusiones y se definirán los trabajos futuros a los que se da paso con este trabajo.

CAPÍTULO 2

Estado del arte

La definición más ampliamente aceptada del término *middleware* se entiende como un software de conectividad que consiste en la habilitación de un conjunto de servicios para la conexión entre distintos componentes software o aplicaciones. El middleware permite abstraer la complejidad y heterogeneidad de las redes de comunicaciones subyacentes, así como de los sistemas operativos y lenguajes de programación, proporcionando una API para la fácil programación y manejo de aplicaciones distribuidas. Su término fue acuñado en la década de los 60 durante la "1968 Nato Software Engineering Conference", fusionando las palabras *middle* (medio) y software, queriendo hacer referencia a un software especial que se colocaba entre las aplicaciones informáticas y el hardware con el objetivo de conectar el nuevo software con sistemas más antiguos. El middleware se enmarca, por lo tanto, en las capas de *presentación* y *aplicación* (6 y 7) dentro de la arquitectura de referencia ISO-OSI.

Durante las décadas previas a los 90, el middleware fue solamente descrito como un software para la gestión de conexión en redes. Para cuando las tecnologías en redes alcanzaron una presencia y visibilidad suficiente, el software middleware había evolucionado en un conjunto de paradigmas y servicios. De esta forma, se estaba ofreciendo una manera más fácil, robusta y controlable para construir aplicaciones distribuidas. Actualmente el uso del middleware como plataforma básica para el desarrollo de software distribuido se ha generalizado y se ha incorporado el modelo de programación basado en componentes como técnica básica para el desarrollo de software distribuido. Por ejemplo, los middleware proporcionan componentes reutilizables que facilitan: el empaquetado y desempaquetado de la información que se transmite por la red, la gestión de recursos compartidos por las aplicaciones distribuidas, invocación remota

de procedimientos, acceso transparente a recursos locales o remotos, comunicación entre procesos de un grupo, notificación de eventos y replicación de datos compartidos. Una ventaja adicional de los middleware modernos es que facilitan el desarrollo de software mediante componentes escritos en diversos lenguajes y que garantizan su correcto funcionamiento en los diferentes sistemas operativos.

Los requerimientos básicos con los que cuentan hoy día la mayoría de middleware, se pueden clasificar en cinco tipos:

- Comunicación de red: Debido a que gran variedad de tipos de middleware acceden a recursos remotos del sistema, los sitemas distribuidos requieren de una comunicación por red.
- Coordinación: En los sistemas distribuidos, la coordinación es requerida para controlar la comunicación entre múltiples puntos. Es por ello que existen mecanismos para lograrlo mediante la utilización de políticas de sincronización y activación.
- Fiabilidad: Los sistemas distribuidos necesitan incluir mecanismos de detección y corrección de errores para eliminar causas indeseadas debido a errores.
- Escalabilidad: La principal tarea es proveer cambios en un sistema distribuido sin cambiar su arquitectura o diseño.
- Heterogeneidad: Referente a la capacidad para interactuar con componentes de distinto tipo.

2.1. Middleware de comunicación

Este tipo de middleware proporciona el medio de comunicación necesario para que las aplicaciones puedan "conversar" entre sí. Algunos ejemplos que existen actualmente son: Java Message Service (JMS) que proporciona soporte al paradigma de comunicación distribuida *publicación-suscripción*, Remote Method Invocation (RMI) que ofrece invocación remota de métodos en Java y Common Object Request Broker Architecture (CORBA) que proporciona soporte al paradigma de la programación *cliente-servidor* orientada a objetos para sistemas distribuidos heterogéneos.

Si bien todos ellos están orientados a comunicaciones, tal y como se describe en [LHD], los middleware de comunicación se pueden clasificar en cuatro tipos representativos en los que pueden estar: orientado a transacciones, orientado a mensajes, basado en RPC y orientado a objetos distribuidos. En los siguientes apartados se puede ver una descripción relativa a cada uno de ellos.

2.1.1. Orientado a transacciones

El middleware orientado a transacciones se caracteriza por ofrecer el desarrollo de sistemas que requieren transacciones distribuidas. Estas transacciones tienen la misión de asegurar que todas las operaciones que se realizan se ejecuten al completo por todos los nodos participantes o por ninguno de ellos. Para realizar esta tarea se utiliza un protocolo de confirmación en dos fases llamado Two-Phases Commit Protocol (2PC). Este protocolo utiliza mensajes de confirmación entre el nodo máster y los nodos receptores que indican el estado de las ejecuciones. Si aparece alguna incidencia (como puede ser un nodo receptor que no ha recibido bien todas las operaciones a ejecutar o bien que ha sucedido un problema durante el procesamiento), el nodo máster ordena realizar la operación *rollback* para evitar que el resto aplique los cambios indicados. Este modo de funcionamiento es generalmente utilizado por aplicaciones que necesitan asegurar la consistencia de estado entre los diferentes componentes distribuidos. Sin embargo, el inconveniente de este middleware reside en la eficiencia ya que el rendimiento se ve afectado con el uso del protocolo 2PC sobre sistemas con restricciones de ancho de banda. Esto es debido a la sobrecarga de comunicación que este tipo de protocolos genera.

2.1.2. Orientado a mensajes

El middleware orientado a mensajes "Message-Oriented Middleware (MOM)", permite a las aplicaciones distribuidas comunicarse mediante el envío de mensajes asegurando que todos ellos lleguen siempre a su destino. La comunicación entre emisor y receptor se realiza de forma asíncrona por lo que nunca llegan a estar conectados directamente entre sí. Sin embargo, este aspecto no implica necesariamente que las aplicaciones que lo utilizan deban ser también de la misma naturaleza. Existen soluciones que trabajan con conexiones asíncronas o pseudo-síncronas que son creadas como capas adicionales a más bajo nivel incorporando mecanismos de detección de fallos, gestión de prioridad de mensajes, etc.

Dentro de esta categoría se encuentran middleware como **MQSeries** y **JMS**. El primero de ellos está desarrollado por IBM y permite la integración de aplicación gestionando el intercambio asíncrono de información a través de diversos sistemas operativos y procesadores. Un aspecto a destacar es lo que definen como *procesado independiente del tiempo* que significa que los mensajes son tratados puntualmente, incluso si alguno de los recursos está temporalmente inaccesible. Por otro lado, JMS que está desarrollado por Sun Microsystems [HB02], proporciona soporte a dos paradigmas de comunicación distribuida: *producto-consumidor* y *publicación-suscripción*.

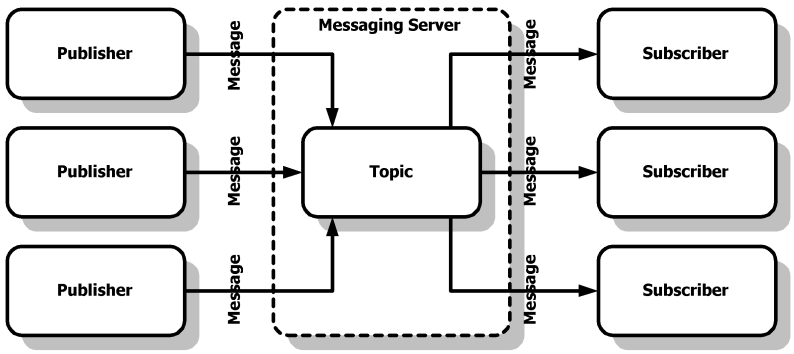

Figura 2.1: Sistema de mensajería basado en el modelo Publicación-Suscripción

La comunicación asíncrona entre componentes mediante el paradigma de productor-consumidor está disponible incluso cuando el consumidor no está activo en el momento del envío del mensaje. El soporte para publicación-suscripción asegura que todo mensaje publicado en el servicio JMS será recibido por todos los componentes suscritos al JMS al que se envió el mensaje (ver figura 2.1). A nivel de aplicación, la principal ventaja de este tipo de redes es que los clientes reciben solamente la información de aquellos que consideran de interés, evitando así malgastar recursos en la gestión de información innecesaria.

2.1.3. Basado en RPC

Las llamadas remotas a procedimiento, término conocido en inglés como Remote Procedure Call (RPC) es un protocolo que permite a un programa ejecutar código en otra máquina remota sin la preocupación relativa al tipo de comunicaciones existentes entre ambos. Son utilizadas dentro del paradigma cliente-servidor. En este modelo, el cliente realiza una llamada de procedimiento y espera una respuesta del servidor. La llamada de procedimiento se traduce a una comunicación en red mediante el mecanismo RPC subyacente. El servicor recibe la solicitud, ejecuta el procedimiento y le devuelve los resultados al cliente. Se trata por lo tanto de un sistema síncrono y diseñado para el acceso remoto a recursos. Actualmente suponen un avance sobre los *sockets* utilizados ya que las comunicaciones vienen encapsuladas dentro de las RPC por lo que el programador no necesita hacer gestión alguna sobre ellas. No obstante, las RPC permiten definir la interfaz de comunicación con los componentes mediante un "lenguaje de definición de la interfaz" (Interface Definition Language (IDL)). A partir de esta definición existen herramientas automáticas de generación de código que proporcionan lo necesario para realizar el empaquetado/desempaquetado de los mensajes y gestionar la comunicación a través de la red. El middleware basado en RPC implementa este protocolo proporcionando una gestión eficiente de las llamadas remotas a procedimiento.

2.1.4. Orientado a objetos distribuidos

El middleware orientado a objetos distribuidos proporciona la abstracción de un objeto que puede (o no) estar ubicado remotamente pero cuyos métodos pueden ser invocados como si se tratara de un objeto ubicado en el mismo espacio de memoria de quién lo invoca. Los objetos distribuidos ofrecen todas las ventajas de la Programación Orientada a Objetos (POO) como son la encapsulación, herencia, poliformismo,

soporte a excepciones o la referencia a objetos. Uno de los objetivos de este tipo de middleware es permitir la comunicación transparente y síncrona con los componentes tanto locales como remotos y siempre utilizando el mecanismo de transporte más eficiente. Por componente se entiende a aquellas entidades independientes que pueden interactuar e interoperar a través de redes, aplicaciones, lenguajes, herramientas y sistemas operativos.

Algunos middleware que se pueden encontrar dentro de esta clasificación son Java RMI, CORBA o Data Distribution Service for Real-time Systems (DDS) entre otros. **Java RMI** pertenece a Sun Microsystems y proporciona invocación remota de métodos en Java. Permite construir aplicaciones distribuidas donde la máquina virtual de Java (Java Virtual Machine (JVM)) puede realizar invocación remota de objetos disponibles en otras JVM localizadas en otros nodos de la red (ver figura 2.2). Esta flexibilidad es posible gracias a la arquitectura de la máquina virtual de Java, simplificada por el uso de un único lenguaje de programación. A partir de este tipo de middleware pueden encontrarse implementaciones de RMI - Hard Real-Time (RMI-HRT) que están enfocadas a sistemas de tiempo real crítico con requisitos de alta integridad. Un ejemplo sería el descrito en [TCA08] donde se define un nuevo modelo de RMI basado en los principales perfiles de "Real-Time Specification for Java (RSTJ)".

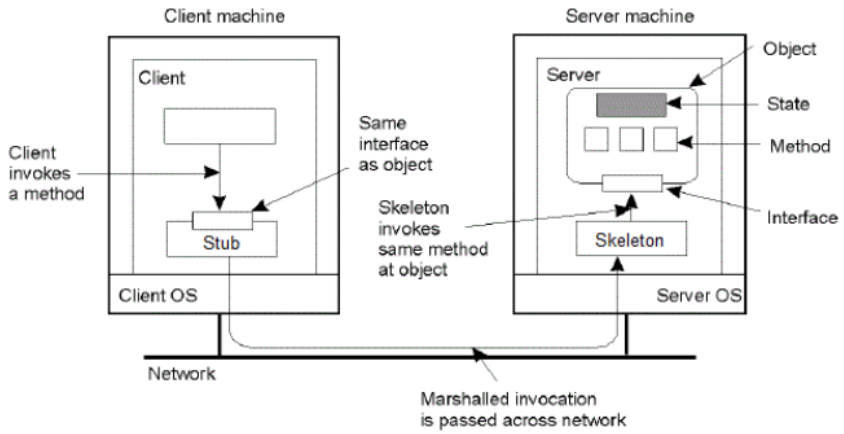

Figura 2.2: Arquitectura RMI

Por otro lado **CORBA**, tal y como se describe en [COR95] es una especificación de la arquitectura middleware que proporciona soporte al paradigma de la programación *cliente-servidor* orientada a objetos para sistemas distribuidos heterogéneos. Proporciona toda la infraestructura de comunicaciones necesaria para identificar y localizar objetos, gestionar las conexiones y entregar los datos. Su funcionalidad básica consiste en redirigir las peticiones de los clientes al objeto remoto invocado.

En el caso de **DDS**, se trata de un estándar del *Object Management Group (OMG)* [Gro08] publicado en junio del 2003. Es la especificación del API de un middleware basado en el modelo de publicación-suscripción, que tiene como objetivo principal los sistemas de tiempo real y es ampliamente utilizado en los sectores de defensa y aeroespacial como puede verse en [FC08]. El término "tiempo real" no sólo significa alto rendimiento, sino determinismo y predictibilidad. Por tanto, los sistemas distribuidos de tiempo real necesitan un middleware de mensajería que proporcione alto rendimiento (baja lantecia y alta tasa de transferencia) con un comportamiento predecible y determinista (bajo jitter, consumo de recursos de memoria y procesador acotados, etc). DDS fue diseñado para cumplir con estos objetivos y es por ello la elección más recomendable en este tipo de sistemas.

El principal inconveniente de todos estos middleware comentados, al igual que ocurre con los orientados a RPC es la escabilidad. No obstante, los orientados a objetos distribuidos cuentan con un gran número de ventajas que los convierten en middleware potentes y flexibles que soportan peticiones síncronas pero que también incluyen soporte para comunicación asíncrona, transacciones distribuidas y mensajería, por lo que en muchos aspectos, pueden utilizarse para sustituir cualquiera de los middleware descritos en apartados anteriores.

2.2. Middleware de control

En este apartado se recogen algunos middleware de control orientados tanto robótica como a control de entornos industriales en general.

2.2.1. Orientados a control de entornos industriales

Dentro de esta categoría existen algunos desarrollos orientados al mundo de la industria como el presentado en [RDLS13] donde se describe un middleware *open-souce* de control específicamente diseñado para dar soporte al desarrollo de sistemas de control orientados a la fabricación industrial. Siguiendo esta línea, en el trabajo [FPLS11] se propone una arquitectura de un *núcleo de control* que permite ejecutar aplicaciones de control empotradas y que expone los principales problemas de este tipo de desarrollos. Si bien es cierto que ha sido aplicado sobre robots móviles para su validación, el diseño de este sistema presenta generalidades que pueden ser aplicadas a cualquier proceso industrial que precise de control en tiempo real.

Relacionados con el control en tiempo real, en el trabajo [GRHR07] se presenta una solución basada en DDS para sistemas de control distribuidos que pretende aplicar los beneficios de DDS con el fin de gestionar los requisitos de tiempo real. Lo mismo ocurre con el middleware descrito en [PCMC11] que utiliza nuevamente esta tecnología para ajustarse a un diseño basado en tiempo real para sistemas empotrados gestionados en entorno de una empresa.

2.2.2. Orientados a robótica

Algunos ejemplos de implementaciones middleware que se encuentran en el campo de la robótica son Open Robot Control Software (Orocos), CARMEN, NAOqi y Robot Operating System (ROS) entre otros.

Orocos consiste en un middleware específico que utiliza un modelo basado en componentes que ofrece una infraestructura de comunicación y soporte de tiempo real para sistemas de control. Además, proporciona herramientas para el intercambio de datos y servicios basados en eventos. Este middleware forma parte del proyecto que lleva su mismo nombre [Bru01] y que nace a partir de la necesidad de crear una alternativa de *tecnología abierta* para el desarrollo de arquitecturas de control para

robots. Un ejemplo de la arquitectura Orocos para su toolkit de tiempo real se muestra en la siguiente figura.

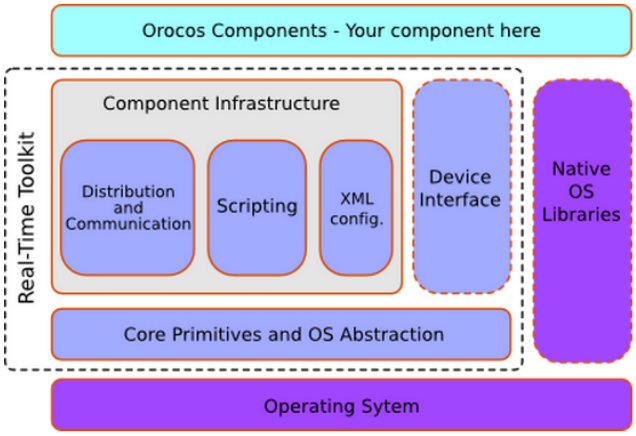

Figura 2.3: Arquitectura Toolkit Orocos de tiempo real

Actualmente, son numerosas las implementaciones basadas en Orocos como la descrita en [VCM+13] donde se realizan distintos algoritmos para el control dinámico de robots utilizando para ello una de las librerías de este middleware, *Orocos Toolchain*.

CARMEN consiste en una colección software de código abierto que está diseñado para proporcionar un conjunto básico de primitivas para robots abarcando una amplia variedad de plataformas comerciales. Fue diseñado en el año 2002 y como se describe en [MRT03] está organizado como una estructura de tres niveles de capas. La capa base se encarga de interactuar con el hardware ofreciendo un conjunto de abstracciones para las funciones base y de sensorización. También proporciona los bucles de control de bajo nivel para la rotación simple y movimiento en línea recta. La capa intermedia implementa las primitivas de navegación que incluyen localización, seguimiento de objetos dinámicos y planificación de movimientos. Por último, la capa restante está reservada para las tareas de nivel de usuario empleando primitivas de la capa media.

NAOqi es un middleware específicamente desarrollado para la plataforma NAO. Los NAO son robots humanoides con rasgos mecánicos, electrónicos y cognoscitivos que incorporan un *framework* de desarrollo que permite dotar al robot de funcionalidades abstrayéndose del hardware y centrándose únicamente en el desarrollo de software de control. Se compone de una estructura software basada en módulos que ofrecen un conjunto de métodos para la creación de comportamientos. Un ejemplo de representación se muestra en la siguiente figura:

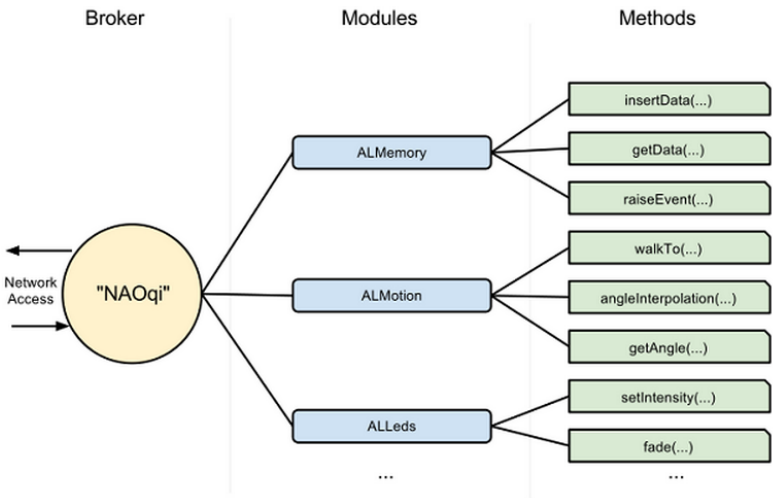

Figura 2.4: NAOqi: Estructura software basada en módulos

ROS es un middleware de código abierto para robots que permite el control de dispositivos a bajo nivel, paso de mensajes entre procesos y gestión de paquetes todo en uno. Además también incluye herramientas y librerías para escribir y ejecutar código a través de varios ordenadores. Este middleware funciona como una red P2P de procesos que se comunican entre sí utilizando una infraestructura de comunicación interna. ROS implementa diferentes estilos de comunicación, entre los que se incluyen las comunicaciones síncronas RPC como *Servicios*, el paso de mensajes asíncrono como *Topics* y un sistema de almacenamiento de datos que en ROS se conoce como el servidor de parámetros "Parameter Server". Por otro lado, gracias su condición de *open-source* actualmente existen numerosos repositorios en Internet que permiten a la comunidad de usuarios compartir sus desarrollos. En ocasiones estos desarrollos in-

corporan funcionalidades capaces de realizar tareas complejas como podría ser el caso de la localización y cartografía simultáneas. En este aspecto, el ámbito de aplicación de ROS continúa en aumento pues siempre es posible desarrollar librerías específicas para el uso de robots comerciales. Una visión general de este middleware se describe en [QCG+09].

2.3. Interfaces para configuración de middleware

Debido a la heterogeneidad de los middleware actuales, el desarrollo de una herramienta genérica que permita abarcar la configuración de cada uno de ellos resulta una tarea casi imposible. No obstante, existen interfaces específicamente diseñadas para ser utilizadas con determinados middleware. En este apartado se muestran algunos ejemplos.

Interfaz POLCOMS

En [WA05] se presenta el desarrollo de una herramienta gráfica en Java para el sistema POLCOMS (Proudman Oceanographic Laboratory Coastal-Ocean Modelling System) que implementa CORBA para la comunicación entre cliente y servidor. La interfaz está diseñada para permitir al usuario la visualización y parametrización en tiempo real de los datos de entrada y salida del sistema desde una estación de trabajo remota. El trabajo en el que se centra la GUI es en la adición de capacidades de visualización así como en el desarrollo de un puente CORBA entre la interfaz y POLCOMS. Este sistema se ejecuta en un servidor remoto mientras que la interfaz con la que interactúa el usuario lo hace desde el cliente. La estructura de la interfaces de POLCOMS y CORBA se muestran en la siguiente figura.

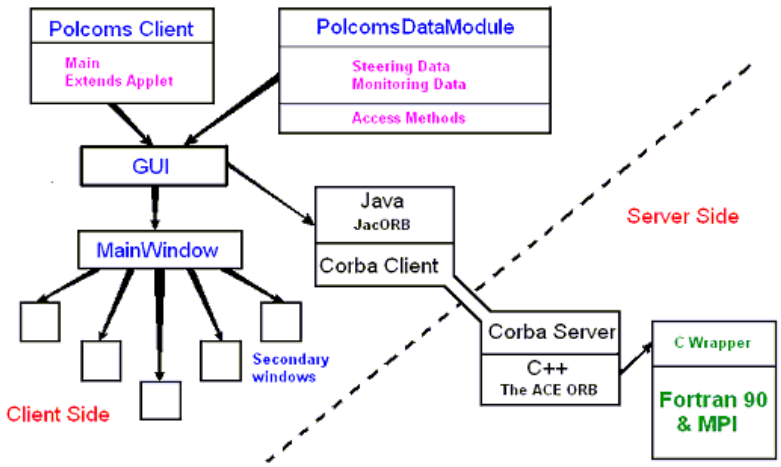

Figura 2.5: Estructura de la interfaz POLCOMS y la interfaz de CORBA

Entre todas las funcionalidades de la herramienta se destaca la sección de monitorización que como se muestra en la figura 2.6 permite especificar el tipo de dato y frecuencia a visualizar. Los datos son mostrados en tiempo real y se reciben vía CORBA.

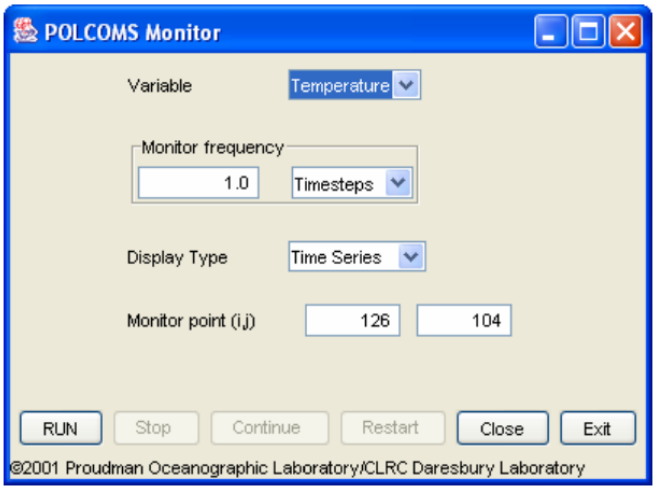

Figura 2.6: Ventana de monitorización de la interfaz POLCOMS

Interfaz *Choregraphe* para los robots NAO

Los robots NAO desarrollados por la empresa Aldebaran Robotics cuentan con una interfaz de programación y simulación llamada Choregraphe que permite a los usuarios definir los comportamientos del robot sin necesidad de tener conocimientos de programación. A través de un entorno de diagramas de bloques, cualquiera puede componer comportamientos con sólo arrastrar y soltar. Cada uno de estos bloques ofrece una conducta preprogramada que es fácilmente configurable a tavés de un sencillo editor que permite modificar los movimientos del robot.

La combinación entre los diferentes bloques ofrece grandes oportunidades en la programación del NAO sin tener que entrar en la complejidad de código. No obstante para usuario más avanzados, Choregraphe "entiende" el lenguaje de programación Python con el que es posible realizar llamadas a módulos propios desarrollados por

separado en lenguaje C++. Una visión general de Choregraphe se muestra en la siguiente figura.

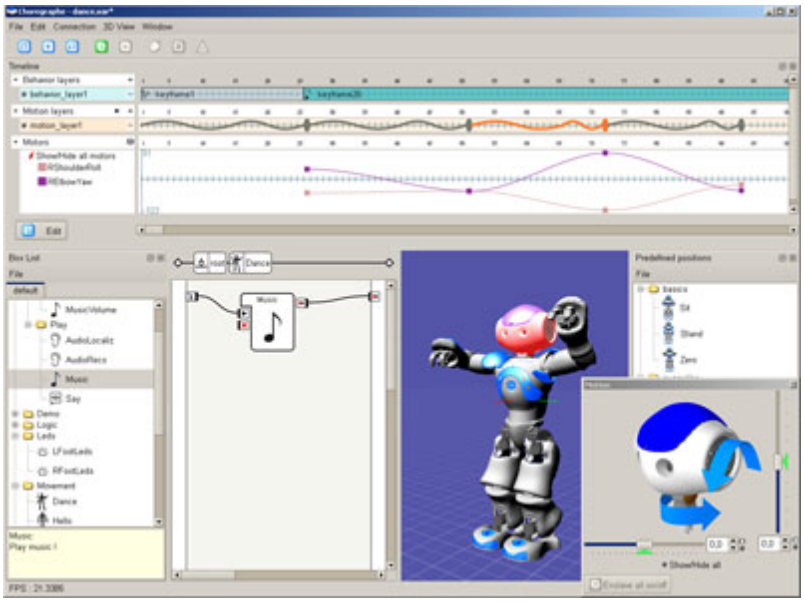

Figura 2.7: Interfaz Choregraphe para la configuración de robots NAO

rxDeveloper

El middleware ROS (comentado en el apartado 2.2.2) dispone de una herramienta gráfica desarrollada por [MHB] llamada rxDeveloper que permite la modificación de parámetros de los nodos forma interactiva. Gracias a su sencilla interfaz, un usuario puede simplemente arrastrar y soltar nodos nuevos (u otros ya existentes), modificar el valor de sus parámetros y guardar la configuración resultante en un fichero.

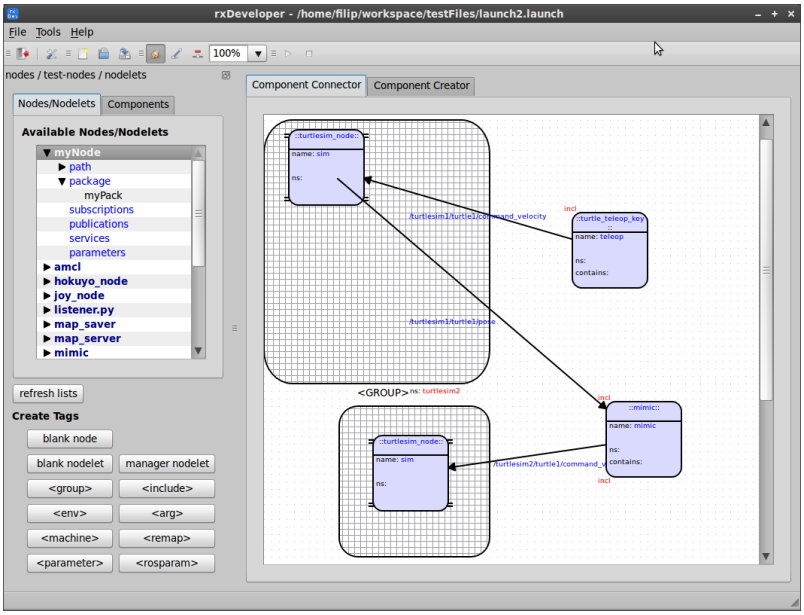

Figura 2.8: Interfaz rxDeveloper

CAPÍTULO 3

Control Kernel Middleware

3.1. Descripción

En el presente capítulo se describe el middleware basado en "Núcleo de control" sobre el que se ha desarrollado la herramienta de configuración. El término núcleo de control (o Control Kernel (CK) por las siglas en inglés de *Control Kernel*) es un concepto que se presentó en [ACS06] y que surge como una solución orientada al control de procesos. Se define como un conjunto de servicios e interfaces que se utilizan para la gestión de los componentes básicos de un sistema de control como son los sensores, actuadores y controladores. Desde hace varios años, el término CK se ha venido utilizando en diferentes publicaciones de artículos tales como [ACS06], [CAB+06] y [LCB+06] pero no ha sido hasta [CBSN08] cuando se ha propuesto un diseño de kernel de control estudiando cuales deberían ser las principales características y funcionalidades mínimas que se deberían incorporar. Tal y como se relata en el artículo, el propósito del CK consiste en *alcanzar la transparencia en el diseño y desarrollo de aplicaciones de tiempo real, ofreciendo una interfaz para la configuración de comportamientos, modos de trabajo y ejecución de los servicios que conforman el kernel.* Se entiende por lo tanto que se pretende diseñar una capa de software intermedia situada entre las aplicaciones de usuario y el sistema operativo (o la capa de abstracción del hardware, también conocida como Hardware Abstraction Layer (HAL)). En la figura 3.1 se muestra el esquema general del kernel de control definido por los autores.

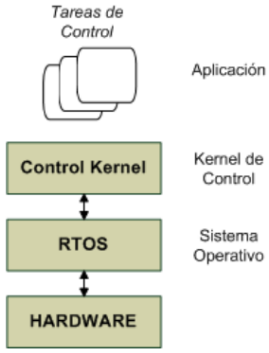

Figura 3.1: Esquema general del Kernel de Control

Desde el punto de vista de la implementación, el CK puede ser creado como parte del sistema operativo o como middleware. Implementarlo como parte del sistema operativo implica agregar los servicios y funcionalidades que requiere el kernel de control a los ya existentes en el propio SO. El inconveniente, es la reducción de la portabilidad entre sistemas empotrados debido principalmente a las modificaciones necesarias sobre el código nativo del SO. No obstante, una implementación basada en la tecnología middleware permite un diseño más rápido y transparente. Por este motivo, se decidió basar el desarrollo del kernel en este último para la creación de aplicaciones de control de tiempo real (esctrictas o no). Esta implementación bautizada como *Control Kernel Middleware* parte del conjunto de requerimientos y funcionalidades descritos en [CBSN08] y que se exponen a continuación:

Requerimientos

- El Control Kernel Middleware (CKM) debe asegurar una operación segura y fiable de todo el sistema de control de tiempo real. Para ello se deben incorporar mecanismos de detección y aislamiento de fallos.

- La actualización y distribución de la información dentro del CKM debe ser minimizada.

- Los retardos en las tareas de control dentro de CKM deberán ser tenidos en cuenta en los bucles completos de control, los cuales serán debidos a que al-

gunos de los recursos deberán ser compartidos dinámicamente entre diferentes actividades de cómputo.

- La utilización de la CPU deberá ser monitorizada y regulada mediante acciones que permitan su optimización.
- Cuando CKM se ejecute en entornos autónomos, el sistema deberá adaptar sus actividades a la energía disponible.
- Para permitir un comportamiento autónomo se deberá considerar la inclusión de mecanismos que ofrezcan capacidades de distribución, reconfiguración, y actualización de componentes software dentro del CKM para lo cual se apoyará en el paradigma de delegación de código.
- Es necesario considerar que en entornos dinámicos la prioridad, las señales disponibles, la cantidad de memoria y el tiempo de asignación de recursos para las actividades de control podrán cambiar dependiendo de las condiciones de trabajo.
- Los cambios en el entorno y la disponibilidad de recursos deberán ser controlados por un nivel de supervisión y resolución que involucre decisiones tales como la selección de las tareas de control más apropiadas en función del estado y el modo de operación actual del sistema.
- Aunque la mayoría de los algoritmos de control son manejados por tiempo mediante acciones separadas por intervalos constantes de tiempo, en entornos con cierta incertidumbre muchas veces es necesario disponer también de mecanismos para controlar eventos asíncronos.

Funcionalidades

- Adquisición de datos. Las medidas de sensores serán adquiridas según los períodos de tiempo requeridos por la aplicación y los apropiados para cada sensor. Los retardos en la adquisición de datos no deberán reflejarse como retardos en el sistema, es decir, el instante en que se capturan los datos no debe afectar al inicio en que se empieza el cómputo del controlador ni el instante en que se entregan las acciones de control.
- Se deberá asegurar que las acciones de control del sistema sean entregadas a tiempo, aún si estas no son el resultado del cálculo de controladores sino en su

lugar un *backup* de acciones de control o acciones de control seguras calculadas con datos de sensores previos.

- Degradado y estrategias de control seguro. En casos de emergencia se considerará la degradación del comportamiento del sistema o la aplicación de acciones de control grabadas anteriormente (*backup*) o la ejecución de determinadas acciones seguras (desconectar, llevar al sistema a un determinado nivel, etc) previamente establecidas. Junto a la configuración y delegación del código de control en el CKM es necesario definir las actividades seguras para los procesos a controlar.

- Distribución óptima del cómputo. Basándose en la delegación de código, los reguladores enviados al middleware se repartirán en el sistema distribuido en función de los requisitos de calidad de servicio acordados. Los reguladores enviados al middleware podrán delegarse en tiempo de ejecución entre los nodos del sistema distribuido para ser usados en diferentes situaciones.

- El middleware proporcionará transparencia en el acceso a datos distribuidos y en las comunicaciones entre aplicaciones de control, y entre éstas y dispositivos externos como sensores y actuadores, de tal modo que se puedan invocar componentes distribuidos sin importar su localización.

- Se utilizará un monitor para analizar el *performance* de los bucles de control del sistema con el objeto de cumplir los niveles de calidad contratados con la aplicación, tomando para ello las decisiones oportunas tales como conmutación de controladores, cambio de modo, desconexión, etc. Y para ello se monitorizará igualmente la utilización de los recursos del sistema.

- Se empleará una gestión de recursos adaptativa utilizando técnicas de calidad de servicio. Se establecerán niveles de calidad de servicio de ejecución para las aplicaciones de control mediante una negociación previa con el middleware.

- Las tareas serán planificadas de forma óptima basados en los objetivos actuales de control, los niveles de calidad contratados y la cantidad de recursos disponibles. Así mismo, se considerará en determinados casos el uso de tareas opcionales que consistirá en la división de algoritmos de control en partes obligatorias y opcionales, siendo esta última parte llevada a cabo en función del excedente de recursos.

- Adicionalmente a la delegación de diversos algoritmos de control, también se delegarán diferentes versiones de este código para ser ejecutado en múltiples arquitecturas hardware. De tal modo que se puedan formar *paquetes binarios universales* que permitan mover libremente el código de controladores dentro del sistema distribuido.

- Posibilidad de cambios en los parámetros de las tareas. Es bien sabido que en general la calidad de control disminuye si el período de muestreo se incrementea, pero también es sabido que a menor período de muestreo mayor serán los recursos requeridos. Por consiguiente, si la disponibilidad de datos sensoriales cambia, el período de la tarea de control también debería cambiar por optimización, lo cual implicaría cambios en los parámetros del controlador y de la información guardada.

3.2. Estructura del middleware

Como se comenta en el apartado anterior, el *Control Kernel Middleware* está diseñado pensando en el desarrollo de aplicaciones sobre sistemas empotrados de tiempo real. Actualmente existen gran variedad de estos sistemas, por lo que es aconsejable establecer diferentes niveles de servicios middleware en función de los recursos disponibles por cada uno de ellos. En este sentido el CKM incorpora una estructura basada en dos grupos de servicios. El primero está basado en una versión reducida (y más barata) pensando en las limitaciones del sistema donde se va a implantar. A esta versión se le conoce como Tiny Control Kernel Middleware (TCKM). El segundo grupo conocido como Full Control Kernel Middleware (FCKM) incorpora una versión completa que incluye las funcionalidades del TCKM más algunas propias. En los apartados 3.2.1 y 3.2.2 se muestra una descripción relativa cada uno de ellos.

3.2.1. Tiny Control Kernel Middleware

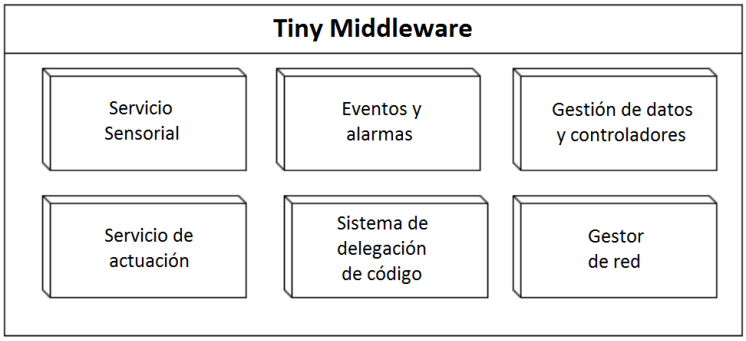

Figura 3.2: Servicios ofrecidos por Tiny Control Kernel Middleware

El TCKM está diseñado para trabajar en nodos de bajo rendimiento donde los recursos están limitados tanto a nivel de red como en capacidad de cómputo. Por lo general, estos nodos están físicamente más cerca del proceso a controlar y son los responsables de las operaciones a bajo nivel tales como las tareas de actuación o sensorización. La información que se considera relevante es intercambiada con el nodo de servicio (que implementa FCKM) y es el responsable de las tareas de alto nivel y de

delegación de código.

Los TCKM proporcionan también mecanismos de seguridad con el fin de mitigar y alertar acerca de operaciones inconsistentes. Para detectar estas situaciones, el TCKM incorpora tareas de vigilancia que supervisan el estado del sistema. El conjunto de servicios que ofrece esta versión de middleware se muestran en la figura 3.2 mientras que en 3.3 se encuentra la descripción relativa a cada uno de ellos.

3.2.2. Full Control Kernel Middleware

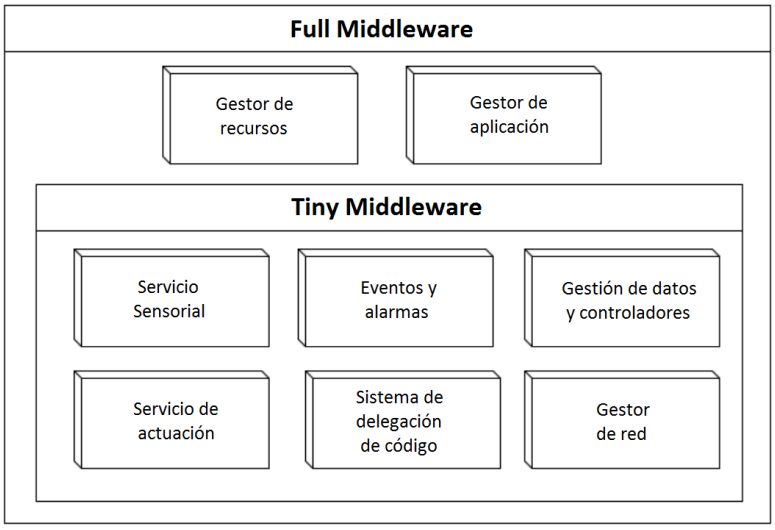

Figura 3.3: Servicios ofrecidos por Full Control Kernel Middleware

El FCKM es una extensión del TCKM que tiene por objetivo ofrecer una interfaz más completa a la aplicación de control. Los nodos de servicio de gran capacidad que implementan esta versión de middleware cuentan con procesadores empotrados potentes con sistemas operativos completos de tiempo real y con capacidades altas de entrada/salida y red. Estos nodos son los responsables de la ejecución de los bucles de control más complejos con el fin de proporcionar una calidad de control mayor. El

conjunto de servicios que conforman el FCKM están mostrados en la figura 3.3 y al igual que en el apartado anterior, sus funcionalidades se describen en el apartado 3.3.

3.2.3. Topología

Un ejemplo representativo sin pérdida de generalidad sobre la topología de una red de control distribuida que implementa CKM es el mostrado en la figura 3.4.

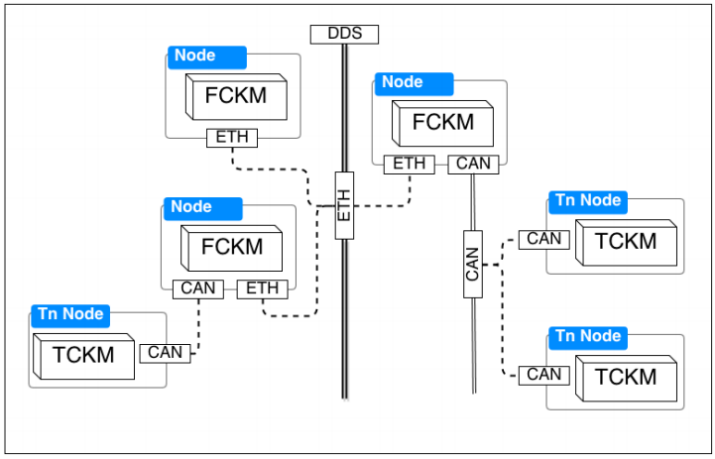

Figura 3.4: Topología de un sistema de control de tiempo real

En este caso se tiene una red distribuida donde cada nodo puede estar compuesto por limitaciones de recurso variable. Partiendo de esta premisa se distinguen dos grupos de nodo. Por un lado los conocidos como *nodos ligeros* caracterizados por estar más limitados a nivel de recursos. Estos nodos implementan el bus de comunicaciones CAN (cumpliendo su papel de bus específico de bajo coste). Por otro lado se distinguen los *nodos de gran alcance* que incorporan comunicación a través de red Ethernet.

3.3. Servicios del middleware

A continuación se describen cada uno de los servicios (descritos en [CBSN08] y [MMB+13]) por los que están compuestos el TCKM y FCKM mostrados en las figuras 3.2 y 3.3 respectivamente. Dichos servicios son: el gestor de recursos, gestor de aplicación, servicios sensorial y de actuación, eventos y alarmas, gestión de datos y controladores, sistema de delegación de código y el gestor de red.

3.3.1. Gestor de recursos

Se trata de un componente que se encarga de administrar el uso de los recursos físicos de los sistemas de cómputo tales como memoria, CPU, ancho de banda y consumo de energía entre otros. Los recursos deben ser gestionados y adaptados con el fin de satisfacer la calidad de servicio (Quality of Service (QoS)) especificada. La calidad se encarga de determinar el nivel de satisfacción de un servicio garantizando un cierto grado de rendimiento en su ejecución. Por esta razón, se establecen diferentes niveles de QoS cuyos criterios son acordados antes de que comience la ejecución. También deben definirse las condiciones para identificar situaciones de alta carga computacional con el fin de llevar a cabo mecanismos de adaptación adecuados. Una actuación temprana sobre situaciones anómalas puede evitar llegar a situaciones de mal funcionamiento que puedan resultar dificiles de resolver. Por otra parte, se garantiza un alto rendimiento del sistema ya que un nodo nunca estará sobrecargado mientras otro nodo del sistema presenta un alto nivel de recursos disponibles.

3.3.2. Gestor de aplicación

Define a una entidad software que ofrece una interfaz inmediata a las aplicaciones de control para acceder a los servicios ofrecidos por el CKM. A cada aplicación de control se le asignará un componente de este tipo. Este componente tendrá el *rol* principal de gestionar el acceso de la aplicación al entorno middleware.

3.3.3. Servicio de sensorización

Las mediciones del sensor son adquiridas de acuerdo con los periodos de tiempo requeridos por la aplicación. Los posibles retrasos en la adquisición de los datos no se deben ver reflejados como retrasos del bucle de control, por lo que el instante en el que el dato es capturado no debe afectar al cálculo del controlador ni al instante

en que es entregada la acción de control. Cualquier variación puede provocar lo que se conoce como *jitter* [ACES00] [MFFR01] lo que implica pérdidas en la calidad de control.

3.3.4. Servicio de actuación

El servicio de actuación es el encargado de suministrar la acción de control calculada en cada momento al actuador correspondiente. Para evitar posibles fallos en el sistema, se ha implementado un mecanismo de seguridad que ejecuta una rutina preestablecida en caso de que no se reciba cualquier acción de control desde el controlador. En tal situación, el programador puede elegir entre varias opciones tales como aplicar el último valor conocido o aplicar la operación de parada de seguridad. Siguiendo esta línea, el servicio de actuación permite utilizar el algoritmo descrito en [SCSB08] donde el modelo de control propuesto permite realizar un conjunto de actividades básicas para garantizar el funcionamiento seguro del sistema controlado. A fin de proporcionar esta funcionalidad, un conjunto de acciones de control futuras se extrapolan utilizando un control predictivo generalizado (GPC) del controlador, y se almacena por el servicio de actuación. Si no es posible cerrar el bucle, el servicio utiliza estas acciones guardadas hasta que el problema de pérdida de datos está resuelto.

3.3.5. Eventos y alarmas

En este componente se informa al nivel de aplicación acerca de los eventos que ocurren dentro del middleware de control. La notificación de estos eventos al nivel de aplicación se hace mediante interfaces preestablecidas por el gestor de aplicación. Las aplicaciones recibirán por defecto notificaciones de eventos propios del middleware tales como perdida de plazos de ejecución, errores de ejecución o cualquier otro tipo de situación anómala. Además, es posible instalar notificadores de eventos específicos a través de mecanismos de escucha conocidos como *listener*.

3.3.6. Gestión de datos y controladores

Se basa en la especificación de OMG DDS (comentado en el apartado 2.1.4) para la distribución de datos de forma predecible dentro del sistema distribuido de tiempo real. Este componente utiliza un modelo de intercambio céntrico de datos y más específicamente del modelo publicación/suscripción, el cual se construye sobre el concepto de espacio de datos global que es accesible por todas las aplicaciones intere-

sadas. En resumen, este servicio trata de obtener la transparencia entre los datos en la memoria local y los objetos distribuidos.

3.3.7. Sistema de delegación de código

Este componente permite el intercambio de fracciones de código entre los distintos nodos de cómputo del sistema distribuido. Sin embargo, sólo algunas de las funciones o módulos específicos son adecuados para ser intercambiados. El intercambio de secciones de código puede ser realizado en tiempo de ejecución por lo que el código puede ser ejecutado en cualquier momento sin necesidad de desconectar el nodo ni de dejar de controlar el proceso para delegar el nuevo controlador. Las principales delegaciones de código que se suelen realizar son del tipo para ejecutar un nuevo controlador. Esta funcionalidad permite que el middleware pueda mover código de un nodo concurrido a otro diferente sin carga. Los criterios para realizar estos movimientos de código dependen del componente de aplicación y del gestor de recursos, y más específicamente, de las peticiones propias de las aplicaciones, de la disponibilidad de recursos locales para el cumplimiento de las restricciones temporales de las aplicaciones, de la distribución física de sensores y/o actuadores y de sobrecargas en redes de comunicaciones, entre otras. Un ejemplo práctico donde se ha utilizado la delegación de código es posible encontrarlo en [CBSN08] donde se dispone de una implementación del tipo TCKM sobre un dsPIC30. En este caso, la delegación es efectuada por el Full Middleware de un nodo de altas prestaciones a través del bus de campo CAN [CBP+10].

3.3.8. Gestor de red

Se encarga de administrar el hardware de red, accediendo a los controladores ofrecidos por el sistema operativo. Gestiona el transporte de mensajes en el sistema distribuido ofreciéndole al middleware servicios de transmisión y recepción de datos a través de diversos buses de comunicaciones.

CAPÍTULO 4

Especificación del modelo

En el apartado 1.1 del documento se hacía referencia a la necesidad de disponer de una herramienta informática capaz de ofrecer al usuario la posibilidad de configurar cada uno de los elementos del sistema de una forma cómoda, rápida e intuitiva. Para la realización de esta tarea es necesario definir previamente un modelo que contemple cuales van a ser las capacidades de configuración asumibles por cada uno de los componentes implicados. En este sentido se presenta a lo largo del capítulo el modelo propuesto para la configuración del *Control Kernel Middleware*.

4.1. Modelo del nodo

Los nodos se caracterizan por no tener una funcionalidad específica asociada sino que representan un elemento físico que contiene a otros componentes como son sensores, controladores, actuadores, canales y mecanismos de comunicación. Cada nodo dispone de una relativa capacidad de cómputo (procesador, microprocesador) y una conectividad (Ethernet, CAN) en función de sus capacidades. La especificación del modelo de nodo plasma el conjunto de componentes contenidos así como la interrelación existente entre éstos con la red, con canales de entrada/salida y entre ellos mismos. Una representación gráfica del modelo sería la mostrada en la siguiente figura.

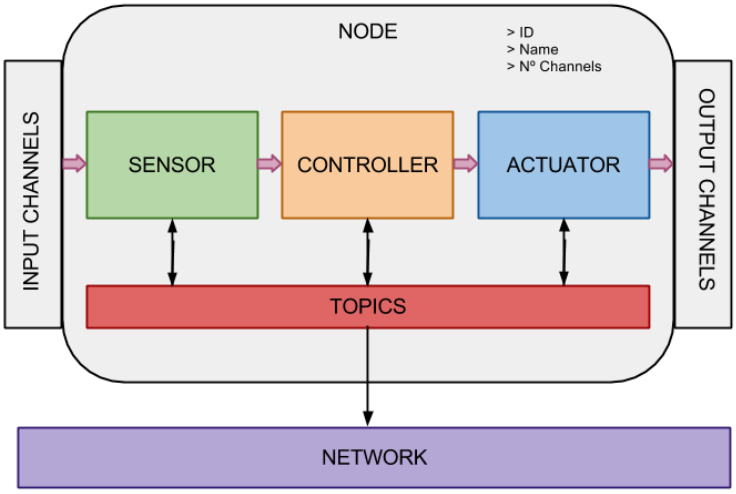

Figura 4.1: Representación modelo de nodo

En el modelo los **parámetros propios** que caracterizarán al nodo son:

- **ID**: Es el identificador único que lo diferencia entre el resto de nodos del sistema y que es representado mediante un número entero mayor o igual a cero. Se exige que todos los nodos tengan un ID definido.
- **Nombre**: Cadena de caracteres utilizada para etiquetar cada uno de los nodos utilizando un lenguaje natural. A diferencia del identificador, dos o más nodos pueden estar etiquetados bajo un mismo nombre. La especificación de este parámetro es opcional siendo una cadena vacía su valor por defecto.
- **Nº canales**: Indica el número de canales que dispone un nodo. No se establecen restricciones sobre el máximo y mínimo que un nodo puede contener por lo que el valor de este parámetro se sitúa entre 0 (valor por defecto) y 'n'.

A ambos extremos de la figura se representan los canales de entrada y salida de un nodo. Un canal es una representación lógica de un elemento físico de entrada/salida. Su configuración se realiza a través del parámetro "typeIO" que admite los siguientes tipos:

- **IN**: Indica que un canal está configurado de tipo entrada. Es el caso de los conectados a un sensor.
- **OUT**: Al contrario que el caso anterior, configura los canales como salida. Esta configuración es utilizada por los canales conectados a actuadores.
- **ND**: Es el tipo establecido por defecto que representa un estado de "no definido". A este tipo pertenecen los canales que no están conectados a ningún componente del nodo.

Las comunicaciones entre componentes se clasifican como internas o externas. Las internas hacen referencia a las existentes entre componentes de un mismo nodo mientras que las externas corresponde a comunicaciones con componentes de nodos distintos. En ambos casos se hace uso del concepto de topic. Un topic se define como un canal de comunicación lógico que sirve para transferir datos entre diferentes componentes. Desde el punto de vista del modelo, las capacidades de configuración de un topic se definen en el apartado 4.5.

4.2. Modelo del sensor

Los sensores son componentes lógicos que se conectan a los canales de entrada de un nodo para la recepción de la señal adquirida. El modelo de sensor plasma la configuración del conjunto de funcionalidades disponibles para el tratamiento de esta señal. La información resultante tras el proceso se publica en topics con el fin de ser accesible por el resto de componentes. La representación gráfica del modelo de sensor se muestra en la siguiente figura.

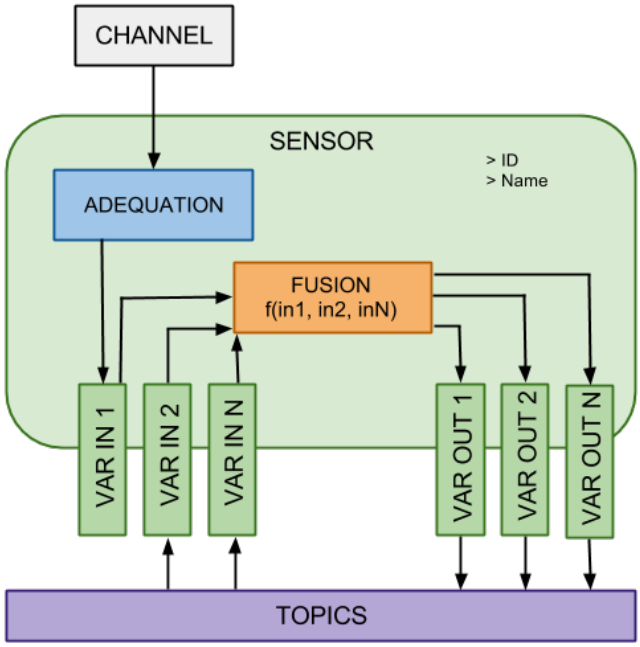

Figura 4.2: Representación modelo de sensor

Como se muestra en la figura anterior, existe una funcionalidad llamada "adecuación"que se encarga de ajustar escalas y rangos de la señal adquirida. El modelo dispone de 3 posibles adecuaciones:

- **Rango**: Coordenadas x,y de dos puntos que definen una recta. Se configura mediante 4 parámetros representados por valores mayores o igual a cero.

- **Saturación**: Coordenadas x,y de dos puntos que definen una recta. La peculiaridad respecto al caso anterior reside en que los valores 'y' son los máximos alcanzables.
- **Usuario**: Esta configuración permite al usuario definir por él mismo el tipo de adecuación a aplicar.

Cada sensor dispone de al menos una variable de entrada (donde almacena la señal tratada) y una variable de salida (que contiene el dato a publicar en el topic). El modelo no contempla restricciones sobre el número de variables máximas admitidas pues variará en función de la capacidad de memoria del nodo y por lo tanto, es el middleware quién internamente se encargará de gestionarlo.

Las capacidades de configuración que puede adquirir un sensor son variables. En el caso más sencillo el sensor puede simplemente adecuar la señal adquirida y publicarla en uno o varios topics bajo unas determinadas calidades de servicio. En configuraciones más complejas es posible tener más de un dato de entrada a tener en cuenta. En este caso se dispone de una funcionalidad llamada "fusión"que permite indicar el tratamiento que se debe realizar sobre el conjunto de datos de entrada. Las posibles configuraciones permitidas son:

- **Last**: Envía al topic el último dato recibido de entre todas las variables de entrada.
- **Mean**: Calcula la media entre los últimos datos de entrada recibidos. No utiliza 2 valores procedentes de una misma variable de entrada por lo que la media es calculada para cada último valor de cada variable.
- **User**: Permite al usuario especificar el tipo de fusión a realizar.

Finalmente, al igual que los nodos, los sensores disponen de parámetros propios que los caracterizan sobre el resto. Estos son:

- **ID**: Es el identificador único que lo diferencia entre el resto de sensores del sistema y que es representado mediante un número entero mayor o igual a cero. Se exige que todos los sensores tengan un ID definido.
- **Nombre**: Cadena de caracteres utilizada para etiquetar cada uno de los sensores utilizando un lenguaje natural. A diferencia del identificador, dos o más sensores pueden estar etiquetados bajo un mismo nombre. La especificación de este parámetro es opcional siendo una cadena vacía su valor por defecto.

4.3. Modelo del actuador

Los actuadores son componentes lógicos que se conectan a los canales de salida de un nodo para el envío de la señal de actuación a realizar. Del mismo modo que ocurre con el sensor, el modelo de actuador se encarga de definir las capacidades de configuración disponibles para la fusión y adecuación de los datos. La representación gráfica del modelo se muestra en la siguiente figura.

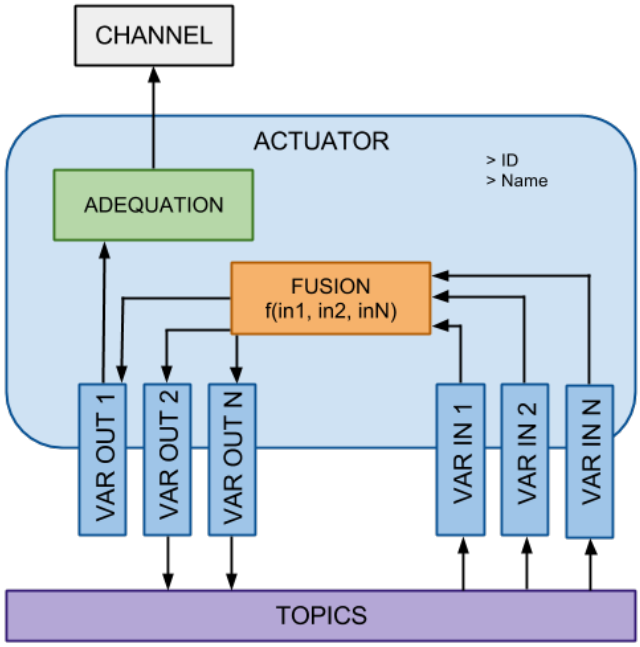

Figura 4.3: Representación modelo de actuador

El funcionamiento del actuador es el contrario al del sensor. Primero lee los datos de entrada procedentes de *topics* y les aplica una acción de "fusión" determinada en función de la configuración elegida por el usuario (*mean*, *last*, usuario). El resultado obtenido puede ser publicado en un *topic* a la vez que se aplica la adecuación determinada para ser enviada por el canal de salida del nodo.

Por último, los parámetros propios que definen al actuador son el "ID" y "nombre" cuyas características son idénticas a las ya definidas al final del apartado 4.2.

4.4. Modelo del controlador

El controlador es un componente puramente lógico puesto que no tiene ningún canal hardware asociado. Se encarga de realizar los cálculos necesarios para transformar los datos de entrada en datos de salida. La representación gráfica del modelo de controlador se muestra en la siguiente figura.

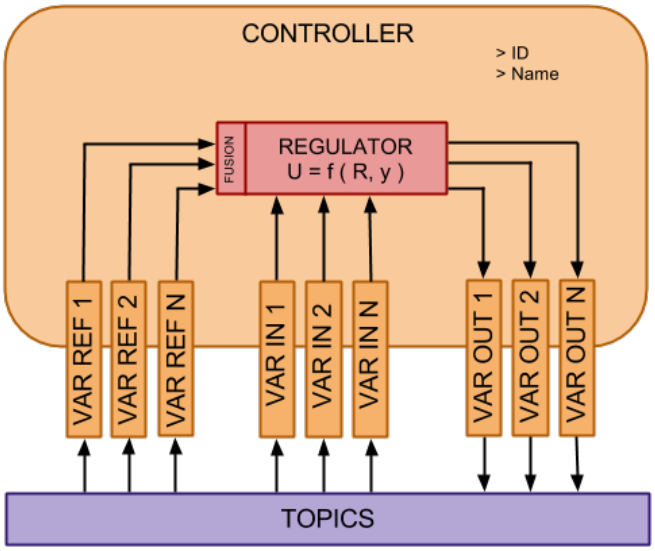

Figura 4.4: Representación modelo de controlador

El controlador se compone de un mínimo de dos entradas y una salida. El número máximo de salidas permitidas no se establece pues variará en función de la gestión del middleware de comunicaciones que tendrá que verificar si la agregación de un nuevo *topic* permitirá asegurar el correcto funcionamiento del sistema a apartir de las calidades de servicio ya existentes.

La fusión de datos es un paso previo al cálculo de la acción de control. Se refiere a los casos donde se tengan definidas más de una referencia y deba decidirse el tra-

tamiento a realizar. Las configuraciones posibles de este parámetro (*mean*, *last*) son idénticas a las ya descritas en apartados anteriores.

En el regulador los parámetros configurables son:

- P: Definido por 2 parámetros (K_P,T).
- PI: Definido por 4 parámetros (K_p, K_i, T, I_{max})
- PD: Definido por 3 parámetros (K_p, K_d, T)
- PID: Definido por 5 parámetros (K_p, K_i, K_d, T, I_{max})

Por último, los parámetros propios que definen al controlador constan de las mismas características que los ya definidos en el sensor y actuador (ver apartado 4.2).

4.5. Modelo de comunicaciones

Las comunicaciones entre componentes se realiza mediante el modelo publicación-suscripción. El middleware CKM actualmente hace uso de la implementación DDS que permite ofrecer una determinada calidad de servicio. La calidad se encarga de determinar el nivel de satisfacción de un servicio garantizando un cierto grado de rendimiento en su ejecución. Para ello se utiliza lo que se definen como parámetros de calidad que se emplean para describir características del servicio como el ancho de banda o el plazo temporal que se requiere de los mensajes, por medio de la lectura del mismo. En el ámbito de las redes de comunicaciones la calidad de servicio se entiende en términos como la capacidad de asegurar una tasa de datos en la red o ancho de banda, un retardo de los mensajes o una variación del retardo o *jitter*.

En el modelo de comunicaciones los parámetros de configuración establecidos son:

- **ID**: Identifica de forma única cada *topic* independientemente de su uso (interno o externo). Se representa mediante un valor numérico entero mayor o igual a cero. Se exige que todos los *topics* tengan un ID definido.
- **Deadline**: Se configura su valor pudiendo especificar valores en segundos, milisegunos y microsegundos. Se representan mediante enteros mayores o iguales a cero.

4.6. Representación del modelo en el lenguaje XSD

La configuración del middleware CKM hasta hace poco se realizaba a través de ficheros binarios previamente cargados en la memoria de los nodos. Una de la motivación de este trabajo consiste en utilizar un formato más legible y cómodo para el usuario, motivo por el cuál se ha migrado a ficheros de configuración representados en el lenguaje xml.

En este sentido, la representación del modelo se ha plasmado utilizando el lenguaje de esquemas XSD que permite definir la correcta estructura de los elementos en un documento xml. Este lenguaje también sirve de referencia para validar los datos que en él aparecen. Las características de este lenguaje se describen en el apartado 5.1.2.

CAPÍTULO 5

Interfaz gráfica

En este capítulo se presenta la herramienta informática que ha sido desarrollada a partir de la especificación del modelo definido en el capítulo anterior con el objetivo de facilitar la manipulación de los ficheros de configuración utilizados por el middleware. Una vista general de la interfaz se muestra en la siguiente figura.

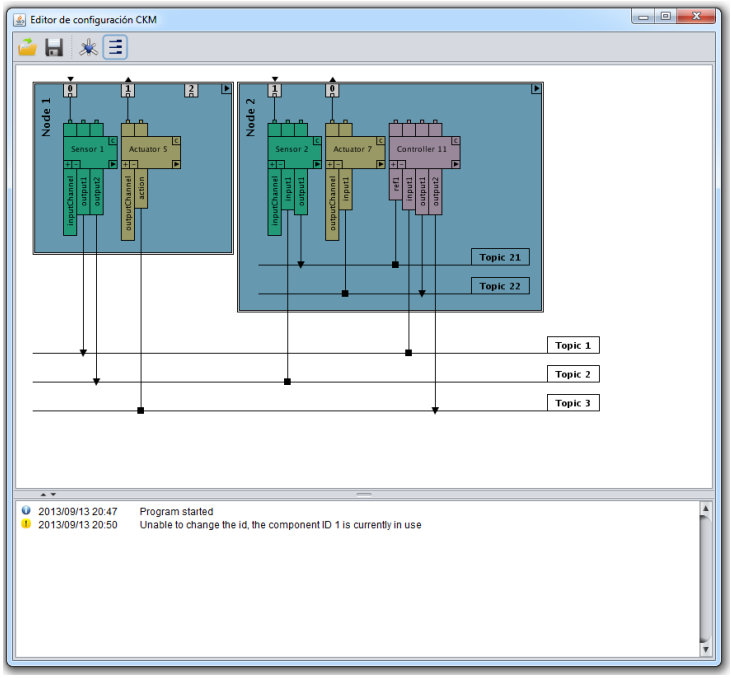

Figura 5.1: Vista general del editor

5.1. Criterios de diseño

En esta sección se realiza la presentación de los principales elementos software involucrados en el transcurso del trabajo aquí descrito con el fin de mostrar una perspectiva general de las herramientas usadas para llevar a cabo la implementación.

5.1.1. Entorno de desarrollo

El lenguaje de programación utilizado para la implementación de la interfaz ha sido Java. El motivo de esta elección se basa en las ventajas que ofrece la programación orientada a objetos así como la posibilidad de ejecutar la aplicación en cualquier máquina virtual de Java sin importar la arquitectura del sistema subyacente. Por otro lado, la elección de un entorno de desarrollo orientado a este lenguaje de programación no ha sido compleja. Actualmente tanto Eclipse como Netbeans están especialmente orientados al desarrollo de aplicaciones basadas en Java. No obstante, la familiaridad adquirida con Eclipse durante trabajos previos ha sido el motivo por el cuál se ha escogido a este entorno de desarrollo para la implementación de la interfaz.

5.1.2. Plug-ins y librerías

El entorno de desarrollo eclipse permite la adhesión de nuevas funcionalidades a través de lo que se conoce como *plug-in*. En el desarrollo de esta herramienta se han hecho uso de algunos de ellos con el propósito de facilitar su implementación. A continuación se muestra una descripción relativa a los *plug-ins* y librerías utilizados:

- **WindowBuilder**: *Plug-in* disponible en Eclipse que permite la creación de aplicaciones GUI de Java de forma rápida e intuitiva. Mediante la técnica de arrastrar y soltar es posible añadir o eliminar componentes visuales generándose de forma automática el código Java equivalente. Este *plug-in* también incorpora un editor para gestionar eventos si bien no se ha llegado a hacer uso de él principalmente porque han sido definidos desde cero pensando en las necesidades de la aplicación.
- **RSyntaxTextArea**: Librería disponible en Java que permite el resaltado de sintaxis para ayudar al usuario con su desarrollo.

- **XSD Core**: Es una herramienta similar a WindowBuilder que permite definir las restricciones y estructuras de forma visual en un XSD. Su principal utilidad (y para lo que se ha usado realmente este *plug-in*) es la posibilidad de generar ficheros xml automáticamente a partir de las estructuras definidas. De forma inversa, permite validar un fichero xml con un determinado XSD. En el apéndice C del documento se incluyen los XSD creados en este trabajo.
- **JAXBBuilder**: Este *plug-in* disponible en Eclipse permite la generación de clases Java a partir de ficheros XML, XSD o JSON.
- **Log4j**: Es una librería adicional de java que permite a nuestra aplicación mostrar mensajes de información sobre lo que está sucediendo en ella. Habitualmente se conoce como *log*.

5.2. Elementos de interacción

En las siguientes secciones se muestran cada una de las representaciones gráficas de los componentes descritos en el modelo junto con su modo de configuración. Así mismo, se detallan restricciones adicionales que se han tenido en cuenta a nivel de programación debido a que no pueden controladas durante la validación del xml sobre el xsd. Ejemplos de este tipo son la detección de valores repetidos en los identificadores o conexiones de variables simultáneas a canales y *topics*.

5.2.1. El nodo

El nodo es el elemento sin funcionalidad específica que contiene a sensores, actuadores, controladores y canales. Se presenta al usuario en tono azul (ver figura 5.2) y con dos modos de vistas posibles, una minimizada (Nodo 1) diseñada para la comodidad del usuario (los nodos alcanzan una anchura considerable) y otra vista expandida (Nodo 2) donde se visualizan los componentes contenidos. La ventana de configuración mostrada a la derecha de la imagen permite al usuario modificar los parámetros propios del nodo que de acuerdo a la especificación del modelo son el ID, nombre y número de canales.

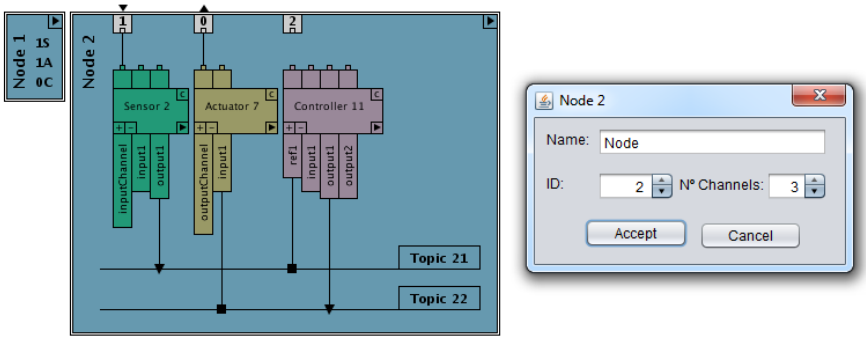

Figura 5.2: Representación gráfica del nodo

La configuración de los canales se establece automáticamente en función del componente al que se conecta (entrada en caso de sensores y salida para actuadores) y se representa gráficamente mediante una flecha que indica el sentido de canal. Los establecidos como no defnidos (ND) no se les aplica representación como es el caso

del canal 2.

A partir del modelo, las restricciones internas aplicadas al nodo son:

- El valor por defecto de un canal se establece en "ND". A este tipo también pertenecen canales que has sido desconectados de un componente.
- El valor del parámetro ID es obligatorio, por tanto se generan identificadores por defecto tanto para nodos como el resto de componentes. La herramienta incorpora una funcionalidad que detecta si un identificador no cumple con los requisitos establecidos (número entero mayor o igual a cero y no repetido). En caso de error se gestiona notificando al usuario del fallo junto con una sugerencia de ID válido y disponible para ser utilizado.

5.2.2. El sensor

La representación del sensor se puede abordar desde dos puntos de vista: la interacción con el resto de componentes y la configuración de parámetros del mismo. Representado en tono verde (ver figura 5.3) un sensor conecta con un canal para la adquisición de la señal mientras que el resto de sus variables conectan a *topics* para la recepción o envío de datos a otros componentes. Las conexiones se crean gráficamente arrastrando y soltando una variable sobre el canal o *topic* al que se quiere conectar.

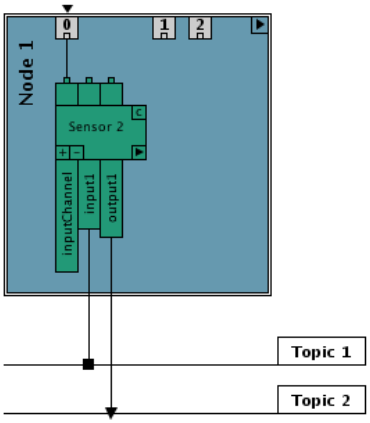

Figura 5.3: Representación gráfica del sensor

Siguiendo la especificación del modelo, la capacidades de configuración se agrupan entre parámetros propios, adecuación y fusión. La figura 5.4 muestra el estilo en que los datos son presentados al usuario donde la parte superior permite la edición de los parámetros "nombre" e "ID", la sección central añade, modifica o elimina las variables asociadas al sensor y en la zona inferior se configura la funcionalidad del bloque "fusión".

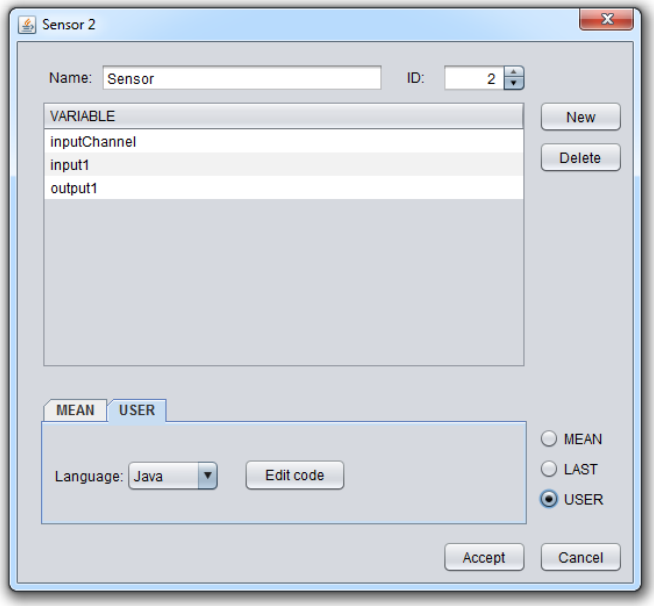

Figura 5.4: Ventana de configuración de parámetros del sensor

En el modelo la adecuación de señal está directamente relacionada con la variable asignada al canal. En este sentido la herramienta gestiona de forma automática esta asignación. Esto implica una total libertad para el usuario a la hora de gestionar indistintamente el uso de variables. La ventana de configuración para el bloque de adecuación se muestra en la siguiente figura.

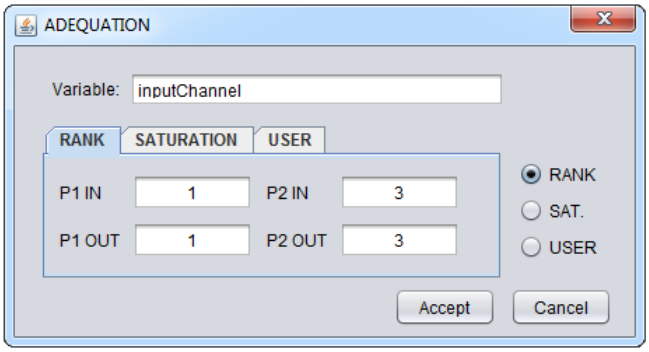

Figura 5.5: Ventana de configuración para la adecuación de señal

Tanto las funcionalidades de adecuación como fusión permiten al usuario definir un comportamiento propio. Es por ello que la herramienta incorpora una ventana de inserción de código haciendo uso de la librería *RSyntaxArea* que soporta la mayoría de lenguajes de programación si bien únicamente tan sólo han sido configurados Java, C++ y XML.

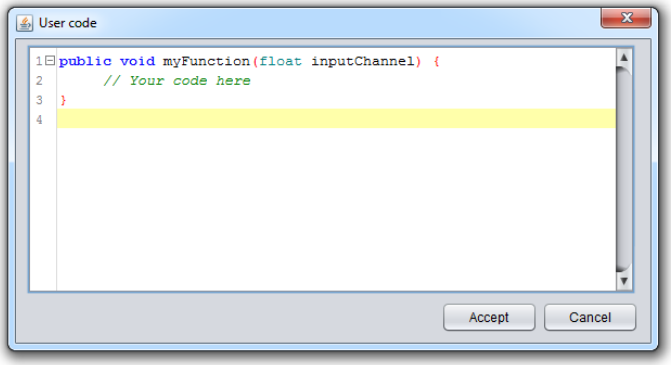

Figura 5.6: Ventana inserción de código de RSyntaxArea

Finalmente, las restricciones aplicadas a nivel de aplicación sobre el sensor son:

- Un sensor no puede estar conectado a más de un canal del nodo.

- Las variables no pueden ser bidireccionales (entrada y salida), por tanto se restringe la conexión con uno u otro componente (canal o *topic*).
- Aún siendo una variable únicamente de entrada, no puede estar asiganada a dos o más *topics* simultáneamente.
- Cuando un sensor es instanciado se incluye por defecto un mínimo de una variable de entrada y una de salida.

5.2.3. El actuador

Desde el punto de vista de diseño, actuadores y sensores son considerados como uno solo. Ambos incorporan funcionalidades para la adecuación y fusión de la información, contienen los mismos parámetros propios y comparten restricciones en el modo que las variables conectan a canales o *topics*. La diferencia más notable es el sentido de flujo de datos configurado en el canal del nodo. Desde el punto de vista de la programación ambos componentes son instanciados sobre la misma clase de modo que a efecto de ventanas de configuración los actuadores utilizan las ya descritas en las figuras del apartado anterior. En la siguiente figura se muestra la representación del actuador en la interfaz.

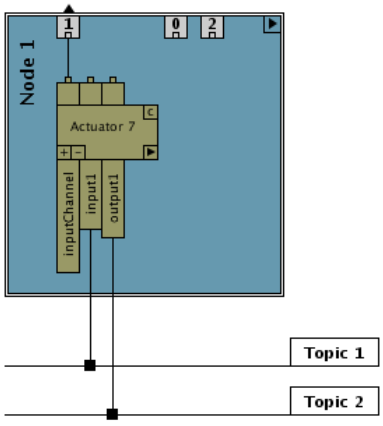

Figura 5.7: Representación gráfica del actuador

5.2.4. El controlador

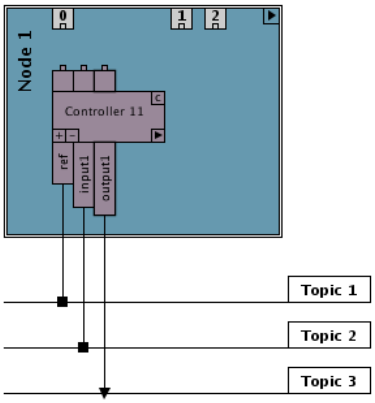

Figura 5.8: Representación gráfica del controlador

Como se muestra en la figura anterior el controlador tiene un aspecto idéntico a los componentes descritos anteriormente. Representado en tono morado limita sus conexiones a *topics* exclusivamente. Esto se debe a su condición de componente puramente lógico que implica no tener ningún canal hardware asociado. Siguiendo la especificación del modelo el controlador dispone únicamente de la funcionalidad de fusión. La ventana de configuración propia de este componente se muestra en la figura 5.9. La parte superior se corresponde al igual que en sensores y actuadores a la gestión de variables y edición de parámetros propios. La parte inferior incluye los parámetros configurables sobre el tipo de control a aplicar. Como es habitual se permite al usuario definir una configuración propia.

Las restricciones establecidas a nivel de aplicación sobre el controlador son:

- No se permite conectar variables con los canales del nodo.
- Cuando se crea un controlador se dispone por defecto de un mínimo de una variable de entrada y una de salida.
- Cuando un controlador es instanciado se incluye por defecto un mínimo de una variable de entrada y una de salida.

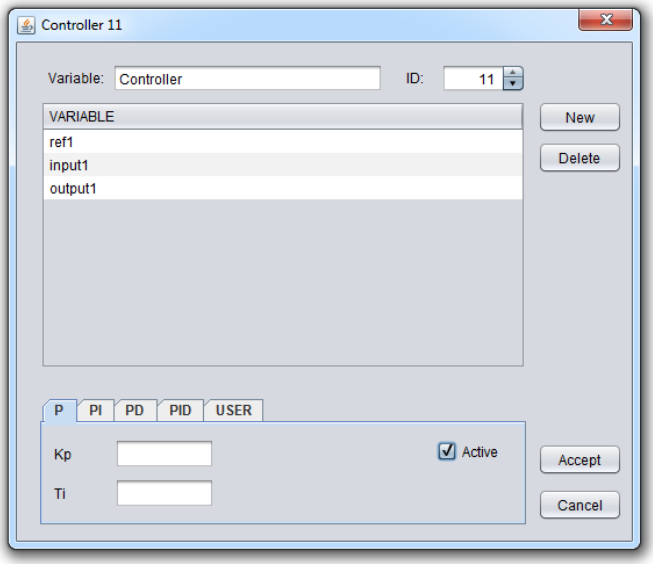

Figura 5.9: Ventana de configuración de parámetros del controlador

5.2.5. El topic

Los *topics* son representados en la interfaz como simples líneas donde las variables pueden enlazarse. En la parte inferior de la figura 5.10 se muestra la simbología utilizada para diferenciar si una variable se enlaza a modo lectura o escritura. Para el primer caso, se utiliza un cuadrado mientras que la escritura se reprenseta mediante un triángulo.

En la parte superior de la figura se muestra la ventana de configuración del *topic* donde se especifican las calidades de servicio. Actualmente tan sólo se configura el parámetro *deadline* si bien en el futuro se irán ampliando.

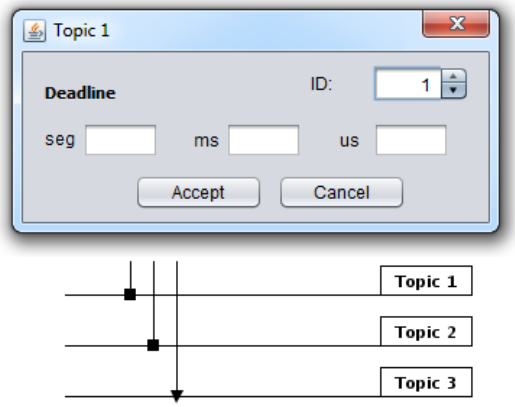

Figura 5.10: Configuración y representación del topic

5.3. Funcionalidades

Hasta ahora se ha mostrado la interfaz desde el punto de vista de representación y capacidades de configuración de los componentes descritos en el modelo del capítulo 4. En esta sección se pretende describir las funcionalidades hasta ahora no comentadas que incluye este editor.

Funcionalidades

- Escritura en fichero. La herramienta permite la generación automática del fichero xml equivalente a partir del diseño establecido en la interfaz. Dado que la aplicación instancia las clases generadas de forma automática a partir del modelo xsd, la garantía de que el xml resultante esté validado respecto al modelo se puede considerar del 100 %.

- Lectura de fichero. Del mismo modo que es posible generar el xml a partir del diseño el paso inverso también está contemplado. El procesamiento del fichero se hace en máximo dos fases. En un primer procesamiento se valida que las estructuras y tipos de datos introducidos son correctos. Del mismo modo se valida que los enlaces entre componentes cumplen con las restricciones correspondientes. En caso de error se identifica el motivo. Si es un error en la estructura o tipo de dato especificado se muestra un aviso por consola indicando que ha fallado y

por qué ha fallado. En caso de ser un error en conexiones entre componentes se hace un segundo procesamiento donde se genera el escenario especificado pero no sus conexiones.

- Consola de errores. Mediante el uso de la librería *log4j* se ha incluido una consola de errores a modo de *log* que notifica al usuario de posibles errores cometidos.

- Vistas. Pese a que la aplicación permite redimensionar el tamaño de la ventana se puede dar el caso donde el número de nodos sea tal que resulte incómodo trabajar con la herramienta. En este sentido se ha definido un sistema de vistas con dos modos posibles. La idea es que los nodos sobre los que no se está trabajando ocupen el menor espacio posible. Se representan como pequeñas cajas con una leyenda que indica el número de sensores, actuadores y controladores que tiene definidos. La segunda vista es la utilizada por defecto donde se visualiza el contenido de los nodos.

- Distribución de nodos. Arrastrando de forma lateral cada nodo es posible intercambiarlo con su compañero inmediato. Esta funcionalidad se incluye para facilitar al usuario una organización de nodos durante su diseño.

- Reorganización de componentes. Con el objetivo de ofrecer un diseño estético al usuario los componentes de un nodo se reorganizan de forma automática cuando se enlazan a canales. De esta forma los canales utilizados se sitúan sobre el componente enlazado y lo mismo ocurre con la organización de las variables de un componente.

- Menús. Todos los elementos de interacción del nodo disponen de menús accesibles mediante el botón secundario del ratón con el fin de ofrecer un fácil acceso a todos los elementos de configuración.

5.4. Ejemplo de configuración

En esta sección se presenta el diseño realizado sobre la interfaz gráfica para la configuración de los tipos a y b de vehículo de Braitenberg mostrado en la figura 5.11. Estos vehículos tienen sensores primitivos que miden algún estímulo en un punto y ruedas dirigidas por sendos motores. Dependiendo de cómo los sensores y las ruedas están conectados, el vehículo exhibe diferentes comportamientos. Esto quiere decir que parecen esforzarse por alcanzar determinadas situaciones y evitar otras, cambiando de rumbo cuando la situación cambia.

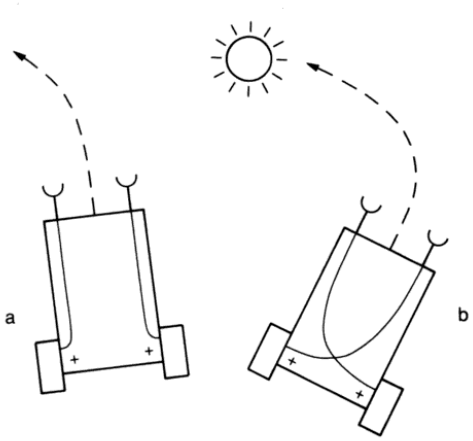

Figura 5.11: Vehículo de Braitenberg

El tipo 'a' de vehículo conecta los sensores con el motor de su mismo lado. Cuando el sensor capta un foco de luz la velocidad de la rueda se incrementa y por tanto provoca que el vehículo se aleje. La equivalencia gráfica sobre la interfaz de esta configuración de vehículo se muestra en la siguiente figura.

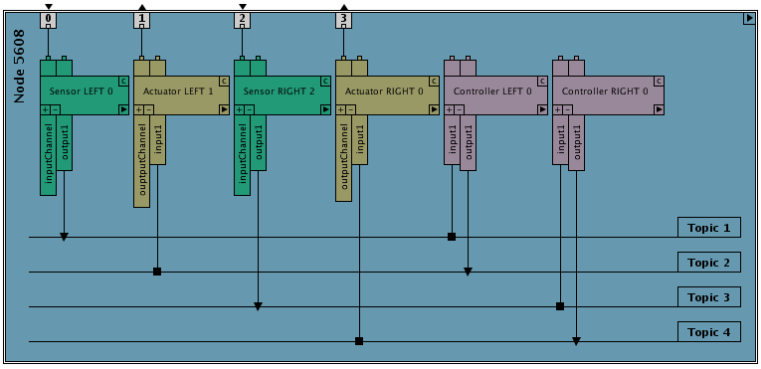

Figura 5.12: Configuración vehículo a

Por otro lado, cuando la conexión entre sensores y ruedas se intercambia el efecto conseguido es el contrario. El sensor que detecta un aumento de luz provoca que su rueda opuesta acelere por lo que el vehículo termina acercándose. En la siguiente figura se representa la configuración para el tipo b.

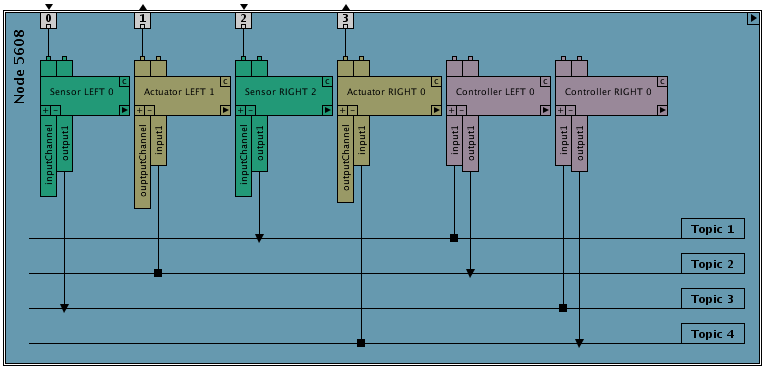

Figura 5.13: Configuración vehículo b

Como se puede observar, el único cambio para pasar de una configuración a otra ha sido modificar las conexiones de los dos sensores respecto al topic donde estos publican. Del mismo modo, si en lugar de modificar las conexiones de los sensores se hubiese hecho en los actuadores también habría sido una configuración válida.

CAPÍTULO 6

Conclusiones

Para finalizar con el documento se procede a recopilar las conclusiones sobre el trabajo aquí expuesto, en ellas se recordarán los principales objetivos propuestos al inicio del documento y se compararán con los resultados obtenidos a fin de sopesar cual ha sido el nivel de cumplimiento alcanzado. Tras ello, se realizará una revisión de las líneas de trabajo futuro originadas gracias a la contribución de este trabajo.

6.1. Conclusión

A la conclusión de este trabajo, se ha realizado una revisión del estado del arte donde se han presentado diferentes middleware tanto de comunicación como de control describiendo sus principales características así como algunos ejemplos de interfaces basadas en los mismos. Esta revisión ayuda a obtener una perspectiva general sobre el desarrollo basados en middleware hoy día.

Posteriormente se han analizado las características del *Control Kernel Middleware* que ha servido como base para el desarrollo de este trabajo. Su estructura ha sido expuesta mediante la enumeración de los servicios proporcionados así como su distinción entre ambas versiones (TCKM, FCKM).

El análisis sobre las capacidades de configuración soportadas por el middleware de control ha derivado en la especificación de un modelo que contiene información precisa sobre la definición de los parámetros requeridos para realizar una solicitud de servicio y su configuración.

El diseño de la interfaz gráfica ha logrado abstraer al ingeniero de control de la necesidad de conocer los detalles de implementación del modelo. Mediante el aprovechamiento de librerías desarrolladas para Java se han podido interpretar el modelo definido en el lenguaje de esquemas XSD así como gestionar y almacenar distintas configuraciones del sistema usando ficheros xml.

Finalmente se ha mostrado un ejemplo de configuración en el cuál se ha validado esta implementación usando el paradigma de los robots de Braitenberg. Gracias a este experimento se ha podido comprobar la facilidad que ofrece la interfaz para realizar cambios de configuración en tiempo de ejecución pudiendo intercambiar entre los dos modelos de robot (tipo a y tipo b) de forma instantánea. El uso de la interfaz evita al usuario la modificación de código o recompilación del mismo.

6.2. Trabajo futuro

En el desarrollo del trabajo se han constatado que existen diferentes aspectos en los que se podría continuar su mejora como son:

- La ampliación en las capacidades de comunicación del sistema de comunicaciones.
- La definición de ficheros de descripción de recursos de nodo que contendrían el número y tipo de canales de entrada/salida del nodo físico, canales de comunicación, etc. Estos ficheros serían integrados en la aplicación para ofrecer restricciones en el diseño de acuerdo a lo especificado.
- Ficheros de *log* que permitan mantener un histórico de operaciones realizadas de modo que la herramienta sea capaz de deshacer un determinado número de acciones aplicadas.

APÉNDICE A

Acrónimos

DCS Distributed Control Systems 1

XSD XML Schema Definition 3

2PC Two-Phases Commit Protocol 9

JMS Java Message Service 9

RMI Remote Method Invocation 9

CORBA Common Object Request Broker Architecture 9

MOM Message-Oriented Middleware 10

RPC Remote Procedure Call 11

IDL Interface Definition Language 11

POO Programación Orientada a Objetos 11

JVM Java Virtual Machine 12

DDS Data Distribution Service for Real-time Systems 12

RSTJ Real-Time Specification for Java 12

RMI-HRT RMI - Hard Real-Time 12

OMG Object Management Group 13

Orocos Open Robot Control Software 14

ROS Robot Operating System 14

HAL Hardware Abstraction Layer 23

CK Control Kernel 23

CKM Control Kernel Middleware 24

TCKM Tiny Control Kernel Middleware 28

FCKM Full Control Kernel Middleware 28

QoS Quality of Service 31

Bibliografía

[ACES00] K-E Arzén, Anton Cervin, Johan Eker, and Lui Sha. An introduction to control and scheduling co-design. In *Decision and Control, 2000. Proceedings of the 39th IEEE Conference on*, volume 5, pages 4865–4870. IEEE, 2000.

[ACS06] P Albertos, A Crespo, and José Simó. Control kernel: a key concept in embedded control systems. In *4th IFAC Symposium on Mechatronic Systems*, 2006.

[Bru01] Herman Bruyninckx. Open robot control software: the orocos project. In *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, volume 3, pages 2523–2528. IEEE, 2001.

[CAB+06] A Crespo, Pedro Albertos, P Balbastre, M Vallés, M Lluesma, and José Simó. Schedulability issues in complex embedded control systems. In *Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control, 2006 IEEE*, pages 1200–1205. IEEE, 2006.

[CBP+10] JO Coronel, F Blanes, P Pérez, M Albero, G Benet, and JE Simó. Scocan: Un protocolo de comunicaciones de tiempo real para sistemas empotrados distribuidos. aplicación al control de robots. *RIAII*, 3(2):71–78, 2010.

[CBSN08] Javier O Coronel, Francisco Blanes, José Simó, and Vicente Nicolau. Middleware de kernel de control para el desarrollo de aplicaciones en sistemas empotrados de tiempo real. *XXIX Jornadas de Automática*, 2008.

[COR95] Object Management Group CORBA. The common object request broker (corba): Architecture and specification. Technical report, 1995.

[FC08] Jaime Martín (eProsima) y Gerardo Pardo (RTI) Fernando Cases, OT TICS. Conceptos de middleware: DDS como tecnología emergente en Defensa. Technical report, Septiembre 2008.

[FPLS11] Adel Fernández Prieto and Orestes Llanes-Santigao. Núcleo de control para sistemas empotrados de control: Una propuesta de arquitectura. 2011.

[GRHR07] Tarek Guesmi, Rojdi Rekik, Salem Hasnaoui, and Houria Rezig. Design and performance of dds-based middleware for real-time control systems. *IJCSNS*, 7(12):188–200, 2007.

[Gro08] Object Management Group. Uml profile for dds specification. *OMG Technical Meeting, Otawa, Canada*, 2008.

[HB02] M Hapner and R Burridge. Java message service. Technical report, 2002.

[LCB+06] Manuel Lluesma, Anton Cervin, Patricia Balbastre, Ismael Ripoll, and Alfons Crespo. Jitter evaluation of real-time control systems. In *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, pages 257–260. IEEE, 2006.

[LHD] [online]URL: `http://www.iuma.ulpgc.es/users/lhdez/inves/tesis/memoria-tesis/node2.html` [cited 2013].

[MFFR01] Pau Martí, Josep M Fuertes, Gerhard Fohler, and Krithi Ramamritham. Jitter compensation for real-time control systems. In *Real-Time Systems Symposium, 2001.(RTSS 2001). Proceedings. 22nd IEEE*, pages 39–48. IEEE, 2001.

[MHB] Filip Müllers, Dirk Holz, and Sven Behnke. rxdeveloper: Gui-aided software development in ros.

[MMB+13] Manuel Muñoz, Eduardo Munera, Francisco Blanes, José Simó, and Ginés Benet. Event driven middleware for distributed system control. 2013.

[MRT03] Michael Montemerlo, Nicholas Roy, and Sebastian Thrun. Perspectives on standardization in mobile robot programming: The carnegie mellon navigation (carmen) toolkit. In *Intelligent Robots and Systems, 2003.(IROS*

2003). Proceedings. 2003 IEEE/RSJ International Conference on, volume 3, pages 2436–2441. IEEE, 2003.

[PCMC11] Yunjung Park, Duckwon Chung, Dugki Min, and Eunmi Choi. Middleware integration of dds and esb for interconnection between real-time embedded and enterprise systems. In *Convergence and Hybrid Information Technology*, pages 337–344. Springer, 2011.

[QCG+09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

[RDLS13] Olivier Roulet-Dubonnet, Morten Lind, and Amund Skavhaug. Icehms, a middleware for distributed control of manufacturing systems. In *Industrial Applications of Holonic and Multi-Agent Systems*, pages 95–105. Springer, 2013.

[SCSB08] Raul Simarro, J Coronel, Jose Simó, and Juan F Blanes. Hierarchical and distributed embedded control kernel. In *17th IFAC World Congress*, 2008.

[TCA08] Pablo Daniel Tejera Carballa and Alejandro Alonso. Middleware de comunicación para sistemas distribuidos de tiempo real en java. 2008.

[VCM+13] Marina Vallés, Jose I Cazalilla, Vicente Mata, et al. Implementació n basada en el middleware orocos de controladores dinámicos pasivos para un robot paralelo. *Revista Iberoamericana de Automática e Informática Industrial RIAI*, 10(1):96–103, 2013.

[WA05] R Wain and M Ashworth. *A Java GUI and Distributed COBRA Client-server Interface for a Coastal Ocean Model*. Council for the Central Laboratory of the Research Councils, 2005.

Ficheros XSD

C.1. kmConfig.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/km"
        elementFormDefault="qualified" xmlns:km="http://www.example.org/km">

        <include schemaLocation="node.xsd" />

        <element name="kmConfig">
                <complexType>
                        <sequence>
                                <element name="node" type="km:typeNode"
                                maxOccurs="unbounded" minOccurs="0" />
                                <element name="topic" type="km:typeTopic"
                                maxOccurs="unbounded" minOccurs="0" />
                        </sequence>
                </complexType>
        </element>
</schema>
```

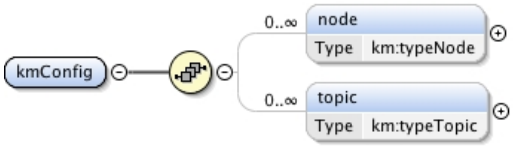

Figura C.1: Esquema XSD: kmConfig

C.2. node.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/km"
        elementFormDefault="qualified" xmlns:node="http://www.example.org/km">

<include schemaLocation="sensor.xsd" />

<complexType name="typeNode">
        <sequence>
                <element name="name" type="node:kmStringName" />
                <element name="channel" type="node:typeChannel"
                                        maxOccurs="unbounded" minOccurs="0" />
                <element name="sensor" type="node:typeSensor"
                                        maxOccurs="unbounded" minOccurs="0" />
                <element name="actuator" type="node:typeActuator"
                                         maxOccurs="unbounded" minOccurs="0" />
                <element name="controller" type="node:typeController"
                                        maxOccurs="unbounded" minOccurs="0" />
                <element name="topic" type="node:typeTopic"
                                        maxOccurs="unbounded" minOccurs="0" />
        </sequence>

        <!-- Attributes -->
        <attribute name="id" use="required" type="node:nodeID" />
</complexType>
```

```
<!-- NODE ID -->
<simpleType name="nodeID">
	<restriction base="int">
		<minInclusive value="0"></minInclusive>
	</restriction>
</simpleType>
</schema>
```

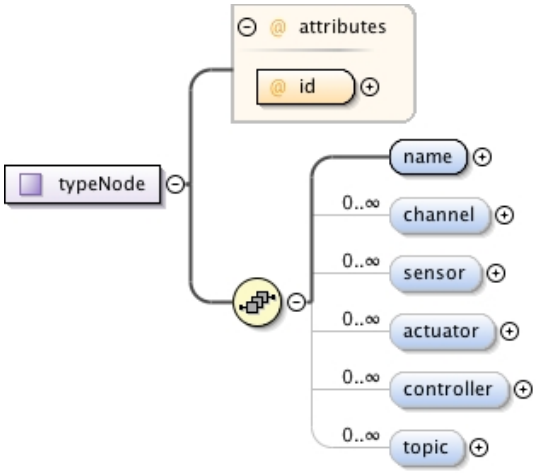

Figura C.2: Esquema XSD: typeNode

C.3. sensor.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/km"
        elementFormDefault="qualified" xmlns:sensor="http://www.example.org/km">
        <include schemaLocation="variable.xsd" />
        <complexType name="typeSensor">
                <sequence>
                        <element name="name" type="sensor:kmStringName" />
                        <element name="vlist">
                                <complexType>
                                        <sequence>
                                                <element name="variable"
                                                type="sensor:typeVariableSensor"
                                                        maxOccurs="unbounded"
                                                        minOccurs="0" />
                                        </sequence>
                                </complexType>
                        </element>
                        <element name="adequation" type="sensor:adequation" />
                        <element name="fusion" type="sensor:fusion" />
                </sequence>

                <!-- Attributes -->
                <attribute name="id" use="required" type="sensor:sensorID" />
        </complexType>

        <!-- SENSOR ID -->
        <simpleType name="sensorID">
                <restriction base="int">
                        <minInclusive value="0"></minInclusive>
                </restriction>
        </simpleType>
</schema>
```

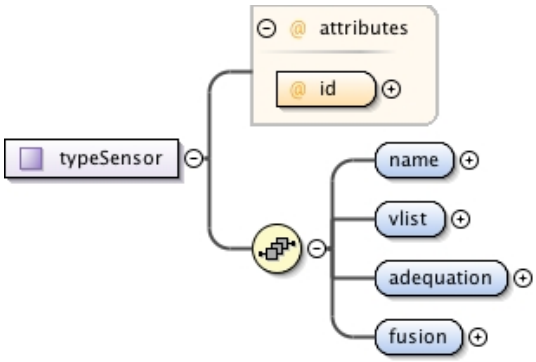

Figura C.3: Esquema XSD: typeSensor

C.4. actuator.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/km"
        elementFormDefault="qualified"
        xmlns:actuator="http://www.example.org/km">
        <include schemaLocation="variable.xsd" />
        <complexType name="typeActuator">
                <sequence>
                        <element name="name" type="actuator:kmStringName" />
                        <element name="vlist">
                                <complexType>
                                        <sequence>
                                                <element name="variable"
                                                type="actuator:typeVariableActuator"
                                                        maxOccurs="unbounded"
                                                        minOccurs="0" />
                                        </sequence>
                                </complexType>
                        </element>
                        <element name="adequation" type="actuator:adequation" />
                        <element name="fusion" type="actuator:fusion" />
```

```
            </sequence>

            <!-- Attributes -->
            <attribute name="id" use="required" type="actuator:actuatorID" />
        </complexType>

        <!-- ACTUATOR ID -->
        <simpleType name="actuatorID">
            <restriction base="int">
                <minInclusive value="0"></minInclusive>
            </restriction>
        </simpleType>
</schema>
```

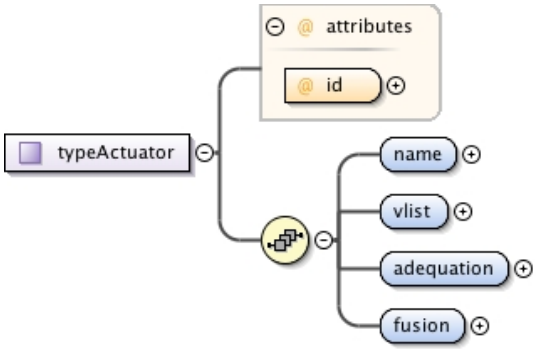

Figura C.4: Esquema XSD: typeActuator

C.5. controller.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/km"
        elementFormDefault="qualified"
        xmlns:controller="http://www.example.org/km">
        <include schemaLocation="variable.xsd" />
        <complexType name="typeController">
                <sequence>
                        <element name="name" type="controller:kmStringName" />
                        <element name="vlist">
                                <complexType>
                                        <sequence>
                                                <element name="variable"
                                        type="controller:typeVariableController"
                                                        maxOccurs="unbounded"
                                                        minOccurs="0" />
                                        </sequence>
                                </complexType>
                        </element>
                        <element name="fusion" type="controller:fusion" />
                </sequence>

                <!-- Attributes -->
                <attribute name="id" use="required" type="controller:controllerID"/>
        </complexType>

        <!-- CONTROLLER ID -->
        <simpleType name="controllerID">
                <restriction base="int">
                        <minInclusive value="0"></minInclusive>
                </restriction>
        </simpleType>
</schema>
```

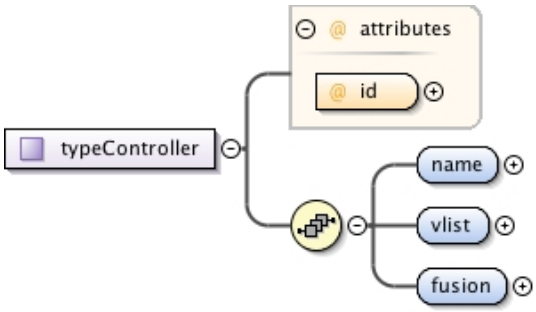

Figura C.5: Esquema XSD: typeController

C.6. topic.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.example.org/km"
        elementFormDefault="qualified"
        xmlns:topic="http://www.example.org/km">
        <include schemaLocation="coomon.xsd" />

        <complexType name="typeTopic">
                <sequence>
                <element name="deadline" type="common:typeDeadline"
                        minOccurs="1" maxOccurs="1"/>
                </sequence>

                <!-- Attributes -->
                <attribute name="id" type="topic:topicID" use="required" />
        </complexType>



        <!-- TOPIC ID -->
        <simpleType name="topicID">
                <restriction base="int">
```

```
                    <minInclusive value="0"></minInclusive>
                </restriction>
        </simpleType>

</schema>
```

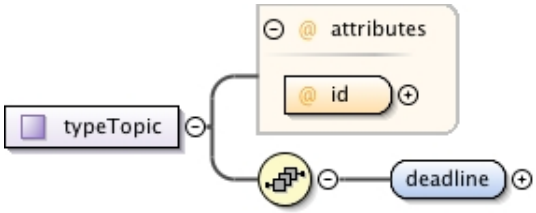

Figura C.6: Esquema XSD: typeTopic

C.7. variable.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
                targetNamespace="http://www.example.org/km"
                elementFormDefault="qualified"
                xmlns:var="http://www.example.org/km">

    <include schemaLocation="common.xsd" />
        <include schemaLocation="topic.xsd" />
        <include schemaLocation="channel.xsd" />
        <include schemaLocation="actuator.xsd" />
        <include schemaLocation="controller.xsd" />

                <complexType name="typeVariable">
                        <sequence>
                                <element name="configuration"
                                        type="var:modeVariable" />
                                <element name="connections"
                                        type="var:variableConnections" />
```

```
                </sequence>

                <!-- Attributes -->
                <attribute name="name" type="string" use="required" />
        </complexType>

        <!-- variableConnections -->
        <complexType name="variableConnections">
                <sequence>
                        <element name="idChannel"
                                type="var:typeConnectionChannel"
                                        minOccurs="0" maxOccurs="1"/>
                        <element name="idTopic"
                                type="var:typeConnectionTopic"
                                        minOccurs="0" maxOccurs="unbounded"/>
                </sequence>
        </complexType>

        <!-- typeConnection -->
        <complexType name="typeConnectionChannel">
                <!-- Attributes -->
                <attribute name="id" type="int" use="required" />
        </complexType>

        <!-- typeConnectionTopic -->
        <complexType name="typeConnectionTopic">
                <!-- Attributes -->
                <!-- internal | external -->
                <attribute name="id" type="int" use="required" />
                <attribute name="location" type="string" use="required" />
                <!-- read | write -->
                <attribute name="mode" type="string" use="required" />
        </complexType>
</schema>
```

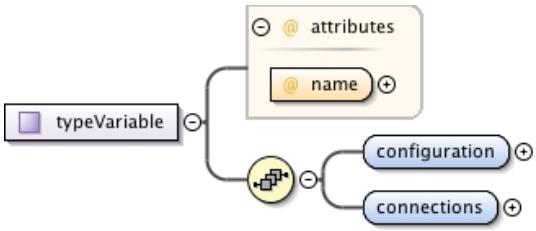

Figura C.7: Esquema XSD: typeVariable

C.8. channel.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.example.org/km"
elementFormDefault="qualified"
xmlns:channel="http://www.example.org/km">

        <include schemaLocation="common.xsd" />

        <complexType name="typeChannel">
                <sequence>
                        <element name="typeIO" type="channel:typeIO" />
                </sequence>

                <!-- Attributes -->
                <attribute name="id" use="required" type="channel:channelID" />
        </complexType>



        <!-- CHANNEL ID -->
        <simpleType name="channelID">
                <restriction base="int">
                        <minInclusive value="0"></minInclusive>
                </restriction>
```

```
        </simpleType>

        <!-- IOtype -->
        <complexType name="typeIO">
                <choice>
                        <element name="ND" type="channel:label_ND" />
                        <element name="IN" type="channel:typeIO_IN" />
                        <element name="OUT" type="channel:typeIO_OUT" />
                </choice>
        </complexType>

        <!-- typeIO_IN -->
        <simpleType name="typeIO_IN">
                <restriction base="string">
                </restriction>
        </simpleType>

        <!-- typeIO_OUT -->
        <simpleType name="typeIO_OUT">
                <restriction base="string">
                </restriction>
        </simpleType>

        <!-- label_ND -->
        <simpleType name="label_ND">
                <restriction base="string">
                </restriction>
        </simpleType>

</schema>
```

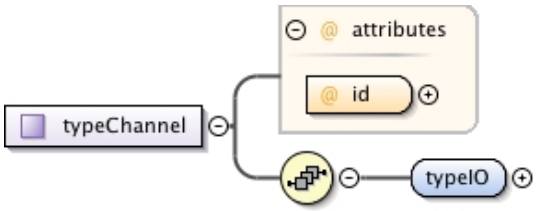

Figura C.8: Esquema XSD: typeChannel

APÉNDICE D

Estructura editor

A causa de la extensión de código fuente que compone la herramienta informática en esta sección se muestra una relación de la estructura utilizada.

Paquetes

- conf: Contiene las clases generadas automáticamente por JAXBBuilder a partir del modelo definido en XSD.
  - Adecuation.java
  - AdecuationRank.java
  - AdecuationSaturation.java
  - FusionLast.java
  - FusionMean.java
  - FusionUser.java
  - KmConfig.java
  - ObjectFactory.java
  - package.info.java
  - TypeChannel.java
  - TypeConnectionChannel.java
  - TypeConnectionSensor.java
  - TypeConnectionTopic.java
  - TypeIO.java

- TypeActuator.java
- TypeController.java
- TypeNode.java
- TypeSensor.java
- TypeTopic.java
- VariableConnections.java

- dialog: Contiene las clases relacionadas con las ventanas de configuración de los componentes.
  - DlgComponent.java
  - DlgNode.java
  - DlgRsyntaxTextArea.java
  - DlgTopic.java
- drw: Son las clases responsables de mostrar gráficamente cada uno de los componentes así como la relación existente entre ellos.
  - DrwActuator.java
  - DrwChannel.java
  - DrwComponent.java
  - DrwController.java
  - DrwNode.java
  - DrwSensor.java
  - DrwTopic.java
  - DrwVariable.java
  - Interaction.java
- km: Contiene las clases principales de la aplicación tales como el panel principal donde se hace el diseño así como la clase que contiene el menú *toolbar* y la consola de errores.
  - LoggerGUI.java
  - MainGUI.java

- MainPanel.java

- util: En este paquete se encuentran las clases encargadas de almacenar los colores utilizados en la interfaz así como los mensajes de aviso del usuario.
  - ColorProfile.java
  - Messages.java