Programa Oficial de Postgrado en Ciencias, Tecnología y Computación
Máster en Computación
Facultad de Ciencias - Universidad de Cantabria

.

Tecnología Ada-CCM para el desarrollo de aplicaciones basadas en componentes de tiempo real

Pablo Pacheco Munguía

pachecop@unican.es

Directores:
Patricia López Martínez y
José María Drake Moyano.
Grupo de Computadores y Tiempo Real
Departamento de Electrónica y computadores.

Santander, Julio, 2008

Este trabajo ha sido subvencionado en el marco de los siguientes proyectos:

- FRESCOR (Framework for Real-time Embedded Systems based on COntRacts) financiado por la EU Sixth Framework Programme (FP6/2005/IST/5-034026).
- ARTIST2 (Network of Excellence on Embedded Systems Design) financiado por la EU Sixth Framework Programme IST-004527.
- THREAD (Soporte integral para sistemas empotrados de tiempo real distribuidos y abiertos) financiado por Plan Nacional de I+D+I 2004-2007 (TIN 2005-08665-C03-02).

INDICE

INDICE DE FIGURAS

CAPÍTULO 1

ÁMBITO Y OBJETIVOS DEL TRABAJO

1.1 Tecnologías de componentes como base de las aplicaciones de Tiempo Real

El diseño del software de tiempo real para sistemas embebidos es una necesidad estratégica de la industria actual. En muchas áreas de aplicación como la robótica, el control industrial o la electrónica del automóvil, etc., los sistemas se construyen ensamblando subsistemas (controlador, sistema de visión, sistema de carburación, etc.) cada uno de ellos dotado de su propio procesador embebido que controla el hardware en el que se integra, y todos ellos intercomunicados por una red dedicada (Ethernet, bus Can, bus firewire, etc). Esta arquitectura proporciona una considerable modularidad, reconfigurabilidad y minimiza y estandariza el cableado.

La capacidad que actualmente tienen los procesadores y la gran cantidad de memoria con que están dotados, hace posible que el software de estos sistemas sea complejo, la plataforma sea distribuida y su funcionalidad exija requisitos de tiempo real.

La utilización en estos sistemas de una arquitectura basada en componentes para el diseño de su software, proporciona muchas ventajas:

- Proporcionan una arquitectura simple que se basa en interfaces y no en protocolos entre subsistemas.
- Hace posible escalar y reconfigurar los sistemas con sólo modificar el plan de despliegue y sin tener que modificar código de bajo nivel.
- Simplifica la evolución y el versionado de los equipos, al sólo requerir la sustitución y en su caso el desarrollo de componentes con funcionalidad y especificación bien definida.

Las tecnologías convencionales de componentes (EJB de Sun Microsystem, COM+ y .NET de Microsoft, o CORBA 3 de OMG COM+, .NET, EJB, etc.) son difícilmente aplicables a estos sistemas, ya que requieren sistemas operativos, sistemas de ficheros,

middleware y redes de comunicación que no son compatibles con la limitación de recursos que son habituales en estos sistemas.

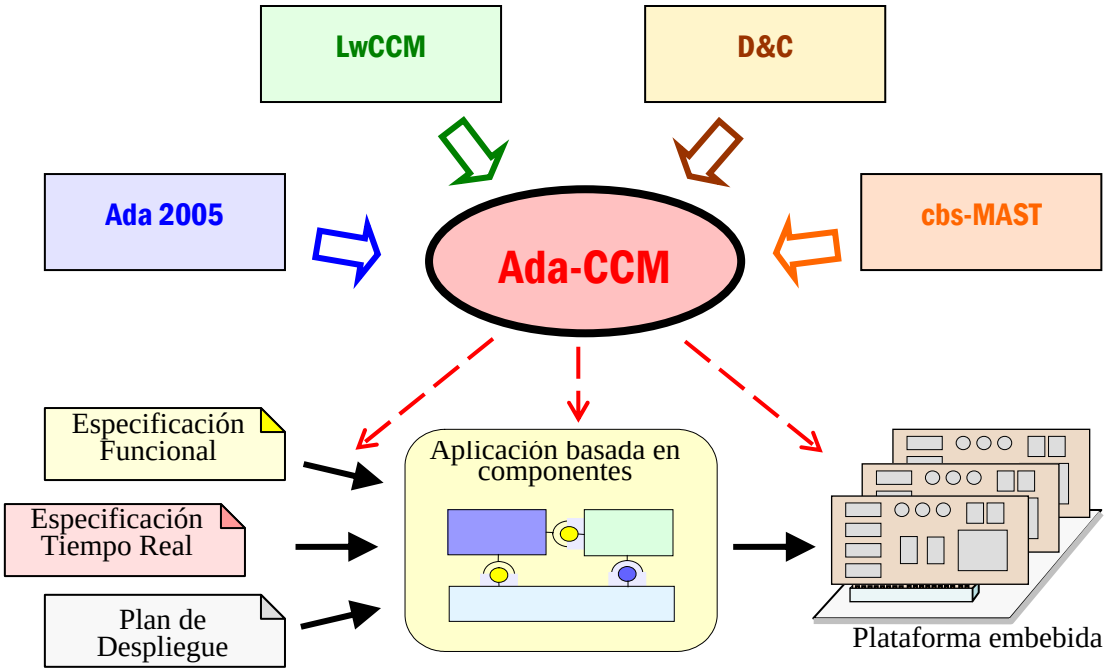

Figura 1.1: Bases y objetivos de la tecnología Ada-CCM.

En esta memoria se propone una tecnología de componentes que denominamos Ada-CCM que está específicamente concebida para sistemas embebidos mínimos, distribuidos y de tiempo real. La tecnología aprovecha las características del lenguaje de programación Ada 2005 para implementar aplicaciones con comportamiento temporal predecible, que pueden ser ejecutadas sobre máquina desnuda con sólo un ligero run-time, y que por tanto pueden ser embarcadas en plataformas embebidas basadas en procesadores con perfil mínimo. Hace uso de la especificación D&C de OMG para describir los componentes, las aplicaciones y las plataformas de ejecución. Se utiliza el modelo de referencia LwCCM de OMG como base de la arquitectura interna de los componentes y de las aplicaciones, pero en ella, se sustituye el middleware CORBA por componentes distribuidos que se denominan conectores, que encapsulan el proceso de invocación remota entre componentes y permiten reutilizar el código de negocio de los componentes en plataformas de ejecución basadas en diferentes middleware.

Para que las aplicaciones basadas en componentes soporten especificaciones de tiempo real, se necesitan extender las especificaciones CCM y D&C con nuevos elementos que describan el comportamiento temporal de los componentes, y nuevas herramientas que evalúen los recursos que se requieren para satisfacer los requisitos temporales. Aspectos concretos de las especificaciones que se han introducido son:

- Los componentes incluyen como parte de su descripción un modelo que permite predecir su comportamiento temporal.
- La plataforma debe ser de tiempo real, y disponer de un modelo que describa sus recursos y el comportamiento temporal de sus servicios.
- En el diseño de las aplicaciones se tienen que especificar los parámetros de concurrencia, de planificación, de sincronización, etc. (*Threading Model*). Los valores que resulten de estos parámetros, deben estar incorporados en el *Plan de Despliegue* de la aplicación.

La tecnología remarca e independiza las responsabilidades y el conocimiento que se requiere de los actores Specifier, Developer, Packager, Assembler, Planner y Executor y para cada uno de ellos, define los elementos de entrada y de salida sobre los que realiza su trabajo.

1.2 CCM (Container Component Model)

A finales de los 90, la OMG (Object Management Group) detectó limitaciones en CORBA en base a la aplicación de los nuevos paradigmas de programación basados en modelos MDA (Model Driven Architecture). Con este motivo se creó un grupo con el fin de evolucionar hacia un modelo basado en componentes, que finalmente se tradujo en el nacimiento de la especificación CORBA 3.0, en la cual se define una nueva tecnología de componentes denominada CCM (CORBA Component Model)[1], que da soporte al concepto de componente como una extensión del modelo de objeto, y establece un estándar que define la implementación, el empaquetado, el ensamblado y el despliegue de las instancias de los componentes en aplicaciones distribuidas.

En la actualidad, las aplicaciones evolucionan creciendo en su complejidad, requiriendo nuevos requisitos no funcionales y necesitando ser ejecutada sobre sistemas distribuidos complejos y sistemas empotrados con recursos limitados. En particular, las tecnologías basadas en componentes que actualmente se encuentran en el mercado (EJB de Sun MicroSystems, COM+ y .NET de Microsoft o CORBA 3 de OMG) no son tecnologías apropiadas para soportar requisitos de tiempo real, y mucho menos, si las plataformas sobre las que se ejecutan disponen de recursos limitados como en el caso de las plataformas embebidas.

En el modelo de referencia CCM, se especifica a CORBA como el middleware de comunicaciones que debe utilizarse. En los últimos años se han propuesto en varios proyectos europeos MERCED[2], COMPARE[3] y FRESCOR[4], buscar alternativas a CORBA menos pesadas, y que sean compatibles con sistemas embarcados y de perfil mínimo. En ellos se ha sustituido CORBA por otros middlewares de comunicaciones menos pesado compatible con requisitos de tiempo real, pasando a formar una nueva extensión de CCM denominada Container Component Model. Una de estas alternativas es LwCCM[5] contiene una versión ligera de CORBA, lo que le permite ejecutarse en plataformas embebidas si bien no es compatible con los requisitos de tiempo real.

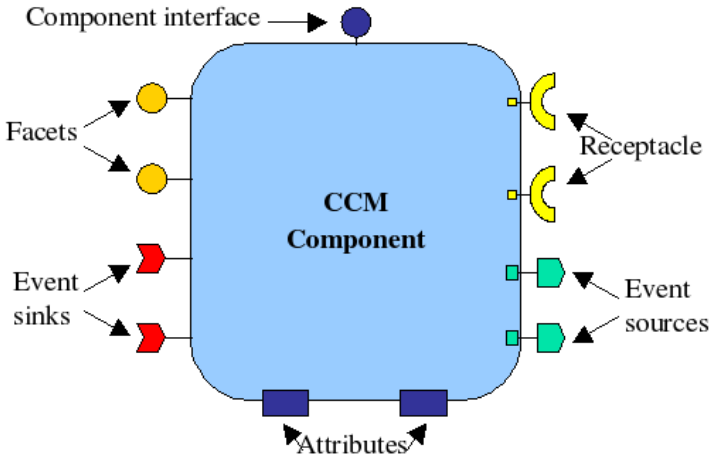

Figura 1.2: Esquema de los puertos de un componente CCM.

1.3 Implementación de CCM sobre Ada 2005

La reciente modificación de la especificación del lenguaje Ada, Ada 2005, y el desarrollo de plataformas que lo soportan, hace posible implementar la tecnología de componentes CCM tomándolo como base. Esta es la opción que se desarrolla en este trabajo.

El uso de Ada 2005 para implementar la tecnología de CCM, proporciona las siguientes ventajas:

- El lenguaje Ada ha sido concebido para el desarrollo de aplicaciones de tiempo real, y desde él, se puede formular el modelo de concurrencia, las políticas de planificación y los mecanismos de sincronización.
- El lenguaje Ada ofrece un modelo de distribución, y desde él, se pueden formular los mecanismos de interacción entre componentes desplegados en diferentes procesadores.
- En la versión Ada 2005 se introduce el concepto de interfaz, y a través de él, se pueden implementar la encapsulación de los servicios que ofrecen los componentes (Facetas en CCM) o las referencias de los servicios que se requieren (Receptáculos en CCM).

El lenguaje Ada'95 estaba soportado por CORBA, sin embargo, la nueva especificación de Ada 2005 no ha sido aún soportada. Por ello, en este trabajo, se ha tenido que modificar y extender el mapeo existente de IDL hacia Ada[6], con el fin de incorporar todas las ventajas de Ada 2005 en esta tecnología.

1.4 Especificación de D&C de OMG Y Extensión RT-D&C

Con objeto de desarrollar la especificación de despliegue y configuración[7] de las aplicaciones basadas en componentes, la OMG mantiene un grupo de trabajo que se encarga de estandarizar la descripción y modelado de los componentes, de las plataformas de ejecución y de las aplicaciones basadas en componentes. Con ellos se busca hacer compatibles los componentes desarrollados por diferentes empresas, hacer interoperables los entornos y las herramientas que soportan su ciclo de desarrollo.

Desde el punto de vista de la distribución comercial, un componente es un paquete que lleva asociado un modelo, esto es la información necesaria sobre el componente para que las herramientas automáticas puedan instanciar el componente en la plataforma, así como componer éste con otros componentes.

En el despliegue, cada instancia debe ser instalada en el procesador que va a ejecutar su código, configurada, enlazada con otros componentes locales y remotos, y finalmente, lanzada su ejecución. Para realizar esta tarea, son necesarias herramientas que en base al modelo asociado a los módulos de código, gestionen de forma adecuada esta información para lograr el despliegue.

El proceso de despliegue de una aplicación basada en componentes comienza después de que el software haya sido desarrollado, empaquetado y publicado por el proveedor de software y adquirido por el propietario del software, que lo desplegará.

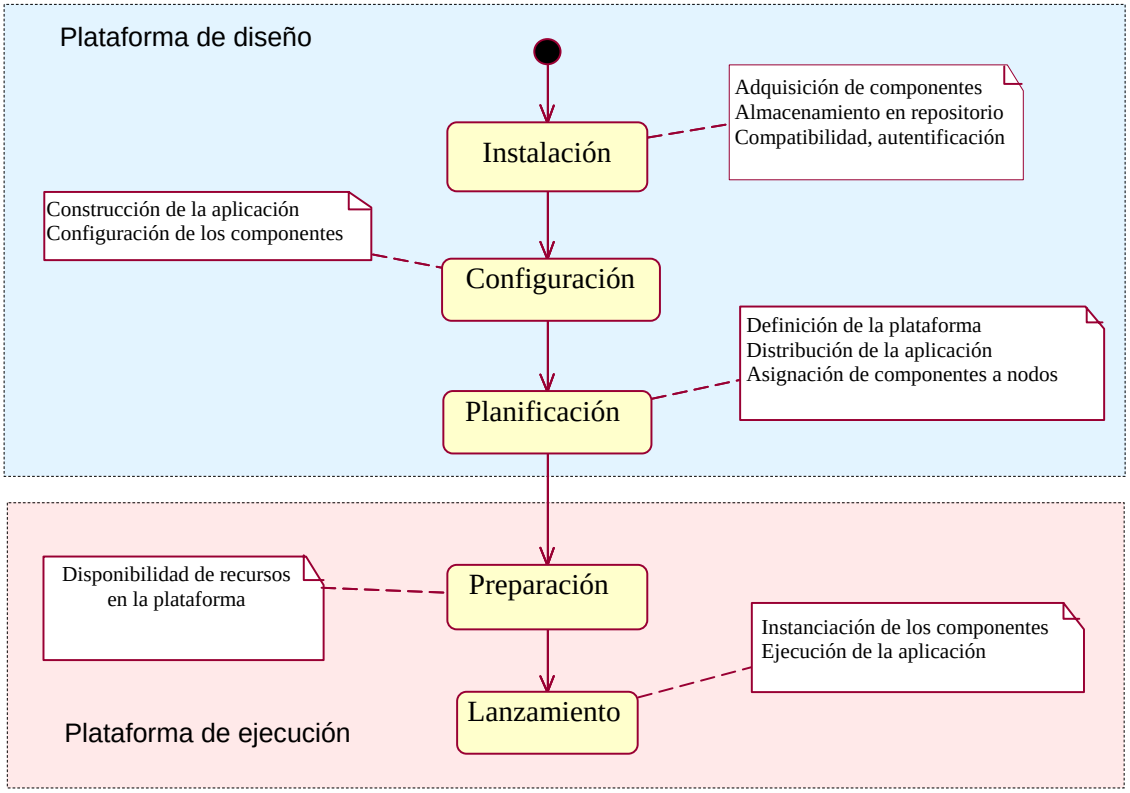

Figura 1.1: Fases del proceso de despliegue según D&C.

Para aplicar este proceso al desarrollo de aplicaciones de tiempo real, se necesita extender las especificaciones CCM y D&C con nuevos elementos que describan el comportamiento temporal de los componentes, y nuevas herramientas que evalúen los recursos que se requieren para satisfacer los requisitos temporales. Aspectos concretos de las especificaciones que se han introducido son:

- Los componentes incluyen como parte de su descripción un modelo que permite predecir su comportamiento temporal [8].
- La plataforma debe ser de tiempo real, y disponer de un modelo que describa sus recursos y el comportamiento temporal de sus servicios.
- En el diseño de las aplicaciones se tienen que especificar los parámetros de concurrencia, de planificación, de sincronización, etc. Los valores que resulten de estos parámetros, deben estar incorporados en el Plan de Despliegue de la aplicación.

1.5 Objetivos de la Tesis

El objetivo de este trabajo es explorar la potencialidad del lenguaje Ada 2005 para realizar una implementación de la tecnología de componentes CCM. Al basarse en un lenguaje de programación moderno que cubre todos los aspectos de la tecnología, da lugar a implementaciones más sencillas y eficientes que otras basadas en múltiples recursos (lenguaje de programación, sistema operativo y middleware de comunicaciones).

Entre los objetivos concretos que se han llevado a cabo en este proyecto, se pueden destacar las siguientes:

1. Se ha desarrollado una arquitectura de clases y objetos Ada que implementan el modelo de referencia (framework) de la tecnología de componentes RT-CCM. La arquitectura especifica los elementos que constituye un componente y para cada uno de ellos, se describe la funcionalidad que debe implementar, y los patrones de interacción que requieren. El criterio principal que ha guiado la especificación de la

arquitectura ha sido facilitar la generación automática del código. La mayoría de las clases están relacionadas con los aspectos definidos en la tecnología y son comunes a todos los componentes. De estas clases se han definido un código parametrizado que sirve de base para la generación automática del código final de los componentes.

2. Se ha propuesto una extensión para Ada 2005, de las interfaces, componentes y tipos de datos que se formulan en el lenguaje IDL e IDL3. OMG tenía definida la correspondencia entre IDL y el lenguaje Ada'95. La introducción de la nueva versión de Ada ha requerido revisar esta correspondencia. Dado que Ada 2005 es un lenguaje en la que se han incorporado muchos de los conceptos modernos de la ingeniería software, la correspondencia que se ha introducido es más sencilla, eficiente y directa que la que hasta ahora estaba definida.
3. Se ha desarrollado la especificación para la creación futura de una herramienta para la generación automática del código Ada 2005 de los componentes que se implementan en la tecnología Ada-CCM. La herramienta procesa como entrada las especificaciones D&C de los componentes y genera el código Ada de los elementos que constituye el componente. La herramienta se basa en disponer de un conjunto de plantillas parametrizadas de código, y en introducir elementos de las mismas en el código final del componente, de acuerdo con la información encontrada en los ficheros IDL de especificación.
4. Se ha propuesto un modelo de comunicación entre componentes distribuidos entre diferentes nudos que se basa en el protocolo de comunicaciones RTEP que soporta tiempo real. Así mismo, se ha generado una herramienta que tomando como entrada el plan de despliegue de la aplicación genera el código de los conectores entre componentes que se especifican en él.
5. Se ha utilizado como demostrador de la tecnología un ejemplo denominado JetFollower. El ejemplo es completo y permite describir muchas de las características que se implementan en la tecnología.

CAPÍTULO 2

ESPECIFICACIÓN, DISEÑO Y EMPAQUETADO DE COMPONENTES Ada-CCM

2.1 Especificación de un componente Ada-CCM.

El agente encargado de realizar la especificación de un componente será un experto en el dominio de su aplicación, y será el encargado de crear la especificación del nuevo componente.

La especificación de los componentes en esta tecnología está realizada de acuerdo con la especificación D&C, mediante su descripción en un fichero .ccd (Component Interface Description). En ella se definen los puertos a través de los cuales se ofrecen servicios a los clientes y obtienen servicios de los servidores para proporcionar funcionalidad. Los tipos de puertos que se pueden especificar son:

- Facetas (facets): Son puntos de conexión a través de los cuales se ofrece a los clientes los servicios implementados. Por cada faceta tendrá asociada una interfaz que describirá su funcionalidad.
- Receptáculos (receptacles): Son los puntos de conexión a través de los cuales el componente obtiene los servicios necesarios para conseguir su funcionalidad. Como en el caso de las facetas, cada receptáculo tiene asociado la interfaz que describe la funcionalidad que en este caso se requiere.
- Propiedades de configuración: valores o características que forman parte del estado del componente, y que normalmente son utilizados para realizar su configuración.

Como se ha indicado en la sección 1.4 la especificación D&C ha sido extendida [8] para incorporar requerimientos de tiempo real e información introspectiva relacionada con el comportamiento temporal del componente. Además, con el propósito de gestionar los parámetros de planificabilidad de los threads que utiliza el componente, se ha incluido en la extensión de tiempo real un mecanismo para que los componentes hagan explícitos los threads que utilizan. Cuando la implementación del negocio del componente necesita threads

para implementar su funcionalidad, declaran un tipo específico de puerto que hemos denominado puertos de activación.

A fin de hacer más amigable en la memoria, la explicación de los recursos y formatos que se utilizan para describir un componente, se va utilizar el componente ServosController que se ha desarrollado para el demostrador que se describe en el capítulo 6.

Como se muestra en la figura 2.1, este componente ofrece la faceta *controllerPort*, que implementa la interfaz *I_Control*. Por otra parte tiene varios receptáculos: Requiere a través del receptáculo *logger* los servicios de un componente que implemente la interfaz *I_Logger*; a través del puerto *ioCard* de otro componente que ofrezca la interfaz *I_AnalogIO* y por último, a través del puerto *speaker* de un componente que implemente la interfaz *I_Player*. En la especificación también viene definida una propiedad de configuración *numServos* que en este caso le dará al código de negocio la información del número de servos que tiene que controlar este componente.

Por otra parte, debido a la extensión de tiempo real, en la especificación se definen 2 puertos de activación periódicos (*controlThread* y *pollingThread*), así como el periodo y su prioridad como propiedades de configuración del mismo.

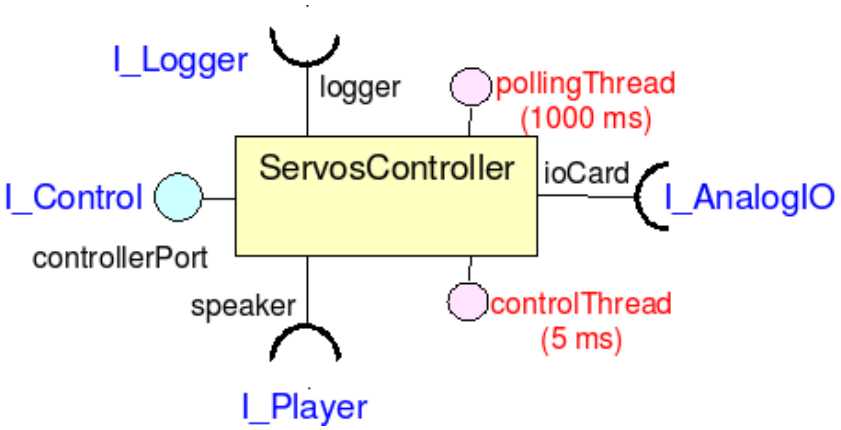

Figura 2.1: Especificación de los puertos del componente ServosController.

La descripción formal de la vista externa o interfaz del componente, se formula a través del fichero XML denominado *ServosController.ccd.xml*. Este documento es el que es procesado e interpretado por las herramientas que necesitan gestionar el componente.

2.2 Descripción de la implementación de un componente Ada-CCM.

En la tecnología Ada-CCM, el desarrollador implementa el código de negocio del componente como un conjunto de paquetes .ads y .adb con el código fuente Ada. El desarrollador del componente, diseña código que implementa la funcionalidad de negocio de un componente, como si programara en un entorno monoprocesador. En la tecnología Ada-CCM, el código e negocio de un componente no tiene dependencias de la tecnología, por lo que se libera al desarrollador de tener que conocer detalles de la tecnología a la hora de desarrollar el código de negocio. Él concibe, diseña e implementa el código como si fuese a ejecutarse en un entorno monoprocesador sencillo.

El único requisito que tiene que satisfacer el código de negocio de un componente, para poder integrarse como implementación de un componente, es que implemente la interfaz de gestión. Esta interfaz será generada a partir del fichero de especificación del componente, usando la herramienta de generación de código

AdaCCM_BusinessInterfaceGenerator. Esta interfaz define los métodos que se deben implementar de cara a que el contenedor pueda gestionar la implementación:

- Dispone de funciones de acceso a las referencias de las implementaciones de las facetas: (get_[NombreFaceta]).
- Procedimientos a través de los cuales se establecen las referencias a los puertos con los que se deben enlazar sus receptáculos ([NombreReceptáculo]_Connect).
- Métodos de acceso a las implementaciones de los puertos de activación (que contienen el método *update()* en el caso de activaciones periódicas y el método *run()* en el caso de activaciones OneShot): (get_[NombreActivacion]).
- Operaciones de lectura y escritura de las propiedades de configuración definidas en la especificación: (get_[NombrePropiedad] y set_[NombrePropiedad].
- Ofrece un conjunto de métodos que permiten gestionar el ciclo de vida de los componentes: El método *complete()* notifica que el componente está configurado y enlazado, y en consecuencia, dispuesto para operar; el método *activate()* inicia la operación del componente; *passivate()* suspende la ejecución del componente y *remove()* elimina el componente.

Además de implementar esta interfaz una correcta implementación de un componente debería incluir:

- Por cada faceta ofrecida por el componente, se debería agregar un objeto que la implemente. En el caso de componentes sencillos, esa misma clase puede implementar las interfaces soportadas por las facetas.
- Por cada activación, se debería agregar otra instancia que la implemente.
- Todos los elementos de la implementación (Implementación de las facetas, de los puertos de activación, etc.), deberán operar de acuerdo al estado del componente, que es único por cada instancia del mismo. Basado en esto, el estado puede ser implementado como una clase agregada que puede ser accedida por el resto de los elementos, eliminado de esta forma las dependencias cíclicas.

En el caso del componente ServosController este sería el diagrama UML que representa su interfaz de gestión (ServosController_Mngr.ads):

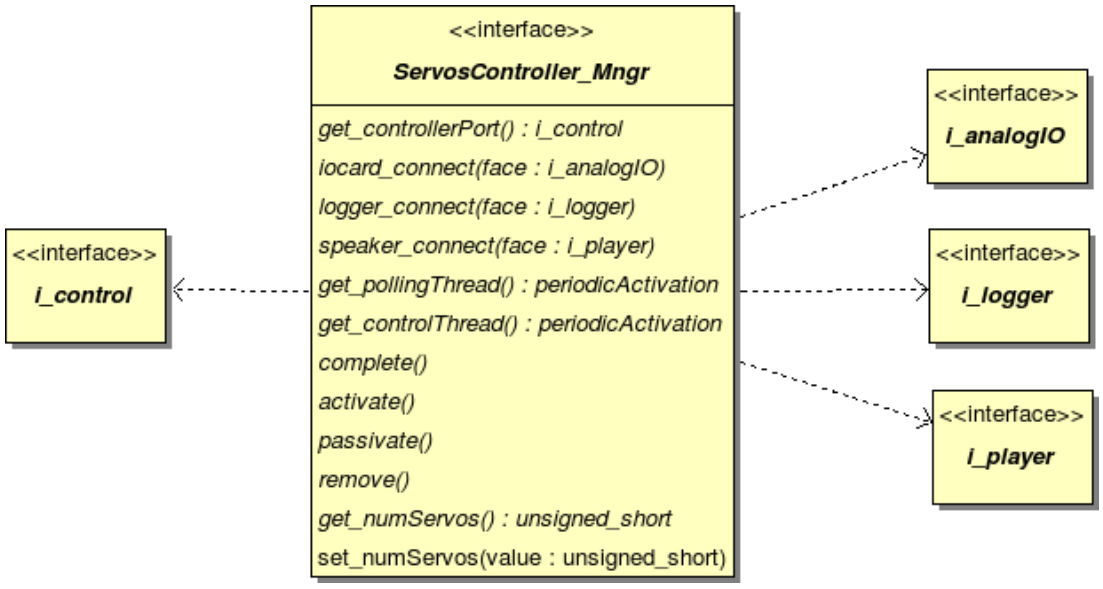

Figura 2.2: Interfaz de gestión del componente ServosController.

El desarrollador tiene un completo conocimiento de los detalles de la implementación, por lo que es el único que puede definir, junto con el código, el modelo de comportamiento

de tiempo real del componente. Además, el desarrollador deberá especificar los requerimientos que impone a la plataforma en la que se ejecuta. Toda esta información estará contenida en el fichero *AdaServosController.cid.xml* cuyo formato y contenido se corresponde con el elemento "ComponentIimplementation Description" definido por D&C, y formalizado en la plantilla W3C-Schema *DnC_CCM_ComponentDataModel.xsd*. Todos estos ficheros, correspondientes a las plantillas de los ficheros XML utilizados, pueden encontrarse en el anexo de esta memoria.

2.3 Empaquetamiento de un componente Ada-CCM

El empaquetador, es el responsable de empaquetar el componente a fin de su distribución comercial, y de agrupar toda la información que se necesita para reutilizar el componente en una aplicación.

Los ficheros correspondientes al contenedor del componente (.ads y .adb) serán generados mediante la herramienta *AdaCCM_ContainerGenerator*, tomando para ello la información contenida en la especificación del componente (fichero .ccd). Estos ficheros serán compilados junto con los correspondientes a la implementación del componente, y se creará una librería (fichero .a) que contendrá los ficheros objeto del componente precompilados. Esta librería es lo único que se necesita para poder ejecutar el componente en la plataforma.

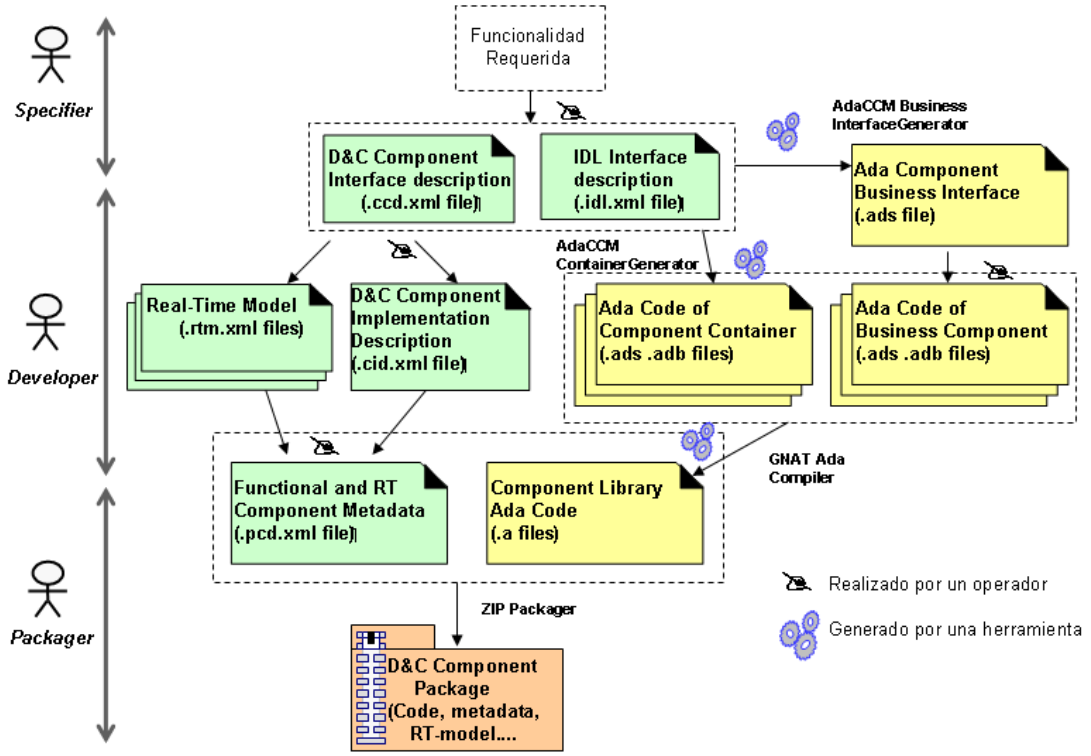

Figura 2.3: Actores y elementos que intervienen en el proceso de desarrollo de los componentes

Finalmente, el empaquetador recopilará toda la información disponible sobre el componente que podrá ser incluido en un catálogo para ser utilizado en el desarrollo de futuras aplicaciones. Este paquete incluirá además del código precompilado (fichero .a) del componente, la información del modelo de comportamiento de tiempo real, la descripción de la implementación y la información para la instanciación y ejecución del componente. Esta información estará contenida de acuerdo al formato del fichero .pcd (*Package Configuration Description)* definido en el D&C.

Para realizar el empaquetamiento del componente que estamos poniendo como ejemplo (ServosController), se ha creado un paquete ZIP que contiene el fichero *ServosController.a* correspondiente al código binario y el fichero *ServosController.pcd.xml*, el cual contendrá la información relativa al comportamiento de tiempo real (*ServosController.rtm.xml*), la descripción de la implementación (*ServosController.cid.xml*) y la información para la instanciación, configuración enlazado y ejecución.

Cuando un componente va a utilizarse, el paquete del componente debe ser instalado en el repositorio del entorno de desarrollo. En él se desempaqueta la información de una forma organizada, de forma que las herramientas de diseño de las aplicaciones encuentren, y ensamblen el componente en ellas.

CAPÍTULO 3

DISEÑO DE UN CONTENEDOR DE UN COMPONENTE Ada-CCM

3.1 Modelo de referencia CCM. Elementos y funcionalidad

La arquitectura de un componente propuesta ha sido generada de acuerdo al modelo de referencia de la tecnología. Ésta se basa en el Container Component Framework propuesto en LwCCM, pero ha sido extendido con algunas nuevas características requeridas para hacer que el comportamiento de las aplicaciones sea predecible:

- Con el fin de hacer que los threads y los parámetros de planificación de una aplicación sean configurables, y con ello controlar la planificabilidad, el código de negocio de los componentes no posee threads internos. La actividad interna de un componente es definida en base a un conjunto de puertos declarados en su especificación. Estos puertos son reconocidos por el contenedor, el cual crea y activa los correspondientes threads que ejecutan la actividad del componente una vez éste haya sido instanciado, conectado y configurado. Estos puertos de activación pueden implementar una de las interfaces predefinidas: PeriodicActivation o OneShotActivation. La interfaz OneShotActivation ofrece el procedimiento run(), el cual será ejecutado una vez que el thread haya sido creado, mientras la interfaz PeriodicActivation declara un procedimiento update(), que será invocado periódicamente. Un componente puede declarar varios puertos de activación, cada uno de ellos representará un hilo de control.
- Interceptores: Es un mecanismo que proporciona la capacidad de incorporar los servicios del entorno en el proceso de invocación de una operación, esto se traduce en la ejecución de determinadas acciones antes y después de la invocación de dicha operación. La introducción de este elemento es opcional y depende del plan de despliegue que instancia los componentes de la aplicación. Las actividades realizadas en el interceptor dependen de la tecnología de componentes utilizada, no obstante no deberán modificar la lógica de negocio, pudiendo por ejemplo realizar

modificaciones de los recursos que el entorno proporcione para la ejecución de dicha operación. En el caso de la tecnología Ada-CCM son usados para controlar los parámetros de planificación. Nuestra tecnología permite aplicar el modelo transaccional de la aplicación [9], de manera que un servicio puede ser ejecutado con diferentes parámetros de planificación en función del punto, dentro de una misma transacción, desde la que es invocada.

La idea básica del modelo de referencia de la tecnología es sustituir la comunicación basada en el middleware CORBA que propone OMG, por mecanismos de comunicación entre componentes que son apropiados a cada plataforma de ejecución que se utilice y que se definen como nuevos componentes que se denominan conectores.

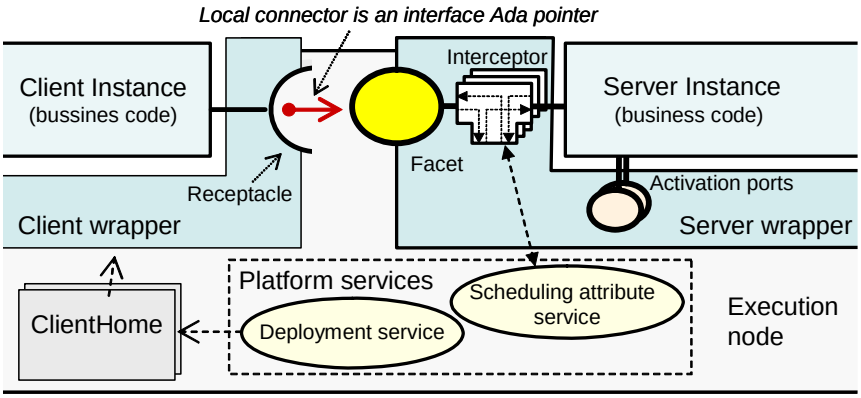

Figura 3.1: Esquema de los elementos básicos del modelo de referencia.

A Continuación se detalla la funcionalidad de los elementos más destacables:

- Contenedor (wrapper): Representa la parte del código de un componente que proporciona el soporte de la plataforma de ejecución al código de negocio. Constituye la forma a través de la que se libera al diseñador del componente, de conocer los detalles de la tecnología y de los recursos de la plataforma de ejecución. Todos aquellos aspectos derivados de la tecnología o de la plataforma se incluyen en el adaptador y su código se genera mediante herramientas automáticas, permitiendo que el diseñador se dedique únicamente al desarrollo del código de negocio.
- Conector (connector): La función de este elemento es realizar la comunicación entre los puertos de los componentes. A través de él, en una conexión distribuida, un componente que se encuentra en un procesador, puede realizar las invocaciones requeridas a otro componente situado en otro procesador. Si ambos componentes se encontraran en el mismo procesador la conexión es local, y el conector se reduce a facilitar la referencia del servidor al cliente. Un conector es un tipo de componente mas, constituido por dos elementos: el proxy (que se instancia en el procesador del cliente) y el servant (que se instancia en el procesador del servidor). El código de los conectores es generado completamente de forma automática por herramientas. En esta tecnología hemos desarrollado los conectores utilizando directamente el protocolo RTEP[10], que es un protocolo de tiempo real sobre red ethernet. También han sido realizados conectores sin características de tiempo real, usando GLADE[11], del anexo de sistemas distribuidos de ada (DSA).
- Servicios de hilos de ejecución (thread service): Tiene como función crear y gestionar todos los hilos que ejecutan actividades en el componente. Algunos de estos son aplicados por el entorno para implementar respuestas asíncronas a la invocación de una determinada operación. Este servicio puede realizar la creación de los hilos bajo demanda, esto es se crearán según vayan siendo necesitados, o puede disponer de un grupo configurable de hilos que aplica a una u otra instancia de un componente en función de la aplicación.

3.2 Implementación Ada del contenedor

Se encuentra implementado en el paquete Ada de nombre *[NombreComponente]_Wrapper*. Está constituido por un conjunto de recursos que constituyen el soporte del componente desde el entorno y la conexión con otros componentes de la aplicación. Representa el adaptador del componente al entorno de ejecución en que se instancia. Su código será generado de forma completa por las herramientas de generación del código.

Se basa en la clase que hemos denominado *Wrapper*, la cual implementa la interfaz equivalente del componente, encargada de adaptar el componente a la plataforma en que se ejecuta, y como establece LwCCM siendo la única interfaz desde la cual los clientes o la herramienta de despliegue pueden acceder a sus servicios. Con éste propósito, la clase wrapper implementa la interfaz CCMObject, la cual ofrece las siguientes operaciones:

- Configuration_complete: será invocada por el entorno para verificar la correcta configuración del componente durante la fase de instanciación de los componentes.
- Provide_facet: Es una operación ofrecida al exterior del componente, mediante la cual un cliente puede obtener una referencia a la implementación de la faceta del componente cuyo nombre se pasa como parámetro.
- Connect: Realiza la conexión receptáculo-faceta, pasándole la referencia a una faceta, obtenida por ejemplo a través de *provide_facet*, y el nombre del receptáculo del componente al que se va a conectar. La referencia a la faceta se almacena en el contexto, para que posteriormente el componente pueda acceder a ella.
- Disconnect: pone a null las referencias almacenadas en el contexto correspondientes al receptáculo cuyo nombre se pasa como parámetro.

Además, la clase *wrapper* define la capacidad de incorporar interceptores mediante la implementación de las interfaces *Client/ServerContainerInterceptorRegistration*, que son una versión modificada de las interfaces definidas con el mismo nombre en la QoSCCM[12]

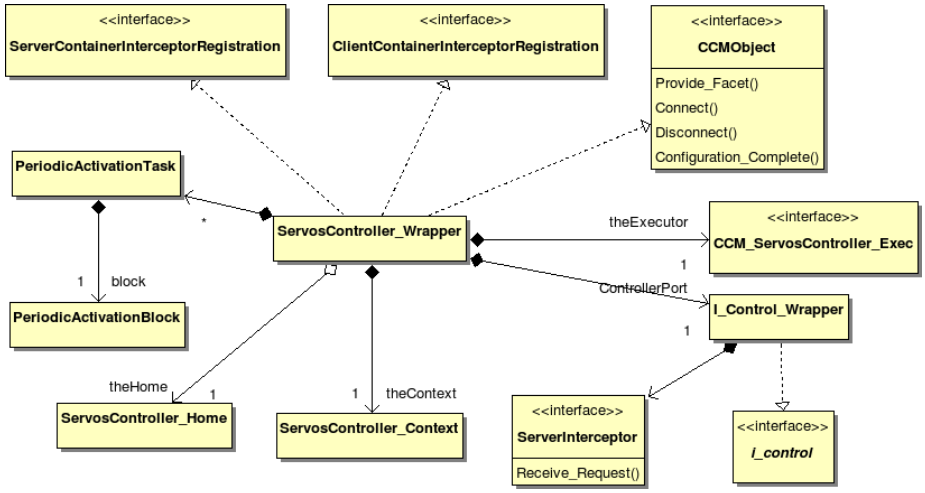

Figura 3.2: Estructura de la clase wrapper

Como se ve en la figura, esta clase es un contenedor que almacenará como elementos agregados o referencias todos los elementos que forman un componente:

- Se almacenarán referencias a los envoltorios de las facetas (denominados *[NombreInterfaz]_Wrapper*. Las clases correspondientes a estos envoltorios o *wrapper* de las facetas estarán definidos en la parte privada del paquete del *wrapper*. Estas clases serán las que capturen las llamadas que se realicen sobre las facetas del componente, y las trasladen a las correspondientes implementaciones. En ellas se incluirán los interceptores que se añadan a las operaciones de un componente con el

objetivo de gestionar propiedades no funcionales de la aplicación. En el ejemplo, correspondiente al componente ServosController, que se muestra en la figura, vemos la referencia *controllerPort* del envoltorio *I_Control_Wrapper*.

- Se almacenará una referencia al ejecutor del componente que será instanciado en el *wrapper* y a través del cual, el *wrapper* tiene acceso a la implementación del componente.
- Existirá una referencia al contexto, necesaria para almacenar la información acerca de todas las conexiones del componente (realizadas a través de las operaciones *connect* y *disconnect*).
- Por cada puerto de activación, habrá una referencia a la tarea Ada que representa el thread que implementa la funcionalidad. Para implementar estos threads se han definidos dos diferentes tipos tarea de Ada. El tipo *OneShotActivationTask* correspondiente a las activaciones no periódicas y el tipo *PeriodicActivationTask* para las periódicas. Ambos tipos reciben en su instanciación una referencia a una estructura de datos (*OneShotActivationBlock* o *PeriodicActivationBlock*) que caracteriza su ejecución, como los parámetros de planificación, el periodo y el procedimiento que ejecutará.
- Por último existirá una referencia a la interfaz equivalente del *home* del componente. A través de la cual se crearán las instancias del componente.

3.3 Implementación Ada del Ejecutor

El ejecutor del componente, posee visibilidad sobre el contexto, y representa la implementación real del componente, tal como se indica en la especificación CCM. Una operación recibida a través de la interfaz equivalente del componente, será tomada por el ejecutor cuya implementación podrá utilizar los receptáculos requeridos debido a la visibilidad que posee sobre el contexto.

En el ejecutor del componente se definen los *skeletons* (clases raíces abstractas) y las interfaces que, siguiendo es estándar CCM, sirven de base para desarrollar las implementaciones de los componentes y de los *home* del componente. Todo este conjunto de clases e interfaces se definen en el paquete [*NombreComponente]_Exec*. Por otra parte, en el paquete *[NombreComponente]_Exec_Impl* contiene la implementación de las clases que heredan de las interfaces definidas en el paquete anterior, y por tanto es dependiente de la tecnología.

3.3.1 El paquete [Nombre_Componente]_Exec

Haciendo uso del concepto de interfaz que proporciona la nueva especificación Ada 2005, en este paquete definiremos en primer lugar, una serie de interfaces locales, denominadas facet executor interfaces que tendrán como nombre *CCM_[NombreInterfaz]* y heredarán de las correspondientes interfaces de las facetas.

En este paquete también se definirá, siguiendo la normativa CCM, una interfaz local denominada *monolithic executor interface*. El nombre que tendrá esta interfaz será *CCM_[NombreComponente]* y heredará de la interfaz *SessionComponent*, definida en el estándar CCM.

Como consecuencia de la herencia de *SessionComponent* esta interfaz implementa la operación *set_session_context*, a través de la cual el contenedor le entregará al ejecutor del componente una referencia al contexto, mediante el que el componente puede tener acceso a

la información acerca de las conexiones que se han realizado en el componente (esta clase se explica con mas detalle a continuación).

Además de las operaciones heredadas, ésta interfaz deberá incluir:

- Operaciones de lectura (accesor) y escritura (mutator) de cada uno de los atributos definidos en el componente: *get_[NombreAtributo]* y *set_[NombreAtributo]* respectivamente.
- Una operación adicional por cada faceta, denominada *get_[NombreFaceta]*, que posee el componente para poder acceder a ellas y que devuelven una referencia a la facet ejecutor interface correspondiente.
- Una operación *get_[NombrePuertoActivacion]* por cada puerto de activación definido en el componente. A través de esta operación, el contenedor del componente podrá tener visión sobre la implementación de la interfaz de activación que corresponda, y de esta manera poder configurar las tareas definidas en el contenedor para que ejecuten dichas implementaciones.
- Un método *configuration_complete*, que será invocado por el contenedor para finalizar la configuración del componente. Esta operación no ha sido impuesta por CCM, se trata de una estrategia de implementación.

A continuación se muestra el diagrama UML de la interfaz *monolithic executor interface* así como de las *facet executor interfaces* para el caso del componente ServosController, que en su especificación ofrece una faceta (*ControllerPort*), tiene 2 puertos de activación (*PollingThread* y *ControlThread*) y posee un atributo (*ServosNum*). En la interfaz *CCM_ServosController* se observan las operaciones que se añaden como consecuencia de ello.

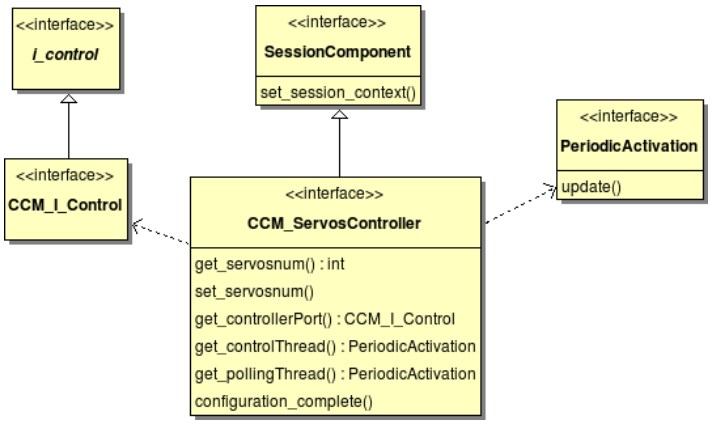

Figura 3.3 Diagrama UML de la Monolithic executor interface.

Dentro del paquete del ejecutor se definirá también la interfaz de contexto del componente. Tendrá por nombre *CCM_[NombreComponente]_Context* y heredará de la interfaz *SessionContext*. Esta interfaz ofrecerá operaciones a través de las cuales un componente podrá obtener las referencias a los componentes que se encuentren conectados a sus receptáculos. Las operaciones serán del tipo *get_connection_[NombreReceptáculo]*, y devolverán la referencia a un objeto del tipo de la interfaz esperada en el receptáculo si hay algún objeto conectado a él.

A continuación se muestra el diagrama UML de la interfaz de contexto del mismo componente ServosController que hemos tomado como ejemplo. Cabe destacar, que en la especificación del mismo se indica que posee tres receptáculos, correspondientes a referencias a implementaciones de las interfaces *I_Logger*, *I_Player* e *I_AnalogIO*.

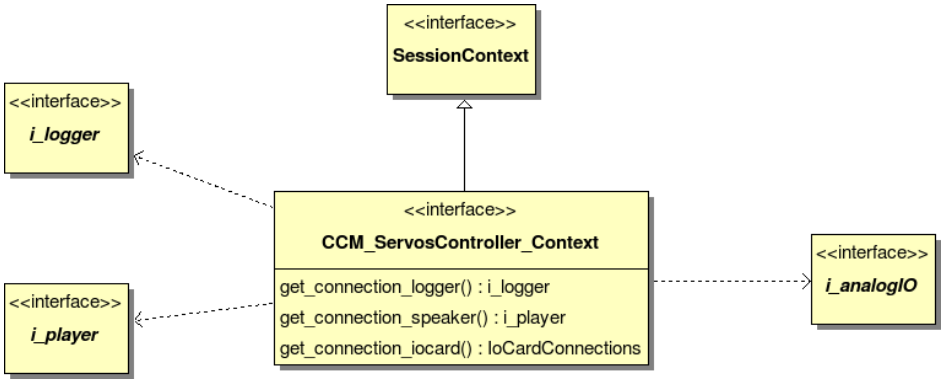

Figura 3.4: Diagrama UML de la interfaz de contexto del paquete del ejecutor.

Finalmente, se definirán tres interfaces correspondientes al gestor, o home, del componente.

- La primera de las interfaces se denomina Home Explicit Executor Interface, y en nuestra tecnología definiremos una interfaz Ada 2005 de nombre CCM_[NombreHome]Explicit, que contendrá todas las operaciones definidas explícitamente para el home.
- La segunda de las interfaces del home será la Home Implicit Executor Interface, que denominaremos CCM_[NombreHome]Implicit y que ofrecerá la operación create(), que será invocada por el contenedor para generar una nueva instancia del componente.
- Por último estará la Home Main Executor Interface, a la que hemos llamado CCM_[NombreHome] que implementará las dos anteriores.

A continuación se muestra el diagrama UML de las clases correspondientes al gestor del componente GralComponent, definido en el ejecutor del mismo:

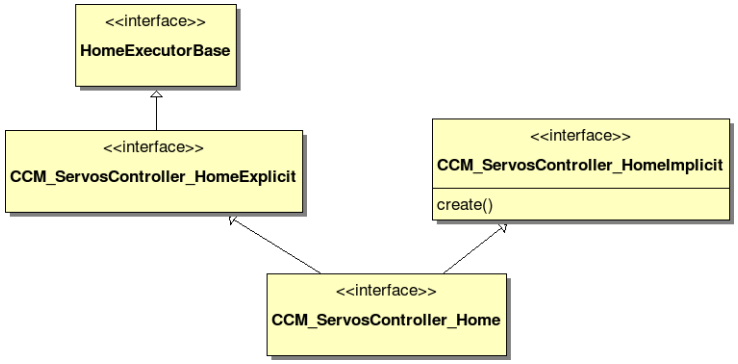

Figura 3.5: Diagrama UML de las clases correspondientes al Home del ejecutor.

3.3.2 El paquete [NombreComponente]_Exec_Impl

En este fichero se declara la clase que representa la implementación del componente. En realidad se trata de una clase intermedia, que contiene como agregada la clase que realmente incorpora el código que implementa la funcionalidad del componente (patrón tie). Esta clase es necesaria para implementar el componente siguiendo la normativa impuesta por CCM, según la cual todo componente debe heredar de los correspondientes ejecutores definidos en el fichero anterior. Todo su código es generado automáticamente. En este paquete se declaran dos clases: la clase *Instance* y la clase *Home*.

La clase *Instance* implementa la interfaz *Object* y la interfaz *CCM_[NombreComponente]* correspondiente a la interfaz monolítica definida en el paquete *[NombreComponente]_Exec*. Esta clase incorpora como atributos:

- Una referencia al contexto, que le pasará el contenedor cuando la instancia sea creada a través del método *set_session_context* que se implementa como consecuencia de la herencia de la interfaz monolítica.
- La referencia a la implementación del código de negocio a través de la interfaz de gestión *[NombreComponente]_Mngr*.
- Las referencias a las implementaciones de cada una de las facetas que posea.

El código de casi todas las operaciones es una reinvocación del mismo método sobre la referencia a la clase agregada, excepto las operaciones *set_session_context* y *configuration_complete*, cuyo código es generado de forma automática directamente en esta clase.

En la figura que se muestra a continuación, se muestra el diagrama UML de esta clase para el componente ServosController que estamos tomando como ejemplo durante este capítulo:

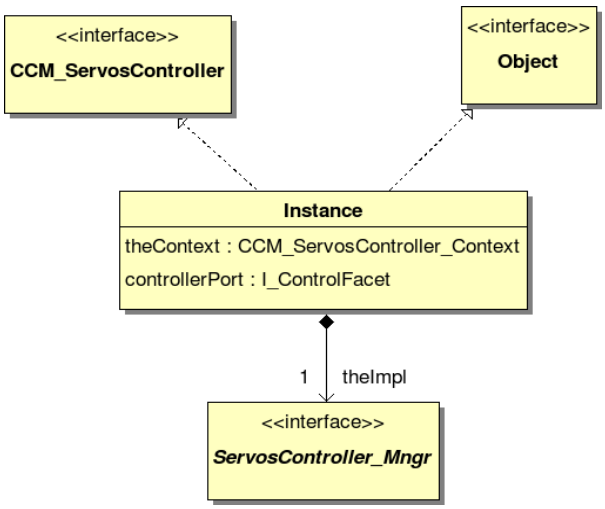

Figura 3.6: Diagrama UML de la clase Instante del paquete Exec_Imp.

La otra clase definida en este paquete es la clase *Home* que implementa la interfaz *CCM_[NombreHome]* definida en el paquete del ejecutor. No posee atributos. La operación *create* que se ha de implementar en esta clase (heredada de *CCM_[NombreHome]*) genera una instancia de la clase *Instance*, que a su vez instancia un objeto de la clase *[NombreComponente]_Impl*.

Se añade en el fichero la operación *create_home* para la creación externa de objetos de la clase *Home*, necesaria durante la fase de instanciación de los componentes.

3.4 Implementación Ada del Contexto

El contexto del componente, es el lugar donde se almacenan las referencias a las implementaciones de las interfaces requeridas por los receptáculos, las cuales son establecidas a través de la conexión en la fase de despliegue de los mismos a otro componente por medio de los conectores.

La clase que representa la implementación concreta del contexto, de nombre *[NombreComponente]_Context*, está definida dentro del paquete *[NombreComponente]_Wrapper*, y realiza la implementación de la interfaz *CCM_[NombreComponente]_Context*, definida en el paquete *[NombreComponente]_Exec* del ejecutor comentado en el apartado anterior.

Debido a que realiza la implementación de la interfaz del contexto definida en el ejecutor, implementará las operaciones *get_connection_[NombreReceptaculo]* que proporcionan el mecanismo de acceso a las referencias de los receptáculos a través de los envoltorios de los receptáculos (*stubs*). Estas referencias se establecieron a través de las operaciones *connect* de la *clase [NombreComponente]_Wrapper* que hemos comentado en el capítulo 3.1. Los interceptores del lado del cliente (*ClientInterceptor*) se encontrarán contenidos en dichos envoltorios de los receptáculos. De esta manera, las invocaciones sobre las operaciones ofrecidas por los receptáculos podrán ser interceptadas.

A continuación se muestra el diagrama UML de esta clase para el caso del componente ServosController:

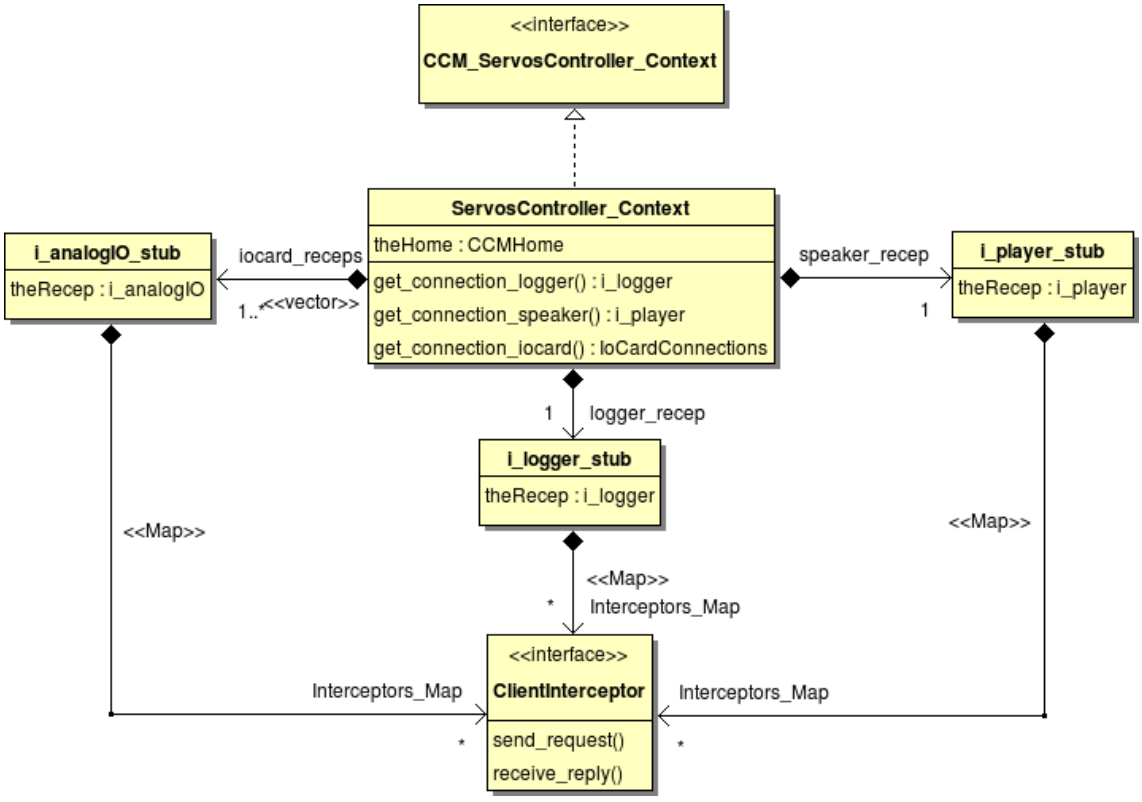

Figura 3.7 Diagrama UML de las clases del contexto del wrapper.

3.5 Implementación Ada del Home

El *Home* representa una clase que podrá ser accedida por las herramientas de despliegue y por otros componentes. Cada *Home* dispone de operaciones para acceder a las instancias del componente que han sido creadas por él. Estas operaciones retornan referencias a la interfaz del componente (*equivalent interface*), mediante la cual se puede obtener la referencia de cualquier otra interfaz del componente.

La clase correspondiente al *Home* está definida en el paquete *[NombreComponente]_Wrapper* y se llama *[NombreComponente]_Home_Wrapper*. Esta clase hereda de *HomeConfiguration* (para poder realizar configuración de los componentes), y de las interfaces *KeylessCCMHome* (para poder crear la instancia del componente) e *I_Set_Home_Executor*. Estas tres clases/interfaces, de las que hereda/implementa, no dependen de los componentes de la aplicación, sino de la propia tecnología, no es código generado sino reutilizado.

Se muestra a continuación el diagrama UML de la clase *ServosController_Home_Wrapper* para el caso del ejemplo:

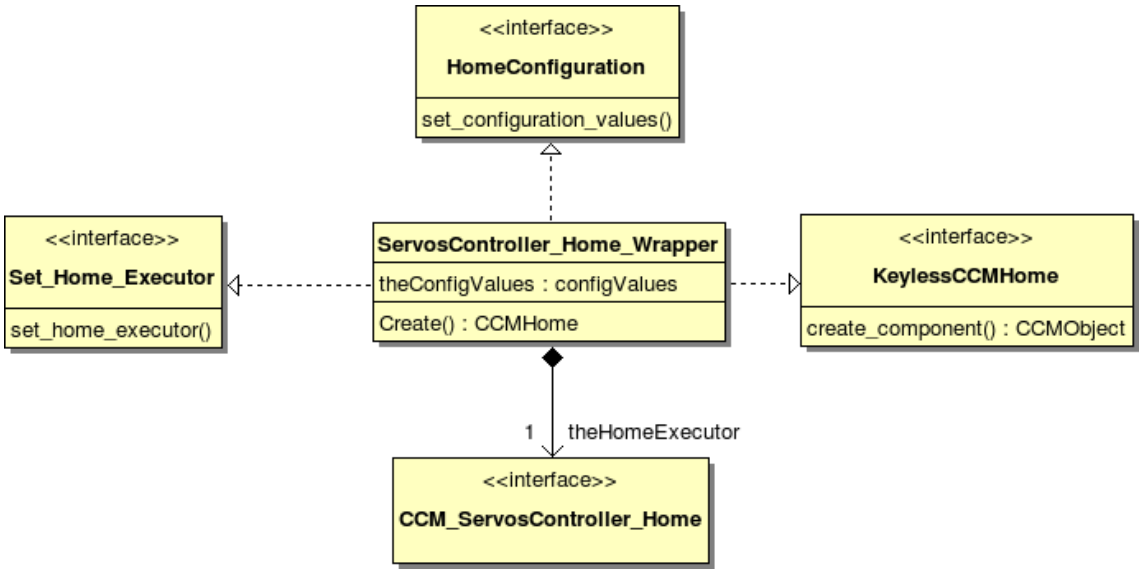

Figura 3.8: Diagrama UML de las clases correspondientes al Home del wrapper.

Hay que incluir en esta clase un procedimiento de creación para las *home*, ya que se utilizará como punto de entrada (entry Point), para poder desde la herramienta de despliegue crear la instancia del *home* y poder comenzar la creación de componentes.

Esta clase contiene dos elementos, una referencia al *home* del ejecutor y una lista con los valores de configuración (*config values*) del componente. La herramienta de despliegue se encargará de fijar estos valores de configuración a través de la operación *set_configuration_values*, definida en esta clase como herencia de *HomeConfiguration*. Posteriormente, estos valores serán empleados para configurar el componente, a través de los procedimientos *set_[attributeName]*, definidos para cada uno de los atributos del componente. Será también tarea de la herramienta de despliegue, almacenar la referencia al *home* del ejecutor una vez que lo cree, mediante la operación *set_home_executor* implementada en esta clase como consecuencia de la implementación de la interfaz *I_Set_Home_Executor*.

Como herencia de *KeyLessCCMHome* deberá implementar la operación *create_component*, que será ejecutada también por parte de la herramienta de despliegue y que realizará la instanciación del componente, siguiendo los siguientes pasos:

- Crea el *wrapper*.

- Crea el ejecutor (a través de la referencia *theHomeExecutor*) y almacena una referencia al mismo en el *wrapper*.
- Crea el contexto, pasándole su referencia al ejecutor (a través de la operación *set_session_context*, que dicho ejecutor implementa).
- Crea las instancias correspondientes a cada una de las facetas (de las correspondientes clases *wrapper_[TipoInterfazFaceta])*.
- Configura el componente con los valores de configuración previamente introducidos por la herramienta de despliegue.

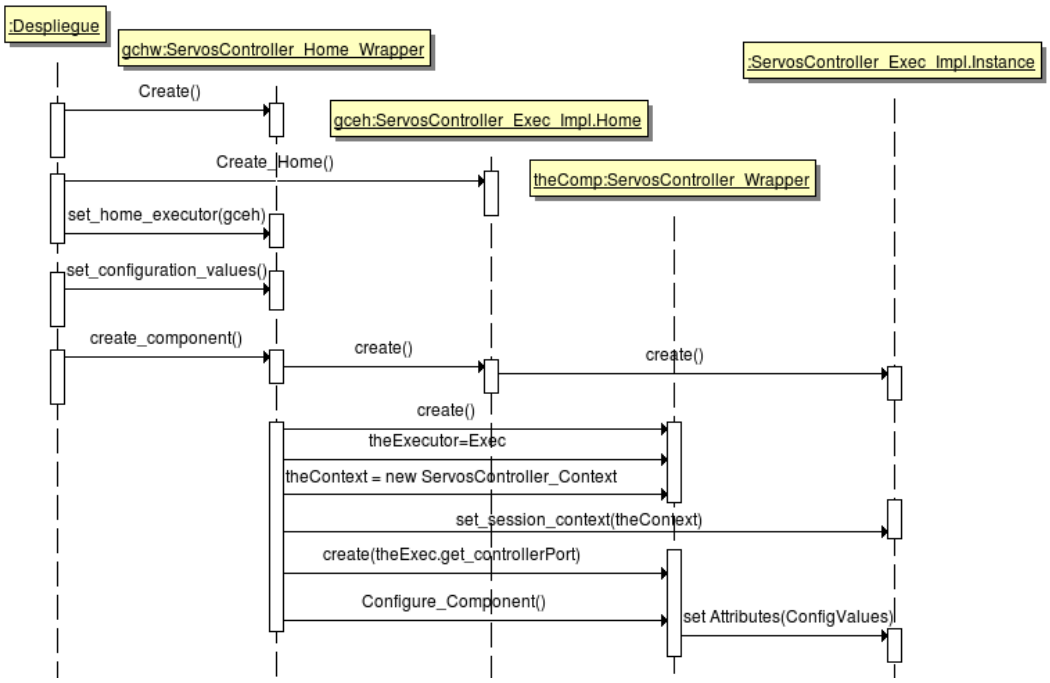

Figura 3.9: Diagrama de secuencias de la instanciación de un componente.

CAPÍTULO 4

CONECTORES BASADOS EN RTEP

4.1 Concepto y funcionalidad de los conectores

En la tecnología Ada-CCM, los conectores constituyen el mecanismo a través del que un componente cliente invoca un servicio de un componente servidor que se ejecutan en otro nudo de la plataforma. Se construyen como dos componentes que se comunican entre sí a través de un mecanismo de comunicación propio.

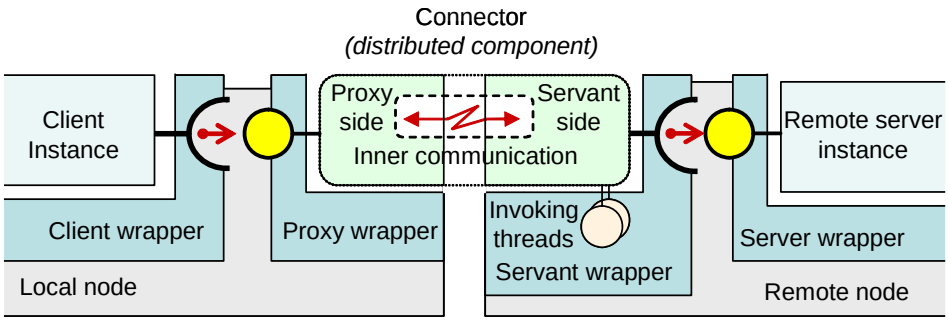

Figura 4.1: Estructura del modelo de referencia con los conectores.

Al componente del conector que se ejecuta en el nudo del cliente lo denominamos *proxy*, y la faceta que ofrece se enlaza al receptáculo del cliente. El componente que se ejecuta en el nudo del servidor se denomina *servant*, y el receptáculo que ofrece se enlaza a la faceta del componente servidor. Su funcionalidad es muy simple, cuando el cliente invoca un método del *proxy*, el thread del cliente se suspende en él, y el *proxy* elabora un mensaje que codifica el método invocado y los valores de los parámetros de entrada pasados en la invocación, y lo envía al *servant*. En respuesta a su recepción el *servant* dedica uno de los threads de que dispone para invocar sobre el componente servidor una invocación idéntica a la que ejecutó el cliente. Finalizada la ejecución del servicio en el servidor, el *servant* elabora un nuevo mensaje destinado al *proxy* con el que le comunica que el servicio ha sido realizado y le transmite los valores de los parámetros de salida retornados. Tras su recepción el *proxy*

concluye la invocación que había realizado el cliente, y le retorna los valores de los parámetros de salida que fueron generados por el servidor.

Los códigos de *proxy* y del *servant* se generan automáticamente por la herramienta de despliegue, en función de la descripción de la interfaz de los puertos que conecta (faceta y receptáculo) y del mecanismo de comunicación que usan.

En función del tipo de conexión que se implementa, el conector debe resolver los siguientes aspectos:

1. Si el conector debe establecer la comunicación entre componentes desarrollados en diferentes lenguajes de programación, el conector implementa los mecanismos de serialización y deserialización (marshalling y unmarshalling) necesarios para que los argumentos de invocación y retorno puedan ser procesados por el correspondiente lenguaje. Esto nunca es necesario en la tecnología Ada-CCM, ya que todos los componentes están realizados en Ada.
2. El Proxy debe disponer de los mecanismos de sincronización necesarios para que el cliente quede suspendido mientras que se realiza la invocación remota del servicio (invocación sincronía), o proporcionar el medio para que el cliente continúe su ejecución, y pueda recibir cuando concluya, los valores retornados por el servidor (invocación asíncrona).
3. Debe ofrecer los threads para la ejecución remota de las invocaciones, esto es, disponer de threads que ejecuten en el nudo del servidor los servicios que se han invocado sobre él.
4. Debe implementar el mecanismo a través del que las invocaciones son traducidas a mensajes que se envían a través de la red.

En la figura 4.2, se muestran los diferentes tipos de conectores que resultan en función de la localización relativa de los componentes conectados, y el tipo de invocación de las operaciones (síncrona o asíncrona) que se realiza:

- En el caso de que la invocación sea local y síncrona, el conector, o no existe, o tiene una funcionalidad mínima. No son necesarios los mecanismos de sincronización, pues es el thread del cliente el que ejecuta la invocación, ni son necesarios los mecanismos de comunicación. Por el contrario, si los componentes están desarrollados en diferentes lenguajes de programación, el conector tiene que implementar los de serialización y deserialización de parámetros en la invocación y en el retorno.

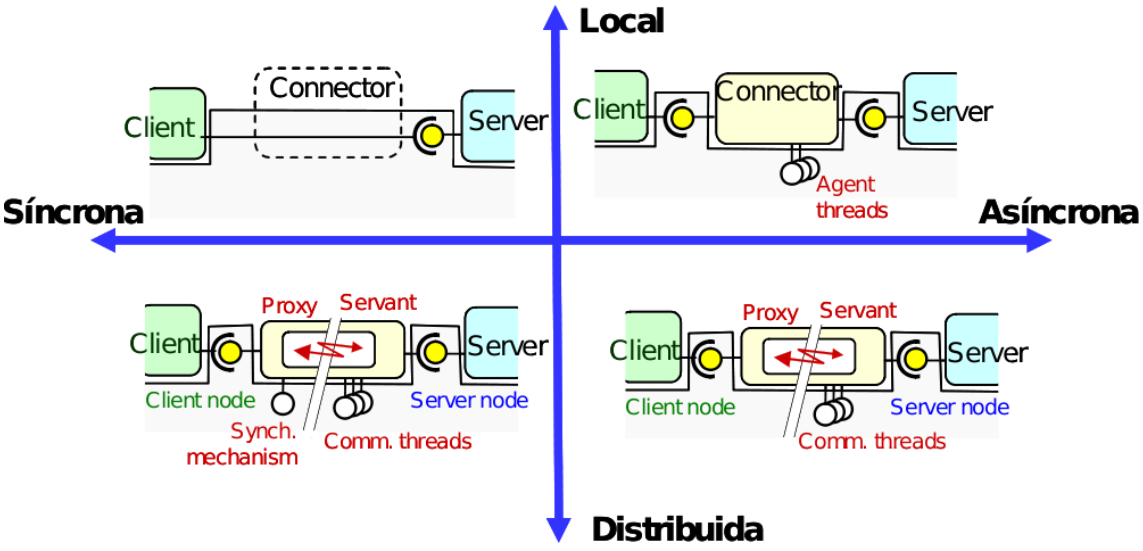

Figura 4.2: Diferentes tipos de conectores.

- En el caso de que la invocación sea local pero asíncrona, el conector no requerirá mecanismos de comunicación, pero sí requerirá por cada invocación un thread del entorno (a través de un puerto de activación) con el que llevar a cabo la operación invocada, y así, retornar de inmediato al cliente el control de flujo.
- Si el cliente y el servidor están instanciados en procesadores diferentes, como se muestra en la figura 4.2, el conector se compone de dos componentes: el componente *proxy*, que se instancia en el procesador del cliente, y el componente *servant*, que se instancia en el procesador del servidor.

4.2 El Protocolo RT-EP

El protocolo RT-EP (*Real Time Ethernet Protocol*)[13], se introduce con el fin de que los servicios de comunicación a través de *Ethernet* tengan un tiempo de latencia acotada y predecible, y evitar la aleatoriedad que introduce el mecanismo de arbitraje no determinista de las redes *Ethernet*. RT-EP implementa un protocolo de paso de testigo sobre *Ethernet* basado en prioridades fijas, con lo que conseguimos predicibilidad completa sobre la comunicación a través de *Ethernet*, utilizando hardware estándar y sin que se necesite introducir cambios en él.

RT-EP ha sido diseñado para evitar las colisiones en *Ethernet* por medio del uso de un testigo que arbitra el acceso al medio compartido. Implementa una capa de adaptación por encima de la capa de acceso al medio en *Ethernet* que comunica directamente con la aplicación. Cada estación (ya sea un nodo o CPU), posee una cola de transmisión y varias de recepción. Todas ellas son colas de prioridad donde se almacenan, en orden descendente de prioridad, todos los mensajes que se van a transmitir y los que se reciben. En el caso de tener mensajes de la misma prioridad se almacenan con un orden FIFO (*First in First Out*). El número de colas de recepción puede ser configurado a priori dependiendo del número de threads (o tareas) en el sistema que necesiten recibir mensajes de la red. Cada thread debe tener su propia cola de recepción. La aplicación debe asignar un canal, denominado *channel_id*, a cada thread que requiera comunicación a través de la red. Este *channel_id* se utiliza para identificar los extremos de la comunicación (las tareas o threads), es decir, proporciona el punto de acceso a la aplicación.

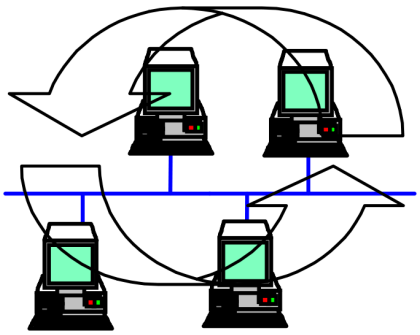

Figura 4.3: Anillo Lógico.

La red está organizada en un anillo lógico sobre un bus como se muestra en la figura anterior. Existe un número fijo de estaciones que serán las que formen la red durante la toda la vida del sistema. La configuración de las estaciones se hace sobre su dirección MAC de 48 bits. Cada estación conoce quién es su sucesora, información suficiente para construir el anillo. Además todas las estaciones tienen acceso a la información de configuración del anillo.

El protocolo funciona circulando un testigo a través del anillo. Este testigo contiene la dirección de la estación que tiene el paquete de mayor prioridad en la red en ese momento. También almacena la prioridad de ese paquete. Existe una estación, llamada token_master, que crea y controla la circulación del testigo. La asignación de esta estación es dinámica. Otra característica importante del protocolo es que, por sencillez, no admite fragmentación de mensajes, por lo tanto, es la aplicación la responsable de fragmentar los posibles mensajes de tamaño superior al máximo permitido, que son 1492 bytes.

El protocolo ofrece 256 prioridades. La prioridad 0, la más baja, se reserva para uso interno del protocolo y por lo tanto proporciona 255 prioridades para los mensajes de la aplicación.

Las operaciones para el envío y recepción (*Send_Info* y *Recv_Info*) utilizan *Streams* para contener el mensaje que se transmite. Se define también el canal (*channel_ID*) utilizado para enviar/recibir y la prioridad con la que se realizará el envío en la red. La operación *Receive*, será una operación bloqueante, de ahí que se necesite un thread por cada canal para poder atender los mensajes distintos. El formato de estas operaciones se muestra a continuación:

```
procedure Send_Info
   (Destination_Station_ID : in Station_ID;
    Channel_ID : in Channel;
    Data : in Stream_Element_Array;
    Data_Priority : in Priority);
```

```
procedure Recv_Info
   (Source_Station_ID : out Station_ID;
    Channel_ID : in Channel;
    Data : out Stream_Element_Array;
    Last : out Stream_Element_Offset;
    Data_Priority : out Priority);
```

4.3 Estructura e implementación de los conectores Ada-RTEP

Tal y como se describió en la sección 4.1, un conector consta de dos componentes: *proxy* y *servant*, cuya estructura e implementación se desarrolla en este capítulo. Ambos componentes al utilizar RT-EP necesitan configurar los canales de envío y recepción, que se realizará por medio de propiedades de configuración de los componentes. Es necesario de igual forma especificar la estación en la que se instancian para identificarse en el anillo. En cuanto a la estrategia de instanciación de estos componentes, necesitaremos una instancia por cada puerto (Una instancia *proxy* por cada faceta y una instancia *servant* por cada receptáculo).

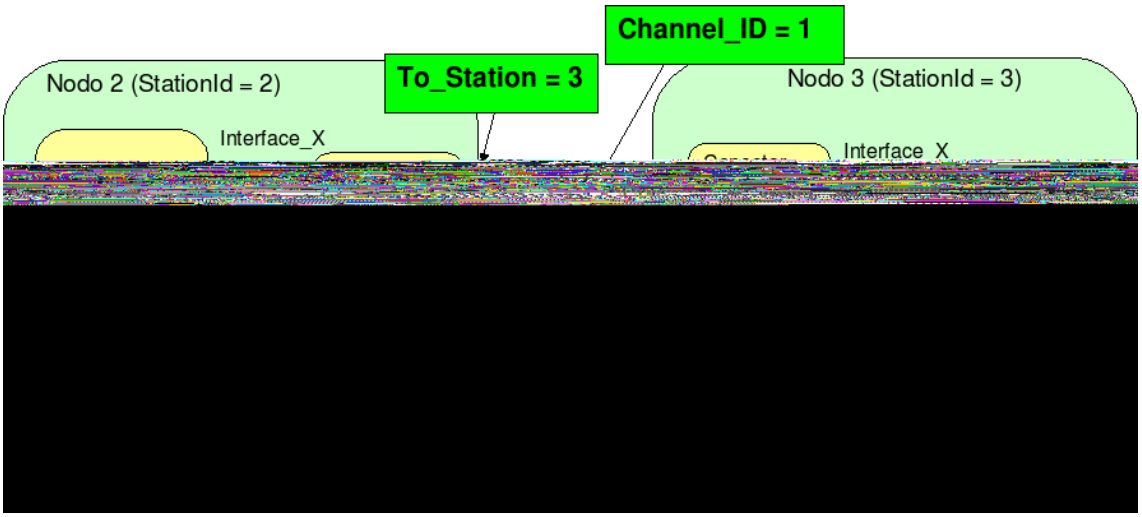

Figura 4.4: Estrategia de conexión de los conectores

4.3.1 El fragmento Proxy del conector RT-EP

Se trata de un componente con la estructura propia de cualquier componente en esta tecnología. La función del código de negocio de los conectores es implementar el mecanismo de comunicación a través del protocolo RT-EP. Este componente poseerá dos propiedades que habrá que configurar en cada una de las instancias: *Channel_ID* y *Station_ID*.

Como estrategia de implementación el código de negocio de cada operación sigue el diagrama de actividad que se muestra en la figura 4.5.

Cada invocación de una operación del cliente se ejecutará sobre la instancia del componente *proxy* de forma local. El proceso de reinvocación de esta petición se detalla a continuación:

1. Leemos el identificador de la transacción (*SchedID*) que ha sido configurado.
2. A través de un servicio de comunicación que hemos implementado obtenemos el canal (*Channel_ID*) en el que recibiremos la respuesta.
3. Creamos el *Stream* que contendrá la información a enviar: Identificador de operación (*OperationID*); Identificador de transacción (*SchedID*); Parámetros de la operación si existen; Canal de respuesta (*Channel_ID*); Identificador del nodo (*Station_ID*).

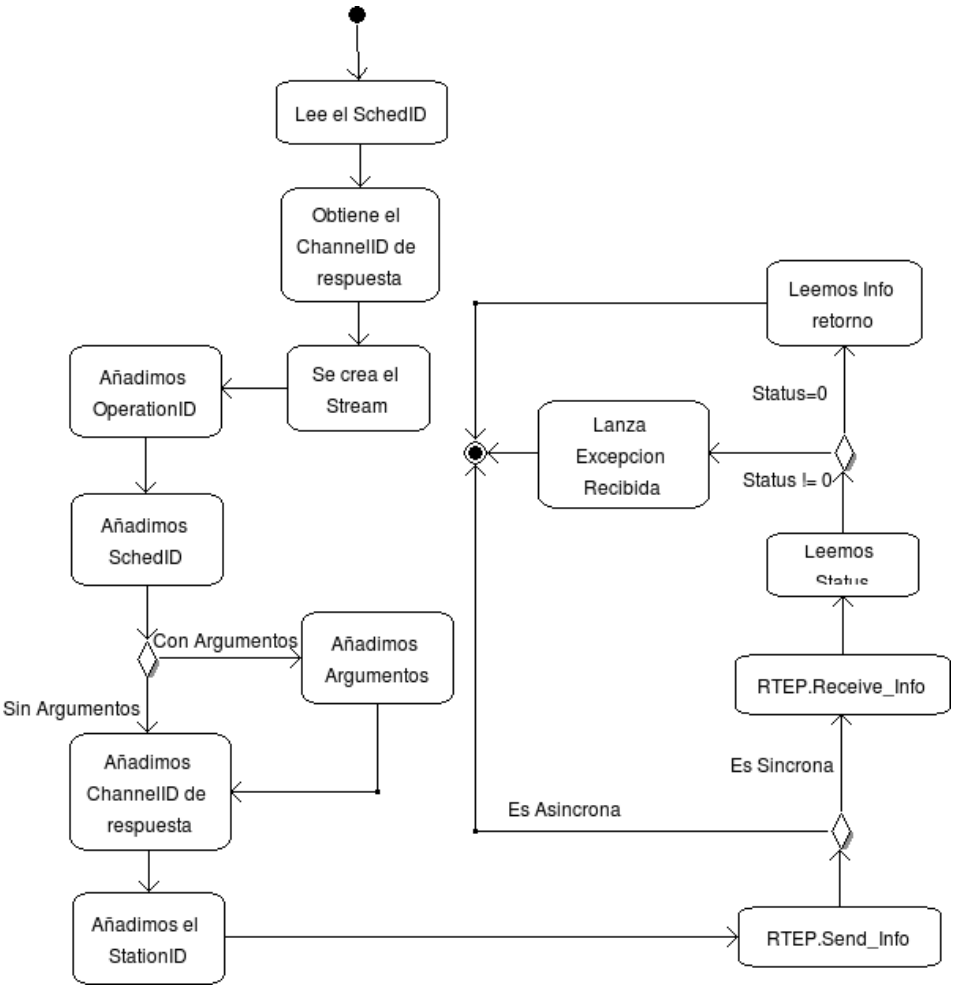

Figura 4.5: Diagrama de actividad de la implementación del Proxy

4. Transmitimos por RT-EP (*Send_Info*) dicha información con la prioridad que corresponda y el canal de envío que ha sido configurado.
5. Si la operación es asíncrona no hacemos nada más, pero si es síncrona nos suspendemos a la espera de respuesta (*Receive_Info*) en el canal de recepción obtenido en el punto 2.

6. Leemos la información de respuesta: Si es estado (*Status*) = 0 leemos la información de retorno y retornamos al componente cliente su petición, en caso contrario se ha producido una excepción en el componente servidor, leemos la excepción de la información de vuelta y propagamos dicha excepción.

4.3.2 El fragmento Servant del conector RT-EP

El elemento *servant* de un conector es también un componente estándar de la tecnología Ada-CCM. Su código de negocio implementa el mecanismo para recepción de los mensajes que representan las invocaciones remotas, ejecuta localmente el servicio en el servidor, y genera y transmite el mensaje con los valores de retorno proporcionados por el servicio invocado. En el caso del *servant*, se necesita disponer de un conjunto (pool) de threads creados que serán los que ejecutarán las invocaciones locales sobre los componentes servidores, correspondientes a las peticiones remotas. El *servant*, posee como propiedades de configuración el canal de recepción (*Channel_ID*), el número de threads del pool y la prioridad de los mismos.

En el diagrama de actividad de la figura 4.6 se muestra la funcionalidad del *servant*:

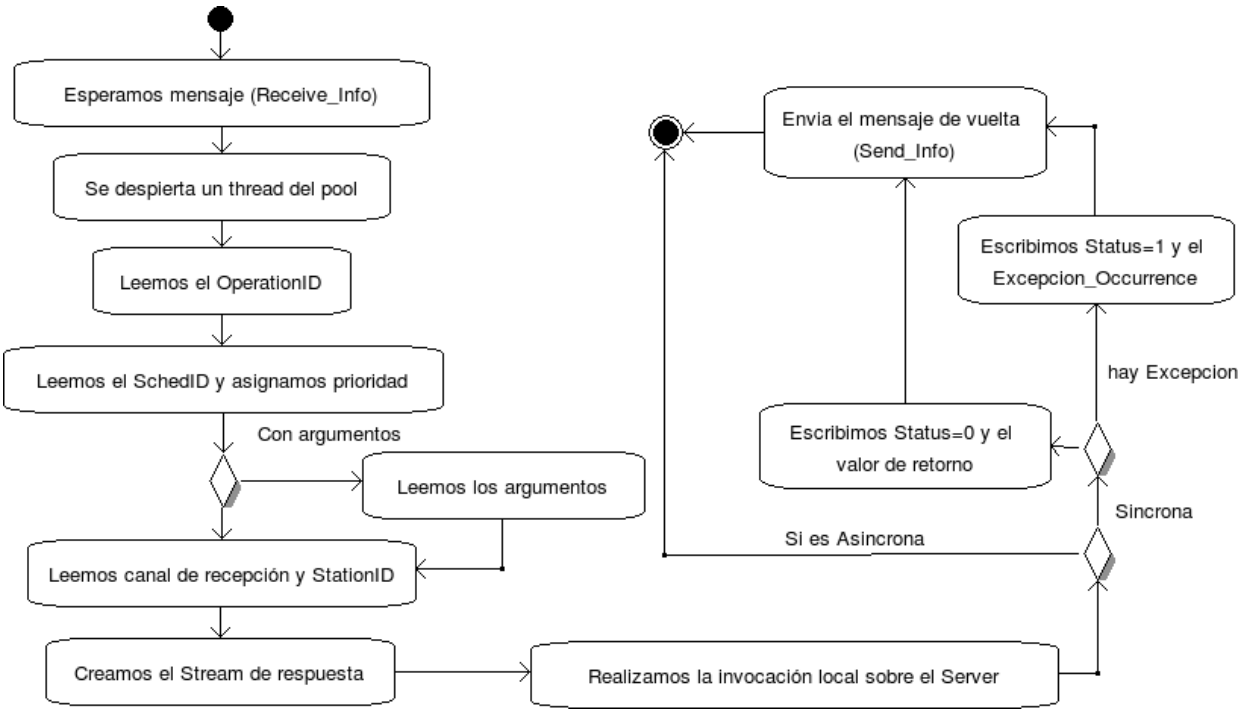

Figura 4.6: Diagrama de actividad de la implementación del servant

A continuación se detalla este proceso:

- Una tarea del pool se bloquea a la espera de un mensaje en el canal (*Channel_ID*) en el que está configurado para esperar. Se ejecuta para ello la instrucción *Receive_Info*
- Una vez recibido un mensaje, lo primero se despierta otra tarea del pool para que el servicio siga pudiendo recibir invocaciones remotas.
- Leemos la información contenida en el *Stream*: Identificador de la operación (*OperationID*); Identificador de la transacción (*SchedID*); Canal de respuesta; Argumentos de la operación (Si tiene).
- Creamos el *Stream* de respuesta.

- Realizamos la invocación local de la operación correspondiente (*OperationID*) sobre el componente servidor conectado al fragmento *servant* del conector.
- Si es una invocación síncrona y la operación produce una excepción, escribimos en el *Stream* de respuesta el estado de excepción (Status=1) y enviamos el identificador de dicha excepción (*Exception_Occurrence*). En el caso de no producirse excepción escribimos Status=0 y la información de retorno. Si la invocación es asíncrona no hacemos nada.

CAPÍTULO 5

APLICACIÓN DE LOS INTERCEPTORES PARA EL CONTROL DE PLANIFICACIÓN

5.1 Concepto y funcionalidad de los Interceptores

Los interceptores constituyen un mecanismo propuesto en la especificación QoSforCCM [12] para incorporar los servicios del entorno en el proceso de invocación de una operación, de manera que se realicen determinadas acciones antes y después de la invocación de dicha operación. En la tecnología Ada-CCM se utiliza para gestionar y controlar los parámetros de planificabilidad de los thread que ejecutan

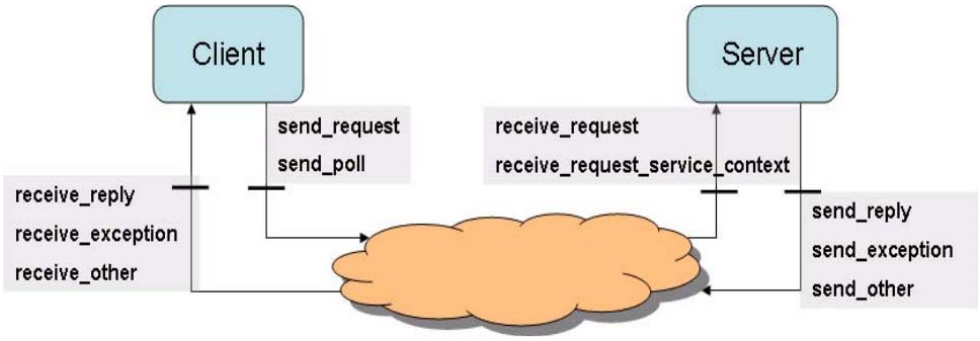

Figura 5.1: Puntos de intercepción según la especificación QoSforCCM

La introducción de estos elementos no es obligatoria y su configuración se realiza en el plan de despliegue de la aplicación.

Los interceptores se introducen a nivel del adaptador, y por tanto, desde ellas se tiene acceso a los recursos del entorno. Al introducirse al nivel del adaptador, el código de negocio de los componentes no se ve afectado.

La especificación QoSforCCM define una serie de puntos de intercepción en los que se pueden introducir dichos interceptores, que son los que se reflejan en la figura 5.1.

5.2 Estrategias de gestión de la planificación de las aplicaciones de tiempo Real

En la bibliografía se describen tres políticas básicas de asignación de los parámetros de planificación:

1. *Parámetros transmitidos por el cliente*: Los parámetros de planificación con los que se invoca una operación son los del cliente que invoca (o los equivalentes en el procesador remoto si la invocación es distribuida). En este caso, se ejecutan con los mismos parámetros de planificación todas las invocaciones de un servicio que se ejecuta dentro de una misma transacción.
2. *Parámetros establecidos en el servidor*: Los parámetros de planificación están definidos en el servidor y son los mismos en cualquier invocación, con independencia de la transacción dentro de la que se invoque.
3. *Parámetros definidos por la transacción*: Los parámetros de planificación de cada invocación se establecen en función de la transacción y de la actividad a las que corresponde la invocación. Este modo hace posible planificar de forma más detallada y eficiente las aplicaciones de tiempo real[14].

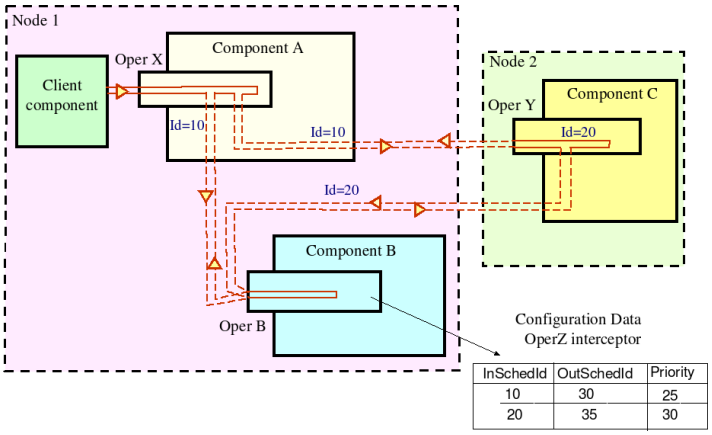

Figura 5.2: Modelo transaccional

El mecanismo basado en interceptores que se propone en la tecnología Ada-CCM permite implementar los tres mecanismos anteriores de asignación de los parámetros de configuración. Si tomamos como referencia la especificación de CORBA de tiempo real, la tecnología Ada-CCM incorpora como nuevo el tercer mecanismo.

5.3 Estructura e implementación de los Interceptores en la tecnología Ada-CCM

En la tecnología Ada-CCM se han implementado cuatro puntos de intercepción:

- En un Componente Cliente:
    1. Cuando el cliente invoca una operación de un receptáculo (*Send_Request*).
    2. Cuando el cliente recibe la respuesta a una invocación (*Receive_Reply*).
- En un Componente Servidor:
    3. Cuando el servidor recibe una invocación (Receive_Request)
    4. Cuando el servidor envía una respuesta (Send_Reply)

En la tecnología Ada-CCM la inclusión de los mecanismos de intercepción se realizará en el contenedor (*Wrapper*), de esta forma desde ellas se tiene acceso a los recursos del entorno y en particular al servicio *SchedulingControlService*, que posee la capacidad de modificar los parámetros de planificación del thread que lo invoca. Para que el contenedor pueda añadir interceptores, éste ha de implementar interfaces *ServerContainerInterceptorRegistration* y *ClientContainerInterceptorRegistration*, ambas definidas en QoSforCCM. La declaración de estas interfaces se describe en el siguiente código IDL:

```
module Components {
  module ContainerPortableInterceptor {
    local interface ClientContainerInterceptorRegistration {
      Components::Cookie register_client_interceptor (
                          in ClientContainerInterceptor ci);
      ClientContainerInterceptor unregister_client_interceptor (
                          in Components::Cookie cookie)
      raises(InvalidRegistration);
    };
  };
};

module Components {
  module ContainerPortableInterceptor {
    local interface ServerContainerInterceptorRegistration {
      Components::Cookie register_server_interceptor (
                          in ClientContainerInterceptor ci) ;
      ServerContainerInterceptor unregister_server_interceptor (
                          in Components::Cookie ck)
      raises(InvalidRegistration);
    };
  };
};
```

En la tecnología Ada-CCM se ha introducido la capacidad de especificar el puerto y la operación concreta a la que se aplicar el correspondiente interceptor, para ello la operación de registro tiene dos parámetros adicionales que identifican el puerto y la operación.

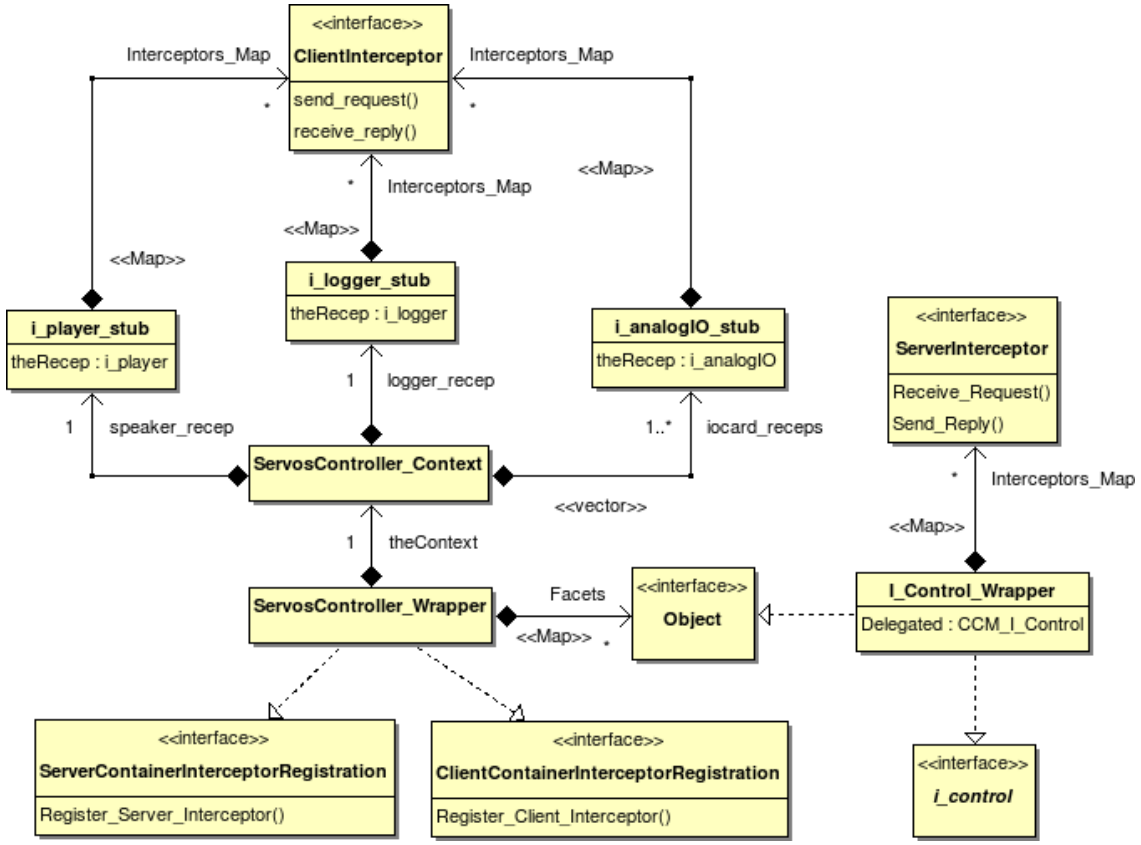

Figura 5.3 : Diagrama UML implementación interceptores en Ada-CCM.

Los interceptores que van a ser registrados, deben verificar las correspondientes interfaces *ClientContainerInterceptor* y *ServerContainerInterceptor*.

En la figura 5.3 se muestra una vista de la implementación de los interceptores en la clase *Wrapper*. La figura corresponde al componente ServosController que ya se ha utilizado en capítulos previos.

En los interceptores Server, es decir los que se ejecutan en el componente ServosController en su función de servidor de la faceta *controllerPort*, cada vez que se recibe una invocación a través de la faceta, se transmite al envoltorio de la interfaz correspondiente (*I_Control_Wrapper*), el cual posee la lista de interceptores asociados a dicha invocación (*Interceptors_Map*), que fueron registrados en el despliegue (*Register_Server_Interceptor*), y puede tanto realizar la invocación de los puntos de control (*Receive_Request y Send_Reply*), como la propia invocación sobre la implementación de la faceta.

Por otra parte, en cuanto a los interceptores que se ejecutan en su papel de componente cliente en la invocación de operaciones sobre sus receptáculos, cuando se produce una invocación de una operación sobre el contexto (*theContext*), ésta se traslada al envoltorio del receptáculo que corresponda (*I_Player_Stub*, *I_Logger_Stub* o *I_Analog_Stub* en el caso del componente ServosController) , y dicho *Stub* poseerá la lista de interceptores correspondiente (*Interceptors_Map*), con lo que podrá acceder del mismo modo tanto a los puntos de control (*Send_Request* y *Receive_Reply*) como a la referencia del receptáculo.

5.4 Funcionalidad, estructura e implementación del servicio de gestión de parámetros de planificabilidad

En la tecnología Ada-CCM, los interceptores controlan las características de planificación del thread que ejecuta la operación que se invoca.

A través de ellos se asigna a los thread las prioridades de acuerdo al modelo transaccional de la aplicación, esto es, los parámetros de planificación de cada invocación se establecen en función de la transacción y de la actividad a las que corresponde la invocación. Este modo hace posible planificar de forma detallada y eficiente las aplicaciones de tiempo real y es el utilizado en el entorno MAST que se utiliza como entorno de análisis en esta tecnología.

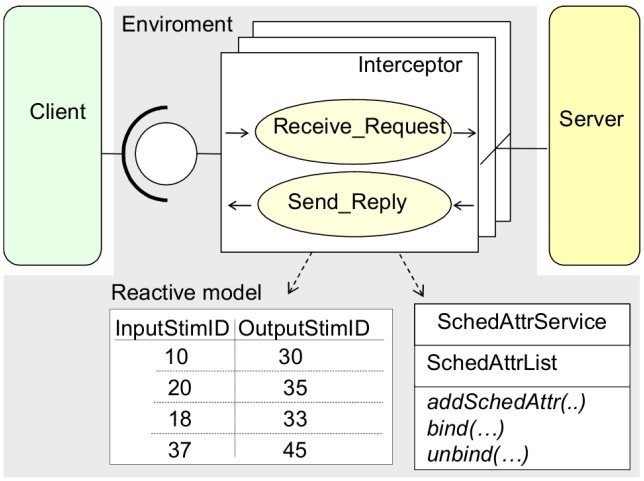

Figura 5.4: Elementos de un interceptor

La gestión de los parámetros de planificación se realiza utilizando un identificador (*StimulusID*), que es transmitido por la invocación e identifica al cliente que lo invoca y la transacción y actividad del modelo reactivo a la que corresponde:

En la operación *Receive_Request()*, el interceptor hace uso del servicio de la plataforma *SchedulingControlService* para, en función del identificador asociado a la invocación, obtener el nuevo *stimulusID* que será asociado a la actividad que se inicia, y el identificador del parámetro de planificación (*schedID*) que define las características de planificación que se asocian al thread que ejecuta la operación. Haciendo uso del servicio *SchedulingControlService*, el thread adquiere las características de planificación que corresponden al *schedID* que le ha sido asignado.

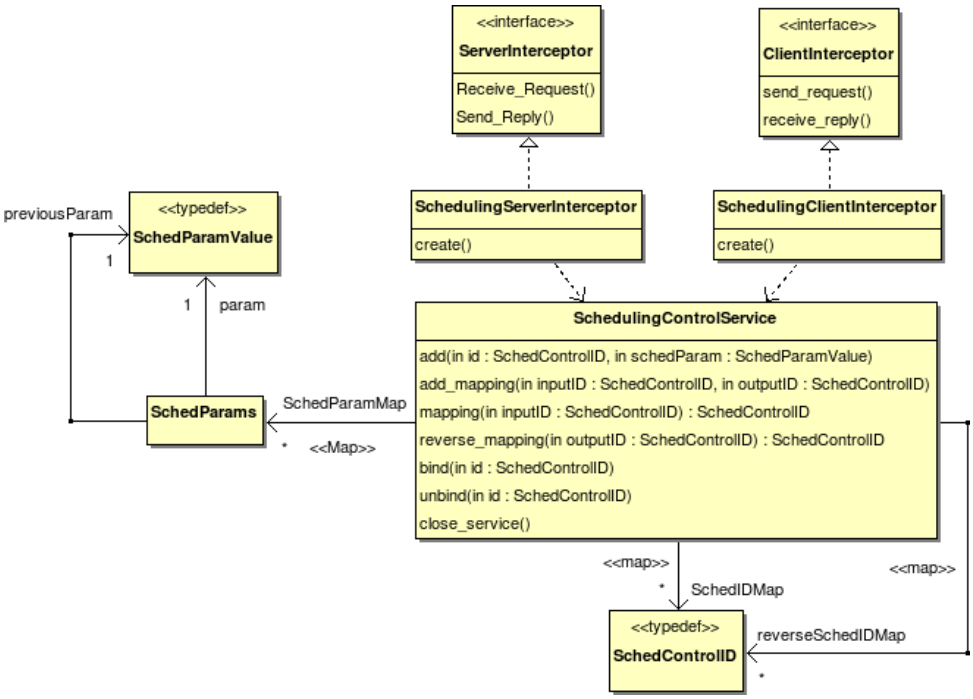

Figura 5.5: Diagrama de clases del gestor de parámetros de planificación

En la operación *send_Reply()* el thread recupera el *stimulusID* y los parámetros de planificación que tenía en la invocación.

La estructura del servicio de gestión de parámetros de planificación sigue la estructura que se muestra en el diagrama de clases UML que se muestra en la figura 5.5.

En la figura 5.6 se muestra el diagrama de secuencia del papel del gestor en esta tecnología:

1. El cliente invoca una operación sobre un componente Servidor.
2. El servidor invoca la operación *Receive_Request* sobre el servidor de planificación de los interceptores. Éste a partir del gestor de parámetros de planificación obtiene el *stimulusID* de la tarea, a partir del cual obtendrá el *SchedID* (*mapping*), con el que establecerá (*bind*) los parámetros de planificación del thread que ejecutará la operación.

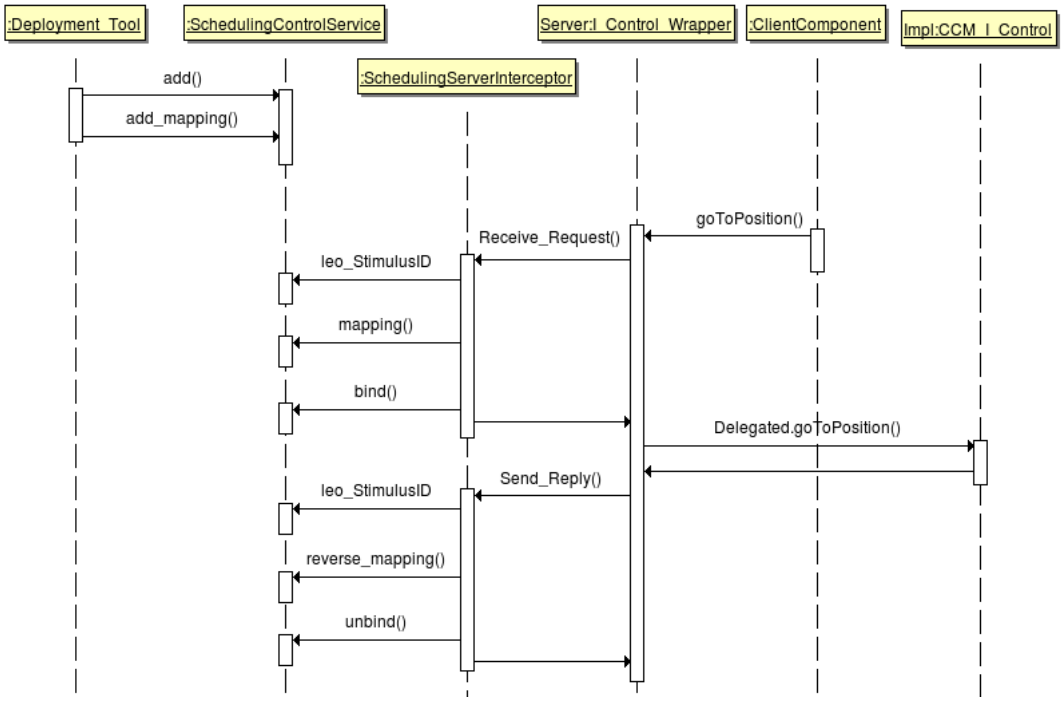

Figura 5.6: Diagrama de secuencia del gestor de planificación

3. El servidor ejecuta la operación solicitada por el cliente sobre la implementación (*Delegated.goToPosition*).
4. El servidor invoca la operación *Send_Reply* sobre el servidor de planificación de los interceptores. Éste leerá de nuevo el *StimulusID*, a partir del cual recuperará el *SchedID* inicial (*reverse_mapping*), y reestablecerá (*unbind*) los parámetros de planificación del thread que se encargó de la ejecución de la operación solicitada.

CAPÍTULO 6

EJEMPLO DE APLICACIÓN: JET FOLLOWER

6.1 Demostrador Jet Follower: Especificación y Arquitectura

El objetivo del demostrador, es poner de manifiesto la capacidad de la tecnología Ada-CCM para dar soporte a aplicaciones basadas en componentes de tiempo real en plataformas embebidas con recursos limitados, y distribuidas en diferentes nodos. Se utilizarán diferentes tipos de componentes con requerimientos temporales, se ejecutará en un sistema distribuido con 2 nodos que operarán bajo sistema operativo MaRTE.

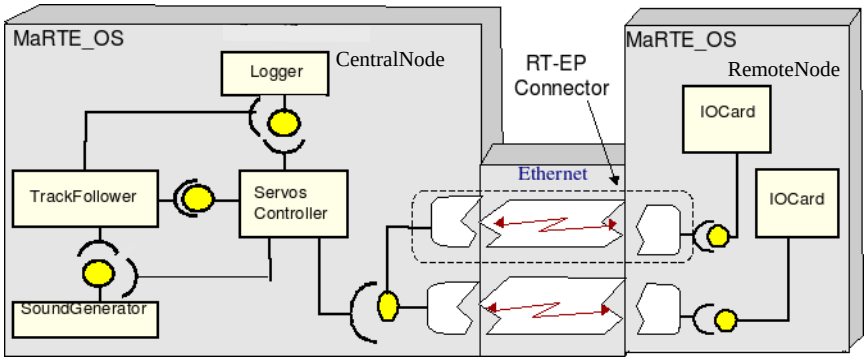

Figura 6.1: Arquitectura del demostrador.

La aplicación es un sistema que controla la posición de un conjunto de n servos, que controlan la posición y orientación de un dispositivo mecánico hacia un objeto móvil cuya posición es determinada por un dispositivo externo, al que denominamos *Tracker*. Ejemplos de aplicaciones a los que puede corresponder este sistema, es el de control de la orientación de una antena situado sobre un vehículo móvil hacia un satélite de comunicaciones fijo, o la orientación de un sistema antiaéreo hacia un avión que se mueva en el espacio.

La funcionalidad básica del sistema es el seguimiento continuo del objetivo, pero también bajo demanda del operador. El sistema registra en un *Logger* la trayectoria que realmente sigue el sistema (trayectoria de los servos), y así mismo genera señales acústicas cuando el sistema está operando o cuando los errores de posición de los servos superan unos ciertos umbrales preestablecidos.

Para realizar esta aplicación, desde esta tecnología necesitamos de un componente cliente al que hemos llamado TrackFollower. Necesitaremos un componente intermedio que implemente la interfaz *I_Control*, al que hemos llamado ServosController, y tres componentes servidores finales SoundGenerator, Logger e IOCard, que implementarán las interfaces *I_Player*, *I_Logger* e *I_AnalogIO* respectivamente.

El componente TrackFollower requiere utilizar operaciones definidas en las interfaces *I_Logger*, *I_Control* e *I_Player*, por lo tanto tendrá que conectarse a los componentes Logger, ServosController y SoundGenerator. Por su parte el componente ServosController deberá conectarse a los componentes SoundGenerator, Logger y IOCard para desarrollar su funcionalidad.

6.2 Catálogo de componentes

La aplicación hace uso de 5 componentes para implementar su funcionalidad: TrackFollower, Logger, IoCard, SoundGenerator y ServosController. El componente TrackFollower es específico de esta aplicación, Los restantes componentes son característicos de su dominio de aplicación y han sido desarrollados con independencia de esta aplicación.

6.2.1 El Componente TrackFollower

Este componente representa al cliente que controla la aplicación. Poseerá 3 receptáculos y no ofrecerá ninguna faceta. Tiene capacidad para ejecutar concurrentemente las siguientes actividades:

- Atender el teclado y en función del comando que se pulsa realizar la acción que corresponda. Para ello necesitará el puerto de activación *keyboardThread* de tipo *OneShotActivation*.

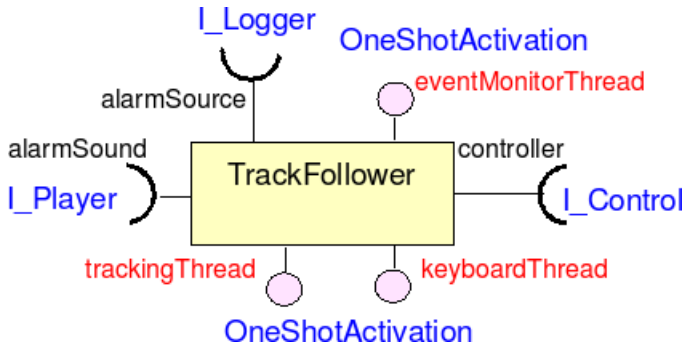

Figura 6.2: Especificación del componente TrackFollower

- Recibe del dispositivo externo *trackingSystem*, posiciones del objetivo que deben ser seguidas. Esta actividad se realiza haciendo uso del thread de activación *OneShot* que se obtiene del puerto *trackingThread*. Este thread permanece a la espera de que

el *trackingSystem* proporcione una posición del objetivo, y cuando la recibe planifica trayectoria que deben seguir los servos para su seguimiento.

- Visualiza en el monitor las alarmas y errores que se registren en el Logger, y en el caso de que se encuentre habilitada, las coordenadas de la trayectoria de los servos que se registran. Esta actividad se implementa con el thread que proporciona el entono a través del puerto *OneShot eventMonitorThread*. Este thread se suspende en el Logger y a cada registro que se produzca en él, retorna el nuevo registro.

6.2.2 El Componente Logger

Este componente es un componente servidor (no requiere receptáculos), que representa a un objeto pasivo (no posee activaciones) que permite el acceso a un registro de eventos para registrar de forma persistente los eventos que se generan en la aplicación. Implementa la interfaz *I_Logger* y la ofrece a través de su faceta *loggerPort*.

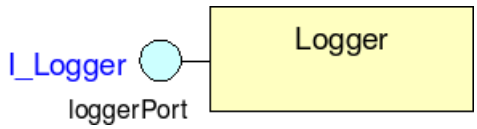

Figura 6.3: Especificación del componente Logger

La interfaz *I_Logger* proporciona las operaciones de acceso y escritura en el Logger:

- *log(type: in EventType; mssg: in Unbounded_String)*: almacena en el Logger el registro correspondiente. Los eventos se almacenan en una lista *EventList*, asociando a cada registro la fecha, el tipo de evento y el mensaje asociado al mismo.
- *get(option: in RequiringEventOption; numEvent : in Natural)*: retorna una lista de *numEvent* eventos del tipo que se solicita registrados en el Logger.
- *awaitEvent(option: in RequirinEventOption)*: Operación que se queda a la espera de la llegada de un evento al Logger. Es ejecutada por el thread de un componente conectado al componente que implemente esta interfaz (por ejemplo el componente TrackFollower mediante el thread *eventMonitorThread*).

6.2.3 El Componente IoCard

Es un componente servidor que implementa el acceso a la tarjeta de IO analógica, la cual contiene los convertidores D/A y A/D con los que respectivamente se controla los servos y se leen sus posiciones. Es un componente pasivo que sirve las invocaciones de lectura y establecimiento de las señales analógicas, a través de su faceta *ioPort* que implementa la interfaz *I_AnalogIO*.

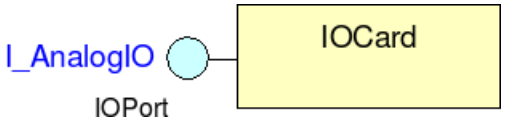

Figura 6.4: Especificación del componente IOCard

Las principales operaciones que ofrece *I_AnalogIO* son :

- *Ai_NumberOfLine()* y *AoNumberOfLine()*: Devuelve el número de líneas de entrada/Salida de la tarjeta.

- *Ai_SetGain()*: Devuelve el rango de los valores de entrada de la tarjeta.
- *Ai_GetValue(line: Natural)*: Devuelve los voltios leídos en la línea especificada.
- *Ao_SetValue(line: Natural; value: Float)*: Establece los voltios deseados en la línea de salida especificada.

6.2.4 El Componente SoundGenerator

Es un componente servidor (no posee receptáculos) que permite generar notas musicales o melodías utilizando el altavoz existente en el PC. Ofrece la interfaz *I_Player* que permite controlar los sonidos que genera. Es un componente activo que requiere ejecutar código en cada cambio de nota.

La actividad que ejecuta la obtiene de la activación periódica a través del puerto *soundThread*. Periódicamente, con un periodo preconfigurado como propiedad de configuración del componente, se ejecutará la operación *update()*, que irá recorriendo la lista de notas y las irá tocando en función de la duración y características de las mismas.

Este componente ofrece la interfaz *I_Player* a través de la faceta *playerPort*. En esta aplicación los componentes ServosController y TrackFollower requerirán de esta implementación a través de sus receptáculos, con el fin de generar sonidos.

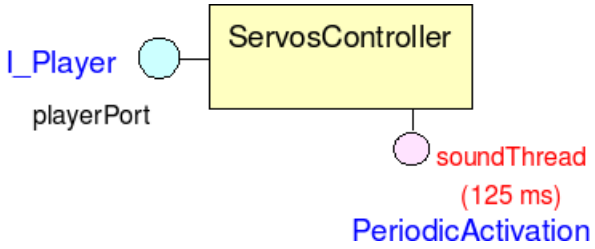

Figura 6.5: Especificación del componente SoundGenerator

La interfaz *I_Player* ofrece las siguientes operaciones:

- *setTimeBase(newTempo: in Tempo)*: Establece el tiempo musical con el que se tocarán las notas. Por tanto, la elección del tiempo condicionará la velocidad en que se reproducirán las notas modificando el marcapasos.
- *playNote(note: in Scale; numTick: in Natural)*: Tocará una nota musical durante un número de *ticks* proporcionado.
- *play(mldy: in String)*: Interpretará la melodía o conjunto de notas musicales.
- *manyPlay(mldy: in String)*: Interpretará la melodía repetidamente hasta que se indique lo contrario.
- *silent()*: Parará la reproducción de sonidos que se estén reproduciendo en el momento de ejecutarse esta operación.
- *tick()*, *tickTock()*, *success()* y *fail()*: Son sonidos preconfigurados que pueden ser reproducidos ejecutando los procedimientos correspondientes.

6.2.5 El Componente ServosController

Este componente realiza las siguientes funciones:

- Implementa el algoritmo de control PID (Proporcional integral derivativo) para el conjunto de servos del sistema. Necesitará el puerto de activación periódico *controlThread*, para realizar el algoritmo de control cada 5ms.

- Genera sonidos en el dispositivo SoundGenerator cuando el sistema se está moviendo o el error de posición de los servos supera el valor umbral establecido. Por lo tanto, requerirá por medio del receptáculo *speaker*, de un componente que ofrezca la funcionalidad de la interfaz *I_Player*.
- Ofrece un modo de monitorización que registra periódicamente en el Logger la posición de los servos o los errores de posición de los mismos que superan un umbral. Por una parte necesitará un receptáculo para acceder al Logger, y por otro lado, para realizar esta monitorización requiere del entorno a través del puerto *pollingThread* un nuevo thread, que registra cada segundo las posiciones actuales en el Logger.

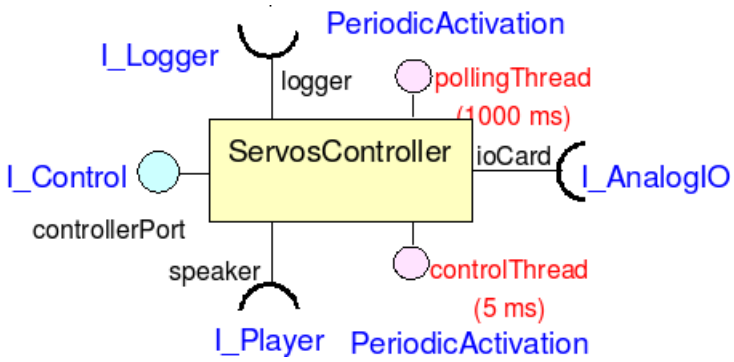

Figura 6.6: Especificación del componente ServosController.

La interfaz *I_Control* ofrece las siguientes operaciones:

- *goToPosition(newPosition: in Position)*: Establece una nueva posición a la que debe conducirse los servos desde la situación actual. Para ello se realiza una interpolación lineal de acuerdo con los parámetros de trayectoria.
- *getCurrentPosition()*: Retorna la posición actual de los servos.
- *startLogging()*: Establece el modo de de registro de la trayectoria. Tras ser invocado este procedimiento el sistema registra en el Logger como *warning* la posición de los servos cada segundo.
- *cancelLogging()*: Deshabilita el modo de monitorización de trayectoria en el Logger.

6.3 Modelo de la plataforma: Descripción D&C

La especificación D&C de la plataforma (ficheros .tdm) ha sido extendida para incluir el modelo de tiempo real. El modelo de la plataforma define los modelos los recursos software (Sistema operativo, mutexes, drivers, etc.) y los recursos hardware (Procesadores, redes, timers, etc.), que caracterizan y cuantifican la disponibilidad de capacidad de procesamiento, las sobrecargas asociadas, las políticas de gestión de las colas, etc.

En esta aplicación la descripción de la plataforma contiene como información más relevante:

- Información de los nodos que se disponen para la ejecución de la aplicación. En nuestro caso tendremos dos nodos: *centraNode* y *remoteNode*. Dentro de la información que concierne a cada uno de los nodos, tenemos los recursos asociados y disponibles que presenta. En el caso de la plataforma disponible para la ejecución del demostrador, se dispone de una plataforma con dos nodos y en cada uno de ellos con una partición. Como información de los recursos de los nodos estará el sistema operativo sobre el que trabajan, que en el caso de los dos nodos será MaRTE OS.

También formará parte de la información relativa al nodo la IP del mismo y la red a la que pertenece.

- Información de la redes de interconexión entre los nodos. Incluye el nombre de los nodos que interconecta y la información relativa al protocolo que se utiliza. En este caso, el protocolo utilizado será el protocolo RT-EP.

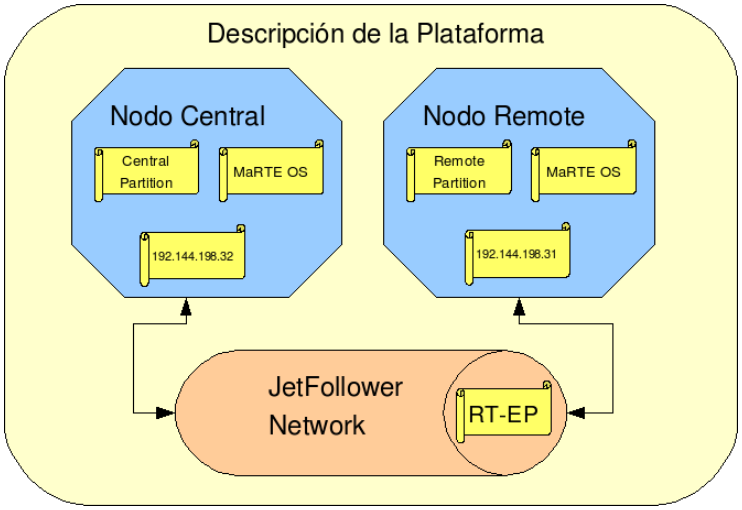

Figura 6.7 : Descripción de la plataforma en la que se ejecutará la aplicación

6.4 Plan de despliegue. Descripción D&C

A partir de la descripción contenida en el fichero .cad (Component Assembly Description) en el que se define el ensamblado de los componentes de la aplicación, y de las características del conjunto de transacciones de tiempo real que se ejecutan concurrentemente definidas en el fichero .wld (Workload Description) como una extensión de tiempo real, se formará la información contenida en el plan de despliegue de la aplicación (Fichero .cdp).

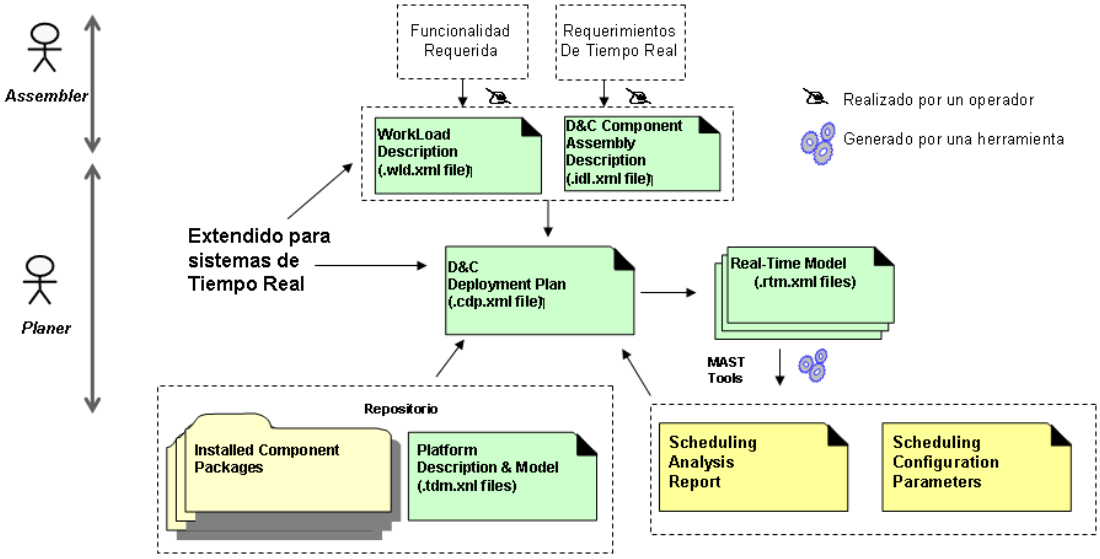

Figura 6.8: Composición del plan de despliegue de la aplicación.

En la figura 6.8 se muestran las fases del desarrollo de una aplicación. Se compone básicamente de 3 fases: la definición por el ensamblador de la aplicación como un conjunto

de instancias de componentes enlazadas entre si; la elaboración por el planificador del plan de despliegue que determina la forma en que la aplicación se instala en la plataforma, y la generación por el ejecutor del código ejecutable para cada procesador de la plataforma que participe en la ejecución.

El plan de despliegue contiene la siguiente información:

- Instancias que intervienen en la aplicación: Se requerirá una instancia (*jetFollower*) del componente TrackFollower, para la realizar la interfaz de usuario de la aplicación. Para generar los sonidos que se requieren en la aplicación se necesitará una instancia (*theSpeaker*) del componente SoundGenerator, y para registrar los eventos que se producen tenemos que crear una instancia (*theLogger*) del componente Logger. Por otra parte se necesitará controlar dos servos, la base y el brazo móvil, por lo que se requerirá una instancia del componente ServosController (*theController*) y otras dos instancias (*baseCard* y *armCard*) del componente IOCard.
- Localización de los nodos en que se ejecutarán las instancias: Debido a la localización remota de los servos cuyo control realizamos, esta aplicación estará distribuida en dos nodos. En el nodo central se ejecutarán las instancias de los componentes TrackFollower, SoundGenerator, Logger y ServosController. Y en el nodo remoto las instancias *armCard* y *baseCard* correspondientes al componentes IOCard.
- Conexiones a realizar entre los puertos de instancias que lo requieran: La instancia *JetFollower* requiere de la instancia *theSpeaker* para generar sonidos y de la instancia *theLogger* para monitorizar los eventos. A su vez el controlador requerirá de la instancia *theSpeaker* para generar sonidos y de la instancia *theLogger* para almacenar los eventos que se vayan produciendo. Por último el controlador requerirá también la conexión con las instancias *armCard* y *baseCard* para realizar su funcionalidad.
- Asignación de las propiedades de configuración: Se configurarán los puertos de activación que contengan los componentes con los parámetros de planificación obtenidos como resultado de la utilización de las herramientas de modelado y análisis de tiempo real, que en nuestro caso será MAST.

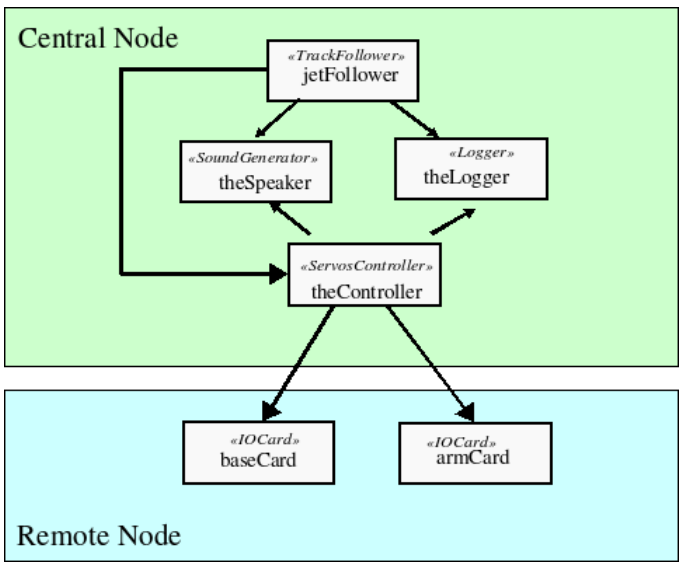

Figura 6.9: Localización y conexión de las instancias que intervienen en la aplicación.

CAPÍTULO 7

CONCLUSIONES Y LÍNEAS FUTURAS DE TRABAJO

7.1 Conclusiones

Se ha realizado con éxito un demostrador sobre plataforma distribuida de tiempo real estricto. Hemos utilizado dos targets (PC's a 200Mhz) corriendo en MaRTE sobre máquina desnuda. Tanto el código correspondiente al contenedor de esta tecnología, como la implementación del código de negocio, han sido implementados sobre lenguaje Ada 2005.

La utilización de Ada 2005 hace que esta tecnología sea adecuada para plataformas embebidas con recursos limitados (sin sistema de ficheros, middleware de comunicaciones, etc.), interconectable mediante redes de comunicaciones de tiempo real (RT-EP). Además, el soporte de Ada para concurrencia y sincronización proporciona la capacidad de desarrollar el código de negocio con comportamiento temporal predecible.

Se han realizado dos implementaciones de los conectores: la primera fue realizada utilizando la implementación GLADE del anexo distribuido de Ada (DSA), la cual no poseía comportamiento temporal predecible. La segunda implementación de los conectores ha sido realizada utilizando directamente RT-EP como mecanismo de comunicación, que cumple los requerimientos de tiempo real. Esta experiencia nos ha servido para certificar la independencia de los componentes sobre el mecanismo de comunicación utilizado en los conectores.

Se ha realizado la especificación y documentación completa de la herramienta de generación de código automática correspondiente a los componentes para la tecnología Ada-CCM. Para esta tarea se ha tenido que realizar una extensión en el mapeo de Idl a la nueva especificación de Ada.

Los resultados obtenidos en este trabajo, están incluidos en las siguientes tres publicaciones:

- P. López Martinez, P. Pacheco, J.L Medina y J.M. Drake: “RT-CCM: Tecnología de componentes de tiempo real basada en Ada 2005” Congreso Español de Informática (CEDI’07), Zaragoza, 2007.
- P. López Martinez, J.M. Drake, P. Pacheco y J.L Medina:”An Ada 2005 Technology for Distributed and Real-Time Component-based Applications” Lecture Notes on Computer Science, Springer, LNCS 5026 ISBN: 3-540-68621-7, p.p. 254-267, June, 2008.
- P. López Martinez, J.M. Drake, J.L Medina y P. Pacheco: “Ada-CCM: Component-based Technology for Distributed real-time systems” 11th International Symposium on Component Based Software Engineering (CBSE, 2008) Karlsruhe October, 2008.

7.2 Líneas de trabajo futuras

Las líneas de investigación que quedan abiertas tras este trabajo son muchas y abarcan múltiples aspectos: Optimización del código, incorporación de nuevos servicios, elaboración del entorno de trabajo, estandarización de la tecnología dentro de la organización OMG, etc.

De entre todas ellas, la que voy a abordar como parte de mi Tesis Doctoral es el desarrollo de componentes de tiempo real autoajustables para el diseño óptimo de aplicaciones de tiempo real en plataformas distribuidas abiertas. El diseño de aplicaciones en plataformas distribuidas abiertas (en las que no se conoce la carga de trabajo), requiere la utilización de servicios de reserva de recursos en la plataforma, de forma que con independencia de la carga que se esté ejecutando en ella, proporcione a la aplicación los recursos que necesita para ser planificada de acuerdo con sus requisitos temporales. La evaluación de los contratos de reserva de recursos se realiza en función del modelo de tiempo real de la aplicación. Sin embargo, la evaluación de los valores de los parámetros de estos modelos en componentes reutilizables en muchas aplicaciones y bajo situaciones muy diferentes, es una tarea ardua sino es imposible. Por ello, la línea de trabajo que vamos a emprender es la especificación, el diseño, y la implementación de servicios estandarizados que se puedan incorporar a los componentes, a fin de monitorizar la actividad y el comportamiento temporal de las instancias de los componentes, durante la propia ejecución de la aplicación, y por tanto en un entorno real. De la valoración estadística de esta información, se puede estimar de forma precisa los valores de los parámetros de su modelo de tiempo real, y los requerimientos de recursos que necesita la aplicación. Con modelos de tiempo real exactos, se pueden ajustar de forma óptima los contratos de reserva de recursos que la aplicación debe establecer con la plataforma de ejecución para garantizar su planificabilidad.

CAPÍTULO 8

REFERENCIAS BIBLIOGRÁFICAS

[1] OMG (Object Management Group)
Corba Components Model Specification, Version 1.2
www.ddj.com/article/printableArticle.jhtml?articleID=184
Formal/05-01-04.

[2] ITEA project MERCED (Market-Enabler for Retargetable COTS components in Embedded Domain).
http://www.itea-merced.org.

[3] IST projects: COMPARE (Component-based approach for real-time and embedded systems).
http://www.ist-compare.org.

[4] IST projects: FRESCOR (Framework for Real-time embedded Systems based on Contracts).
http://www.frescor.org.

[5] OMG (Object Management Group)
Lightweight Corba Component Model.
pct/03-11-03.

[6] OMG (Object Management Group)
Ada Language Mapping Specification - Version 1.2.
October 2001.

[7] OMG (Object Management Group)
Deployment and Configuration of Component-based Distributed Applications Specification, Version 4.0.
http://www.omg.org
formal/06-04-02.

[8] P. López, J.M. Drake and J.L. Medina
Real Time Modelling of Distributed Component-Based Applications
32th Euromicro Conference on Software Engineering and Advanced Applications
Croacia, August 2006.

[9] M.González Harbour, J.J. Gutiérrez, J.C. Palencia y J.M.Drake
MAST: Modeling and Análisis Suite for Real Time Applications.
Euromicro Conference on Real-Time Systems
http://mast.unican.es
June 2001.

[10] J.M.Martínez and M. González
RT-EP: A Fixed-Priority Real Time Communication Protocol over Standard Ethernet
10th Int. Conference on Reliable Software Technologies, Ada-Europe 2005
York(UK), June 2005.

[11] L. Pautet and S. Tardieu.
GLADE: a Framework for Building Large Object-Oriented Real-Time Distributed Systems.
3rd IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing,
Newport Beach (USA), March 2000.

[12] OMG (Object Management Group)
Quality of Service for CORBA Components
ptc/06-04-05.

[13] Martínez, J.M y González Harbour, M.
Diseño, implementación y modelado de un protocolo multipunto de comunicaciones para aplicaciones de tiempo real distribuidas y analizables sobre Ethernet. (RT-EP)".
Proyecto Fin de Carrera. Ingeniería de Telecomunicación.
Septiembre 2002.

[14] J.J.Gutiérrez García and M. González Harbour
Prioritizing Remote Procedure Calls in Ada Distributed Systems.
9th Intl. Real-Time Ada Workshop, ACM Ada Letters, XIX, 2, pp. 67 72
Junio 1999.

CAPÍTULO 9

ANEXO

Como parte del anexo, se incluye en el CD en el que va contenido este documento (Directorio XML), los esquemas que formalizan la definición de los ficheros XML que se han hecho mención en la redacción del documento.

Los esquemas incluidos en el anexo son:

- DnC_CCM_BasicTypes.xsd
- DnC_CCM_CommonTypes.xsd
- DnC_CCM_ComponentDataModel.xsd
- DnC_CCM_ExecutionDataModel
- DnC_CCM_TargetDataModel
- WorkloadModel.xsd

Además se incluye el código de los componentes que forman la aplicación JetFollower descrita en el capítulo 6. Estos ficheros se incluyen en el directorio SCS donde:

- El directorio *SCS\CCM* se incluyen los paquetes correspondientes a la tecnología. Éstos son comunes a todas las aplicaciones desarrolladas sobre la tecnología Ada-CCM.
- Los directorios *SCS\Control*, *SCS\io*, *SCS\media* y *SCS\database* contienen los paquetes correspondientes a los diferentes dominios de la aplicación. Contendrán por tanto el código de las interfaces utilizadas en la aplicación.
- Los directorios *SCS\iocard*, *SCS\logger*, *SCS\servoscontroller* y *SCS\soundGenerator* contienen los paquetes correspondientes a los componentes que forman la aplicación.
- Los directorios *SCS\iocard_conector_proxy* y *SCS\iocard_conector_servant* contienen el código de los fragmentos *proxy* y *servant* del conector utilizado en el ejemplo.