
# UNIVERSIDAD POLITÉCNICA DE VALENCIA

FACULTAD DE INFORMÁTICA
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN


# DESARROLLO DE APLICACIONES DISTRIBUIDAS Y MÓVILES DESDE UN MODELO ARQUITECTÓNICO ORIENTADO A ASPECTOS

**Proyecto Final de Carrera**
Septiembre de 2006


**Autor:**
Carlos Millán Belda

**Dirigido por:**
Nour Ali Irshaid, Isidro Ramos Salavert

# *Agradecimientos*

*A Nour, por por su esfuerzo y dedicación, pero también por sus lecciones; esto no sería nada de no ser por todo lo que he aprendido de ella.*

*A Cristóbal, por su paciencia y simpatía, pero también por tenderme una mano cuando surgían las dudass.*

*A Jennifer, por su atenta ayuda y por todas sus recomendaciones.*

*A Javi C. y Javi G., por su colaboración indispensable en los momentos difíciles.*

*A Pepe Carsí e Isidro Ramos, por ofrecerme la oportunicad de realizar este proyecto y por compartir sus conocimientos y sus consejos.*

*A Ismael, Carlos, Rogelio, Manolo, José Antonio, Rafa, Gonzalo, Alejandro, Abel, Pascual, Javi J., Elena, Raquel y José Miguel, por tantos y tan buenos ratos…*

# ÍNDICE GENERAL

# ÍNDICE DE FIGURAS

# ÍNDICE DE TABLAS

A mis padres y a mi hermano

*...i a Paula*

CAPÍTULO I

# INTRODUCCIÓN

## Contenidos del capítulo

# INTRODUCCIÓN

En las últimas décadas, la sociedad de información ha experimentado cambios importantes debido a la aparición de nuevas tendencias culturales y nuevos dominios de negocio e industria, provocados por la incorporación en la vida diaria de nuevas tecnologías y la implantación de Internet como un marco global de conocimiento. Por estas razones, se han promovido dos ideas importantes: el mundo se considera una única unidad sin límites, y la gente trabaja de una manera colaborativa desde distintos lugares del mundo sin necesidad de reunirse físicamente.

Estas ideas han causado la necesidad de que los procesos de desarrollo de los sistemas software actuales traten con estructuras complejas, nuevos requisitos no funcionales, la adaptación dinámica y con nuevas tecnologías. Además, muchos de los sistemas software requieren la capacidad para trabajar con diversos dispositivos (PC, computadoras portátiles, PDAs, teléfonos móviles, etc.) a través de redes de comunicación de manera distribuida y segura. Consecuentemente, los procesos de desarrollo *software* deben tener en cuenta la naturaleza distribuida, ubicua y móvil de los sistemas software desde las etapas más tempranas del ciclo de vida software. Considerar la distribución a un nivel alto de abstracción permite la definición de modelos de distribución independientes de plataforma y la especificación de patrones de generación de código. Esta manera de tratar la distribución permite desarrollar sistemas *software* distribuidos de forma eficiente, fiable, ahorrando tiempo y costes.

No obstante, la mayoría de enfoques posponen el tratamiento de dichos requisitos relacionados con la distribución y la movilidad a la etapa de implementación, no ofreciendo soporte para éstos durante las etapas de requisitos, análisis o diseño. Esta forma de abordar estos requisitos no funcionales tiene graves consecuencias en el producto software, ya que al introducir cambios en la implementación no se preserva la trazabilidad entre las distintas etapas del ciclo de vida software y se crea una gran dependencia entre el modelo de distribución y la plataforma de desarrollo. Por lo tanto, el mantenimiento y reutilización de código del producto software resulta muy complicado.

Estos problemas generan un gran desafío, que es el definir un enfoque de desarrollo que permita la especificación de propiedades de distribución y movilidad de forma abstracta, evitando así la dependencia a una plataforma tecnológica. Además, también hay que destacar que dicho enfoque debe

mantener de forma consistente el estado distribuido de las aplicaciones de este tipo.

Las arquitecturas software se consideran el puente entre las etapas de requisitos e implementación del ciclo de vida software. Éstas describen la estructura de los sistemas software utilizando Lenguajes de Descripción de Arquitecturas (LDAs). Sin embargo, los LDAs no proporcionan primitivas para definir requisitos de distribución o movilidad, así como propiedades relacionadas con estos requisitos.

El objetivo de este proyecto es la implementación de un modelo que dé soporte al desarrollo de aplicaciones distribuidas y móviles de modelos arquitectónicos orientados a aspectos. Concretamente, modelos que han sido definidos siguiendo el enfoque PRISMA. PRISMA permite definir arquitecturas software distribuidas, haciendo uso de Aspectos de Distribución y canales transparentes que permiten la comunicación distribuida. También es objetivo, dar soporte al desarrollo de aplicaciones distribuidas y móviles haciendo uso del modelo Ambient-PRISMA, una ampliación del PRISMA que incorpora en su LDA primitivas de distribución y movilidad en las primeras etapas de desarrollo software mediante el uso de Ambient Calculus [Ali06b].

## 1.1. Contexto

PRISMA [Pe05] es un modelo para la definición de arquitecturas software basado en la integración de dos enfoques de desarrollo software: el Desarrollo de Software Orientado a Aspectos (DSOA) [Aosd] y el Desarrollo de Software Basado en componentes (DSBC) [Szy02]. PRISMA utiliza los aspectos para describir las diferentes características (distribución, seguridad, coordinación, etc.) de los elementos arquitectónicos (componentes, conectores, sistemas) que componen la arquitectura. PRISMA permite la especificación de sistemas complejos, distribuidos y altamente dinámicos. Para ello, proporciona un Lenguaje de Descripción de Arquitecturas Orientado a Aspectos (LDAOA) [Pe06]. En lo que respecta a las características distribuidas del modelo, en PRISMA se pueden definir elementos arquitectónicos móviles mediante la inclusión de un Aspecto de Distribución que encapsule la lógica de la movilidad. Además, PRISMA tiene como primitivas los canales de comunicación entre los diferentes elementos arquitectónicos que proporcionan comunicación distribuida de forma transparente.

La funcionalidad básica del modelo PRISMA ha sido implementada en anteriores Proyectos Finales de Carrera [Cos05] para su ejecución sobre el *framework* .NET de Microsoft©, utilizando el lenguaje de programación *C#* y *.NET Remoting* para implementar las características de comunicación remota y movilidad. El resultado de la implementación es PRISMANET [Pe05b], un *middleware* que permite la ejecución de arquitecturas PRISMA en el marco de .NET. PRISMANET está formado por un conjunto de librerías que dan soporte a cada uno de los tipos del modelo PRISMA y que proporciona servicios de gestión para la ejecución de instancias de dichos tipos. PRISMANET permite

además, la ejecución de configuraciones PRISMA distribuidas mediante la ejecución del *middleware* en distintas máquinas.

Trabajos posteriores han ampliado el modelo PRISMA mediante la incorporación de primitivas específicas para la especificación de la movilidad y distribución [Ali05]. Estas primitivas están basadas en los formalismos expresados en el Cálculo de Ambientes [Car98]. Como resultado de la incorporación de los conceptos del Cálculo de Ambientes a PRISMA se obtiene la aproximación Ambient-PRISMA [Ali06b]. Una de las principales características de Ambient-PRISMA es la incorporación del concepto de ambiente al metamodelo PRISMA [Ali05b]. De esa forma, el modelo proporciona primitivas para modelar la distribución y la movilidad.

## 1.2. Objetivos

El presente proyecto está dividido en dos partes bien diferenciadas.:

1. Por un lado se pretende ampliar el modelo de distribución del PRISMANET, mejorando el diseño y la implementación de los canales de comunicación existentes, con el objetivo de conseguir un modelo de comunicación distribuida y de movilidad más eficiente y robusto. Para ello, además de modificar dichos canales, se dotará al *middleware* de mecanismos para la gestión del entorno de ejecución distribuida que conforman la unión de todos los *middlewares* sobre los que se puede propagar una configuración arquitectónica PRISMA. Esto propiciará una situación favorable para la gestión de arquitecturas distribuidas, creación de elementos arquitectónicos remotos, resolución de nombres, etc. Finalmente, aunque no menos importante, se pretende ampliar el modelo de ejecución del *middleware* para el soporte de ejecución transaccional se servicios. Mediante la incorporacion de transacciones a PRISMANET se dará soporte a la ejecución de servicios transaccionales que puedan propagarse a través de los límites distribuidos de los *middlewares*, es decir, en el marco del entorno de ejecución PRISMA.

2. El otro gran reto del proyecto es estudiar la forma en que las primitivas del Cálculo de Ambientes aplicadas a PRISMA puedan ser llevadas a la implementación. Para ello se creará el *middleware* Ambient-PRISMANET, que partirá de una versión de PRISMANET mejorada, con gestión del entorno distribuido y transacciones, para dar soporte a las nuevas primitivas que proporciona Ambient-PRISMA. Mediante esta aproximación, se pretende dar un soporte de ejecución sobre la plataforma .NET a las primitivas de movilidad de Ambient-PRISMA, basadas en el Cálculo de Ambientes.

Cabe añadir que la realización de la primera parte es vital para la consecución de los objetivos de la segunda, ya que para incorporar los conceptos del Ambient-PRISMA al middleware es necesario un entorno de ejecución de la

movilidad ampliado, como se explica, y soporte para la ejecución de servicios transaccionales.

## 1.3. Organización

El proyecto está organizado en siete capítulos y la bibliografía. En primer lugar, este capítulo de introducción introduce el contexto en el que se ha desarrollado el proyecto, sus objetivos y su organización.

En el capítulo 2 se presentan los trabajos relacionados. Para ello, se da una breve introducción al DSOA de cara a mostrar el estado del arte en cuanto a las aproximaciones que contemplen la distribución con aspectos. A continuación, se hace un breve análisis de las distintas aproximaciones que contemplan la movilidad, haciendo una mención especial a las plataformas de agentes. Finalmente, se introduce el Cálculo de Ambientes y se habla de sus implementaciones existentes.

En el capítulo 3 se introduce el modelo PRISMA, su Lenguaje de Descripción de Arquitecturas Orientado a Aspectos y su implementación previa sobre la plataforma .NET (PRISMANET) en la que se centra la explicación en los modelos de ejecución para la comunicación distribuida y la movilidad, tal y como estaban implementados.

En el capítulo 4 se detallan las diferentes modificaciones que se han realizado sobre PRISMANET para mejorar los modelos de ejecución de la comunicación distribuida. También se presenta la ampliación del *middleware* para dar soporte a la gestión de un entorno distribuido, y como eso facilita la construcción de un DNS distribuido. Finalmente, se presenta el modelo transaccional de PRISMANET y como se ha implementado en el *middleware* para dar soporte a las transacciones en el entorno de ejecución.

En el capítulo 5, se presenta Ambient PRISMA detallando las ampliaciones propuestas por este modelo. Se muestra también como la incorporación de las nuevas primitivas afecta al Lenguaje de Descripción de Arquitecturas Orientado a Aspectos y, para finalizar, se presenta la especificación de un caso de estudio.

El capítulo 6, se muestran las estrategias seguidas para la implementación de los diferentes conceptos de Ambient-PRISMA sobre el *middleware* PRISMANET, de cara a la consecución de una nueva versión del mismo llamada Ambient-PRISMANET.

Finalmente, en el capítulo 7 se presentan los resultados obtenidos en base a los objetivos propuestos, así como las conclusiones y los trabajos futuros.

CAPÍTULO II

# TRABAJOS RELACIONADOS

## Contenidos del capítulo

# TRABAJOS RELACIONADOS

Como se ha comentado el presente proyecto está dividido en dos fases: Una primera fase en la que se amplia el modelo de distribución y movilidad de un modelo arquitectónico orientado a aspectos. Y una segunda fase en la que se da soporte a una ampliación de dicho modelo que incorpora las primitivas de movilidad basadas en el Cálculo de Ambientes. Por ello, el presente capítulo está dividido en dos secciones. Por un lado se hará una breve introducción a los sistemas distribuidos y a las plataformas más conocidas que les proporcionan soporte. Puesto que estas plataformas están basadas en el paradigma de software orientado a objetos, se mostrarán cuales son los problemas que presenta este paradigma y cómo los solventa el paradigma orientado a aspectos, al que se hará, previamente, una introducción basándose en su lenguaje más extendido, AspectJ. A continuación se explicará, como se da soporte a los sistemas distribuidos mediante el paradigma orientado aspectos, presentando brevemente dos de los *frameworks* existentes para la programación distribuida orientada a aspectos como son D y PaDA. Finalmente, se introducirán los conceptos relacionados con la movilidad en los sistemas distribuidos y se hará un recorrido a través de las plataformas de agentes móviles.

Por otro lado, se introducirán los conceptos del Cálculo de Ambientes y se presentarán brevemente las propuestas para su implementación, con la finalidad de poner al lector en antecedentes para la segunda parte del proyecto.

## 2.1. Desarrollo de Software Orientado a Aspectos

El desarrollo de software orientado a objetos (DSOO) [Eli00] ha aportado al desarrollo de software la introducción de conceptos como la encapsulación y la herencia, proporcionando un grado más alto de reutilización que la programación procedural. No obstante, el DSOO falla al intentar representar algunas de las características de los sistemas complejos de hoy en día como son los requisitos no funcionales del sistema. Frente a esto, se ha introducido el desarrollo de software orientado a aspectos (DSOA) [Kic97] como una nueva técnica de desarrollo a lo largo del ciclo de vida. Entre las mejoras que proporciona el DSOA se encuentran los mayores grados de modularidad, reutilización y mantenibilidad y evolución.

La intención del DSOA es permitir una adecuada modularización de las aplicaciones y posibilitar una mejor separación de conceptos (*separation of concerns*). Gracias al DSOA se pueden capturar los diferentes conceptos que componen una aplicación en entidades bien definidas, eliminando las dependencias inherentes entre cada uno de los módulos. De esta forma, se consigue razonar mejor sobre los conceptos (*concerns*), se elimina la dispersión del código y las implementaciones resultan más comprensibles, adaptables y reutilizables. En la Figura 1, se muestra como se separan los diferentes concerns que se encuentran entremezclados en el código tradicional (Figura 1 (a)) para obtener un código más limpio, organizado, ganando las ventajas antes mencionadas (Figura 1 (b)). Además, la mantenibilidad y evolución es facilitada mediante la manipulación únicamente de un concepto, independientemente de los demás.

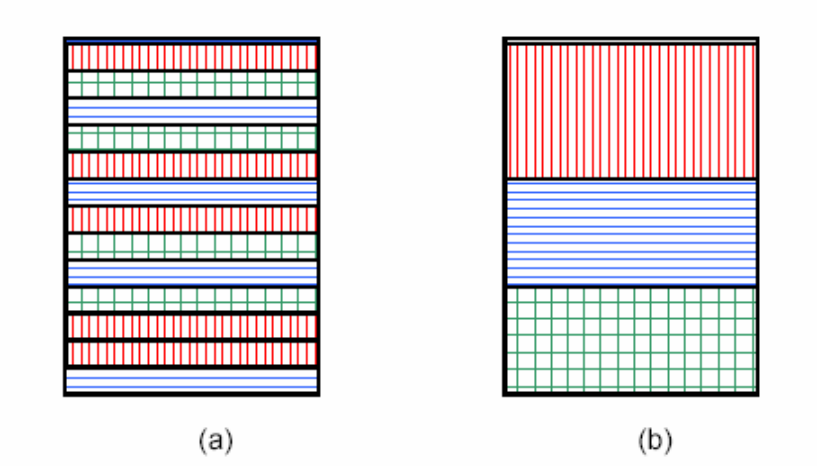



**Figura 1: Separación de concerns en el codigo**

Uno de los lenguajes orientados a aspectos con más aceptación ha sido AspectJ, que es una extensión de Java para la Programación Orientada a Aspectos (POA); Además de ser el que más avanzado se encuentra, es en el que se han basado la mayoría de aproximaciones existentes para otros lenguajes. A continuación, se realizará una breve introducción a AspectJ con el propósito de presentar sus construcciones más características.

AspectJ [AsJ03][AsJ04] utiliza un compilador propio mediante el cual se realizan los enlaces entre aspectos y aplicaciones Java. AspectJ es una tecnología que extiende el lenguaje origen, es decir, el programador que necesite utilizar POA necesariamente deberá aprender una sintaxis adicional para utilizar aspectos en su aplicación.

Los lenguajes basados en AspectJ incorporan las siguientes construcciones: *Join Points*, *Pointcuts*, *Advices* y *Aspects*. Los *joinpoints* están ligados al código base de la aplicación, mientras que los *aspects*, *pointcuts* y *advices* se definen en el código aspecto. Cada una de estas construcciones se explican a continuación:

Un aspecto es una construcción del lenguaje que encapsula un *crosscutting concern*. Su definición es muy similar a una clase de Java, con la salvedad que además añade *advices* y *pointcuts*, estructuras sintácticas que definen cómo y a qué código se enlazarán los aspectos, respectivamente.Un aspecto se enlaza con uno o varios métodos del código base, a través de los *pointcuts* (puntos de corte), y mediante los *advices* se especifica si la ejecución del código del aspecto será antes, después o en lugar de la ejecución del método indicado por el *pointcut*.

La forma en que AspectJ combina los *concerns* con el código origen, es mediante el enlazado en tiempo de compilación (w*eaving*). El código original es transformado por el compilador de AspectJ de forma que los *advices* son transformados en métodos estándar, cuya llamada es insertada en aquellos puntos del código origen indicada por los *pointcuts*. Si existen zonas del código que son resueltas dinámicamente, el código es dirigido a las zonas controladas por los *advices*.

Un *Join Point* (punto de unión) es un concepto semántico introducido en *AspectJ* que se define como un punto bien definido en el flujo de ejecución de un programa. Una llamada a un método o un acceso a un atributo serían ejemplos de *joinpoints*. Existen diferentes tipos de *joinpoints* definidos en AspectJ. Un *joinpoint* debe considerarse como un punto en la ejecución de la aplicación susceptible de ser interceptado por los aspectos, es decir, como un punto de entrada donde podrá introducirse el código de los aspectos para alterar el flujo de ejecución normal del código base. Es importante tener en cuenta que cada *joinpoint* es distinto en tiempo de ejecución, tiene un contexto de ejecución diferente. Por ejemplo, un *joinpoint* correspondiente a la invocación de un método, será diferente según sea el objeto que lo llame. Los *joinpoints* forman parte del código base y no necesitan de ninguna construcción explícita del lenguaje para especificarlos.

Por tanto, una aplicación tendrá un conjunto de *joinpoints*, de los cuales tan sólo un pequeño subconjunto interesará interceptar. Para ello, se definen los *pointcuts* (puntos de corte) mediante los cuales se podrá seleccionar, del conjunto de todos los *Join Points* posibles, aquellos que se desean utilizar para asociarlos con un aspecto concreto. Es por esto que los *pointcuts* son los elementos de AspectJ que permiten definir el *weaving* (enlazado) de un aspecto con el código base.

Un *advice* define el código que deberá ser ejecutado en los *joinpoints* seleccionados por los *pointcuts*. La ejecución de dicho código estará condicionada por el tipo de *advice* que lo contenga. AspectJ dispone de diferentes tipos: *before*, *after* y *around*. El advice *before* describe que el código enlazado se ejecuta antes que cada *joinpoint*. En el *after*, el código enlazado se ejecuta después de la ejecución de cada *joinpoint*. Existen dos subtipos: *after throwing* y *after returning*. El primero es utilizado cuando se lanza una excepción, y el segundo cuando se finaliza con normalidad la ejecución del

*joinpoint* Finalmente, en el *advice around* el código del aspecto reemplaza el código que se iba a ejecutar en el *joinpoint*.

## 2.2. Sistemas distribuidos

Los avances en las redes de telecomunicaciones han impulsado la investigación en sistemas distribuidos. De ese modo se ha establecido una visión donde los dispositivos ya no se ven como entidades autónomas auto contenidas que, ocasionalmente, se comunican con otra, sino que se ven como entidades que forman parte de una plataforma de computación global formada por la sinergia de todos los recursos locales, cuya compartición es proporcionada gracias a las redes de comunicación de banda ancha.

Los sistemas distribuidos hacen uso de un *middleware* [Bern96] de objetos o componentes distribuidos. El papel de este tipo de *middleware* es facilitar la tarea para programar y gestionar aplicaciones distribuidas. En cierto modo se trata de plataformas que abstraen la complejidad y la heterogeneidad de las múltiples tecnologías de red, sistemas operativos, lenguajes de programación y arquitecturas de las máquinas que subyacen.

Dos de las propuestas más conocidas y utilizadas en lo que respecta a plataformas para la programación distribuida son las siguientes:

- **CORBA del OMG**: CORBA (*Common Object Request Broker Architecture*) [CORBA] es un estándar que establece una plataforma de desarrollo de sistemas distribuidos facilitando la invocación de métodos remotos bajo un paradigma orientado a objetos. CORBA utiliza un Lenguaje de Definición de Interfaces (IDL) para especificar los interfaces con los servicios que los objetos ofrecerán. CORBA puede especificar a partir de este IDL la interfaz a un lenguaje determinado, describiendo cómo los tipos de dato CORBA deben ser utilizados en las implementaciones del cliente y del servidor, y abstrayendo el hecho que dichas implementaciones pueden estar hechas en lenguajes de programación diferentes. De esa forma el IDL es el responsable de asegurar que os datos son correctamente intercambiados entre los diferentes lenguajes de programación. CORBA también proporciona un *Object Request Broker* (ORB) que se encarga de redirigir de forma transparente invocaciones a métodos entre objetos remotos, así como de proporcionar otra serie de servicios a los objetos, como nombrado, replicación, comportamiento transaccional, etc.

- **Microsoft ® .NET Remoting**: Remoting [Ramm02] [Scott03] proporciona un *framework* que permite a los objetos interactuar entre ellos a través de dominios de aplicación. Remoting facilita el desarrollo de aplicaciones distribuidas, permitiendo que dos objetos

que se encuentren en máquinas distintas, comunicadas a través de una red cualquiera, puedan comunicarse entre sí. Además de ofrecer una gran cantidad de servicios y protocolos estándar ya implementados, permite usar y/o construir formatos de codificación o protocolos de comunicación personalizados. También proporciona los mecanismos básicos para los Servicios Web de .NET, implementando estándares como SOAP y WSDL.

Ambas tecnologías expuestas están orientadas a objetos. Como se ha explicado en la sección 2.1, el DSOA se presenta como una aproximación que promueve el desarrollo a lo largo del ciclo de vida y la especificación de requisitos no funcionales. A continuación se describen algunas aproximaciones que hacen uso del DSOA para especificar características de distribución.

### 2.2.1. Aspectos de distribución

Si se trata de separar *concerns* de una aplicación, las características distribuidas son un buen ejemplo de código que se pueden separar en un aspecto determinado. En la presente sección se hará un breve recorrido entre los trabajos relacionados con Aspectos de Distribución.

El Aspecto de Distribución en la mayoría de propuestas se ha definido para encapsular las propiedades necesarias para la comunicación remota. Uno de los primeros trabajos realizados en cuanto a la separación de *concerns* relacionados con la distribución usando la POA es el *framework* D [Lo97]. En D se identifican las características distribuidas y se programan de forma externa a la implementación de la funcionalidad de la aplicación. D proporciona primitivas para la separación sintáctica de los conceptos distribuidos, como la creación y coordinación de hilos de ejecución y la comunicación entre espacios de ejecución.

PaDA (*Pattern for Distribution Aspect*) [Soa02] propone un patrón para el *concern* de distribución usando AspectJ. Este patrón proporciona comunicación remota entre dos componentes (un cliente y un servidor) de un sistema. Se proponen tres aspectos: Un aspecto en la parte del cliente (*Client-Side*) para llamar remotamente al componente servidor, un aspecto en la parte del servidor (*Server-Side*) para permitir la recepción de las llamadas remotas y un aspecto de manejo de excepciones (*Exception Handling*). El aspecto de la parte del cliente esta enlazado (*weaved*) con el componente origen mientras que el aspecto de la parte del servidor se enlaza con el componente destino. El aspecto de manejo de excepciones está enlazado con ambos componentes, origen y destino (ver Figura 2). El patrón PaDA consigue un alto grado de modularidad que hace que el código fuente sea independiente de la API de comunicaciones. Esto permite conseguir un alto grado de mantenibilidad en el que el API de la comunicación puede ser cambiado sin afectar al funcionamiento del sistema. Además, la separación de *concerns* facilita las pruebas funcionales; los errores

del código de la distribución no afectarán a las pruebas de la funcionalidad de la aplicación local.

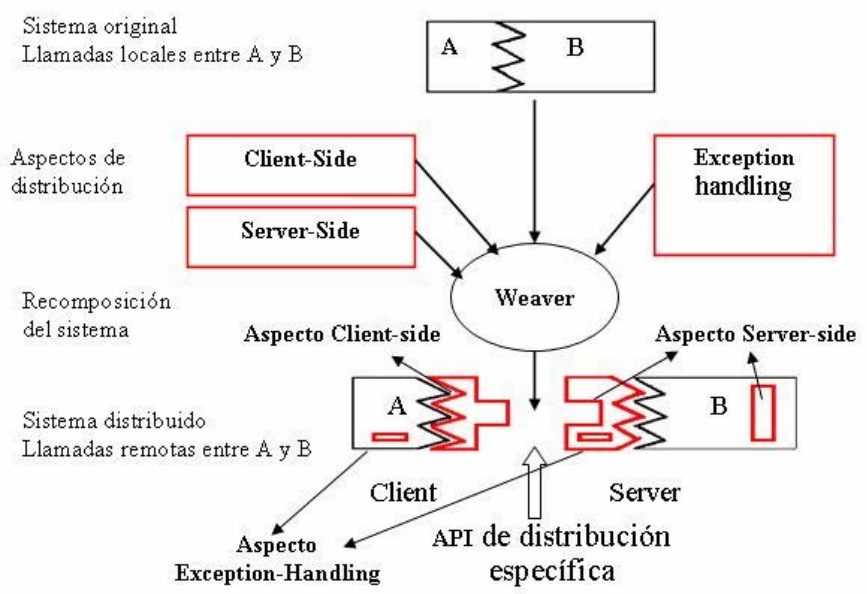



**Figura 2: Estructura de PaDA**

No obstante, la experiencia en el desarrollo de aplicaciones distribuidas con AspectJ pone en relieve una serie de inconvenientes [Soa02b]. El principal inconveniente es que la definición de *pointcut* identifica una parte específica de un sistema determinado, por lo que los aspectos se crean específicamente para ese sistema, perjudicando a la reutilización de éstos.

Como posible solución a este inconveniente, en [He03] se propone una plataforma para el desarrollo de aplicaciones orientadas a aspectos que trabaja con el paradigma orientado a aspectos a un nivel conceptual para generar código dinámicamente. El aspecto de distribución de esta propuesta contiene una tabla de referencias y una plataforma de comunicación. La tabla de referencias mantiene referencias a todos los objetos invocados. La plataforma de comunicación es la responsable de proporcionar los mecanismos necesarios para enviar y recibir mensajes hacia y desde otras plataformas de comunicación.

### 2.2.2. Movilidad

Acordes a una visión basada en la visión global de los dispositivos como conjunto que comparte recursos locales, ha surgido una nueva generación de aplicaciones distribuidas cuya característica distintiva reside en la movilidad de código. Estas aplicaciones se han denominado Aplicaciones de Código Móvil (*Mobile Code Applications* , MCA) y se clasifican [Cu97] en los siguientes tipos:

- **Evaluación remota**: Según esta aproximación, cualquier componente de una MCA puede invocar servicios proporcionados por otros componentes, que se encuentran distribuidos en los nodos

de una red, no sólo proporcionando los datos de entrada necesarios para la invocación del servicio, sino proporcionando el código que describe como realizar dicho servicio. MCA basados en esta aproximación están formados por agentes que se mueven de entorno en entorno con el objetivo de realizar determinadas tareas en cada uno de ellos.

- **Código bajo demanda**: Según esta aproximación, el código que describe el comportamiento de un MCA puede variar con el tiempo. Un componente ejecutándose en un entorno computacional puede descargar y enlazar al vuelo el código para desarrollar una tarea determinada desde un componente remoto que actúa como un servidor de código.

- **Agentes Móviles**: Un agentes se puede ver como una unidad de ejecución. El adjetivo Móvil indica que estas unidades de ejecución que se encuentran en un entorno de computación, pueden moverse a otro entorno reanudando su ejecución en él. La movilidad, por tanto, conlleva datos y código, para que la unidad de ejecución pueda reanudar su trabajo en el destino.

En la presente sección se va a centrar en la aproximación de agentes móviles, pues el objetivo es hacer un recorrido sobre el estado del arte de las aproximaciones de movilidad de código centrado en entidades en ejecución que cambian su ubicación dinámicamente.

Antes de entrar en materia, cabe hacer una distinción más. Como se ha comentado, los lenguajes de código móvil se caracterizan por el hecho de que el segmento de código, estado de la ejecución y espacio de datos de una unidad de ejecución se puede mover de entorno de computación en entorno de computación. No obstante, a la hora de mover dicha unidad de ejecución se puede distinguir entre dos tipos de movilidad: La movilidad fuerte y la movilidad débil [Fug98].

- **La movilidad fuerte (*strong mobility*)** es la habilidad de un lenguaje de código móvil de mover las unidades de ejecución, transmitiendo su código y estado de ejecución. Las unidades de ejecución son suspendidas, enviadas a destino y resumidas, de forma que reanudan la ejecución desde la instrucción en que se quedaron. En resumidas cuentas, se trata de mover el código, los datos y el hilo de ejecución manteniendo su estado.

- **La movilidad débil (*weak mobility*)** denota los casos en que se transmite únicamente el código y los datos de una unidad de ejecución, pero al reanudar la ejecución en destino, esta ha de comenzar desde el principio. Es una movilidad poco transparente en el sentido que hay que preparar la unidad de ejecución para ser movida, salvando sus datos, llevándola a un estado seguro, etc.

Algunos trabajos [Be01] hablan también de la Movilidad Completa (*full mobility*) que supone el movimiento de la unidad de ejecución completa, incluyendo la pila de llamadas del hilo de ejecución, nombres de espacios y otros recursos. Se trata de una generalización de la movilidad fuerte que hace de la migración un proceso totalmente transparente.

En cualquier caso, la movilidad fuerte es muy complicada de implementar, pues depende en gran medida del entorno en que se está ejecutando la aplicación y de cómo éste gestione internamente los *threads*. Por ejemplo, para una aplicación ejecutándose bajo la plataforma .NET el proporcionar una movilidad fuerte supondría por un lado obtener el puntero y el contexto de ejecución de los *threads* que se vayan a mover. Por otro lado, en la máquina destino de la movilidad, se ha de poder regenerar el *thread* usando el contexto de ejecución e iniciarlo por el punto en que se encontraba. Dicho de otro modo, para proporcionar *strong mobility* con .NET es necesario conseguir serialización de *threads*, la cual cosa no es de momento posible en el *framework* .NET.

No obstante son muchas las aproximaciones que proporcionan movilidad débil para la migración de objetos. A continuación, se hace un breve resumen de estas aproximaciones, mostrando sus características más relevantes.

El *framework* basado en Java MobJex [Rya04] proporciona movilidad débil así como acceso remoto a los objetos. La arquitectura del *framework* está dividida en tres componentes: un *System Controler* encargado de la gestión del los modelos y del entorno de ejecución, una consola de administración que interactúa con el *System Controller* y un *Mobility Subsystem*, responsable de la migración transparente de los objetos de la aplicación a través del entorno de ejecución. Las características de movilidad y acceso remoto de los objetos se consigue precompilando los objetos móviles (*mobjects*) con el objetivo de generar dos interfaces: una interfaz remota y otra local. Esto permite que los métodos sean llamados directamente, vía la interfaz local, cuando el invocador se encuentra en el mismo *host* que el invocado, o a través de la interfaz remota cuando no se da ese caso. Dos clases más son generadas: un *proxy* que proporciona un cliente con la referencia al servidor y una clase serializable que representa la clase original que implementa las dos interfaces. Una característica relevante de MobJex es que la peticiones de movilidad no pueden ser requeridas por el mismo objeto que ha de moverse; son invocadas por el *System Controller*. Esto elimina la posibilidad de los objetos de ser autónomos. Además, si un objeto servidor se mueve, se produce una cadena de llamadas para localizar su nueva localización, puesto que el objeto *proxy* no es notificado directamente del cambio.

Otra aproximación que trata la movilidad en Java es la de los *Active Containers* implementada en JACOb [Cha03]. Esta aproximación proporciona un compilador que dinámicamente genera el código para insertar (`set`), eliminar (`remove`) y obtener copias (`get`) de los contenedores de objetos (*Active Containers*). De esa forma, la comunicación entre objetos es transparente pues

se hace a través de una llamada al active container, que alberga la lógica interna de dicha comunicación. Esta llamada genérica (`call`) recibe como parámetro el objeto a que va dirigida la petición, el nombre del método a invocar y una lista con los argumentos de la llamada. Así, los mecanismos de reflexión de Java son utilizados para redirigir la petición al objeto adecuado. Por otro lado, para mover un objeto es necesario indicar tanto el active container en que se encuentra el objeto como el active container al que se quiere mover, siguiendo una secuencia del tipo obtener copia, eliminar de origen y poner en destino:

```
fromActiveContainer.get(key)
fromActiveContainer.Remove(key)
toActiveContainner.put(key, object)
```

Dicha secuencia se encontrará entre dos llamadas genéricas al objeto para detenerlo en origen previamente a la movilidad, y otra para reanudarlo en destino. Esta forma de proporcionar la movilidad reduce la transparencia del proceso e impide a los objetos iniciar su movilidad.

Uno de los pocos trabajos en este campo realizado con tecnología .NET es el que se presenta en [Tro03]. Esta aproximación usa Aspectos para separar las decisiones de movilidad del código del objeto. Esta aproximación proporciona a los objetos la posibilidad de autoiniciar un proceso de movilidad. Los cambios de localización provocados por la movilidad son transparentes a los objetos puesto que existe un modulo encargado de redirigir las peticiones previa localización de la ubicación de un objeto.

Al hablar de aproximaciones que proporcionan movilidad débil y además permiten a los objetos iniciar la movilidad de forma activa, es necesario hacer referencia al campo de los agentes móviles. Los sistemas de agentes móviles, donde un agente es una entidad de código activo, autónomo y dirigido por un objetivo, también se basan en la movilidad de código auto-contenidos a través del sistema.

### 2.2.3. Plataformas de agentes móviles

En esta sección se hará un breve resumen de Aglets y Grasshopper, dos de las plataformas de agentes móviles más importantes. Además se presentará la plataforma ProActive que además de soportar la movilidad, presenta un diseño interesante para la comunicación distribuida. Puesto que estas plataformas de agentes están basadas en Java, también se comentarán en esta sección MAPNET y CAPNET, dos iniciativas que en el campo de agentes móviles se han propuesto para .NET.

#### *2.2.3.1. Aglets*

El paquete Aglets [Cle97] desarrollado por IBM está enfocado a la producción de agentes móviles. Estos agentes móviles se mueven alrededor de la red siguiendo un itinerario determinado, una ruta que obtendrán en el momento de la instanciación.

Un *aglet* es un objeto java móvil. La estructura de un *aglet* consiste en dos partes bien diferenciadas: el núcleo y el *proxy*. El núcleo es el corazón del *aglet* y contiene las variables y métodos internos proporcionando interfaces con que el *aglet* interactuará con su entorno. El núcleo está encapsulado en un *proxy*, que actúa como armadura para evitar intentos de acceder directamente a alguno de los métodos o variables privadas. Un *aglet* contiene además un identificador único y un itinerario (ver Figura 3).

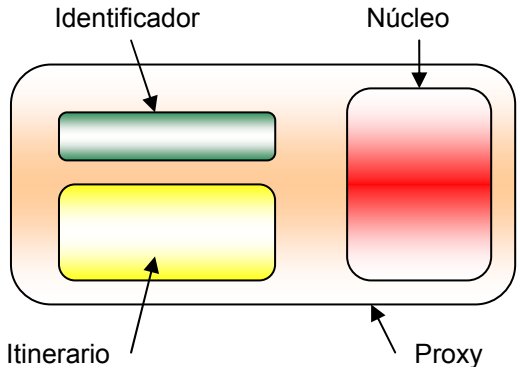



**Figura 3: Esquema de partes de un aglet**

Los *aglets* pueden migrar a máquinas que estén ejecutando un servidor compatible con la plataforma. Estos objetos servidor, denominados contextos, actúan como entornos en los que *aglets* de todo tipo se pueden comunicar. Un contexto es un objeto estacionario que proporciona mecanismos de hospedaje y mantenimiento de *aglets*. Una máquina puede tener una o varias instancias de contexto ejecutándose. Dentro de este contexto, un *aglet* es capaz de obtener información sobre su entorno y comunicarse con otros *aglets* activos próximos.

La comunicación entre *aglets* se realiza mediante mensajes a través del contexto, por lo que si un *aglet* se quiere comunicar con otro, deberá moverse al contexto en el que se encuentre. Esta comunicación puede ser síncrona o asíncrona. Los *aglets*, además, pueden suscribirse con determinados eventos y de esa forma se pueden realizar comunicaciones por *multiscast*.

Hay dos formas de crear un *aglet*: Puede ser creado mediante instanciación o por clonación, copiando un *aglet* ya existente. Otras acciones que se pueden aplicar sobre los *aglets* se listan a continuación: Los *aglets* pueden ser eliminados cuando ya hayan finalizado su funcionalidad, pueden ser parados temporalmente y por tanto reanudados, pueden ser despachados a otro contexto (local o remoto) o recuperados de un contexto en un momento determinado.

La movilidad en el entorno Aglets viene proporcionada por el *Agent Transfer Protocol* (ATP), protocolo desarrollado para Aglets aunque puede extenderse su funcionalidad, y por el *Java Agent Transfer and Communication Interface* (J-ATCI).

#### *2.2.3.2. Grasshopper*

Grasshopper [Bau00b] es una plataforma de desarrollo y ejecución de agentes móviles construida sobre un entorno de procesamiento distribuido. Esto conlleva a la integración del paradigma cliente-servidor con la tecnología de agentes móviles. Grasshopper está implementado en Java, basándose en la implementación Java 2. Es más, la plataforma está basada en el estándar *Mobile Agent System Interoperability Facility* (MASIF) de OMG. Además, se ha ampliado en su última versión para ser conforme con las especificaciones del *Foundation for Inteligent Physical Agents* (FIPA). De esta manera Grasshopper permite a sus usuarios un variado rango de agentes, desde agentes móviles simples que se desplazan a través de los nodos de una red, hasta sistemas estáticos multi-agente comunicándose de forma distribuida entre si para resolver problemas de forma conjunta.

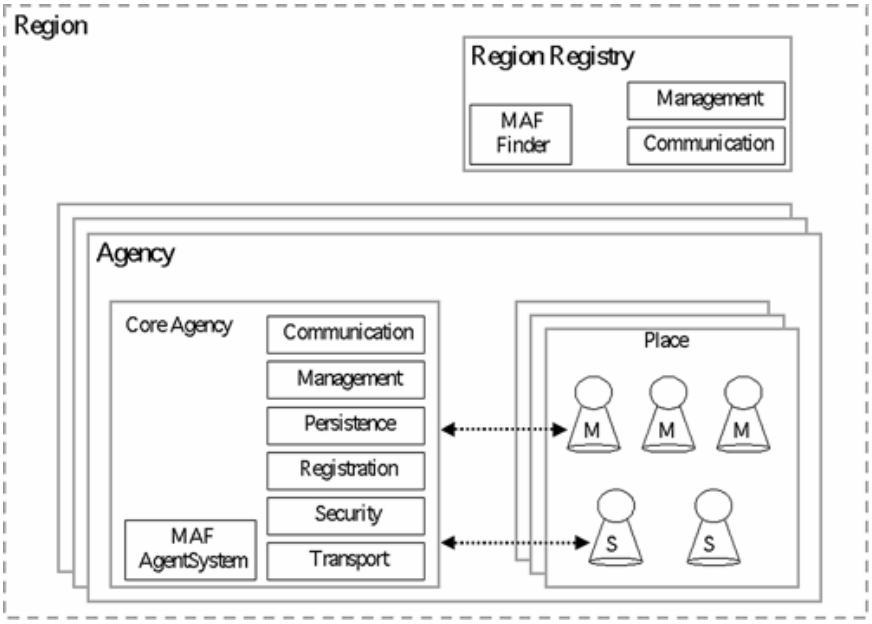



**Figura 4: entorno de agentes distribuidos de Grasshopper**

El Entorno de Agentes Distribuidos (DAE) de Grasshopper está compuesto por regiones, sitios, agencias y diferentes tipos de agentes (ver Figura 4). Un sitio proporciona una agrupación lógica de funcionalidad dentro de una agencia. La región facilita la gestión de los componentes distribuidos (agencias, sitios y agentes) en el entorno Grasshopper. Las agencias, así como sus sitios pueden ser asociados a una región específica, registrándose en el registro que acompaña a la región. Todos los agentes que pertenezcan a dicha agencia se registrarán automáticamente con dicho registro. Si un agente se

mueve a otra localización, la información del registro se actualizará adecuadamente.

En Grasshopper se distinguen dos tipos de agentes: Agentes móviles y agentes estacionarios. El entorno de ejecución de los agentes es la agencia. Una agencia en Grasshopper está formada por el núcleo de la agencia y uno o más sitios. El núcleo de una agencia representa la funcionalidad mínima requerida por una agencia para dar soporte a la ejecución de agentes. Para ello el núcleo proporciona los siguientes servicios:

- *Communications Service*: Servicio responsable de las interacciones remotas que tienen lugar entre los elementos distribuidos de Grasshopper. Proporciona un soporte multi-protocolo (RMI, IIOP y *Sockets*). Ejemplos de estas interacciones son la comunicación transparente, la migración, localización, etc. de agentes. Cabe destacar que la localización de los componentes es transparente en la comunicación, de forma que, para un agente no hay diferencia entre invocar métodos remotos e invocar métodos locales. Entre los diferentes modos de comunicación que se pueden llevar a cabo están la comunicación síncrona, asíncrona, dinámica, la cual supone construir el mensaje en tiempo de ejecución, y *multicast*.

- *Registration Service*: Cada agencia ha de tener información de todos los agentes y sitios que contiene. Además el registro de una agencia está en comunicación con el registro de la región a la que pertenece que contiene información sobre agencias, sitios y agentes en el ámbito de la región.

- *Management Service*: Permite el monitorizado y control de agentes y sitios de una agencia.

- *Security Service*: Grasshopper soporta dos mecanismos de seguridad.

  ■ **Seguridad externa**: Protege las interacciones entre componentes remotos, por ejemplo, la comunicación entre una agencia y la registro de la región. Para este propósito se utilizan los certificados X.509 y SSL, proporcionando confidencialidad, integridad y autentificación.

  ■ **Seguridad interna**: Protege los recursos de una agencia de acceso no autorizado por parte de sus agentes. También se usa para proteger a los agentes los unos de los otros. En este caso se usan los mecanismos de seguridad de Java.

- *Persistence Service*: El servicio de persistencia activa el almacenamiento de agentes y sitios en un medio persistente, para

de esa forma poder recuperarlos en caso de necesidad, por ejemplo cuando la agencia es reiniciada después de un fallo del sistema.

#### *2.2.3.3. ProActive*

ProActive [Bau00] es una libreria Java para computación concurrente y distribuida cuyas principales características son transparencia en las comunicaciones remotas, comunicaciones bilaterales asíncronas y mecanismos de sincronización de alto nivel.

Las aplicaciones concurrentes y distribuidas construidas usando ProActive están compuestas por un determinado número de entidades de granularidad media llamadas objetos activos. Cada objeto activo tiene su propio hilo de ejecución y tiene la habilidad de decidir en que orden servirá las peticiones de servicio entrantes.

Respecto a la migración, cuando un objeto activo que se ha de mover tiene referencias a objetos pasivos, éstos se moverán con él, es decir, no migra únicamente el objeto activo, sino todo el subsistema que conforma. Para la movilidad, ProActive proporciona una movilidad débil (*weak mobility*). Existen dos formas de mover un objeto con ProActive:

La primera consiste en migrar a un *host* remoto, para lo cual el *host* tendrá que tener ejecutando un objeto Java llamado *Node*. El *Node* es un *daemon* que se encarga de recibir, reanudar y mantener una traza de los objetos activos locales.

```
public void moveToHost(String t)
{
            ProActive.migrateTo(t);
}
```

La otra forma de migrar un objeto es anexarlo a otro objeto remoto, aunque la dirección de dicho objeto no se conozca: Tan solo hace falta una referencia a dicho objeto remoto. No obstante no está garantizado que tras la migración ambos objetos terminen en el mismo *host*, puesto que el objeto referenciado puede haber migrado.

```
public void joinFriend(Object friend)
{
            ProActive.migrateTo(friend);
}
```

Una de las características más relevantes del ProActive es que proporciona la capacidad de que un objeto activo pueda realizar acciones diferentes en diferentes *hosts*. Además la lista de *hosts* que ha de visitar un objeto activo (con las acciones que ha de realizar en él) puede variar dinámicamente. Para conseguir esto se han empleado técnicas de reflexión, lo

cual proporciona mucha versatilidad a la hora de elegir el método que se ejecutará al llegar al destino.

Para hacer esto, ProActive define una serie de eventos para lanzarlos cuando se produzcan acciones de movilidad. El siguiente evento se define para ejecutarlo cuando un objeto activo llega a destino. Con el se definirá cual será el método que ejecutará al finalizar la movilidad:

```
ProActive.onArrival(String)
```

La cadena de texto será el nombre del método a ejecutar. Los métodos que podrán ser invocados por este método tienen dos limitaciones: no pueden devolver nada ni recibir parámetros. Esto es comprensible, ya que no tiene sentido que nadie espere una respuesta de un método de este tipo. Sobre el uso de parámetros tampoco hay que tener en cuenta que el método se ejecutará en el destino, y puede haber un *delay* entre el momento en que el método es puesto y cuando se efectuará realmente la llamada, por lo que los valores de los parámetros no son seguros. No obstante este método puede contener en su interior llamadas a métodos normales.

Otro evento interesante es el correspondiente a la partida de un objeto activo. Con este evento se indicará cual será el último método que ejecutará el objeto activo antes de iniciar el proceso de movilidad:

```
ProActive.onDepature(String)
```

Este método resultará de utilidad por ejemplo para liberar recursos utilizados por el objeto activo que se ha de mover.

Otro de los aspectos a tener en cuenta de la implementación de las características de comunicación, es el mecanismo para garantizar que un objeto activo móvil siempre sea alcanzable, incluso si nunca para de moverse de un *host* a otro. Esto ha sido solucionado mediante una cadena de *forwarders*, para lo cual no es necesario saber la localización real de un objeto para comunicarse con él. Para conseguir esto, se construye una cadena de referencias, donde cada elemento será un *forwarder*, esto es, un objeto dejado por el objeto móvil antes de moverse que contiene una referencia a la siguiente localización del objeto móvil. Cuando un mensaje es enviado, sigue la cadena de *forwarders* hasta que llegue a la localización donde se encuentra realmente el objeto al que va dirigido. Por motivos de eficiencia, en ProActive esto ha sido implementado haciendo que en el momento que un objeto activo abandona un *host* se convierta en realidad en un *forwarder*. De esta forma el mecanismo es transparente al llamador, pues el sigue comunicándose con el mismo objeto. Además, este mecanismo permite que respuestas asíncronas puedan ser enviadas a objetos activos sin que estos limiten sus características de movilidad mientras se encuentran en estado de espera.

La comunicación distribuida transparente que se consigue con los *forwarders* también presenta problemas. Por ejemplo, si algún elemento de la

cadena resulta temporal o permanentemente inalcanzable, aislará a todos los objetos activos que tengan *forwarders* en ella. Para reducir la posibilidad de fallo se aplica una técnica para mantener la longitud de la cadena al mínimo. Bajo el principio "cuando más corta la cadena, mejor", ProActive implementa un mecanismo mediante el cual, cuando un objeto activo recibe una petición remota, le enviará su localización actual al llamador si, y solo si, éste no la conoce. Esta información se encapsulará en el propio mensaje, así se evita enviar información innecesaria.

#### *2.2.3.4. MAPNET*

MAPNET [Sta04] es una plataforma de agentes móviles para el *framework* .Net basada en MASIF. El objetivo de la plataforma es dar un soporte para la programación con agentes móviles haciendo uso de las facilidades proporcionadas por el *framework* .Net para la programación distribuida. La plataforma MAPNET es una librería de tipos escrita en C#.

Un agente MAPNET se implementa como un descendiente de la clase abstracta `Agent` (ver Figura 5). Dichos descendientes estarán marcados con el atributo `[Serializable]` para activar la posibilidad de migración. La clase abstracta `Agent` también hereda de `MarshalByRefObject` para permitir el acceso remoto al agente.

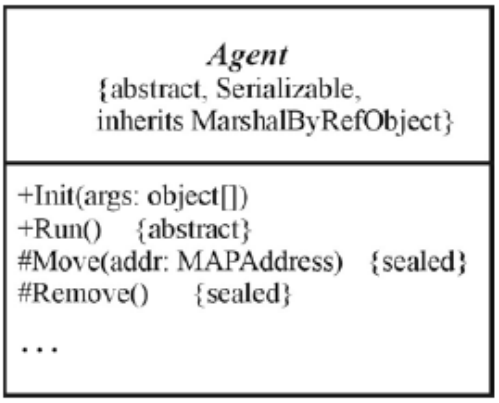



**Figura 5: Clase Agent**

La migración es iniciada por el propio agente haciendo uso del método `Move()`. Una vez finalizada la operación de movilidad, se iniciará el proceso de destrucción del agente con el método `Remove()`. La movilidad viene representada por la creación del agente en el destino y, tras la serialización del estado, la destrucción del agente en el origen. MAPNET proporciona una movilidad débil (*weak mobility*), en el sentido que tan solo mueve el estado del agente y no su hilo de ejecución. Los métodos `Move()` y `Remove()` no pueden ser reescritos en las clases descendientes.

El método `Run()` es el único método abstracto de la clase. En él se implementará el funcionamiento del agente, y por ser su labor tan específica

del dominio, será descrito por el desarrollador en las clases descendientes de la clase abstracta `Agent`.

Otros métodos de la clase pueden ser reescritos opcionalmente en las clases descendientes, como por ejemplo el método `Init()`, que es usado en los agentes en lugar del constructor. Este método es invocado por el Servidor de Agentes tras la creación del agente. Otros métodos definidos en la clase son `BeforeMove()` y `AfterMove()`, llamados respectivamente, inmediatamente antes de la movilidad o después de ella. `BeforeRemove()` es un método que se invocará antes de la llamada al `Remove()` del agente. Tanto `BeforeRemove()` como `BeforeMove()` podrán ser reescritos por el desarrollador para liberar los recursos consumidos por el agente.

Cada agente MAPNET tiene asignado un perfil (*profile*), representado por la clase `AgentProfile`, en el que se incluirá información como el nombre del agente, su estado, el nombre del Agent Server en el que se ha creado, así como en el que se encuentra y el último que ha visitado, etc. El agente móvil lleva consigo este *profile* durante la migración.

Los agentes MAPNET son ejecutados en el contexto de un servidor de agentes (*Agent Server*) [Sta04b]. Una o más instancias del servidor de agentes han de estar ejecutándose en las máquinas que quieran recibir agentes. Algunas de las tareas del servidor de agentes son:

- **Gestión de agentes:** El servidor de agentes es capaz de crear, terminar, suspender y transferir agentes.

- **Nombrado de agentes:** Obtener el nombre único global a los nuevos agentes siguiendo la nomenclatura propuesta por MASIF.

- **Gestión de los hilos de ejecución:** Asignación de hilos de ejecución para que los agentes desempeñen sus acciones. Para la asignación de hilos, el servidor usa el patrón *Thread Pool*, mediante el cual se generan un número de hilos de ejecución previamente a los que luego se les asignan las tareas por orden de llegada. De esa forma se evita la sobrecarga producida por la creación y destrucción de hilos.

- **Registro de agentes:** Servicio para registrar los perfiles de los agentes. Proporciona métodos para buscar agentes locales.

- **Comunicación remota entre agentes:** Servicios para obtención de *proxies* a agentes remotos.

- **Seguridad:** El servidor ha de proporcionar entornos de comunicación segura para los agentes.

El servidor proporciona también una serie de eventos, como `AgentCreatedEvent`, `AgentRemovedEvent` y `AgentMovedEvent` para notificar a los suscriptores de lo que está ocurriendo en el servidor.

Al igual que los agentes, los servidores de agentes tienen asociado un *profile* en el que se encuentra información relevante sobre ellos.

Cuando un agente solicita ser movido a otro *host*, lo hace a través del Servidor de Agentes del *host* en el que se encuentra. En el Servidor de Agentes origen, la migración de un agente supone los siguientes pasos:

1. Terminar la ejecución del agente y devolver su hilo de ejecución al gestor de *Threading*, para que le pueda ser asignado a otro agente.
2. Convertir el estado en un `ByteStream`
3. Se delega la transferencia al servicio de comunicación

El servidor de agentes destino realizará una secuencia de pasos simétrica:

1. El servicio de comunicación recibe el agente
2. Deserializará su estado
3. Se le asignara un hilo de ejecución al agente para que realice las acciones definidas en su `Run()`

La comunicación entre agentes es proporcionada por el Servicio de Comunicación del servidor de agentes, que será el encargado de proporcionar el *proxy* al agente deseado y devolverlo al agente que se lo ha solicitado. Si por ejemplo un agente A quiere llamar al método M de una agente B que reside en la localización *hostB* del que A solo conoce su nombre. Desde el punto de vista del desarrollador la invocación se realiza de la siguiente forma:

```
AgentB b = (AgentB) this.AgentSystem.GetAgent(hostB, nameB);
Result = b.M();
```

La tecnología empleada en MAPNET tanto para la comunicación remota entre agentes como para la migración es .Net Remoting. En concreto se usa el canal HTTP, que proporciona más flexibilidad que el TCP, junto a un formateador binario, más rápido que SOAP.

#### *2.2.3.5. CAPNET*

CAPNET (*Component Agent Platform for .NET*) [Con04] es otra plataforma de agentes para el *framework* .Net y el *Compact Framework* [Bur03] escrita enteramente en C#. CAPNET está basado en el modelo de plataformas de agentes de la especificación FIPA.

El principal objetivo de CAPNET es obtener una estructura integrada que cubra la programación, despliegue, administración e integración con aplicaciones legadas de sistemas Multi-Agente. Por tanto, consiste de un entorno de ejecución que permite el despliegue de sistemas multiagente, un entorno de desarrollo en forma de plantillas de agentes, herramientas de programación, una galería de componentes y algunos conectores que permitan la integración con *Enterprise Services* [Pa02] (ver Figura 6).


**Figura 6: Entorno de ejecución de CAPNET**

El modelo de referencia FIPA considera la plataforma de agentes como un conjunto de cuatro componentes (ver Figura 7):

- **Agentes (*Agents*)**: Parte principal de la plataforma. Encapsula uno o más servicios dentro de un modelo de ejecución unificado e integrado.

- **Directorio (*Directory facilitator*)**: Es un agente que proporciona un servicio de páginas amarillas[1].

- **Sistema de gestión de agentes (*Agent Management System*)**: Agente especial que proporciona servicio de páginas blancas y de ciclo de vida, así como mantiene un directorio de identificadores de los agentes que contiene y sus respectivos estados.

- **Sistema de transporte de mensajes (*Message Transport System*)**: Proporciona el sistema de entrega de mensajes

[1] Un servicio de páginas amarillas (*yellow-pages*) es un directorio cliente-servidor para sistemas distribuidos donde se almacena información centralizada de usuarios, máquinas u otra información relevante.

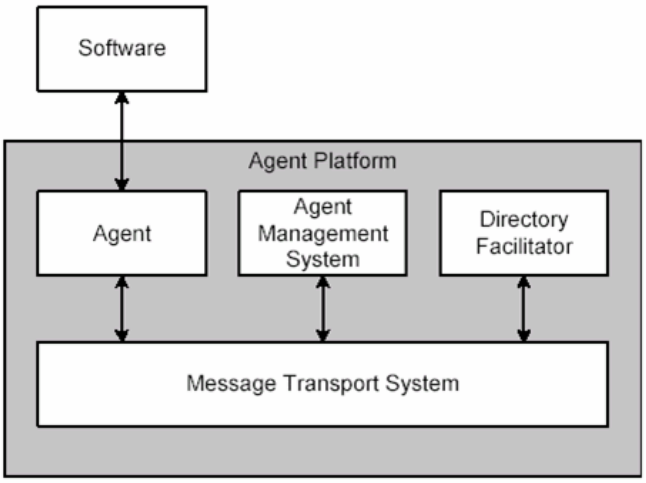



**Figura 7: Componentes FIPA de una plataforma de agentes**

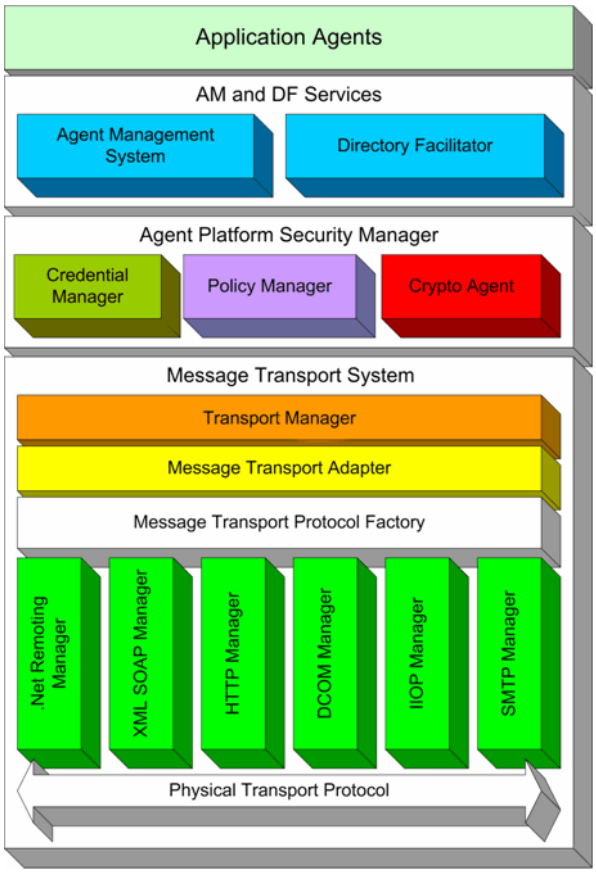



**Figura 8: Arquitectura de CAPNET**

La idea fundamental de CAPNET es permitir a los desarrolladores crear e integrar aplicaciones de agentes distribuidas de una forma consistente y escalable. Como se muestra en la Figura 8, la arquitectura de CAPNET está formada por cuatro grandes bloques: *Application Agents*, *AM and DF Services*, *Agent Platform Security Manager* y *Message Transaport System*.

El *Message Transport System* (MTS) es el núcleo de la plataforma y soporta la comunicación mediante varios tipos de transporte. Esta variedad asegura la interoperabilidad entre otras plataformas de agentes. Cuando la comunicación se hace entra diferentes MTS, aparece la figura del *Massage Transaport Adapter* (MTA) que será el encargado de determinar el tipo de transporte del mensaje que es necesario para completar la operación y así usar la infraestructura apropiada. El MTA usa los servicios del *Message Transport Protocol Factor*y para instanciar el componente que proporcionará el acceso a la capa física. Este será en realidad el componente *Transport Manager*.

Para garantizar la fiabilidad del MTS se han implementado una serie de mecanismos adicionales. Cuando un agente envía un mensaje a través del MTS , el sabrá si el MTS ha entregado el mensaje al destino. Si esto no ocurre, el agente emisor recibirá una notificación de este hecho para que pueda tomar las medidas oportunas.

## 2.3. Una descripción formal para la movilidad

En este apartado se va a explicar como una algebra de procesos llamado Cálculo de Ambientes trata la movilidad. A diferencia de las aproximaciones de movilidad presentadas en las previas secciones, el Cálculo de Ambientes describe la movilidad de forma abstracta e independiente de plataforma. Este apartado es relevante en el desarrollo del presente proyecto, pues gran parte de él se basa en el formalismo que se van a describir a continuación. En esta sección se explicarán las características más relevantes del Cálculo de Ambientes, así como se hará un breve repaso por sus propuestas de implementación.

### 2.3.1. El Cálculo de Ambientes

El Cálculo de Ambientes (*Ambient Calculus*)[Car98] es una álgebra de procesos que extiende el π-cálculo [Mil91] para introducir el concepto de ambiente. Un ambiente es un lugar delimitado en donde se ejecutan una serie de procesos. Por lo tanto, un ambiente puede ser cualquier cosa con una frontera delimitada como un ordenador portátil, una página web, una carpeta del sistema de ficheros, etc. Además, un ambiente puede contener a otros sub-ambientes. Cada ambiente tiene un conjunto de primitivas que permiten gestionarlo. Dichas primitivas son las que permiten la movilidad de ambientes.

En el Cálculo de Ambientes, los ambientes son las entidades móviles. Dicha movilidad se realiza cruzando las fronteras de los ambientes. El ambiente no sólo proporciona primitivas de movilidad, también ofrece primitivas para la comunicación local. Estas primitivas se pueden expresar con una sintaxis textual o gráfica, llamada Cálculo de Carpetas (Folder Calculus) [Lo02]. El cálculo de carpetas es una metáfora gráfica asociada al Cálculo de Ambientes donde los ambientes se representan como carpetas. En la Figura 9

se muestra una tabla con las relaciones entre la sintaxis visual y la sintaxis gráfica.

| Sintaxis Textual | Sintaxis Visual | Comentarios | Sintaxis Textual | Sintaxis Visual | Comentarios |
|---|---|---|---|---|---|
| $(vn)P$ |  | Nuevo nombre *n* en un alcance *P* | 0 |  | Proceso Inactivo |
| | | | $!P$ |  | Replicación de *P* |
| $n[P]$ |  | Carpeta (ambiente) con nombre *n* y contenidos *P*. | $(M)$ |  | Salida *M* |
| $M.P$ |  | Acción *M* seguido por *P* | $(n).P$ |  | Entrada de *n* seguida por *P* |
| $P \mid Q$ |  | Dos procesos en paralelo | $(P)$ |  | Agrupación |

**Figura 9: Sintaxis textual y gráfica del cálculo de ambientes**

Algunas construcciones que utiliza el Cálculo de Ambientes son heredadas del π-Cálculo tales como el nombramiento, la restricción, la composición paralela, el proceso inactivo y la replicación. En la Figura 9 se muestra que la replicación de un proceso en el cálculo de carpetas se representa mediante una fotocopiadora con el proceso que réplica encima.

Sin embargo, a diferencia del π-Cálculo, los nombres del Cálculo de Ambientes son nombres de ambientes en lugar de nombres de canales. Por lo tanto, para describir que un ambiente con el nombre `n` contiene un proceso `P`, se especifica como `n[P]`.

Algunas de las primitivas que proporciona el Cálculo de Ambientes son conocidas como capacidades (*capabilities*). Las capacidades son las acciones que se pueden realizar a los ambientes. Existen tres capacidades principales: `enter`, `exit` y `open`. La capacidad `enter` solicita al ambiente que entre en otro ambiente de su mismo nivel jerárquico (ver Figura 10). La capacidad `exit` solicita al ambiente que salga de su ambiente padre (ver Figura 11). La capacidad `open` disuelve un ambiente dejando libres los procesos que estaban dentro de él (ver Figura 12).

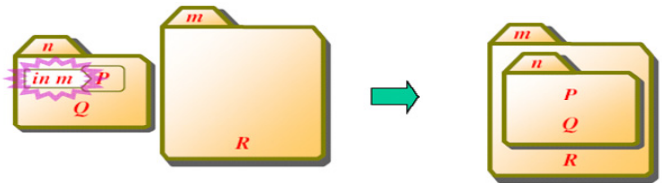



**Figura 10: Aplicando la capability enter al ambiente n**

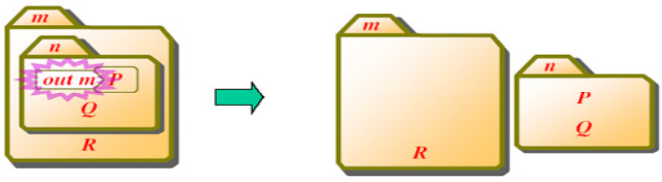



**Figura 11: Aplicando la capability exit al ambiente m**


**Figura 12: Aplicando la capability open al ambiente n**

### 2.3.2. Implementaciones del Cálculo de Ambientes

Son pocas las implementaciones existentes del Cálculo de Ambientes hasta la fecha, no obstante, en la presente sección se resumirán las propuestas que se han localizado para este propósito.

En [Ca97] Cardelli propone un algoritomo para implementar el Cálculo de Ambientes en Java. La implementación describe una estructura de árbol donde cada nodo es un ambiente. La estructura de árbol se modifica mediante operaciones que implementan las *capabilities enter* y *exit*. No obstante esta implementación tan solo pretende mostrar el comportamiento de la estructura de ambientes frente a la ejecución de *capabilities*, y en ningún caso es una implementación completa que considere la interacción entre nodos distribuidos, dejando así muy limitada la propuesta.

Propuestas que den soporte a sistemas software distribuido tan solo se ha encontrado en [Va04], en la que se presenta un esquema para implementar las primitivas del Cálculo de Ambientes en C# sobre el *framework* para el soporte de programación paralela ARC (*Anonimous Remote Computing*) que está construido sobre .NET.

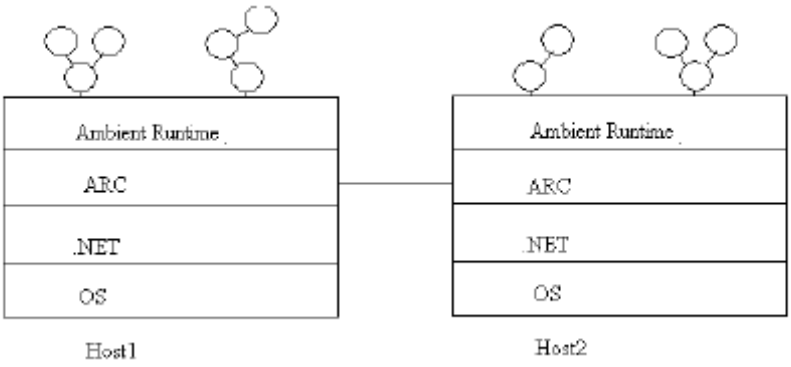



**Figura 13:Arquitectura de la ejecución de ambientes sobre ARC**

ARC es un *framework* orientado a servicios desarrollado en C# sobre .Net que soporta computaciones paralelas y distribuidas con características de movilidad. La idea de la aproximación es explotar las características de movilidad del *framework* ARC para implementar una versión distribuida del cálculo de ambientes. En la Figura 13 se muestra la arquitectura propuesta.

Como se puede observar, el *ambient runtime* actúa como la capa superior en cada nodo, lo que implícitamente la convierte en el padre de todos los ambientes que se encuentren en dicho nodo. El *ambient runtime* es quien decide cuando un ambiente necesita ser movido a un nodo remoto en función de las llamadas y la disponibilidad de la *capability*, es decir, si se cumplen las condiciones para que la *capability* se pueda ejecutar. Si el ambiente va a ser migrado, el *ambient runtime* hace uso del *framework* ARC para transportar al ambiente al *host* destino.

Para la implementación del cálculo de ambientes en C# se ha usado la jerarquía de clases mostrada en la Figura 14. La implementación está basada en el patrón de diseño *composite* [2]de forma que permite componer ambientes con procesos. La clase abstracta `AbstractAmbient` está implementada en las dos clases descendientes, es decir, como un proceso o como un ambiente que puede contener en su interior otros ambientes o procesos.

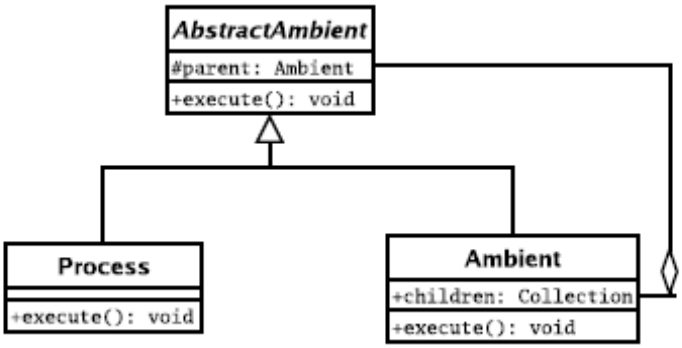



**Figura 14: Jerarquía de la clase Ambient**

Como se puede ver todo ambiente tiene un padre (`parent`) que por defecto es el ambiente que representa el entorno de ejecución ARC. El método `execute()` se implementa en las clases hijas: En el caso de la subclase `Ambient`, la petición es reenviada a todos los elementos hijos, mientras que para la subclase `Process`, el método alberga el cuerpo del proceso, que es implementado usando clases definidas por el usuario usando la herencia de la clase `Process`. Tan pronto como se crea un `Process`, en el momento que su ambiente padre es asignado, la ejecución de su lógica se pone en funcionamiento en un hilo nuevo.

De esta forma, se propone el esquema de implementación de los conceptos del Cálculo de Ambientes haciendo uso del *framework* ARC. No obstante, hasta la fecha no se ha publicado un nuevo trabajo con los avances obtenidos siguiendo esta propuesta.

[2] El patrón *composite* [Gamma95] compone objetos en estructuras de árbol para representar jerarquías parte-todo. El patrón permite a los clientes tratar con objetos individuales o composiciones de objetos uniformemente.

## 2.4. Resumen

En esta sección se ha dado una amplia visión sobre las diferentes aproximaciones que tratan la comunicación distribuida y la movilidad de código. Partiendo de la presentación de los entornos distribuidos orientados a objetos más comunes (CORBA y .NET Remoting) se ha mostrado la necesidad de establecer un paradigma más expresivo que el orientado a objetos para poder representar requisitos no funcionales, con lo que se ha dado paso a la programación orientada a aspectos. La explicación de las peculiaridades de este paradigma de programación, junto a sus primitivas esenciales, se han mostrado a través de su lenguaje más representativo, AspectJ [AsJ03][AsJ04]. Esta explicación es necesaria para la comprensión de los conceptos relacionados con el Desarrollo de Software Orientado a Aspectos a los que se recurrirá continuamente a lo largo del proyecto.

Otra de las secciones trata el uso de aspectos de distribución, que es a la postre uno de los objetivos del proyecto. Así se introducen las propuestas basadas en de separación de los conceptos de distribución (*framework* D [Lo97] ) y el uso de un patrón de generación de código (PaDA [Soa02]) para el *concern* de distribución. Esto contrasta con la separación del concerns de distribución y movilidad realizada en Ambient-PRISMA, en la que se representan por separado los concerns de distribución de los de movilidad, encargándose los primeros de la ubicación del elemento arquitectónico y los segundos de albergar la lógica para modificar dicho *concern*. De esa forma, se establece una separación entre la representación de la ubicación (Aspecto de Distribución) y los mecanismos empleados para modificarlos (Aspecto de Movilidad).

A continuación se han introducido los conceptos relacionados con la movilidad y en concreto la diferencia entre *strong mobility* y *weak mobility*. Esto es importante puesto que la movilidad, tal y como se va a implementar en el presente proyecto va a proporcionar *weak mobility*. No obstante se justifica cuales són las dificultades de implementar *strong mobility* en .NET. A continuación se han mostrado diversas aproximaciones que tratan la movilidad con *weak mobility*, MobJex[Rya04] y JACOb [Cha03], subrayando el hecho que no permiten la movilidad activa, es decir auto invocada por el objeto móvil. De todas formas es interesante remarcar el mecanismo de invocación remota que proporciona JACOb mediante el uso de un método genérico, ya que está relacionada con la solución que se propone para la comunicación distribuida en Ambient-PRISMA.

La enumeración de algunas de las plataformas de agentes móviles pretende establecer una comparación entre las técnicas utilizadas para dotar la movilidad autoinducida a los objetos móviles y las utilizadas en PRISMA para proporcionar esta movilidad a sus elementos arquitectónicos. También se mencionan las aproximaciones para la tecnología .NET, por ser sus implementaciones más cercanas a los que se va a presentar en el proyecto.

Finalmente se introduce el Cálculo de Ambientes [Ca97], el formalismo para la movilidad propuesto por Luca Cardelli. La introducción de esta álgebra de procesos es fundamental para la comprensión de las novedades que aporta el Ambient-PRISMA al modelo PRISMA. Por ello se da una breve descripción del concepto de ambiente y de las *capabilities*. Finalmente se repasan las implementaciones existentes que modelen la movilidad haciendo uso de ese formalismo.

CAPÍTULO III

# PRISMA

## Contenidos del capítulo

# PRISMA

En la actualidad PRISMA [Pe05] es un marco de trabajo o *framework* que incluye un modelo, unos lenguajes, una herramienta y una metodología. PRISMA se presenta como un enfoque integrado y flexible para describir modelos de arquitectura complejos, distribuidos, evolutivos y reutilizables. Para ello, se basa en componentes y aspectos para la construcción de modelos arquitectónicos y se caracteriza por la integración que realiza del Desarrollo de Software Basado en componentes (DSBC), del Desarrollo de Software Orientado a Aspectos (DSOA) y por sus propiedades reflexivas. Además, se ha desarrollado un *middleware* abstracto que se ejecuta en un nivel superior sobre la plataforma .Net que permite la implementación y ejecución de aplicaciones PRISMA llamado PRISMANET.

PRISMA define una metodología guiada y (semi-)automatizada para obtener el producto software final basándose en el paradigma de la prototipación automática de Balzer [Bal85]. En esta metodología, el usuario utilizará una herramienta de modelado (compilador de modelos) que le permitirá validar el modelo mediante la animación de los modelos arquitectónicos definidos en la especificación. PRISMA está basado en un lenguaje de especificación formal Orientado a Objetos llamado OASIS [Let98]. OASIS es un lenguaje formal para definir modelos conceptuales de sistemas de información orientados a objetos, que permite validar y generar las aplicaciones automáticamente a partir de la información capturada en los modelos. PRISMA preserva las ventajas de OASIS, garantizando la compilación de sus modelos, y lo extiende, para poder definir formalmente la semántica de los modelos arquitectónicos orientados a aspectos, ya que OASIS únicamente es capaz de especificar sistemas de información orientados a objetos.

Por otro lado, PRISMANET proporciona un soporte para los diferentes conceptos de PRISMA, así como su metanivel y características de movilidad y comunicación distribuida para sus arquitecturas. Este *middleware* se ha desarrollado haciendo uso de los mecanismos estándar de .Net, es decir, sin necesidad de ampliar la plataforma de desarrollo, aumentando de esa forma su compatibilidad. PRISMANET ofrece funcionalidades y características extra que no ofrece directamente .NET, como por ejemplo la ejecución de aspectos, la reconfiguración de arquitecturas software locales y distribuidas, la gestión de componentes móviles, etc.

En el presente capítulo de presentará el modelo PRISMA junto con su implementación previa, PRISMANET, con la finalidad de poner al lector en antecedentes y, establecer el punto de partida del proyecto.

La estructura del capitulo es la siguiente: En la sección 3.2 se presenta el modelo PRISMA, describiendo cuales son sus ciudadanos de primer orden. En la sección 3.3 se mostrará el Lenguaje de Definición de Arquitecturas Orientado a Aspectos (LDA-OA) que. Finalmente en la sección 3.4 se presentará brevemente el *middleware* PRISMANET subrayando la parte relacionada con la movilidad de elementos arquitectónicos y su comunicación distribuida.

.

## 3.1. El Modelo PRISMA

PRISMA [Pe05] define un modelo para describir arquitecturas software de sistemas complejos mediante la integración de dos aproximaciones de desarrollo de software: El Desarrollo Software Basado en componentes (DSBC) [Szy02] y el Desarrollo Software Orientado a Aspectos (DSOA) [Aosd] Esta integración se consigue definiendo el comportamiento de los elementos arquitectónicos mediante un conjunto de aspectos, de manera que, cada aspecto define el comportamiento de un elemento arquitectónico desde un punto de vista (o *concern*). La mayoría de los modelos arquitectónicos analizan cuáles son las primitivas básicas para la especificación de sus arquitecturas, esto es, la descripción de la estructura y la interacción de los elementos arquitectónicos, y exponen su sintaxis y semántica. El modelo PRISMA, además de definir los elementos arquitectónicos y especificar su sintaxis y semántica, también especifica los aspectos que cubren las necesidades de cada uno de ellos.

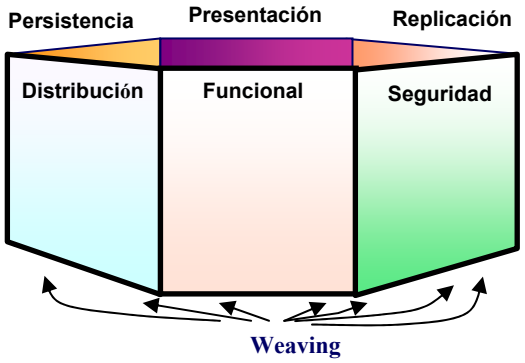



**Figura 15: Vista interna de un elemento arquitectónico PRISMA**

Un elemento arquitectónico PRISMA puede ser observado desde dos puntos de vista: un punto de vista interno y otro externo. La vista interna (Figura 15) define un elemento arquitectónico como un prisma con tantas caras como aspectos necesite para expresar su comportamiento. Dichos aspectos están definidos desde la perspectiva del problema y no de su solución, aumentando el nivel de abstracción y evitando el solapamiento de código que

puede sufrir la programación orientada a objetos. La interacción entre los diferentes aspectos viene dada por el concepto de *weaving* del DSOA.

Por otro lado, la vista externa de un elemento se puede considerar como una caja negra que encapsula el comportamiento y publica mediante una serie de puertos las interfaces de los servicios que ofrece y requiere al resto de elementos arquitectónicos (Figura 16).

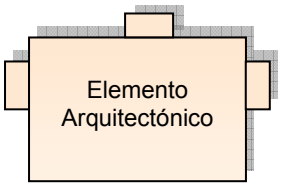



**Figura 16: Vista externa de un elemento arquitectónico PRISMA**

Por tanto, las dos vistas mencionadas se relacionan con cada una de las aproximaciones de desarrollo en las que se basa PRISMA: La vista interna con la orientada a aspectos y la vista externa con la orientada a componentes. A continuación se presentará el modelo desde ambas perspectivas.

### 3.1.1. Visión Orientada a Aspectos

Las técnicas de la Programación Orientada a Aspectos, permiten encapsular aquella funcionalidad que se repite a lo largo de todo el sistema en una entidad llamada aspecto. En el modelo PRISMA un aspecto específica la semántica de la estructura (atributos) y el comportamiento (servicios) de la arquitectura del sistema desde una vista específica (vista funcional, distribución, calidad, etc.). Por ello, cada primitiva de la arquitectura está formada a su vez por varios aspectos que la describen desde diferentes puntos de vista. Algunos tipos de aspectos pueden ser:

- **Aspecto Funcional:** Captura la semántica del sistema de información mediante la definición de sus atributos, sus servicios y su comportamiento

- **Aspecto de Coordinación:** Define la sincronización entre elementos arquitectónicos durante su comunicación.

- **Aspecto de Distribución:** Especifica las características que definen la localización dinámica del elemento arquitectónico en el cual se integra. También define los servicios necesarios para implementar estrategias de distribución de los elementos arquitectónicos (como movilidad, equilibrio de carga, etc.) con el objetivo de optimizar la distribución de la topología del sistema resultante [Ali03].

Existen más tipos de aspectos, pero tan solo se han nombrado los tres anteriores debido al hecho que son los más importantes, como se verá en las siguientes secciones. Es más, el número de tipos no está limitado, puesto que es posible definir nuevos gracias al metanivel: los diferentes tipos de aspectos emergerán de los requisitos de los sistemas a los que se aplique el modelo PRISMA, siendo por tanto estos dependientes del dominio del problema.

No obstante, no es suficiente con definir únicamente los tipos de aspectos, sino que también hay que enlazarlos entre sí: esto se realiza mediante los *weavings*. Los *weavings* indican que la ejecución de un servicio de un aspecto puede provocar la invocación de servicios en otros aspectos. Son una manera de entretejer los *crosscutting concerns* que se localicen en la definición de los aspectos.

A diferencia de otras tecnologías orientadas a aspectos en las que los aspectos se enlazaban al código base, en PRISMA no existe el código base como tal, sino que la funcionalidad global de una primitiva del modelo arquitectónico se define mediante la unión de los distintos aspectos que la forman. Otra diferencia que resulta importante remarcar, es que en las otras tecnologías (como es el caso de AspectJ [AsJ04]) los *pointcuts* se definen en el mismo aspecto, resultando esta práctica en una pérdida de reutilización por parte del aspecto. Es por esto que en PRISMA los *weavings* entre los aspectos se definen fuera del aspecto, en el elemento que los integra, esto es, en el elemento arquitectónico PRISMA.

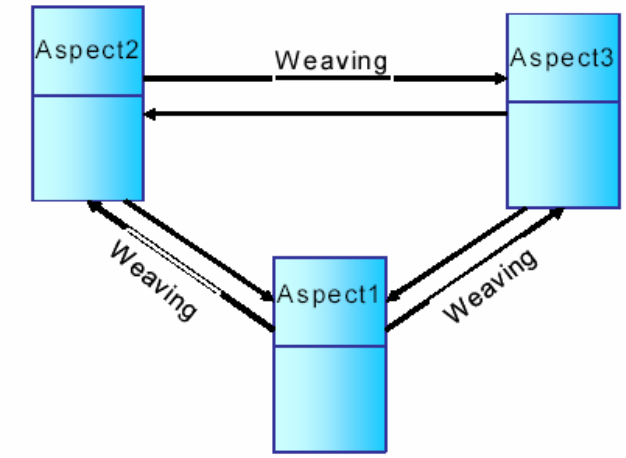



**Figura 17: Weavings entre aspectos**

Como se puede observar en la Figura 17, los aspectos se definen independientemente del elemento arquitectónico donde se vayan a integrar, y son los *weavings* los que unen el conjunto de aspectos de un mismo elemento arquitectónico y forman la vista interna tal y como se ha mostrado anteriormente. De la misma forma que en otras tecnologías orientadas a aspectos, PRISMA dispone de varios tipos de *weavings*:

- **After:** Cuando se especifique *aspect1.service after aspect2.service* se está diciendo que el servicio *aspect1.service* será ejecutado después

de que termine *aspect2.service*, actuando éste último como el disparador del *weaving*.

- **AfterIf (condición):** igual que After, sólo si se cumple la condición.

- **Before: :** Cuando se especifique *aspect1.service before aspect2.service* se está diciendo que *aspect1.service* será ejecutado antes de *aspect2.service* cuando este último vaya a ejecutarse.

- **BeforeIf (condición):** igual que Before, sólo si se cumple la condición.

- **Instead:** Cuando se especifique *aspect1.service instead aspect2.service* se está diciendo que *aspect1.service* será ejecutado en lugar de *aspect2.service* cuando éste último vaya a ejecutarse.

De esta forma es posible sincronizar la ejecución de servicios de diferentes aspectos que conforman un elemento arquitectónico expresando de esta manera las relaciones existentes entre los aspectos. Por ejemplo, se puede pensar en la ejecución de un servicio que muestre el dinero de una cuanta bancaria en el aspecto de presentación tras modificarlo mediante una operación del aspecto funcional: el servicio del aspecto de presentación se ejecutaría después de la ejecución del servicio del aspecto funcional. Además, al estar los *weavings* definidos a nivel del elemento arquitectónico que encapsula los aspectos, se favorece la reutilización de los aspectos, ya que no dependen de aquellos a los que están entretejidos.

El hecho que los aspectos PRISMA sean altamente reutilizables favorece la mantenibilidad, pues un cambio en un *concern* específico (por ejemplo el cambio en la estrategia de distribución de elementos arquitectónicos) sólo ha de realizarse en un tipo de aspecto y no en todo el conjunto del sistema.

### 3.1.2. Visión Basada en Componentes

La visión externa de los elementos arquitectónicos está orientada hacia la aproximación basada en componentes, a diferencia de la orientación a aspectos de la visión interna. El modelo arquitectónico PRISMA consta de tres tipos de elementos arquitectónicos: componentes, conectores y sistemas. Como se ha visto anteriormente, cada uno de estos tipos a su vez está compuesto por tantos aspectos como se consideren relevantes para definir el sistema de información, no obstante, cada elemento arquitectónico tiene una serie de restricciones respecto a los tipos de aspectos que puede importar y como se verá a continuación, estos tipos de aspectos son los que definen la semántica del elemento arquitectónico. A continuación se describen cada uno de los elementos arquitectónicos con sus características más relevantes.

#### *3.1.2.1. Componentes*

Un componente PRISMA se define como un elemento arquitectónico que captura la funcionalidad del sistema de información y no actúa como coordinador entre otros elementos arquitectónicos. Esto se traduce en la obligación de incorporar un Aspecto Funcional donde se defina esta funcionalidad. Un componente puede verse como una parte del sistema que no se puede disgregar en partes más simples, arquitectónicamente hablando. Está formado por:

- Un identificador único
- Un conjunto de aspectos, que le proporcionan su funcionalidad. Como mínimo uno: El Aspecto funcional
- Un conjunto de relaciones de *weaving*, que proporcionan la cohesión entre los aspectos que lo forman
- Una serie de puertos, cuyo tipo es una interfaz específica (un conjunto de servicios) cuya semántica es dada por los aspectos que contiene en su interior

Los puertos son los elementos que permiten la interacción del componente con los demás elementos arquitectónicos. Un puerto puede ofrecer un comportamiento servidor, cliente o cliente/servidor, en función de los servicios que lo definan. El comportamiento servidor está representado por los servicios que el componente ofrece al resto de los demás elementos mientras que el comportamiento cliente tiene su representación en los servicios que puede requerir de los demás elementos arquitectónicos.

Como se ha comentado, los componentes PRISMA han de cumplir ciertas restricciones:

- Todo componente debe especificar su aspecto funcional, excepto en el caso de componentes externos (COTS[3])
- Un componente nunca puede contener un aspecto de coordinación
- Un componente nunca puede contener dos aspectos del mismo tipo
- Los tipos de los puertos de un componente sólo podrán ser aquellas interfaces cuya semántica se defina en los aspectos que formen a dicho componente

#### *3.1.2.2. Conectores*

Un conector PRISMA es un elemento arquitectónico que actúa como coordinador entre otros elementos arquitectónicos. El conector permite describir interacciones complejas entre componentes mediante su aspecto de

[3] Los componentes externos o COTS (*Commercial Off-The-Self* ) son componentes cuya funcionalidad es realizada por terceras personas y se presenta precompilada.

coordinación. Este hecho impone la necesidad de incorporar un aspecto de coordinación que es quien coreografiará dichas interacciones. Todo conector está formado por:

- Un identificador
- Un conjunto de aspectos, que le proporcionan su funcionalidad, necesariamente uno de coordinación.
- Un conjunto de relaciones de *weaving*, que proporcionan la cohesión entre los aspectos que lo forman
- Una serie de puertos, cuyo tipo es una interfaz específica (un conjunto de servicios)

Los conectores sincronizan y conectan componentes, otros conectores o sistemas a través de sus puertos, que de la misma forma que los puertos de los componentes, definen un conjunto de servicios que el conector ofrece (comportamiento servidor), que el conector necesita (comportamiento cliente) o un comportamiento mixto (cliente/servidor).

Al igual que ocurría con los componentes, existen una serie de restricciones que todo conector ha de cumplir:

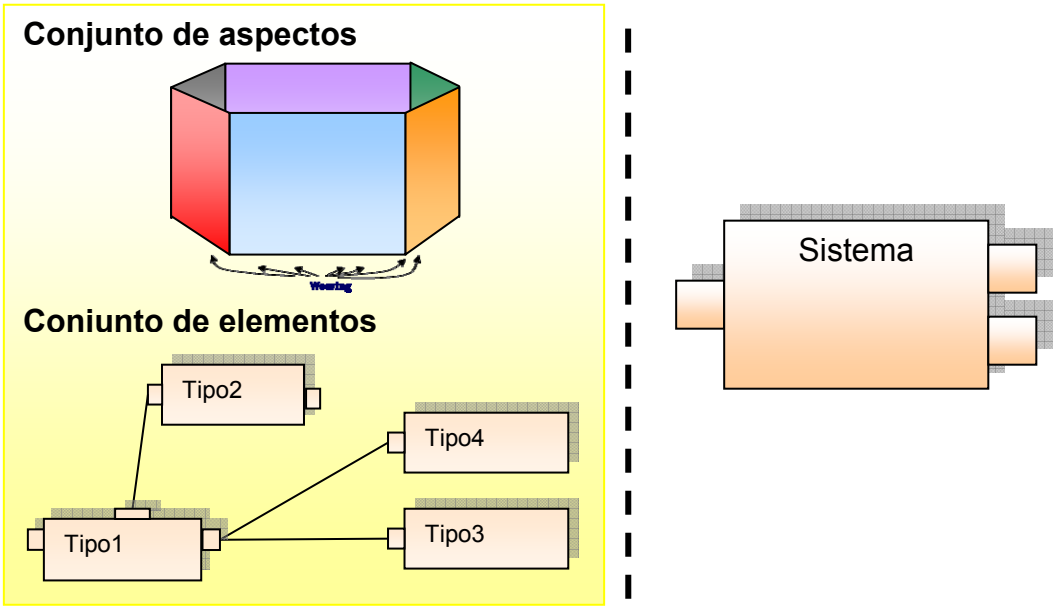
- Un conector siempre ha de tener un aspecto de coordinación,
- Un conector nunca puede tener un aspecto funcional,
- Un conector nunca puede tener dos aspectos del mismo tipo,
- Los tipos de los puertos de un conector sólo podrán ser aquellas interfaces que reciban semántica en alguno de los aspectos que formen a dicho conector,
- Una instancia de un conector ha de conectar al menos dos elementos arquitectónicos PRISMA

#### *3.1.2.3. Sistemas*

En la mayoría de los modelos arquitectónicos, surge la necesidad de tener mecanismos de abstracción que permitan tener elementos de mayor granularidad, aumentando la modularidad, composición y reutilización de los elementos arquitectónicos. En PRISMA esto se consigue mediante los sistemas. Un sistema permite encapsular un conjunto de conectores, componentes y otros sistemas, en definitiva, un conjunto de elementos arquitectónicos correctamente conectados entre sí. Pero un sistema PRISMA no sólo es capaz de encapsular, sino que, fruto de la composición de elementos arquitectónicos pueden surgir propiedades emergentes que se representan como nuevos aspectos del sistema.

En este caso particular, la vista interna no sólo está formada por un conjunto de aspectos y una serie de *weavings* definidos sobre ellos, sino que a dicha vista se le añade el conjunto de elementos arquitectónicos encapsulados por el sistema, con sus respectivos conexiones y como se muestra en la Figura

18(a). De otro modo, para el resto de elementos arquitectónicos ubicados en el exterior del sistema, la visión de dicho elemento arquitectónico es la visión externa de la Figura 18(b).




**(a) Visión interna** **(b) Visión externa**

**Figura 18: Vistas de un sistema PRISMA**

Un sistema está formado por los mismos elementos y restricciones que un componente: una función de identificación, un conjunto de aspectos, una serie de *weavings* que los entreteje, y los puertos. No obstante, al ser un tipo compuesto, dispone también de un conjunto de elementos arquitectónicos que pueden o no estar conectados entre sí y realizan una determinada función para el sistema.

La especificación de la conexión entre estos elementos y el propio sistema se realiza mediante los *bindings*, que son los enlaces entre los puertos del sistema y los puertos de los elementos que encapsula. Permiten mantener un enlace entre los distintos niveles de granularidad de los elementos arquitectónicos que forman un sistema.

Por otra parte, las conexiones entre los distintos elementos que contiene (componentes, conectores y sistemas) se realizan mediante los *attachments*, que establecen la conexión entre componentes y conectores a través de sus puertos.

De la composición de los elementos arquitectónicos contenidos pueden resultar nuevos aspectos. Estos aspectos emergentes y sus *weavings* asociados pertenecen al sistema. Por ello, las interfaces que muestran los puertos de un sistema pueden recibir semántica tanto de un aspecto del sistema como de un elemento arquitectónico definido en su interior.

Los aspectos de un sistema, se encapsulan en un componente especial que encapsulará el sistema, como si de un elemento arquitectónico más en su

interior se tratase. Por tanto, los aspectos emergentes de un sistema se comunicarán con los elementos arquitectónicos ubicados en el exterior del sistema mediante *bindings* con los puertos del sistema. Además, el sistema, ha de ser capaz de interactuar con los elementos arquitectónicos que contiene en su interior. Esta comunicación se realizará a través de *attachments* que conectarán el componente especial que encapsula los aspectos emergentes con el resto de los elementos arquitectónicos internos al sistema respetando así las restricciones del modelo. De esa forma, los elementos arquitectónicos encapsulados en el interior de un sistema, se podrán comunicar con el exterior del sistema a través de sus puertos mediante los *bindings* y con los aspectos del sistema a través de *attachments* conectados al componente especial (ver Figura 19).

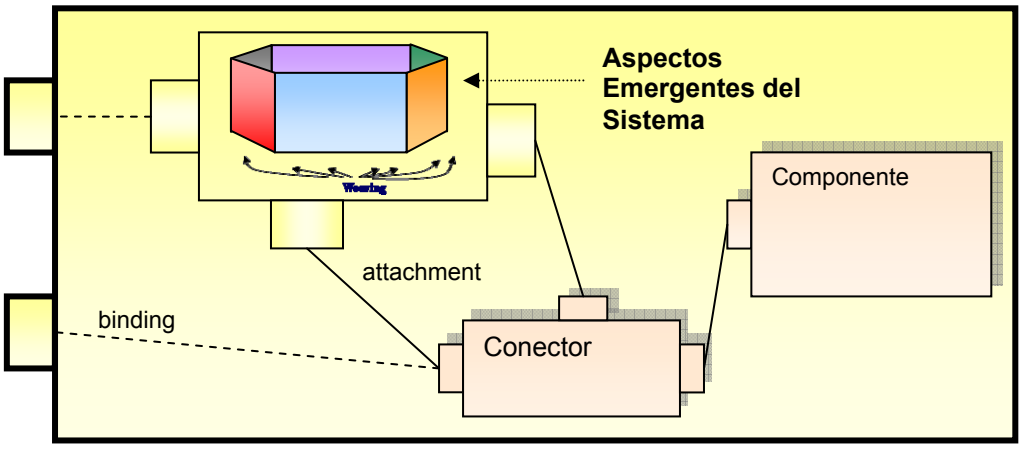



**Figura 19: Aspectos del sistema**

## 3.2. Lenguaje de Definición de Arquitecturas Orientado a Aspectos (LDA-OA)

El Lenguaje de Descripción de Arquitecturas Orientado a Aspectos de PRISMA (LDA-OA)[Pe06] proporciona una la especificación sintáctica y semántica del modelo PRISMA. Por ello, el LDA-OA proporciona construcciones para describir las primitivas del modelo PRISMA como son las interfaces, los aspectos, los elementos arquitectónicos, los *attachments* y los *bindings*. El LDA-OA está basado en OASIS [Let98] para describir formalmente los aspectos en lógica dinámica [Har84] y en Pi-Cálculo [Mil91] para formalizar la ejecución de los servicios del elemento arquitectónico. El LDA-OA de PRISMA está dividido en dos niveles de abstracción: el nivel de definición de tipos y el nivel de configuración.

El nivel de definición de tipos de PRISMA potencia la reutilización y combina el DSBC y DSOA. Este nivel permite definir los tipos necesarios para especificar un sistema arquitectónico. Dichos tipos se guardan en una librería para posteriormente reutilizarlos en la definición de distintos modelos

arquitectónicos. Sus ciudadanos de primer orden son: Interfaces, Aspectos, componentes, conectores y sistemas.

El nivel de configuración permite definir las instancias y especificar la topología del modelo arquitectónico. Para ello, en primer lugar se han de importar todos aquellos tipos elementos arquitectónicos (conectores, componentes y sistemas) definidos mediante el lenguaje de definición de tipos, que se necesiten para un determinado modelo arquitectónico. Después, se ha de definir el conjunto de instancias necesarias de cada uno de los tipos importados. Finalmente, se debe especificar la topología, interconectando adecuadamente mediante *attachments* y *bindings* las instancias del modelo.

Esta diferenciación de nivel de granularidad proporciona importantes ventajas frente a la mezcla de distintos niveles de granularidad dentro de la especificación. Una de ellas es que permite gestionar de forma independiente los tipos y las topologías específicas de cada sistema, incrementando la reutilización y obteniendo un mejor mantenimiento de las librerías de tipos. Además, permite discernir claramente entre la evolución de tipos (nivel de definición de tipos) y la evolución arquitectónica (nivel de configuración), teniendo meta-eventos diferentes en cada lenguaje para hacer evolucionar a los tipos o a las conexiones existentes entre sus instancias.

### 3.2.1. Nivel de definición de tipos

A continuación se describe la sintaxis de los ciudadanos de primer orden de PRISMA: interfaces, aspectos, componentes y conectores.

#### *3.2.1.1. Interfaces*

Las interfaces son el mecanismo usado en PRISMA para establecer canales de comunicación entre los elementos arquitectónicos. Una interfaz describe la visibilidad parcial que tienen el resto de elementos, del comportamiento del tipo que define la semántica de dicha interfaz. Por lo tanto, una instancia sólo puede solicitar y ofrecer aquellas propiedades y servicios publicados en sus interfaces. De este modo, una interfaz actúa como mecanismo de seguridad y permite que los componentes, conectores y sistemas sean vistos como cajas negras, preservando un alto nivel de abstracción.

Las interfaces definen los servicios sin tener en cuenta los puertos o aspectos que las van a utilizar, con el objetivo de favorecer la reutilización. No obstante, se ha de tener en cuenta que todos los servicios que define una misma interfaz han de ser especificadas por completo por un mismo aspecto., pudiendo estar especificada en aspectos de tipos diferentes. Por ejemplo, una interfaz puede estar especificada en un aspecto como servicios requeridos y en otro tipo de aspecto, como servicios ofrecidos.

A continuación, se muestran tres ejemplos de interfaces: Las interfaces `IAccountOperations` y `IDeposit`, que definen las operaciones típicas para el manejo de cuentas bancarias, y la interfaz `IMobility`, que define los servicios de movilidad que puede contener un elemento arquitectónico:

```
Interface IAccountOperations
   withdrawal(input ID: string, input Quantity: currency,
                    output NewMoney: currency);
   balance(input ID: string, output NewMoney: currency);
End_Interface IaccountOperations;

Interface IDeposit
  deposit(input ID: string, input Quantity: currency,
                    output NewMoney: currency);
End_Interface Ideposit;

Interface IMobility
  move(input NewLoc: LOC);
End_Interface Imobility
```

**Tabla 1: Ejemplos de Interfaces**

Como se puede observar, los servicios definen los parámetros que son de entrada mediante la palabra reservada *input*, mientras que los de salida se definen mediante la palabra reservada *output*.

#### *3.2.1.2. Aspectos*

Para definir los aspectos se ha definido una sintaxis específica basada en OASIS [Let98]. La plantilla genérica se presenta a continuación:

```
1  tipo_aspecto Aspect nombre [using interface1, ... interfaceN];

2  Attributes
       [Constant | Variable]
            <nombre_atributo1> : <tipo_atributo>;
            ...
            <nombre_atributon> : <tipo_atributo>;
       [Derived]
            <fórmula_derivación>

3  Constraints
       <restricciones>

4  Services
       begin [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio)];
            Valuations
                 <fórmulas_observabilidad_atributo>

       [in | out] <nombre_servicio> (<arg_servicio>)
            [as <nombre_servicio> (<arg_servicio)];
            Valuations
                 <fórmulas_observabilidad_atributo>

       End [(<arg_servicio>)] [as <nombre_servicio> <arg_servicio)];
            Valuations
                 <fórmulas_observabilidad_atributo>
```

```
5  Preconditions
          <fórmula_precondición_evento>

6  Triggers
          <fórmula_disparo_evento>

7  Transactions
                  <fórmula_transacción>

8  Played_Roles
          <fórmula_played_role>

9  Protocols
          <fórmula_protocolo>

10 End_ tipo_aspecto Aspect name;
```

**Tabla 2: Plantilla de Aspecto**

La cabecera de un aspecto (1) define el tipo del aspecto (funcional, distribución, etc.), el nombre que lo identifica y las interfaces a las que se da semántica. La sección `Attributes` (2) define la lista de atributos que almacenan el estado de un aspecto. Para cada atributo se indica el nombre y el tipo de valores que va a contener. Puede ser de tres tipos:

- **Constante**: su valor se establece en la inicialización y no es modificado
- **Variable**: su valor puede cambiar cuando ocurra una acción relevante (como la invocación de un servicio o la activación de un disparador)
- **Derivado**: su valor se especifica en términos de los valores de otros atributos, mediante una fórmula de derivación.

La sección `Constraints` (3) define las restricciones de integridad que deben cumplirse a lo largo de toda la vida de las instancias del tipo del aspecto. Son fórmulas basadas en el estado del aspecto y se pueden clasificar en estáticas o dinámicas, dependiendo de si se refieren sólo a un estado o relacionan diferentes estados, respectivamente. Las restricciones dinámicas se caracterizan por el uso de operadores temporales como *never, always, since* o *until*.

La sección `Services` (4) define los servicios que especifica el aspecto. Todos los servicios de la interfaz o interfaces a las que da semántica el aspecto deben definirse, además de aquellos servicios propios y no públicos del aspecto. Existen diferentes tipos de servicios: el servicio `begin` es aquel que se ejecuta a la hora de instanciar el elemento arquitectónico que importa el aspecto dando valores iniciales a los atributos, el servicio `end` hace la tarea opuesta, destruyendo al aspecto y finalmente, los servicios de modificación y consulta. Es importante resaltar que los servicios `begin` y `end` no indican que un aspecto sea instanciable, sino que hacen referencia al servicio de creación y destrucción del elemento arquitectónico al cual el aspecto pertenecerá. Además, cada

servicio de modificación o consulta puede tener un comportamiento servidor (*in*), comportamiento cliente (*out*) o un comportamiento servidor y cliente (*in/out*). El comportamiento servidor se caracteriza porque el aspecto proporciona un servicio que puede ser requerido por otros aspectos, mientras que el comportamiento cliente indica que el aspecto va a requerir dicho servicio a otro aspecto/componente.

De la misma forma que se ha visto en la definición de los servicios en la interfaz, los argumentos de los servicios que incluye un aspecto pueden ser de entrada (*input*) o de salida (*output*).

Por otro lado, el LDA-OA también proporciona un mecanismo de renombrado de servicios, mediante el operador `as`, con lo que se consigue que el nombre del servicio definido en una interfaz no tenga que coincidir con el nombre del servicio definido en el aspecto.

La sección `Valuations` especifica la semántica que define el comportamiento de los servicios. Las `Valuations` permiten definir el cambio de estado de los atributos del aspecto ante la ejecución de un servicio determinado. Se definen mediante fórmulas en lógica dinámica del tipo "φ → [a]ϕ" y se interpretan como: "si en un determinado estado del aspecto se satisface φ y ocurre la acción 'a', en el estado inmediatamente posterior se satisface ϕ". Se puede no declarar la condición de evaluación φ asumiéndose *true*.

Las secciones `Preconditions` (5) y `Triggers` (6) definen respectivamente las precondiciones y los disparadores. Las precondiciones definen las condiciones que se deben cumplir para que un servicio se pueda ejecutar, mientras que los disparos permiten especificar la ejecución de un servicio cuando se cumple una determinada condición.

La sección `Transactions` (7) se definen las transacciones. Una transacción PRISMA es un servicio no elemental y atómico, en el que se define una serie de servicios que se van a ejecutar. Por ejemplo, la definición de una operación transaccional para el envío de noticias a una lista de correo en PRISMA sería:

```
Transactions enviarNoticias(listaCorreo:string, listaNoticias:string):
  ENVIARNOTICIAS ::= seleccionarListaCorreo(listaCorreo)→NOTICIAS;
  NOTICIAS ::= seleccionarNoticias(listaNoticias, coleccion)→ ENVIO;
  ENVIO ::= envioSimultaneo(coleccion)→ CONFIRMACION;
  CONFIRMACION ::= out confirmacionLlegada();
```

**Tabla 3: Ejemplo de transacción PRISMA**

En el ejemplo se puede observar cómo se realiza el mapeo de parámetros de la transacción a cada uno de los servicios y cómo se especifica el siguiente estado alcanzado *<nombre_servicio>→ <nuevo_estado>*.

Por último, se especifican los `Played_Roles` (8) y los `Protocolos` (9). El Protocolo define un proceso describiendo un conjunto de acciones cuya

ocurrencia es posible. Está compuesto por un conjunto de procesos que establecen los servicios y transiciones a otros procesos posibles. Además, también permite especificar las prioridades de ejecución de cada uno de los servicios implicados.

Por otra parte, un *played role* es una proyección del protocolo que define el comportamiento parcial de un rol o papel determinado. Por lo tanto, todo *played role* debe ser compatible en signatura y proceso con el protocolo de forma que, dicho papel o rol se desempeña dentro del comportamiento global del protocolo y sus cálculos son un subconjunto de los cálculos del proceso.

Un *played role* es una vista parcial del protocolo que tiene sentido por sí solo, un comportamiento específico y restrictivo que es posible asociarlo posteriormente a un puerto o rol de un componente o conector. Dicha asociación permite definir de forma exacta el comportamiento que debe ejercer un puerto o rol además de los servicios que publica, establecidos por la interfaz a la que da semántica.

Ambos, protocolo y *played role*, se basan en el π-Cálculo Poliádico [Mil91], el cual permite describir de forma sencilla la ejecución de procesos y servicios susceptibles de ejecutarse concurrentemente. Una parte de su sintaxis se muestra en la Tabla 4.

| | |
|---|---|
| `x,y,z` | Nombre (interfaz, servicio, *played_role*, parámetro, identif.) |
| `x.y` | Jerarquía de nombres (x es una interfaz e y un servicio) |
| `All \| *` | Todos los *played_roles* |
| `x(m,n)` | Vector de nombres tal que m y n son parámetros |
| `P,Q,R` | Proceso |
| `P ::= x?(m,n).P` | Prefijo de entrada o acción de recepción |
| `\| x!(m,n).P` | Prefijo de salida o acción de envio |
| `\| P + Q` | Selección no determinista |
| `\| P ^ Q` | Conjunción |
| `\| P \|\| Q` | Composición paralela (concurrencia de procesos) |
| `\| P → Q` | Composición secuencial |
| `\| x(m,n):k.P` | k indica la prioridad del servicio x, tal que k=0,1,… |
| `\| if b then P else Q` | Alternativa (derivado) |
| `\| case(b » P→Q)` | Alternativa múltiple |
| `\| \|x=y\|P` | Comparación o *matching* |

**Tabla 4: Sintaxis del Pi-Cálculo Poliádico**

Para ilustrar estos conceptos, se muestran el conjunto de aspectos (funcional, distribución y coordinación) de un sistema bancario sencillo. El primero de ellos es el aspecto funcional `BankInteraction`, que especifica la funcionalidad básica de una cuenta bancaria.

```
1  Functional Aspect BankInteraction using IAccountOperations, IDeposit

2  Constant
       accountId: string;
       customer: string;

3  Variable
       address: string;
       money: currency;

4  Services

     begin( input ID: string, input Cust: string );
       Valuations
         [in begin( ID, Cust )] accountId := ID; customer := Cust;

     in withdrawal( input ID: string, input Quantity: currency,
                output NewMoney: currency );
       Valuations
         {ID = accountId}
         [in withdrawal( ID, Quantity, NewMoney )]
         money := money - Quantity,
         NewMoney := money;

     in balance( input ID: string, output NewMoney: currency );
       Valuations
         {ID = accountId}[in balance(ID, NewMoney)] NewMoney := money;

     in changeAddress(input NewAdd: string);
       Valuations
           [in ChangeAddress(NewAdd)] address := NewAdd;

     in deposit(input ID: string, input Quantity: currency,
                output NewMoney: currency);
       Valuations
         {ID = accountId}
         [in deposit(ID,Quantity, NewMoney)]
         money := money + quantity,
         NewMoney := money;
     end;

5  Constraints
     static { money >= 0}

6  Preconditions
     in withdrawal(quantity)
       if { quantity <= money };

7  Played_Roles
     OPSERVE for IAccountOperations ::=
     (
       withdrawal?(ID, Quantity, NewMoney)
     +
       balance?(ID, NewMoney)
     +
       changeAddress?(NewAdd)
     );

     DEPSERVE for Ideposit = deposit?(ID, Quantity, NewMoney);

8  Protocol
     BANKINTERACTION ::= begin!(ID,Cust)→VALIDOPERATIONS;
     VALIDOPERATIONS ::=
     (
       OPSERVE.withdrawal?(ID,Quantity, NewMoney):10 +
       OPSERVE.balance?(ID,NewMoney):10 +
       DEPSERVE.deposit?(ID,Quantity, NewMoney):10
```

```
      )→VALIDOPERATIONS
      +
      end!();

End_Functional Aspect BankInteraction;
```

**Tabla 5: Aspecto Funcional BankInteraction**

Como puede observarse, se han definido una serie de atributos (3), dos de ellos constantes (2): El número de cuenta y el nombre del cliente, ambos se definen en el momento de creación y no tiene sentido que se puedan modificar. Se han definido los servicios (4) correspondientes para sacar dinero de la cuenta (`Withdrawal`), para la consulta de saldo (`Balance`), para ingresar dinero (`Deposit`) y para cambiar la dirección (`ChangeAddress`). Obsérvese cómo se ha definido la semántica de dichos servicios mediante la lógica dinámica.

Se ha definido una restricción (5) mediante la cual se fuerza a que en todo momento el saldo sea mayor o igual que cero, y una precondición (6) para el servicio `Withdrawal` por la cual se establece que sólo se podrá sacar dinero si se dispone de dicha cantidad en la cuenta. En este caso, dado que el servicio `Withdrawal` es el único que puede decrementar el valor del atributo `money`, la precondición asociada evitaría la necesidad de añadir la restricción, pues nunca llegaría a darse dicho caso, no obstante se ha dejado a modo de ejemplo para mostrar la expresividad del lenguaje.

En la sección `Protocols` (8) se han definido dos procesos: `BANKINTERACTION`, que es el proceso inicial, y `VALIDOPERATIONS`. Obsérvese la sintaxis en Pi-Cálculo Poliádico para indicar que la petición (comportamiento servidor, mediante el símbolo '?') de cualquier servicio (*PlayedRole.Nombre_Servicio*) se mantendrá en el proceso `VALIDOPERATIONS`.

En segundo lugar se muestra el aspecto de distribución y el aspecto de coordinación. Ambos son importantes, pues formarán parte de los componentes que formarán el sistema que se construirá como ejemplo. La finalidad del aspecto de distribución `ExtMbile` no es más que indicar que el componente que lo utilice puede moverse a una nueva máquina.

```
Distribution Aspect ExtMbile using Imobility;

  Variable
    location: LOC NOT NULL;

  Services
    begin( input InitialLOC: LOC );
      Valuations
        [in begin( InitialLOC )] location := InitialLOC;
    in move( input NewLoc:LOC );
      Valuations
        [in move( NewLoc )] location := NewLoc;
      end;

  Played_Roles
    ACTUAL for Imobility = move?( NewLoc );
```

```
    Protocol
      CREATION = begin→EXTMBILE;
      EXTMBILE = ACTUAL.move?( NewLoc ):10→EXTMBILE + end;

End Distribution Aspect ExtMbile;
```

**Tabla 6: Aspecto de Distribución ExtMbile**

La finalidad del aspecto de coordinación es coordinar las peticiones entre componentes que demanden los servicios `Wihdrawal`, `Deposit` o `Balance` al componente que actúe de servidor:

```
Coordination Aspect BankCoordination using IaccountOperations

  Services
    begin;
    in/out withdrawal(input Quantity: integer, output Money: integer);
    in/out balance(output Money: integer);
    in/out deposit(input Quantity: integer, output Money: integer);
    end;

  Played_Role
    OPCLIENT for IaccountOperations =
    (
      ( withdrawal?( ID,Quantity, Money )
        →
       withdrawal!( iD,Quantity, Money ) )
    +
      ( balance?( ID,Money )
        →
       balance!( ID,Money ) )
      +
      ( deposit?( ID,Quantity, Money )
        →
       deposit!( ID,Quantity, Money ) )
    );
      OPSERVER for IaccountOperations =
    (
      ( withdrawal!( ID,Quantity, Money )
        →
          withdrawal?( ID,Quantity, Money ) )
    +
      ( balance!( ID,Money )
        →
       balance?( ID,Money ) )
    +
      ( deposit?( Quantity, Money )
        →
       deposit!( Quantity, Money ) )
    );
    DEPCLIENT for Ideposit =
      ( deposit?( ID, Quantity, Money )
        →
       deposit!( ID, Quantity, Money ) );


      DEPSERVER for Ideposit =
      ( deposit?( ID, Quantity, Money )
        →
       deposit!( ID, Quantity, Money ) );


    Protocol
      BANKCOORDINATION = begin.COORD;
      COORD =
```

```
        (
          ( OPCLIENT.withdrawal?( ID, Quantity, Money ):10 →
            OPSERVER.withdrawal!( ID, Quantity, Money ):10 →
            OPSERVER.withdrawal?( ID, Quantity, Money ):10 →
            OPCLIENT.withdrawal!( ID, Quantity, Money ):10 ).COORD
        +
          ( OPCLIENT.balance?( ID, Money ):10 →
            OPSERVER.balance!( ID, Money ):10 →
            OPSERVER.balance?( ID, Money ):10 →
            OPCLIENT.balance!( ID, Money ):10).COORD
        +
          ( DEPCLIENT.deposit?( ID, Quantity, Money ):10 →
            DEPSERVER.deposit!( ID, Quantity, Money ):10 →
            DEPSERVER.deposit?( ID, Quantity, Money ):10 →
            DEPCLIENT.deposit!( ID, Quantity, Money ):10).COORD
        +
           end
        );

    End_Coordination Aspect BankCoordination;
```

**Tabla 7: Aspecto de Coordinación BankCoordination**

Con esta especificación, más compleja que las anteriores mostradas en la sección actual, puede observarse la potencia expresiva del Pi-Cálculo Poliádico. Este aspecto se utilizará en un conector para coordinar dos componentes, uno que se comportará como cliente y el otro como servidor. Cada uno de estos roles los desempeñará tanto para los servicios de `IAccountOperations` como para los de `IDeposit`, no obstante, para simplificar, la explicación se centrará únicamente en la coordinación de las peticiones de los servicios de `IAccountOperations`. Se han definido dos *played roles*: uno para el comportamiento cliente (`OPCLIENT`) y otro para el comportamiento servidor (`OPSERVER`). El *played role* `OPCLIENT` describe el flujo de entradas y salidas para el comportamiento cliente. Por ello, cuando se reciba una petición del servicio `Withdrawal` (flujo de entrada, indicado mediante el símbolo '?') la siguiente acción válida sólo podrá ser devolver el resultado de dicha petición (flujo de salida, indicado mediante el símbolo '!'). En cambio, el *played role* `OPSERVER` modela el otro lado de la comunicación: cuando se reciba un flujo de salida (invocar un servicio), la única acción válida sólo podrá ser un flujo de entrada (obtener los resultados de dicho servicio). Los símbolos '!' deben verse como flujos de salida (*output*) y los '?' como flujos de entrada (*input*).

Por otra parte, el protocolo modela el proceso global: coordina y sincroniza los servicios de los distintos *played roles*. En la Figura 20 se puede observar el esquema de funcionamiento de un protocolo de forma gráfica. Cuando llegue una petición de `Withdrawal` (?) al aspecto de coordinación, la siguiente acción que se realizará es transmitir dicha petición (!) al componente que actúa como servidor (el componente Banco). Entonces, el aspecto esperará hasta que se reciba la respuesta por parte del servidor (`SERVER.Withdrawal?`), tras lo cual se retransmitirá al componente cliente (`CLIENT.Withdrawal!`). Una vez se enviase al cliente, el siguiente estado válido sería `COORD`, y podría procesarse otro servicio diferente.

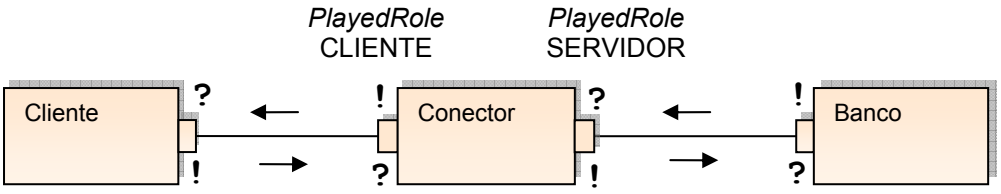



**Figura 20: Esquema de funcionamiento de un protocolo**

Si se quisiese permitir la ejecución de otro servicio (como `Balance`) mientras el aspecto de coordinación espera la respuesta del servidor, se indicaría mediante el uso del operador '| |' (composición paralela), en lugar del operador '+' (selección no determinista), con lo que se especificaría la ejecución concurrente de servicios.

#### *3.2.1.3. Componentes y Conectores*

La plantilla de un componente y de un conector, muy similares entre sí, están formadas por cuatro partes básicas: la cabecera, la definición de puertos o roles de comunicación, los aspectos que lo forman y los *weavings* que entretejen dichos aspectos. A continuación se muestra la plantilla genérica, y se describirá cada sección en detalle:

```
1  [Component | Connector]_type <nombre_Componente|Conector>

2   [Functional | Coordination] Aspect Import <nombre_aspectoj>;
   <tipo_aspectok> Aspect Import <nombre_aspectok>;

3  Port
     <nombrei> : <interfazi>
     Played_Role <nombre_aspecto>.<nom_playedRole_del_aspecto>;
   End_Port;

4  [Weaving
     <nombre_aspectok>.<nombre_serviciok>
     <operador_weaving>
     <nombre_aspecton>.<nombre_servicion>;
   End_Weaving;]

5  Initialize
     New(<argumentos>) {
       <nombre_aspectoi>.begin(<args_constructori>);
     }
   End_Initialize;

6  Destruction
     Destroy() {
       <nombre_aspectoi>.end(<args_destructori>);
       <nombre_aspectoi+1>.end(<args_destructori+1>);
     }
   End_Destruction;

7  End_[Component | Connector]_type <nombre_Componente|Conector>;
```

**Tabla 8: Plantilla de componente y conector**

La cabecera (1) se compone de la palabra reservada `component_type` (para el caso de los componentes) o de `connector_type` (para el caso de los conectores), seguida del nombre que reciba el componente o conector, y que lo identificará unívocamente en la librería PRISMA.

A continuación viene la importación de los aspectos que se realiza indicando el tipo y el nombre del aspecto (2). Como se ha comentado, al menos debe importarse un aspecto funcional, en el caso de los componentes, o un aspecto de coordinación, en el caso de los conectores.

Los puertos de comunicación se definen en la sección `Port` (3), indicando qué interfaz implementan. Además, se le asocia un *played role* al puerto para especificar el comportamiento del conjunto de servicios que forman parte de la interfaz. El *played role* pertenece al aspecto indicado mediante la notación punto *<nombre_aspecto>.<nom_playedRole_del_aspecto>*. De esta forma, pueden crearse varios puertos con la misma interfaz pero con *played roles* diferentes, con lo que cada puerto con idéntica interfaz tendrá un comportamiento diferente: el especificado por el *played role*.

El *weaving* entre los aspectos (4) se define al importar cada aspecto, indicando con qué aspecto van a entretejerse sus servicios. Éstos se declaran especificando el nombre del aspecto en el cual están definidos mediante la notación:

*<nombre_aspecto>.<nombre_servicio>.*

El *weaving* entre los servicios se realiza mediante *<operador_weaving>*, que indica si la sincronización se realiza *after* (después), *before* (antes) o *instead* (en lugar de), como se mostró en el apartado 3.1.1.

Por último, las secciones `Initialize` (5) y `Destroy` (6) describen el proceso de creación y destrucción de los componentes/conectores. En la sección `Initialize` se especifica cómo el servicio `New` desencadena la ejecución del servicio `Begin` de los distintos aspectos que forman el componente, mientras que la sección `Destruction` desencadena la destrucción de los aspectos. Además, en el servicio `New` pueden indicarse los parámetros que necesitan los aspectos para su inicialización, que a su vez pueden ser proporcionados por el elemento arquitectónico que cree el componente/conector.

A continuación, se muestra un ejemplo de un componente: El componente `Account`. Éste importa el aspecto funcional definido anteriormente y realiza el *weaving* del servicio `ChangeAddress` con el servicio `Move` del aspecto de distribución:

```
Component Account

  Functional Aspect import BankInteraction;
  Distribution Aspect Import ExtMbile;
```

```
    Weavings
      BankInteraction.ChangeAddress(NewAdd: string) before
          ExtMbile.Move(NewAdd: string);
    End_Weavings;

    Ports
      Operations_port: IaccountOperations
      Played_Role BankInteraction.OPSERVE;
      deposit_port: Ideposit
       Played_Role BankInteraction.DEPSERVE;
    End_Ports;

    new( ID: integer, initialLocation: LOC )
    {
      BankInteraction.begin( ID );
      ExtMbile.begin( InitialLocation );
    }

    destroy() {
      BankInteraction.end();
      ExtMbile.end();
    }

End_Component Account;
```

**Tabla 9: Componente Account**

Como se observa, se importa un aspecto funcional y un aspecto de distribución, sin embargo, los puertos definidos únicamente muestran puertos asociados a los *played roles* del aspecto funcional. Esto es debido a que los servicios del aspecto de distribución no van a ser invocados desde un componente externo, sino que serán invocados desde el mismo componente al que pertenece el aspecto, ya sea desde el propio aspecto de distribución o a través de un *weaving* desde otro aspecto. En la especificación mostrada para la definición del `Account`, se ha definido un *weaving* mediante el cual se invocará al servicio `Move` de `IMobility` en el aspecto de distribución desde el aspecto funcional. Con esto se pone de manifiesto que las interfaces definidas en un aspecto no tienen porque exteriorizarse al resto de elementos arquitectónicos a través de los puertos. Dicho esto, se puede pensar que, dado que la interfaz define servicios privados al aspecto, no tiene sentido definirla como tal. No obstante, se ha de tener en cuenta que, pese a que no se exteriorice la sintaxis de la interfaz mediante los puertos, si que es posible que sea invocada desde otro aspecto a través de un *weaving*, como ocurre en el ejemplo.

También se muestra un conector cuya función sería la de coordinar a un componente que requiriese las operaciones de las cuentas, por ejemplo, un componente que representase al banco, con las propias cuentas.

```
Connector CnctAccount

  Coordination Aspect import BankCoordination;
  Distribution Aspect import ExtMbile;

  Ports

   BankOp_port: IaccountOperations,
      Played_Role BankCoordination.OPCLIENT;
```

```
    deposit_port: Ideposit,
       Played_Role BankCoordination.DEPCLIENT;

    AccountOp_port: IaccountOperations,
       Played_Role BankCoordination.OPSERVER;

  AccountDep_port: Ideposit,
       Played_Role BankCoordination.DEPSERVER;

  End_Ports


  new( InitialLocation: LOC )
  {
     BankCoordination.begin();
     ExtMbile.begin( InitialLocation );
  }

  Destroy()
  {
     BankInteraction.end();
     ExtMbile.end();
  }

End_Connector CnctAccount;
```

**Tabla 10: Conector CnctAccount**

Como puede observarse, el conector define dos puertos con la interfaz `IAccountOperations` y otros dos con la interfaz `IDeposit`. Es mediante los *played roles* que puede distinguir las peticiones de servicio que recibe (comportamiento cliente de los componentes a él conectados) y los servicios ofrecidos (comportamiento servidor). La semántica de coordinación de dichos *played roles* vendrá definida por el aspecto de coordinación que lo define.

En el conector tiene un Aspecto de Distribución, pero ni se define un puerto con un *played role* asociado a dicho aspecto ni se crea un *weaving* que entreteja la ejecución de los servicios de `ExtMbile` con otro aspecto, por lo tanto, será como si el método `Move` de la interfaz `IMobility` se hubiese definido como un método privado, pues solo se invocará desde el propio aspecto de movilidad.

A continuación se muestra la plantilla para la definición de sistemas PRISMA:

```
1  System_type <nombre_sistema>

2  [<tipo_aspectok> Aspect Import <nombre_aspectok>;]

3  [Weaving
     <nombre_aspectok>.<nombre_serviciok>
     <operador_weaving>
     <nombre_aspecton>.<nombre_servicion>;
   End_Weaving;]

4  Ports
     <nom_puertoi> : <interfazi>;
```

```
    <nom_puertoi+1> : <interfazi+1>;
  End_Ports

5 Import Arquitechtural Elements
    <nom_componente | nom_conector | nom_sistema>;
     ...
  ;

6 Attachments
    <nom_attachmenti> : <nom_EAk>.<nom_puerto> ←→ <nom_EAj>.<nom_puerto>;
    ...
  End_Attachments;

7 Bindings
   <nom_bindingi>: <nom_puerto> ←→ <nom_EAk>.<nom_puerto>;
    ...
  End_Bindings;

8 New() {
      <nom_AEi>= new <(nom_componente | nom_conector | nom_sistema)>(<args>);
      ...
    }

9 Destroy() {
      <(nom_componente | nom_conector | nom_sistema)>.destroy();
      ...
    }

  End_System_type <nombre_sistema>;
```

**Tabla 11: Plantilla de sistema**

En la cabecera de un sistema se indica el nombre identificativo de dicho tipo (1), que lo identificará dentro de la librería de tipos PRISMA. A continuación se importan los aspectos del sistema (2), en caso de haberlos. De la misma manera, se definen los *weavings* entre los aspectos definidos (3).

La definición de puertos (4), permite indicar los puertos de comunicación que tendrá dicho sistema con el resto de elementos arquitectónicos. Mediante los *bindings* se establecerá la relación entre los puertos del sistema y los componentes que les darán semántica a los servicios de dichos puertos.

La importación de elementos arquitectónicos (5) permite especificar los componentes, conectores y/o sistemas que formarán parte del sistema. La construcción/destrucción de dichos componentes se desencadenará en el momento de creación/destrucción del sistema. Estas operaciones se especifican en las secciones `New` (8) y `Destroy` (9) y permiten indicar los parámetros que necesitará cada componente para su creación.

La definición de un sistema especifica las relaciones de conexión (*attachmens*) y composición (*bindings*) entre los elementos arquitectónicos que contiene. Como se definió anteriormente, los *attachments* establecen la conexión entre los puertos de los componentes y los roles de los conectores, mientras que los *bindings* definen la composición entre el sistema y los componentes o subsistemas que contiene. Los *attachments* (6) se especifican mediante una relación entre dos elementos arquitectónicos importados, y a través de la notación punto se indican los puertos que se van a conectar:

*<elemento_arquitectónicoi>.<puerto>* ←→ *<elemento_arquitectónicoj>.<puerto>*

Los *bindings* (7) se especifican de la misma forma, sólo que en lugar de definir la relación entre componentes y conectores, se define entre un componente o conector internos al sistema y un puerto del sistema.

Como ejemplo, en la Figura 21 se muestra el sistema `BankSystem` que encapsula los componentes y conectores definidos anteriormente, junto con la especificación PRISMA.

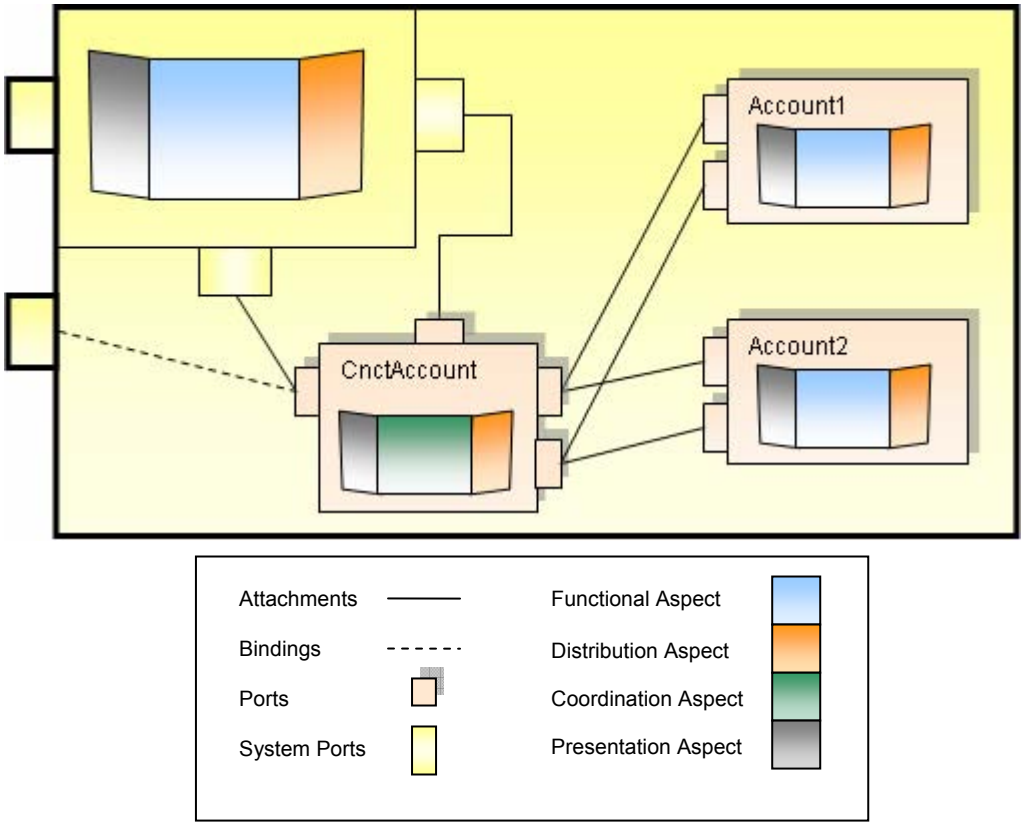



**Figura 21: Ejemplo BankSystem**

A continuación se muestra la especificación del sistema de la Figura 21 haciendo uso de la plantilla mostrada anteriormente del LDA-AO de PRISMA:

```
System BankSystem

  Functional Aspect import BankOperations;
  Distribution Aspect import ExtMbile;

  Ports
    deposit_port : Ideposit;
    Mobility_port : Imobility;
    BankOperations_intport : IaccountOperations;
    deposit_intport : Ideposit;

  End_Ports;
```

```
Import Architectural Elements Account(0,n), CnctAccount(1,1);

Attachments
  AccountOpAtt:
     Account.Operations_port(0..n)←→(1..1)CnctAccount.AccountOp_port;
  AccountDepAtt:
     Account.deposit_port(0..n)←→(1..1)CnctAccount. deposit_port;
  BankOpAtt:
     Account.BankOp_intport(1..1)←→(1..1)CnctAccount.BankSystOp_port;
  BankDepAtt:
    BankSystem.deposit_intport(1..1)←→(1..1)CnctAccount.deposit_port;
End_Attachments;

Bindings
  ExtdepositBind:
    BankSystem.deposit_port (1..1)←→(1..1)CnctAccount.deposit_port;
End_Bindings;

new(Name: string, InitialLocation: LOC, Account,
      AccNumber: integer, CnctAccount, CnctNumber: integer )
  {

    BankOperations.begin(Name);
    ExtMble.begin(InitialLocation);

    new Account(AccountId:integer,InitialLocation:LOC);
    new CnctAccount(InitialLocation: LOC);

    new AccountOpAtt(Account, Operations_port, CnctAccount,
                                        AccountOp_port);
    new AccountDepAtt(Account, Deposit_port, CnctAccount,
                                        Deposit_port);
    new BankOpAtt (Account, BankOp_intport, CnctAccount,
                                        BankSystOp_port);
    new BankDepAtt (Account, Deposit_intport, CnctAccount,
                                        Deposit_port);

    new ExtdepositBind (BankSystem, Deposit_port, CnctAccount,
                                        Deposit_Port);
}


destroy() {
    BankOperations.end();
    ExtMbile.end();

    Account.destroy();
    CnctAccount.destroy();
    AccountOpAtt.destroy();
    AccountDepAtt.destroy();
    BankOpAtt.destroy();
    BankDepAtt.destroy();
    ExtdepositBind.destroy();
}

End_System BankSystem;
```

**Tabla 12: Sistema BankSystem**

Como se puede observar se ha definido un sistema que encapsula dos cuentas bancarias y un conector. El sistema tiene un puerto `IDeposit` conectado con el puerto `Deposit_port` del conector mediante una relación de *binding*. También se ha contemplado la emergencia de nuevos aspectos que se conectarán mediante una serie de puertos a los elementos arquitectónicos que

componen al sistema, como es el caso de aquel por el que se solicitarán operaciones de `IAccountOperations`. En este caso la conexión se ha establecido mediante un *attachment*.

### 3.2.2. Nivel de configuración

El nivel de configuración del lenguaje permite definir las instancias y la topología del modelo arquitectónico final. Es en este nivel en el que se instancian los diferentes elementos arquitectónicos definidos a nivel de tipos. A continuación se puede observar la plantilla genérica para la configuración de arquitecturas:

```
Architectural_Model_Configuration <nombre_configuracion> =
  New <nombre_modelo>{


  <nom_inst_comp>= new <nom_componente> (<args>) ;

  <nom_inst_con>= new <nom_conector> (<args>) ;

  <nom_inst_sistema>= new <nom_sistema> (<args>)
  {
    <instanciación_elementos_sistema>
    <nom_inst_att_sys>= new <nom_attachment_sys> (<args>) ;

    <nom_inst_bind>= new <nom_bindings> (<args>) ;
  };

  <nom_inst_att>= new <nom_attachment> (<args>) ;

}
```

**Tabla 13: Plantilla de Configuración Arquitectónica**

Como se puede observar, para cada uno de los tipos de elementos arquitectónicos se van creando las instancias, a las cuales se les da un nombre (<*nom_instancia_tipo*>). En el caso de los sistemas, las instancias se crean para cada uno de sus elementos internos, con sus conexiones mediante *attachments* y *bindings*. También se instancian los *attachments* entre los conectores, componentes y sistemas instanciados en la configuración.

A continuación se muestra un ejemplo de configuración para el sistema Bancario, ejemplo con el que se ha ido ilustrando esta sección:

```
Architectural_Model_Configuration BankConf = new SimpleBankSystem
{

   mw0 = new LOC("tcp://host0.dsic.upv.es");
   mw1 = new LOC("tcp://host1.dsic.upv.es");
   mw2 = new LOC("tcp://host2.dsic.upv.es");


   BANKSYSTEM = new BankSystem("MyBank", mw0, Account, 2, CnctrAccount,1)
{
       ACCOUNT1 = new Account("000001", mw0);
```

```
            ACCOUNT2 = new Account("000002", mw1);
            CONNECTOR = new CnctrAccount(mw2);

            ATT1 = new AccountOpAtt(ACCOUNT1,Operations_port,AccountOp_port,
                                                              CONNECTOR);
            ATT2 = new AccountOpAtt(ACCOUNT2,Operations_port,AccountOp_port,
                                                              CONNECTOR);
            ATT3 = new AccountDepAtt(ACCOUNT1,Deposit_port,AccountDep_port,
                                                              CONNECTOR);
            ATT4 = new AccountDepAtt(ACCOUNT2,Deposit_port,AccountDep_port,
                                                              CONNECTOR);
            ATT5 = new BankOpAtt(BANKSYSTEM, BankOp_intport, BankSysOp_port,
                                                              CONNECTOR);
            ATT6 = new BankDepAtt(BANKSYSTEM, Deposit_intport, Deposit_port,
                                                              CONNECTOR);

            BIND1 = new ExtdepositBind(BANKSYSTEM, Deposit_port, Deposit_port,
                                                              CONNECTOR);
        };
    };
```

**Tabla 14: Configuración del Sistema Bancario**

En la configuración anterior se ha instanciado un sistema del tipo `BankSystem` indicándole cual será su nombre (*MySystem*) y los elementos que tendrá en su interior: dos `Accounts` y un `CnctrAccount`. También se le indica la localización en el que se ejecutará el sistema. La localización se indica mediante una instancia del tipo abstracto de datos LOC, tipo de datos especial del modelo PRISMA que representa las localizaciónes físicas que componen el sistema distribuido al que se le esta modelando su arquitectura. Como se puede observar, el primer paso de la configuración del modelo arquitectónico es la definición de tres localizaciones diferentes. Las instancias de los elementos arquitectónicos de este modelo deberán localizarse en estas tres localizaciones.

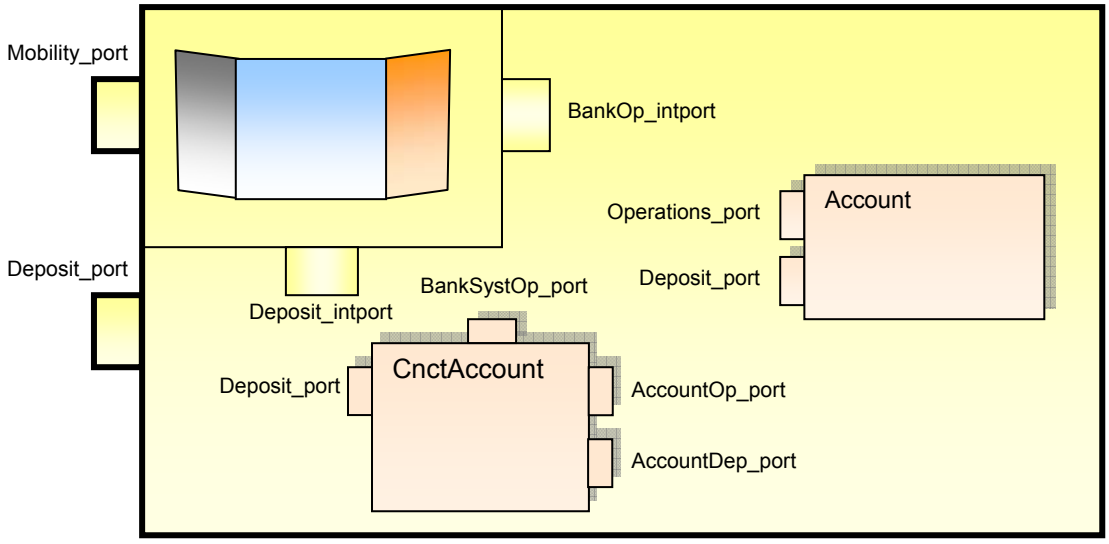



**Figura 22: Puertos del Sistema BankSystem**

Obsérvese cómo se han ido anidando los constructores para inicializar los componentes que forman el sistema. El *framework* de PRISMA sería el encargado, mediante reflexión de código, de obtener los componentes que

forman el sistema y de pedir al usuario los parámetros de inicialización necesarios, bien de forma gráfica mediante interfaces de usuario, o bien por consola. Así pues, se instancian dos `Accounts` y el conjunto de *attachments* y *bindings* necesarios para conectar los puertos del sistema. Para facilitar la lectura, en la Figura 22 se muestran los nombres de puertos de los elementos arquitectónicos implicados.

Con lo explicado hasta este punto se tendría la información necesaria, tanto a nivel de tipos como a nivel de configuración, para poner en ejecución el modelo arquitectónico deseado.

## 3.3. PRISMANET

En la sección anterior se ha presentado como el modelo PRISMA combina el DSOA y el DSBC para especificar arquitecturas software de sistemas distribuidos y complejos en un nivel alto de abstracción. Además, uno de los objetivos del modelo es el generar automáticamente aplicaciones ejecutables desde sus especificaciones. Para poder lograr este objetivo, se ha de elegir una plataforma tecnológica específica. La plataforma especifica elegida ha sido la plataforma .NET de Microsoft© (como pudiese ser cualquier plataforma tecnológica existente). Sin embargo, la plataforma .NET no da soporte a las primitivas que PRISMA proporciona como en el caso de los aspectos, los elementos arquitectónicos o la movilidad de los elementos. Por este motivo, se ha tenido que desarrollar un *middleware* en C# para dar soporte a estas primitivas denominado PRISMANET [Cos05][Pe05b].

PRISMANET es la capa que implementa las correspondencias entre el modelo PRISMA y la plataforma .NET. Para construir una arquitectura PRISMA ejecutable el compilador de modelos PRISMA utilizara el PRISMANET para extender las clases que corresponden con las primitivas PRISMA y proporcionarlas con los servicios necesarios para su ejecución. PRISMANET proporciona a las aplicaciones PRISMA generadas la ejecución concurrente de los elementos arquitectónicos, la comunicación distribuida, la movilidad y la evolución dinámica.

El diseño realizado da soporte a la gestión de los aspectos, los elementos arquitectonicos, *attachments* y *bindings*. Además provee a los elementos arquitectónicos la concurrencia, la comunicación distribuida, la movilidad, la reutilización y la evolución dinámica de código. Éstas son las características del modelo arquitectónico PRISMA que proporcionan a sus elementos una gran potencia expresiva. Sirva como ejemplo de la capacidad que brinda el prototipo desarrollado, la propiedad de los componentes de poder agregar o eliminar aspectos, *weavings* o puertos en tiempo de ejecución sin necesidad de recompilar y, en algunos casos, sin detener la ejecución del componente. Esta propiedad es posible gracias a que los componentes se han implementado de forma totalmente independiente de las partes que lo componen, con lo que además también se favorece la reutilización.

En la presente sección se pretende dar una visión de cómo se ha implementado el *middleware* PRISMANET en la plataforma .NET ya que el trabajo de este proyecto extiende dicho *middleware* para incorporar nuevas características adicionales para el desarrollo de aplicaciones distribuidas y móviles.

### 3.3.1. Diseño de la Arquitectura

Como se ha comentado la plataforma sobre la que se ha creado el *middleware* es el *framework* .NET de Microsoft© [dotNET]. La arquitectura del *framework* .NET está dividida en tres capas. .NET proporciona un lenguaje intermedio (MSIL, *MicroSoft Intermediate Language*) que es interpretado por una máquina virtual (CLR, *Common Languaje Runtime*) para la obtención del código nativo. Los diferentes compiladores de lenguajes que funcionan en la plataforma .NET generan código intermedio haciendo uso de las reglas que proporciona el CTS (*Common Type System*[4]) a través del CLS (*Common Language System*[5]) para la generación de código interoperable.

Una vez elegido el lenguaje se contemplaron varias aproximaciones en relación al nivel en que se debía ubicar el *middleware* PRISMA.

- **Modificación del CLR:** Introducir mecanismos que extendiesen la funcionalidad del CLR. Al modificar en este nivel, las herramientas de depuración existentes no pueden ser utilizadas ya que el CLR ha visto modificado su comportamiento.

- **Alteración del código intermedio generado:** Son las aproximaciones que manipulan de alguna forma el lenguaje intermedio generado. El principal inconveniente, al igual que en el caso anterior, es que tampoco es posible depurar el código fuente.

- **Extensión del CTS/CLS:** El problema que presentan es que los compiladores desarrollados deben actualizarse al mismo ritmo que la evolución de los lenguajes .NET que extienden, y además, se pierde la independencia del lenguaje

Los tres grupos descritos tienen el inconveniente adicional de crear una dependencia con la plataforma .NET, con lo que futuras versiones supondrían una actualización de dichas aproximaciones para mantener la compatibilidad.

[4] El Sistema de Tipos Común permite la interoperabilidad entre los diferentes lenguajes en .Net.
[5] El Sistema de Lenguaje Común define las reglas que han de cumplir todos los compiladores de .Net para generar código interoperable.

En cambio, PRISMA actúa a un nivel superior, ya que utiliza un lenguaje de descripción de arquitecturas independiente de cualquier plataforma de desarrollo, lo que le permite incorporar funcionalidad adicional y un mayor nivel de abstracción. Por otro lado, la implementación de PRISMA se ha realizado basándose en los lenguajes estándar definidos, sin añadir ninguna extensión a la plataforma de desarrollo, con el objetivo de no tener que adaptar la implementación a nuevas versiones del CLR, MSIL, etc.

En la Figura 23 se muestra gráficamente la ubicación de PRISMA en la arquitectura del framework .NET. Se puede afirmar por tanto que PRISMANET [Pe05b] no solo extiende la tecnología .NET incorporándole los conceptos del modelo PRISMA, dando soporte a la reconfiguración dinámica de arquitecturas locales y distribuidas, sino que además esta funcionalidad se ha conseguido haciendo uso de los mecanismos que proporciona .NET.

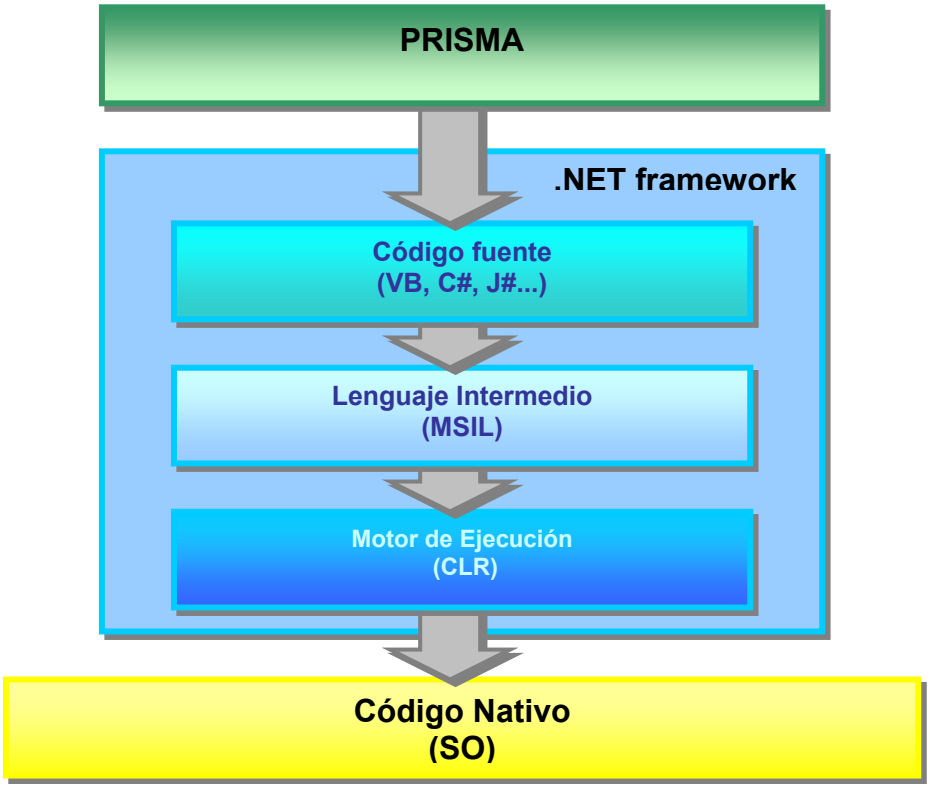



**Figura 23: PRISMA sobre .NET.**

El proceso de crear un modelo PRISMA y ejecutarlo pasa por una serie de etapas (ver Figura 24). En primer lugar, a través de una herramienta visual [6]o mediante el uso del LDA-OA, el analista diseña y especifica el modelo arquitectónico. En la segunda etapa, a partir del modelo arquitectónico y haciendo uso de un conjunto de patrones de generación de código el compilador de modelos genera su representación en un lenguaje de programación en este caso C#, extendiendo los constructores que proporciona el PRISMANET. Por

[6] En la actualidad se dispone de una herramienta visual de modelado para PRISMA basada en DSL Tools de Microsoft©.

ejemplo, para representar un aspecto del modelo PRISMA el compilador de modelos extiende la clase `AspectBase` que el PRISMANET proporciona. Por último, el código generado es puesto en ejecución. Este código hace uso de los servicios de distribución y evolución del PRISMANET requeridos por el modelo.

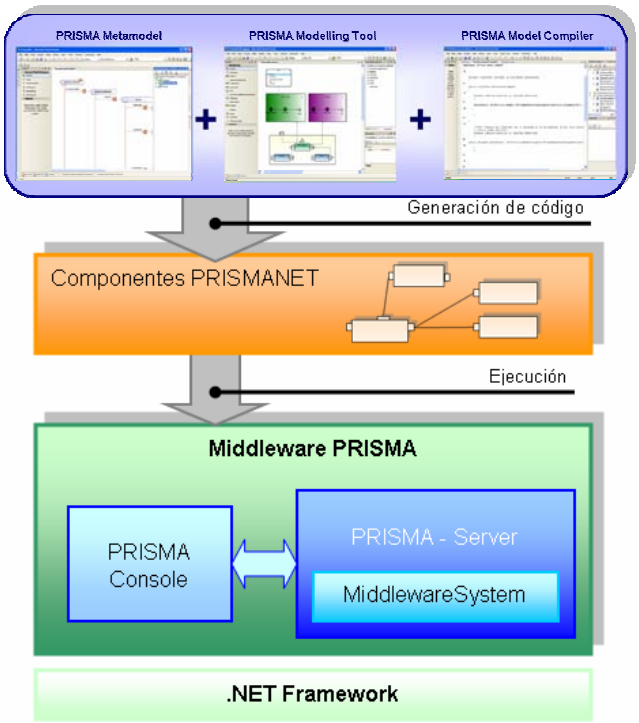



**Figura 24: Arquitectura PRISMANET**

El *middleware* (ver Figura 24) es una capa abstracta que se ejecuta sobre la plataforma .NET, formada a su vez por dos módulos:

- **PRISMA *Server*:** Módulo que contiene el núcleo del *Middleware*, implementado en la clase `MiddlewareSystem`, y proporciona los servicios de gestión, distribución y evolución de los componentes PRISMA.

- **PRISMA *Console*:** Consola de administración del *Middleware* que ofrece al usuario una interfaz para poner en ejecución configuraciones PRISMA, así como gestionar las instancias en ejecución. Además muestra información sobre el entorno de ejecución, y los diferentes mensajes que son lanzados por los diferentes middlewares que lo conforman.

Los dos módulos son independientes entre sí, para que en caso de fallar uno de ellos, el sistema pueda recuperarse fácilmente sin más que volver a cargar el módulo. Para la ejecución de configuraciones PRISMA es necesario que PRISMA Server esté en ejecución, pues es quien proporciona los mecanismos propios de la arquitectura PRISMA necesarios para la ejecución. La ejecución de PRISMA *Console* es optativa, y sólo necesaria en caso que el

usuario quiera lanzar a ejecución una configuración o ver información sobre el estado de una configuración en funcionamiento. En las últimas versiones PRISMA *Console* también proporciona una interfaz para la invocación de servicios de los elementos arquitectónicos en ejecución.

Un modelo PRISMA en ejecución puede ser distribuido, tal como se contempla en el lenguaje de descripción de arquitecturas. Esto implica que en cada nodo donde se quiera ejecutar PRISMA debe estar instalado y cargado el PRISMA *Server*, ya que además de gestionar los componentes PRISMA también proporciona los servicios de distribución, movilidad, evolución, y mantenimiento a las instancias (ver Figura 25).

Los distintos *middleware* se encargan de gestionar los elementos que están ejecutándose en su nodo y de establecer las comunicaciones con sus *middlewares* vecinos. Los elementos compilados del modelo PRISMA se encuentran distribuidos por los distintos nodos, de forma que si uno de ellos falla, el sistema formado por el conjunto de middlewares puede redistribuirse las tareas para mantener estable al sistema.

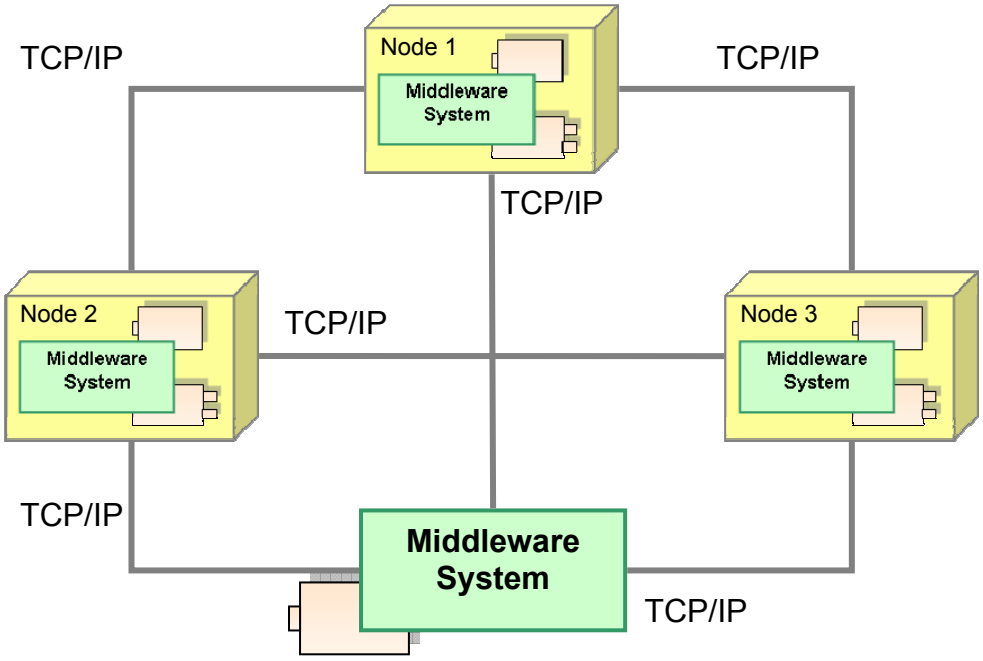



**Figura 25: Middlewares ejecutándose de forma distribuida**

El hecho que todos los *middlewares* que forman el entorno de ejecución estén comunicados entre si, permite que, pese a que las instancias de los elementos arquitectónicos se encuentren distribuidas entre diferentes nodos, todas las instancias sean accesibles desde cualquier *middleware*. En cierto modo se puede considerar como el conjunto de nodos con un *middleware* PRISMANET en ejecución como una parte de un *middleware* virtual en ejecución.

Como se ha visto, la funcionalidad del *middleware* PRISMANET está contenida en la clase `MiddlewareSystem`. Dicha clase es instanciada cuando se pone en ejecución el PRISMA *Server* en el nodo. La clase `MiddlewareSystem` está factorizada en un conjunto de capas que encapsulan las diferentes facetas de su funcionalidad (ver Figura 26).

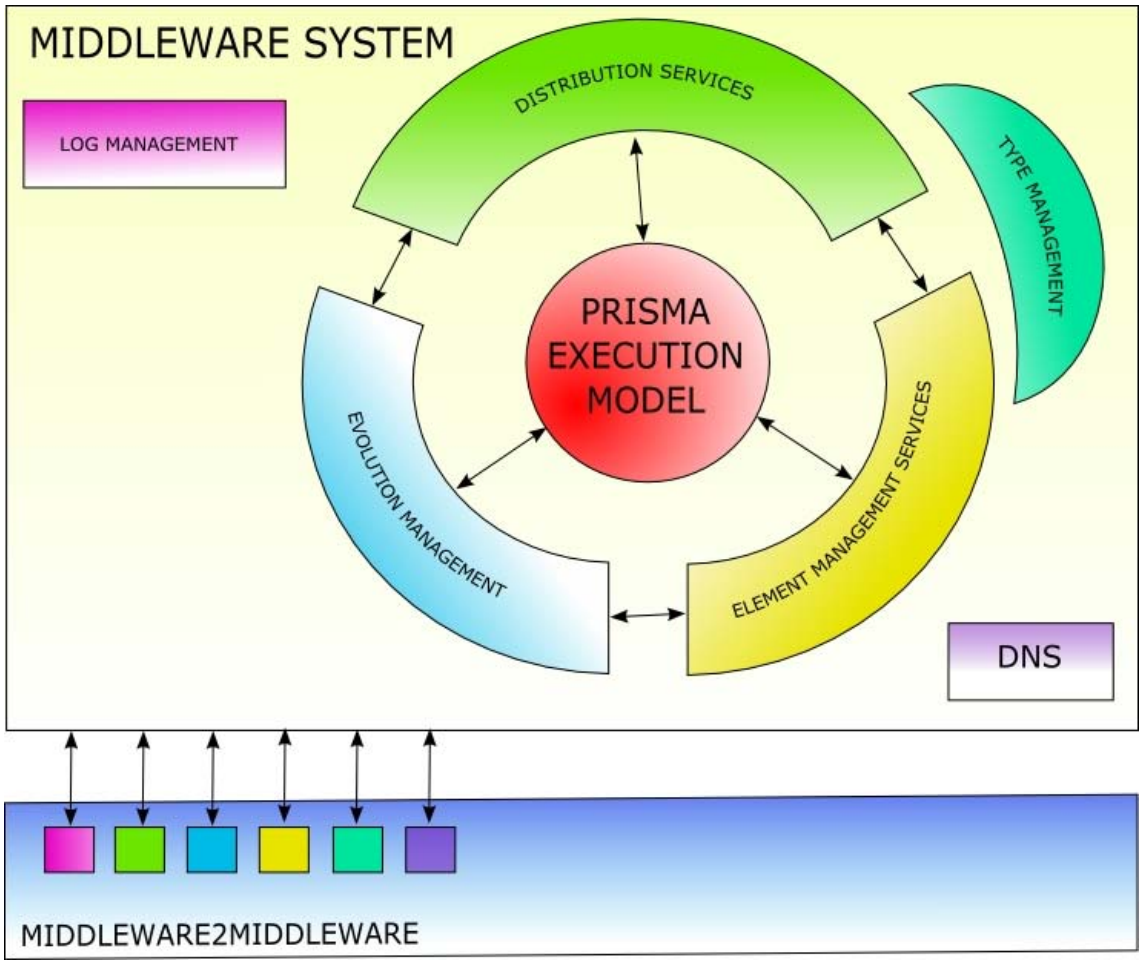



**Figura 26: Capas de Middleware System**

Las capas en las que se divide `MiddlewareSystem` se explican a continuación:

- ***Element Management Services*:** Esta es la capa encargada de la instanciación de los elementos arquitectónicos, de su configuración e interconexión. Guarda la información de los elementos arquitectónicos que se crean, así como de los *attachments* y *bindings*. En la Figura 27 se muestra un esquema con las diferentes clases que forman el *Element Management Services*.

- ***Distribution Services***: Es la que encapsula las operaciones relacionadas con la movilidad de elementos arquitectónicos. Como se ha comentado anteriormente, el middleware proporciona mecanismos para la reconfiguración dinámica de la arquitectura, pues bien, la capa *Distribution Services* es la que implementa los servicios necesarios para mover de forma transparente un elemento arquitectónico de un *middleware* a otro, donde proseguirá su ejecución.

- ***Evolution Management*:** Capa que encapsulará las operaciones para dotar al modelo PRISMA en ejecución de propiedades de evolución dinámica.

- ***Type Management*:** Capa encargada de gestionar los tipos PRISMA. Se encarga de localizar los ensamblados en que están definidos los tipos PRISMA entre los *middlewares* y transferirlos a los middlewares que los puedan necesitar. Así mismo mantiene una lista con los tipos con los que está trabajando el *middleware*.

- ***DNS*:** Realiza la función de DNS y es útil para el resto de capas a la hora de localizar la ubicación en la que se están ejecutando los elementos arquitectónicos.

- ***Log Management*:** La finalidad de esta capa es proporcionar un información al usuario de lo que está ocurriendo en el *middleware*. Mediante ella el resto de capas muestran mensajes por consola.

- ***Middleware2Middleware:*** Capa encargada de establecer la comunicación entre los diferentes *middlewares* distribuidos que componen un entorno de ejecución PRISMA. La capa actúa como un *proxy* en el cual solo se muestran los servicios de las diferentes capas cuya invocación tenga sentido hacerla desde otro *middleware*.

Como se puede ver, cada una de las capas tiene una finalidad específica que la diferencia del resto.

Con todo esto, dada una aplicación PRISMA, se pueden distinguir tres clases de comunicaciones:

- Las realizadas por parte de los elementos arquitectónicos de la aplicación al *Middleware*, en las que se solicitan servicios de movilidad o de replicación. Para ello, todo elemento arquitectónico es capaz de obtener una referencia al *middleware* en el que se está ejecutando. Esta referencia es dinámica, pues como se ha comentado, un elemento arquitectónico puede moverse a través de los nodos donde se ejecutan los *middlewares* que componen el entorno de ejecución.

- Las comunicaciones entre los distintos elementos arquitectónicos, según lo requieran las especificaciones del modelo (solicitud de servicios unos a otros). Este tipo de comunicaciones se realiza mediante *attachments* y *bindings*.

- Las realizadas entre los distintos *Middlewares* para averiguar las localizaciones de los distintos elementos, mover elementos, transferir tipos, evolucionar tipos y arquitecturas, actualizar el

estado, etc. Estas comunicaciones se hacen mediante la capa *Middleware2Middleware*.

Debido a la naturaleza distribuida del modelo, la implementación de las comunicaciones se ha realizado con la tecnología .Net Remoting [Scott03] ya que proporciona implementados la mayor parte de los mecanismos para acceder remotamente a objetos así como para serializarlos y moverlos de una máquina a otra.

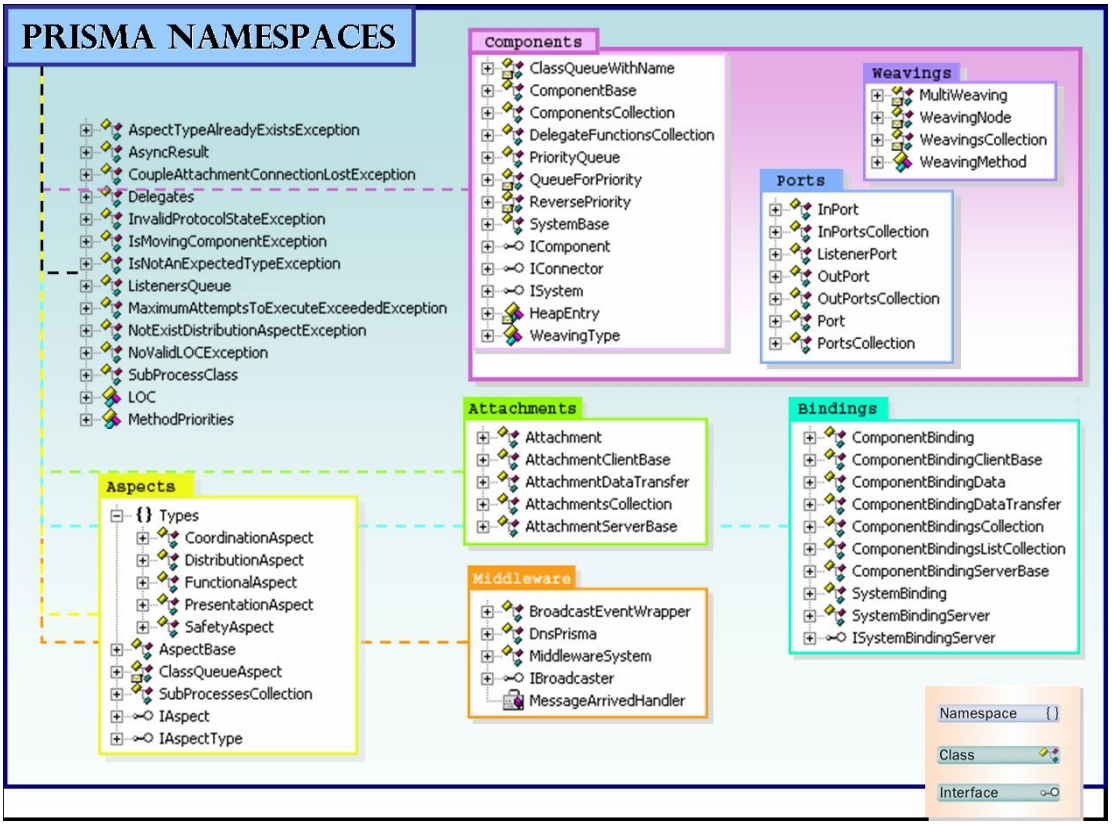



**Figura 27: Clases de ElementManagement Services**

### 3.3.2. Movilidad y comunicación distribuida en PRISMANET

Puesto que la finalidad del presente proyecto es ampliar el *middleware* PRISMANET en materia de movilidad y comunicación distribuida, es necesario presentar como se han implementado estas características. Por ello, a continuación se introducirán brevemente los modelos de comunicación distribuida y movilidad tal y como están implementados en PRISMANET para, de esa forma, establecer la base necesaria para entender las modificaciones presentadas en los siguientes capítulos. No obstante, si el lector está interesado en más características del *middleware* PRISMANET puede consultar [Cos05].

#### *3.3.2.1. Comunicación distribuida entre elementos arquitectonicos*

Como se ha explicado en la sección 3.1.2.3, la comunicación distribuida en el modelo PRISMA viene dada por los *attachments* y los *bindings*. Los *attachments* comunican elementos arquitectónicos (componentes y conectores), vistos como una caja negra, mediante la conexión de sus puertos. Los *bindings* comunican elementos arquitectónicos ubicados en el interior de un sistema con los puertos externos del sistema, para que de esa forma puedan externalizar su interfaz.

- **Attachments**

En el modelo PRISMA, la comunicación entre componentes y conectores se realiza a través de los *attachments*. Cada *attachment* conecta dos elementos (componente y conector) y se encarga de llevar peticiones desde el uno al otro, actuando como si fuera un canal de comunicación. De esta forma, los *attachments* permiten que las comunicaciones distribuidas sean transparentes para componentes y conectores. Es más, los *attachments* abstraen a los elementos arquitectónicos que conectan de las peculiaridades de la comunicación, de forma que, los elementos arquitectónicos conectados no se conocen entre si, y es el *attachment* quien conoce a ambos y los comunica bidireccionalmente. La comunicación es bidireccional, ya que por una parte se encargan de hacer accesible al componente desde otras ubicaciones remotas (comportamiento servidor), y por otra, se encargan de establecer la comunicación hacia los componentes remotos (comportamiento cliente).

Los elementos arquitectónicos a conectar pueden encontrarse en la misma máquina o en distintas máquinas, en cuyo caso la comunicación es distribuida. Además, éstos pueden reconfigurarse dinámicamente, por lo que los *attachments* se han diseñado de forma que pueden actualizar las referencias a los elementos arquitectónicos en tiempo de ejecución.

Por tanto, los *attachments* conectan puertos. En la implementación, los puertos de un elemento arquitectónico están divididos en dos partes: Los puertos de salida y los puertos de entrada. El elemento arquitectónico dejará sus peticiones en los puertos de salida (`OutPorts`)(comportamiento servidor) y recibirá peticiones en los puertos de entrada (`InPorts`)(comportamiento cliente).

De forma análoga, un *attachment* tiene dos comportamientos claramente diferenciados: comportamiento cliente y comportamiento servidor. El comportamiento cliente consiste en escuchar periódicamente las peticiones depositadas en el puerto de salida que está escuchando, para lo cual tiene un hilo de ejecución para tal propósito. El comportamiento servidor consiste en reenviar las peticiones recogidas al puerto de entrada.

Como se ha comentado, los *attachments* pueden conectar elementos arquitectónicos distribuidos, por lo que se ha optado por dividir el concepto en dos entidades, cada una de las cuales representa a un participante de la comunicación (componente o conector) desde el mismo *middleware* en que se están ejecutando. Estas entidades son instancias de la clase `Attachment` y podrán actuar como cliente o servidora. Para separar ambos comportamientos se ha optado por implementarlos en clases separadas: `AttachmentClientBase` para el comportamiento cliente, y `AttachmentServerBase` para el comportamiento servidor.

Por tanto, un *attachment* PRISMA se corresponde con una pareja de instancias de la clase `Attachment`, cada una de ellas asociada a uno de los elementos arquitectónicos involucrados en la comunicación, cada una de las cuales está formada por la agregación de las clases que implementan el comportamiento cliente y el comportamiento servidor.

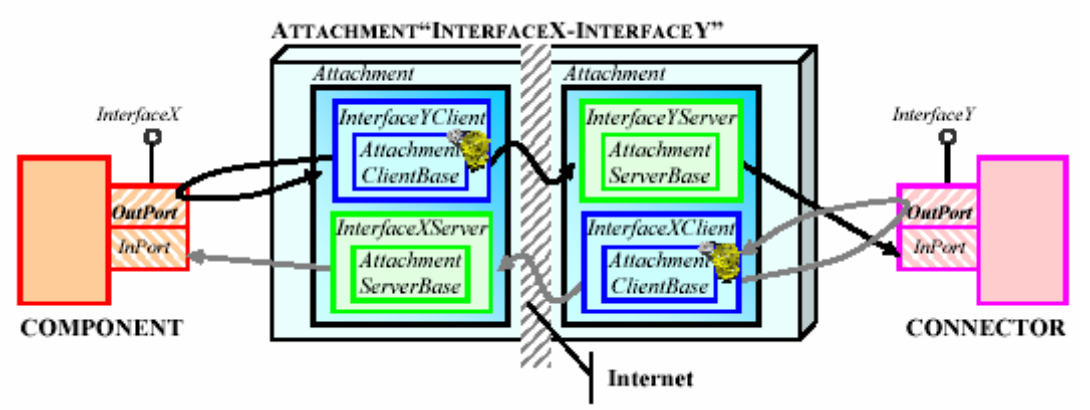



**Figura 28: Modelo de ejecución de los *attachments***

En la Figura 28 se muestra el modelo de ejecución de los *attachments*. Como se puede ver, el *attachment* comunica dos puertos con una interfaz que no tiene por que ser la misma, pero ha de ser compatible. El *attachment* está formado por dos instancias de la clase *attachment* que se encontrarán ejecutándose en el mismo *middleware* en que se encuentre el elemento arquitectónico al que representan. Cuando el componente deje una petición en el `OutPort`, está será recogida por la parte cliente del *attachment* que tiene asociado. Esta parte cliente representa a la interfaz del conector, el elemento al que conecta y reenviará la petición a la parte servidora del *attachment* ubicado en el lado del conector. Esta parte redirigirá la petición al `InPort` del conector. La comunicación a la inversa sigue el mismo planteamiento.

Esta estructura de los *attachments* con parte cliente y parte servidora funcionando de forma independiente en cada uno de los lados de la comunicación se adapta a la reconfiguración de la arquitectura. Si uno de los elementos arquitectónicos que se comunican cambia su ubicación en tiempo de ejecución, bastará con avisar al *attachment* de la otra parte de este hecho para

que actualice su referencia a la parte que se ha movido. Esto se hace gracias a que cada par de *attachment* tiene información sobre la ubicación del otro.

### ○ Bindings

En PRISMA, un *binding* es la entidad encargada de establecer la comunicación entre un elemento arquitectónico y el sistema que lo encapsula. Asocia un puerto de un sistema con un puerto de un componente o conector, actuando como un redirector de peticiones, de forma transparente para los componentes externos al sistema. Cuando un puerto de un sistema recibe una petición de servicio, y éste tiene un *binding* asociado, la petición es reenviada al elemento arquitectónico interno, que puede estar en otro middleware distinto al que se encuentra el sistema. En el sentido contrario, cuando un elemento arquitectónico interno al sistema desea comunicarse con un componente externo, tras dejar la petición en su puerto de salida, el *binding* recoge dicha petición y la reenvía al puerto de salida del sistema.

El diseño se ha realizado de forma similar a los *attachments*. Un *binding* está formado por dos partes, cada una de las cuales reside en la misma máquina que el elemento arquitectónico al que representa. La diferencia radica en que la parte que representa al sistema, se ha diseñado de forma especial, de forma que es común para todos los *bindings* de dicho sistema.

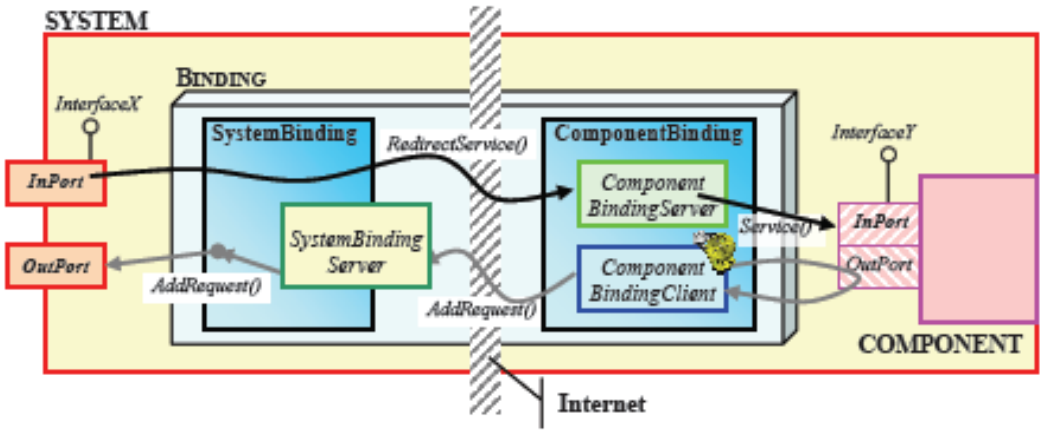



**Figura 29: Modelo de ejecución de los *bindings***

La parte que reside en el lado del elemento arquitectónico interno al sistema, a la que se ha denominado `ComponentBinding`, conserva el mismo diseño de los *attachments*: está formada por un objeto que implementa el comportamiento cliente (de la clase `ComponentBindingClient`) y otro objeto con el comportamiento servidor (de la clase `ComponentBindingServer`). El objeto cliente se encarga de recibir las peticiones de servicio del puerto de salida del elemento arquitectónico y redirigirlas hacia la parte residente en el lado del sistema. El objeto servidor se encarga de recibir las peticiones enviadas por la parte del *binding* del sistema y redirigirlas al puerto de entrada del elemento arquitectónico interno.

La parte del *binding* que reside en el lado del sistema, a la que se ha denominado *System Binding* (clase `SystemBinding`), realiza las funciones complementarias al `ComponentBinding`: Por una parte, recoge todas las peticiones de servicio salientes de los componentes internos, enviadas a través de los `ComponentBinding`, y las reenvía a los puertos de salida correspondientes. Para ello el *System Binding* proporciona una parte servidora (`SystemBindingServer`) que permitirá ser accedido remotamente por los `ComponentBindingClient` asociados a elementos arquitectónicos remotos. Por otra parte, inicializa adecuadamente los puertos de entrada para que redirijan las peticiones de servicio entrantes hacia los componentes internos. Nótese como en este caso, al contrario que los *attachments*, las peticiones que lleguen por los `InPorts` del sistema son redirigidas a los `InPorts` del elemento arquitectónico interno y viceversa. Lo mismo para los `OutPorts`. En cierto modo el *binding* lo que está haciendo es trasladar la petición de un puerto a otro.

Pero el `SystemBinding` también ejerce la función de gestor de *bindings*. De hecho entre sus labores también se encuentra la gestión de los *bindings* del sistema, encargándose de su creación y eliminación. Existe sólo una instancia de `SystemBinding` por cada sistema, y es la encargada de crear los respectivos `ComponentBinding` por cada *binding* definido. También se encarga de actualizar las referencias en caso de movilidad de alguno de los elementos arquitectónicos internos que tenga un *binding* asociado. El *middleware* se encarga de avisar al `SystemBinding` de este hecho para que actualice la referencia al `ComponentBinding` del elemento que se ha movido. Cuando es el sistema el que se mueve, el `SystemBinding` avisa a todos los componentes de este hecho para que actualicen la referencia a él.

El modelo de ejecución de los *bindings* es semejante al de los *attachments*: Cuando el `ComponentBinding` detecta una petición de servicio en el puerto de salida del elemento arquitectónico al cual está escuchando, invoca a través de la referencia que tiene al método `AddRequest()` del `SystemBinding` proporcionándole el nombre del `ComponentBinding` del que proviene la petición. De esta forma el `SystemBinding` identifica a que puerto ha de redirigir la petición: A aquel que esté relacionado con ese `ComponentBinding`. Así, se obtiene el puerto del sistema al cual redirigir la petición y se invoca al método `AddRequest()` del puerto de salida correspondiente del sistema, pasándole los parámetros recibidos desde el `ComponentBinding`. Como se puede ver, se ha redirigido la petición hacia el puerto del sistema.

Por otra parte, la redirección de las peticiones de servicio entrantes hacia los `ComponentBindings` se ha realizado a través de un atributo de los puertos de entrada: `componentBinding`. Cuando se asocia un *binding* a un puerto, el objeto `SystemBinding` inicializa su atributo `ComponentBinding` con una referencia al `ComponentBindingServer` (que puede ser remoto o local). En tiempo de ejecución, cuando llega una nueva petición a un sistema, el puerto de entrada crea un delegado hacia el método `RedirectService` del `ComponentBindingServer` y lo almacena en la cola del sistema, con la finalidad de preservar el orden de

llegada de las peticiones. Posteriormente, al ser ejecutado dicho delegado por el sistema, la petición de servicio será reenviada al puerto de entrada del componente asociado. Este diseño presenta el inconveniente de que ha precisado modificar el puerto de entrada del sistema para que también se almacenase el *binding* asociado, resultando en una pérdida de independencia por parte de los puertos. Sin embargo, en los siguientes capítulos se mostrará como esto ha sido modificado para preservar la independencia de los puertos sin perder el orden de llegada de las peticiones.

#### *3.3.2.2. Movilidad en PRISMANET*

Otra característica de PRISMANET es que da soporte a la movilidad de elementos arquitectónicos entre los diferentes *middlewares* que conforman el entorno de ejecución. Los componentes son entidades software con una funcionalidad determinada que heredan de la clase `ComponentBase`, que representa a los elementos arquitectónicos y reciben la semántica por la incorporación de aspectos. Al igual que en el caso de la comunicación distribuida, las características móviles son proporcionadas por la tecnología *.Net Remoting*.

Todos los elementos que conforman los componentes están marcados con el atributo `Serializable` para denotar que en un momento dado se pueden serializar a otro *middleware*. No obstante *.Net Remoting* no proporciona *strong mobility* y, por lo que, cuando un elemento arquitectónico se serializa lo hace previo una parada del mismo, procesando antes todas las peticiones pendientes. Al llegar al destino, recuperará su estado e iniciará la ejecución desde el principio.

- **Representación de las localizaciones**

Antes de mostrar el modelo de ejecución de loa movilidad, es necesario explicar de qué forma se expresan las localizaciones en PRISMANET. Como se ha explicado el entorno de ejecución PRISMA está formado por un conjunto de *middlewares* trabajando de forma colaborativa, cada uno de esos *middlewares*, o localizaciones donde la ejecución de los elementos arquitectónicos puede tener lugar se representa en PRISMA con el tipo `LOC`. De esa forma, se crea una instancia de `LOC` para cada *middleware* y es con esas instancias con lo que los elementos arquitectónicos tratan, ya sea para localizarse en el momento de la creación o en el momento que se han de mover a otra máquina. La estructura `LOC` garantiza que una URI (*Uniform Resource Identifier*) es válida, es decir que existe y que está ejecutando PRISMANET Server y tiene los servicios necesarios para la gestión de la localización.

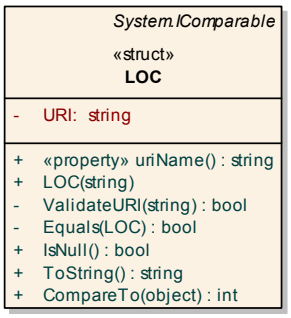



**Figura 30: La estructura LOC**

El tipo `LOC` se ha implementando haciendo uso de una estructura homónima en C# (ver Figura 30) que tiene como propiedad fundamental la URI a la que representa y como servicios asociados a esa propiedad, métodos para su validación, comparación y representación. Todos los servicios del *middleware* que trabajan con localizaciones hacen uso de los `LOC` definidos para representar localizaciones en el entorno de ejecución.

### ○ Modelo de ejecución de la movilidad de los elementos arquitectónicos

En lo que respecta al modelo de ejecución de la movilidad de los elementos arquitectónicos, PRISMANET delega esta funcionalidad a la capa *Distribution Services*, que será la encargada de proporcionar servicios de serialización/deseraialización de elementos arquitectónicos para la movilidad de un *middleware* a otro. El intermediario en las operaciones de movilidad será un Aspecto de Distribución que albergará la lógica para invocar los servicios de la capa *Distribution Services* que mueven al componente.

Al ser la movilidad iniciada desde un aspecto, es posible que el elemento arquitectónico decida cuando moverse, tan solo ha de establecer *weavings* con el Aspecto de Distribución y otros aspectos. De esa forma, el aspecto podrá lanzar una orden de movilidad en el momento que decida oportuno: Si está diseñado para que cuando se produzcan determinadas circunstancias se mueva a otra máquina, o si simplemente, el usuario solicita la movilidad haciendo uso del Aspecto de Presentación. También, al proporcionar los servicios de movilidad a través de un aspecto, es posible publicar dichos servicios a través de los puertos del elemento arquitectónico, de forma que, otros elementos arquitectónicos puedan moverlo.

En cualquier caso, el Aspecto de Distribución establece un puente entre el elemento arquitectónico y la capa que ofrece los servicios de distribución en el *middleware*, por lo que todo elemento arquitectónico que quiera incorporar características de movilidad, ha de importar un Aspecto de Distribución, lo que es coherente con el modelo PRISMA.

La capa *Distribution Services* tiene dos servicios fundamentales para mover elementos arquitectónicos: `Move` y `TransferComponent`. La diferencia radica en que uno es el que uno se ejecuta en el *middleware* origen y el otro en el *middleware* destino:

- `Move(place, sourceObject)`: Servicio que se ejecuta en el *middleware* origen de la movilidad. Es el que invoca el aspecto de distribución, pasándole el destino de la movilidad en un objeto de tipo `LOC` (`place`) y el nombre del elemento arquitectónico que se va a mover (`sourceObject`). Su función es hacer las tareas pertinentes previas a la movilidad, obtener una referencia al *middleware* destino e invocar al servicio `TransferComponent` de éste.

- `TransferComponent(targetComponent, attDataList, binDataList)`: Servicio que se ejecuta en el *middleware* destino de la movilidad. Es el que hace realmente la serialización. Como se ha visto, se invoca desde la capa *Distribution Services* del *middleware* origen, en concreto desde el servicio Move, que le pasa la referencia al elemento arquitectónico a serializar (`targetComponent`) y las listas de *attachments* (`attDataList`) y *bindings* (`binDataList`) para regenerar sus canales de comunicación. Se encarga de recibir el elemento arquitectónico móvil y realizar las tareas pertinentes para ponerlo en ejecución en el nuevo *middleware*.

A continuación se explica paso a paso el funcionamiento del modelo de ejecución de la movilidad. En la Figura 31 se muestran las diferentes entidades que interactúan en el proceso de movilidad de un elemento arquitectónico en PRISMANET.

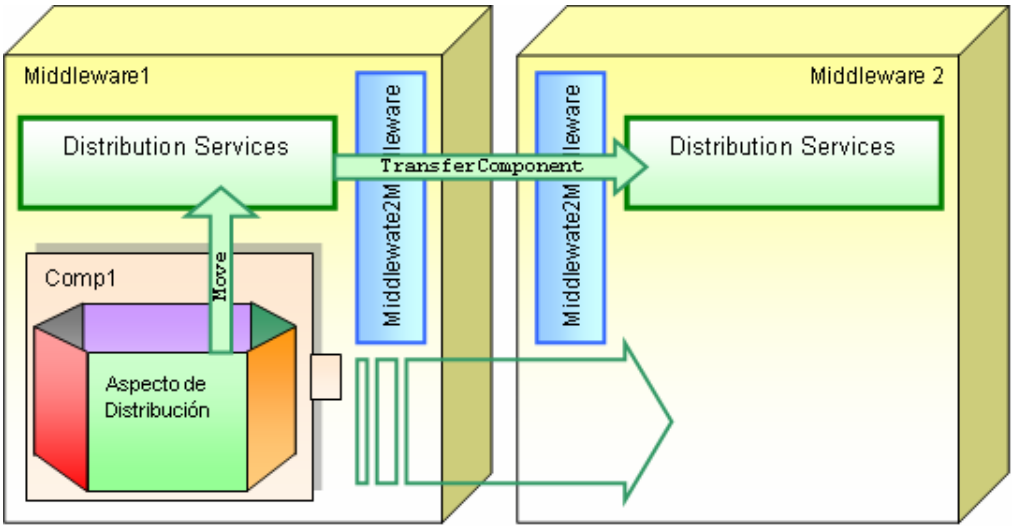



**Figura 31: Modelo de ejecución de la movilidad**

Cuando un elemento arquitectónico ha de moverse, ya sea por decisión propia o porque así se lo solicitan las operaciones del Aspecto de Distribución.

Como se ha comentado, todo elemento arquitectónico móvil ha de tener un aspecto de distribución que le proporciona la semántica de la movilidad. El aspecto de Distribución, al recibir la petición de movilidad la redirige a la capa *Distribution Services*, pasándolo como parámetros la localización que habrá recibido de la petición de movilidad y el componente a mover.

La capa *Distribution Services* recibe la petición del `Move` y lo primero que hace es comprobar que el elemento que le han pasado para que mueva es móvil. Esto se comprueba mirando si tiene aspecto de distribución. Si esta condición se cumple, y el destino es otro *middleware* diferente al actual, comienza las acciones necesarias para la movilidad. Así, se obtiene una referencia al elemento arquitectónico a mover y se marca su *flag* `IsMoving`, para que el elemento arquitectónico sea consciente que se va a mover. Tras eso se obtiene la lista de *attachments* y *bindings* conectados, Una vez se tienen estas dos listas, ya se dispone de la información necesaria para mover al elemento arquitectónico, por lo que se prepara al elemento para detener su ejecución.

El elemento arquitectónico cuando recibe la orden de detener su ejecución detiene la entrada de peticiones desde los puertos y procesa todas las peticiones que se encontrasen en ese momento en la cola del elemento arquitectónico. Una vez se han ejecutado todas las peticiones, el elemento arquitectónico ya está listo para detener su ejecución, y eso es lo que hace.

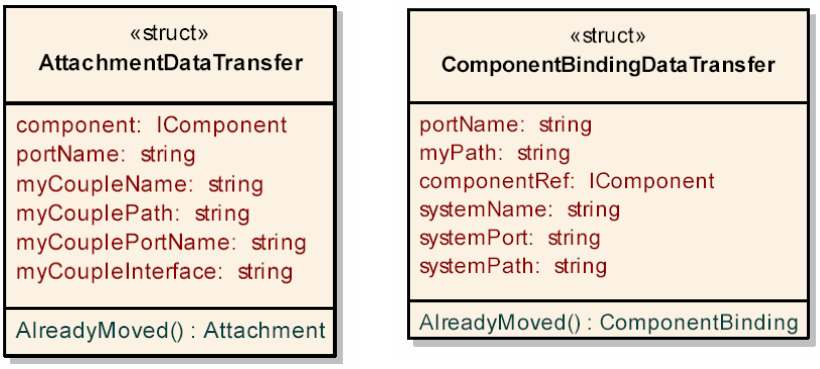



**Figura 32: AttachmentDataTransfer y ComponentBindingDataTransfer**

Una vez el elemento arquitectónico detenido y teniendo la información de los *attachments* y *bindings* guardada en sendas listas, se procede a su detención. Para cada *attachment* o *binding* se guarda la información en una clase especial para la transferencia mediante las estructuras `AttachmentDataTransfer` y `ComponentBindingDataTransfer` respectivamente (ver Figura 32). Tras detener y guardar la información sobre *attachments* y *bindings* conectados al elemento que se va a mover, estos se eliminan del *middleware* las partes asociadas al lado del elemento móvil, pues con la información de los `AttachmentDataTransfer` y los `ComponentBindingDataTransfer` se volverán a crear en el destino. Recuérdese que tanto *attachments* como *bindings* están formados por dos partes, una a cada lado de la comunicación. Durante este proceso se detienen ambos lados pero tan solo se elimina la entidad (`Attachment` o `ComponentBinding`) de la parte móvil.

Con las estructuras de información sobre los *attachments* y *bindings* y la referencia al elemento arquitectónico ya detenido y sin *attachments* conectados a sus puertos, se obtiene una referencia al *middleware* destino, y a través de la capa *Middleware2Middleware* se invoca a su capa *Distributed Services*, en concreto al servicio `TransferComponent`. Como se ha comentado, es este servicio el que en realidad serializa al elemento arquitectónico, pues recibe su referencia como parámetro. Por ello, lo que hace el servicio es agregar el nuevo componente recibido al *middleware* (a través de la capa *Element Management Layer*). A continuación utiliza las listas de información de *attachments* y *bindings* para volverlos a crear en el *middleware* actual. Estos al crearse de nuevo manteniendo la información que tenían en origen, avisan al otro lado de su nueva ubicación al tiempo que ponen en marcha el conjunto del *attachment* o *binding*. El último paso en la máquina destino es poner en marcha el elemento arquitectónico para que continúa funcionando. El elemento mantiene el estado de sus aspectos tras la movilidad, pero la ejecución comienza desde un estado seguro (el elemento arquitectónico se había parado) y no desde el punto que estaba ejecutando cuando se le ordeno moverse, es por eso que la movilidad es débil.

En el *middleware* origen, una vez se tiene constancia de que el proceso ha funcionado correctamente en el destino, se borra el elemento arquitectónico de ese *middleware*. Como se puede observar, en realidad la movilidad se está simulando, pues del modo que se está haciendo lo que se está pasando es una copia del elemento arquitectónico, que sigue existiendo en el *middleware* origen (de ahí la necesidad de eliminarlo al finalizar).

Si algo fuese mal, el método `Move` que es el que orquesta todo el proceso implementa un código de seguridad que deshace los cambios: vuelve a crear los *attachments* y *bindings* en local desde los `AttachmentDataTransfer` y `BindingsDataTransfer` generados, poniéndolos en marcha y finalmente, iniciando de nuevo la ejecución del elemento arquitectónico.

CAPÍTULO IV

# DISTRIBUCIÓN EN PRISMA

## Contenidos del capítulo

# DISTRIBUCIÓN EN PRISMA

La finalidad del presente proyecto es proporcionar un marco para el desarrollo de aplicaciones distribuidas y móviles para un modelo arquitectónico orientado a aspectos. Como se ha explicado en el capítulo anterior, el modelo arquitectónico PRISMA integra la aproximación del desarrollo de software orientado a aspectos (DSOA) y el desarrollo de software basado en componentes (DSBC). Además, el modelo PRISMA está preparado para soportar configuraciones de naturaleza distribuida y gracias a su capacidad para la reconfiguración dinámica de sus elementos arquitectónicos, es capaz de modelar la distribución y la movilidad.

Por otro lado, en trabajos previos se ha proporcionado un marco de ejecución para los modelos PRISMA en el entorno PRISMANET. Es más, existe una herramienta para el modelado gráfico de arquitecturas PRISMA una basada en DSL-Tools. La herramienta facilita al usuario la labor de especificar modelos mediante una interfaz gráfica y es capaz de generar automáticamente las librerías necesarias para poner en ejecución configuraciones del modelo implementado en el middleware PRISMA.

Es por ello que PRISMA se presenta como el entorno ideal para el desarrollo de aplicaciones distribuidas y móviles. En concreto, la implementación del *middleware* PRISMANET servirá como punto de partida para este proyecto. La finalidad será dotar al *middleware* los mecanismos necesarios para conseguir la correcta traza entre los conceptos de distribución y movilidad expresados por el modelo arquitectónico y la plataforma de implementación de .NET.

A continuación, se presentan las diferentes modificaciones que se han llevado a cabo sobre la anterior implementación del *middleware* para dar soporte a las necesidades del modelo y, en general, a la comunicación distribuida y la movilidad. Del mismo modo, se presentarán las ampliaciones de dicho middleware para proporcionar mecanismos para la gestión del entorno distribuido y para dar soporte a las transacciones PRISMA a través de dicho entorno.

## 4.1. Ampliación de la implementación del modelo de comunicación distribuida

Un primer paso hacia la consecución del objetivo marcado supondría mejorar los mecanismos de comunicación distribuida existentes en el *middleware* PRISMANET. Tal y como se ha descrito en el capítulo anterior, en el modelo PRISMA los encargados de gestionar la comunicación distribuida entre los elementos arquitectónicos son los *attachments* y los *bindings*. Los elementos arquitectónicos delegan toda la lógica de dicha comunicación al canal de comunicación, garantizando así su independencia.

Los *attachments* y *bindings* son los que conocen a los elementos arquitectónicos que se comunican, y a los puertos de dichos elementos que en realidad están conectando. Los *attachments* y *bindings* han de ser capaces de adaptarse a los cambios que se produzcan en la configuración de la arquitectura. Es decir, si un elemento arquitectónico se mueve de una máquina a otra, los *attachments* y *bindings* conectados a él han de modificar sus referencias al elemento arquitectónico móvil para garantizar la comunicación distribuida tras la movilidad.

Cada vez que un elemento va a moverse, este se lo notifica a la capa *DistributionServices* del middleware para que le prepare el terreno. El middleware localiza los *attachments* y *bindings* a los que está conectado y los detiene para proceder con la movilidad. Al detenerse, los *attachments* y *bindings* dejan de enviar peticiones a los puertos de ambos lados. Una vez finalizada la movilidad del elemento arquitectónico, el middleware de la máquina destino se encarga de poner en marcha de nuevo los canales de comunicación hacia el elemento que acaba de recibir. Para ello, el middleware localiza todos los *attachments* y *bindings* implicados y los va reanudando mediante su servicio `Start()`. Se ha de tener en cuenta que los *attachments* y *bindings* no se destruyen tras un proceso de movilidad, sino que se reconfiguran para adaptarse a la nueva situación.

No obstante, el hecho de que sea el *middleware* quien deba localizar las partes de los diferentes *attachments* y *bindings* para la detención previa a la movilidad y para reanudar una vez el proceso haya finalizado, resta independencia al canal de comunicación. Por otro lado, el que el middleware haya de localizar las diferentes partes del *attachment* entre el entorno de ejecución conlleva a un sobrecoste, al tener que hacer uso del DNS para localizar las partes.

Esto tiene sentido, pues en ambos casos (*attachments* y *bindings*) el concepto está representado por dos instancias (dos instancias de la clase `Attachment` para los *attachments* y una instancia de `ComponentBinding` y otra de `SystemBinding` para los *bindings*) que se conocen mutuamente. Puesto que ambas tienen constancia de la existencia de la otra parte, no es necesario que el

*middleware* interceda para localizar las partes, sino que ellas mismas tienen la información necesaria para hacerlo.

Por otro lado, los canales de comunicación están continuamente pendientes de lo que los elementos arquitectónicos a los que están conectados dejan en sus `OutPorts`, lo cual, como se verá, puede suponer una sobrecarga innecesaria en el sistema.

A continuación se explicará como se ha modificado el modelo de ejecución de los *attachments* y *bindings* para dotarlos más independencia y, al mismo tiempo, liberar al middleware de las tareas relacionadas con su gestión. Finalmente se añadirá una subsección para explicar como se ha optimizado el consumo de recursos de los *attachments* y *bindings* mediante la aplicación de patrones de programación.

### 4.1.1. Modificación del modelo de ejecución de los attachments

En primer lugar se detallará como se ha modificado el modelo de ejecución de los *attachments*. Las modificaciones que se presentan a continuación parten del modelo del ciclo de ejecución de *attachments* explicados en el capítulo anterior.

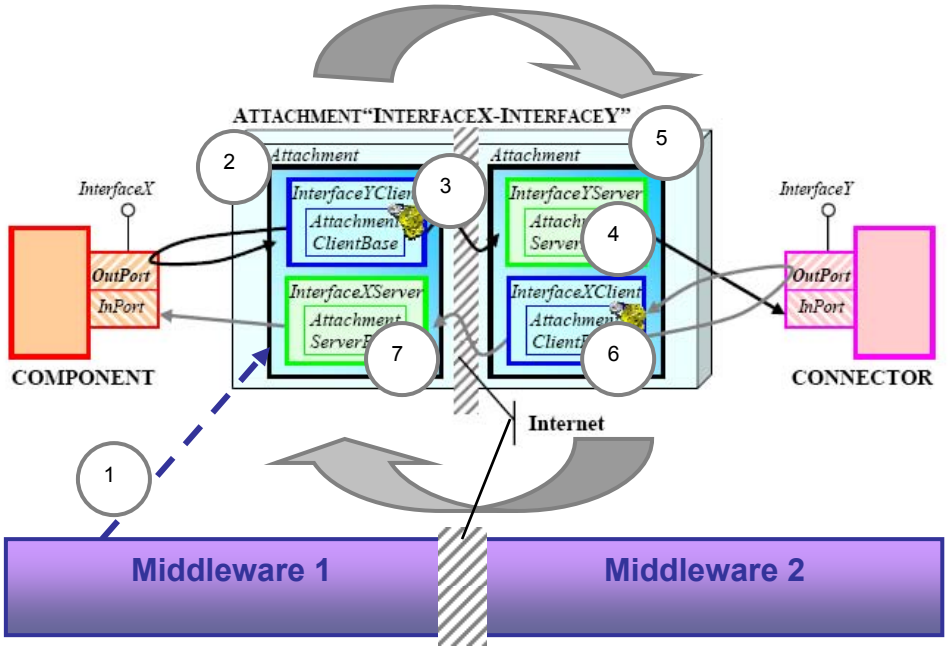



**Figura 33: Nuevo ciclo de ejecución de los *attachments***

Un *attachment* (Figura 33) está formado por dos partes, una por cada elemento arquitectónico que conecta. Cada una de estas partes está formada por otras dos partes: una parte cliente y otra parte servidora. La parte cliente de un *attachment* coge las peticiones encoladas en los `OutPorts` del elemento arquitectónico de su parte y las envía a la parte servidora del *attachment* de la

otra parte. La parte cliente del *attachment* es la que tiene en realidad la referencia o el *proxy* a la otra parte. Puesto que la parte cliente de un *attachment* conoce a la parte servidora de la otra, y la parte servidora, a su vez, tiene una referencia mediante la propiedad `parent` a la instancia de la clase *attachment* que le encapsula. La clase *attachement* tiene referencias tanto a su parte servidora como a su parte cliente.

Como se ha constatado en el párrafo anterior todos los elementos involucrados en el canal de comunicación están relacionados entre si de una forma o de otra. Como se comentó en el capítulo anterior, un *attachment* está en realidad formado por dos partes, una por cada elemento arquitectónico que comunican (*AttachmentInterfaceX* y *AttachmentInterfaceY* ). Estas partes representan las interfaces de los puertos a los que están conectados (*InterfaceX* e *InterfaceY*), que puede ser diferente, pero ha de tener servicios en común. Cada uno de los lados del *attachment* está a su vez formado parte cliente que recibe peticiones (*Client*) y una parte servidora que las envía (*Server*). De esa forma, la parte Cliente del *attachment* de un lado (*InterfaceX*) está conectada a la parte servidora del otro (*InterfaceY*). Lo mismo ocurre en el sentido contrario. De esa forma es posible establecer un ciclo de llamadas que pase por todos los elementos, de forma que, cada elemento que la reciba realice las acciones que debe hacer y propague la llamada al siguiente elemento. Para ello, se ha establecido el siguiente orden en función de las relaciones entre los elementos:

*AttachmentInterfaceX* → *AttachmentInterfaceYClient* →
*AttachmentInterfaceYServer* → *AttachmentInterfaceY* →
*AttachmentInterfaceXClient* → *AttachmentInterfaceXServer* →
*AttachmentInterfaceX*

El ciclo comienza y termina en el mismo punto, de ese modo, cuando vuelva al punto de partida, se detectará y se dará por finalizado. En la Figura 33 se explican detalladamente los pasos que sigue el nuevo ciclo de ejecución propuesto. La explicación se centrará en la puesta en marcha de los *attachments* (`Start()`), pues tanto la detención (`Stop()`) como el aborto (`Abort()`) se realizarán de forma similar.

En la Figura 33, se muestra el nuevo modelo de ejecución de un *attachment*. El *attachment* conecta un puerto con interfaz "*InterfazX*" de un componente con un puerto de interfaz "*InterfazY*" de un conector. En cursiva se muestran los nombres de las clases que definen cada uno de los lados del *attachment*. Nótese como en el caso de la parte del componente (*AttachmentInterfaceX*), la parte cliente sólo proporciona servicios del *InterfazY*, que es la interfaz que le proporciona la parte servidora del lado del conector (*AttachmentInterfazY*). De esta forma, las peticiones que no pertenezcan a la interfaz del servidor son descartadas antes de enviarse por la red; en cualquier caso no serían servidas. Lo mismo ocurre en la dirección contraria: en el lado del conector, la parte cliente solo transmite peticiones de servicios de la *InterfazX*, que es la que puede ser servida en el otro lado.

Supóngase que se parte de una situación en que un *attachment* se encuentra detenido, por ejemplo, tras la movilidad de alguno de los elementos arquitectónicos que está comunicando (ver Figura 33). Por tanto, ambas instancias de la clase *attachment* que representarán el *attachment* PRISMA en la ejecución tendrán sus respectivos *flags* `executing` puestos a `false`. En el caso del ejemplo de la Figura 33, el *attachment* entre los puertos con *InterfaceX* e *InterfaceY* está detenido, por ello, tanto *AttachmentInterfaceX* como *AttachmentInterfaceY* , es decir, ambos lados del *attachment*, se encuentran con el *flag* `executing` a `false` y con los hilos de ejecución de su parte cliente detenidos. En ese punto el *middleware* indica a una de las partes (por ejemplo a *AttachmentInterfaceX*, la que está en el lado del componente) que se ha de reanudar, para ello invoca a su servicio `Start()` (1).

Al procesar la petición del servicio, el *AttachmentInterfaceX* lo primero que hace es comprobar que su *flag* `executing` está puesto a falso, indicando así que la instancia de `Attachment` está parada. A continuación la parte cliente del *attachment* actualizará su referencia a la pareja. Como se ha comentado, la parte cliente del *attachment* es quien tiene en realidad la referencia a la otra parte, en concreto a la parte servidora de la otra parte. Actualizar la referencia es importante, porque puede que durante el tiempo que el *attachment* ha estado parado, él o su pareja se hayan movido. Para ello usa el método del cliente `ConnectToCoupleAttachment` que asigna el atributo del cliente con la referencia al servidor, que puede ser local o remota. Para obtener la referencia usamos el método `ConnectToServerCouple()`, que en función de si el *attachment* es local o distribuido devuelve una referencia local o un *proxy* a la pareja, respectivamente. De esa forma, el funcionamiento es transparente tanto si los elementos arquitectónicos a comunicar actúan de forma distribuida o local.

Una vez hecho esto, la parte cliente del *attachment* está lista para reanudar su hilo de ejecución . Lo primero que hará dicho hilo es registrar el `AttachmentClient` con los `OutPorts` del elemento arquitectónico (2), y antes de ponerse a escuchar a la espera de peticiones que vengan de éstos, utilizará su recién adquirida referencia a la pareja, en el ejemplo *AttachmentInterfaceY*, para invocar al método `Start()` de su `AttachmentServer` (3).

En la parte servidora de la pareja (4), se registra el `AttachmentServer` con los `InPorts` del elemento arquitectónico al que está sirviendo, es decir, el conector. Tras eso, se cambia el *flag* `sttopped` para indicar que el elemento está de nuevo en ejecución y se llama al `Start()` del *attachment* al que pertenece haciendo uso de la propiedad `parentAttachment`.

De nuevo se encuentra el proceso en el método `Start()` de un `Attachment` (5), pero en esta ocasión, en el de la pareja del *attachment* del que ha iniciado el proceso, la que está del lado del conector. Se vuelve a comprobar el *flag* `executing`, y como está valuado a falso, pues esta parte del *attachment* aún no ha sido iniciada, se cumple la condición. Se asigna de nuevo el `coupleServer` del *attachment* cliente (6), se crea el hilo y se llama a su método `Start()` de la

misma forma que se ha hecho en la pareja. Se ha iniciado el hilo del cliente. De la misma forma que se hizo en la pareja, el cliente se suscribirá con los `OutPorts` del elemento arquitectónico al que está conectado, en este caso del conector y llamará al `Start()` del servidor con el que se comunicará, que vuelve a ser el *attachment* de partida, el del lado del componente.

De vuelta al *attachment* inicial, *AttachmentInterfaceX*, en su parte servidora (7), de manera análoga a lo visto anteriormente, se registra con los `InPorts` del elemento arquitectónico (el componente) y llama al `Start()` del *attachment* padre usando la propiedad `parentAttachment`.

En este punto, la petición de reanudación ha llegado a un *attachment* que ya está en funcionamiento (2), puesto que se ha iniciado en la primera parte del proceso, así que ya no tiene sentido continuar. A efectos de implementación, esto se traduce en que el *flag* `executing` estará valuado a cierto, y por tanto el proceso finalizará al llegar a este punto. Como se ha visto, la ejecución de un `Start()` en un *attachment* se ha propagado a través de la estructura de comunicación formada por la pareja de *attachments* y sus respectivas partes cliente y servidora. Al pasar el proceso por todas estas partes, todas se han iniciado o han sido avisadas de que el *attachment* al que pertenece ha sido reanudado, por lo que han hecho las tareas adecuadas. En la Figura 34 se muestra el diagrama de secuencia del ciclo de ejecución de los *attachments* cuando se procesa una petición de reanudación.

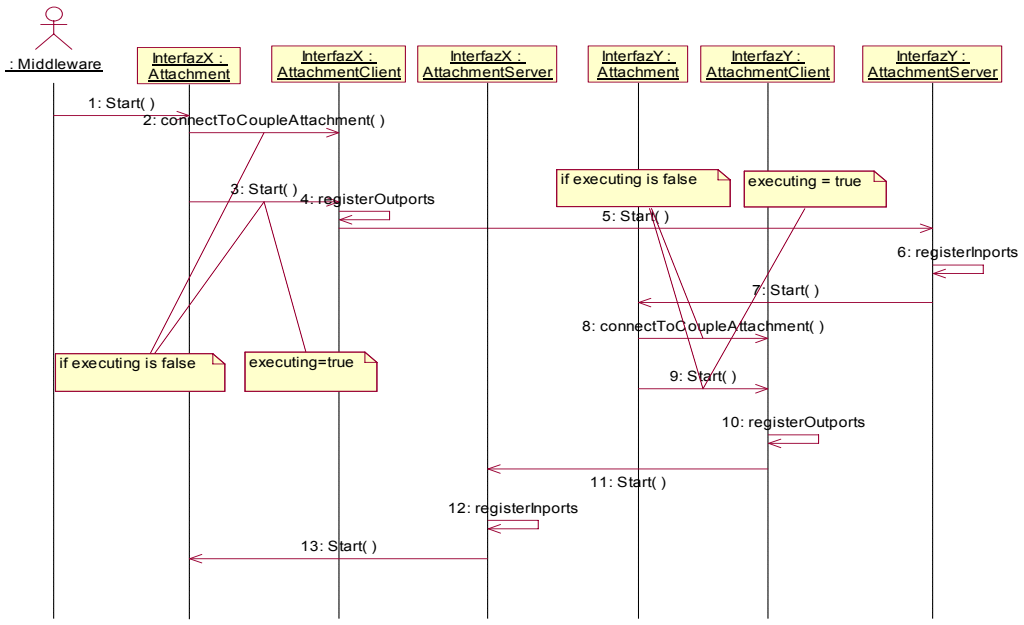



**Figura 34: Diagrama de secuencia del ciclo de ejecución de los *attachments***

Como se ha comentado, este planteamiento cíclico ha sido también implementado de forma análoga, eso si, adaptándolo a las necesidades de cada caso, para detener (`Stop()`) y abortar (`Abort()`) el *attachment*.

Gracias a las modificaciones aplicadas, se ha extraído la lógica de la gestión de la ejecución del *middleware* para encapsularla en el propio *attachment*. De esa forma, se simplifica la implementación del *middleware*, se agiliza el proceso de inicio y detención de los *attachments*, y lo más importante, se encapsula la funcionalidad inherente al *attachment* dentro del propio *attachment*, incrementando su independencia y mantenibilidad.

### 4.1.2. Modificación del modelo de ejecución de los bindings

De forma similar a como se han modificado los *attachments*, se han modificado también los *bindings*. El canal de comunicación también está formado por dos elementos, aunque en este caso son dos elementos distintos: Por un lado está el `SystemBinding` y por el otro el `ComponentBinding`. No obstante, la implementación previa arrastraba el mismo problema que en el caso de los *attachments*, al tener que ser el *middleware* quien localizase a través del DNS la ubicación en el entorno de ejecución distribuido la ubicación de ambas partes para proceder a la parada, reanudación o aborto de la ejecución del *binding*.

Así pues, la idea es modificar el ciclo de ejecución de forma similar a como se ha hecho para los *attachments*, pero adaptando la solución al caso de los *bindings*. Para aplicar el mismo planteamiento que en los *attachments* hay que tener en cuenta que los *bindings* funcionan de forma diferente. En una parte, la del sistema, se encuentra el gestor de *bindings* o `SystemBinding` que guarda referencias a todos los *bindings* conectados con los puertos del sistema, y se ocupa de localizar los componentes conectados a los *bindings* y redirigirles las peticiones. El gestor de *bindings* tiene una parte servidora (`SystemBindingServer`) que es la encargada de recibir peticiones de los componentes y redirigirlas a los puertos de salida del sistema. Solo hay una instancia del `SystemBinding` por sistema. Por otro lado se encuentra el *binding* del lado del componente o `ComponentBinding`. El `ComponentBinding` tiene una parte servidora que recibe peticiones de servicio que redirige a los `InPorts` del componente y una parte cliente, que escucha en los `OutPorts` del componente y reenvía las peticiones al `SystemBindingServer`. Existe una instancia de `ComponentBinding` por cada componente conectado mediante un *binding* al sistema.

Como se ha explicado, un *binding* está formado por dos partes bien diferenciadas: el *binding* de la parte del sistema y el *binding* de la parte del servidor. No obstante, en la implementación anterior, el `InPort` del sistema tenia una referencia directa a la parte servidora del `ComponentBinding`. Esto suponía el problema de la falta de independencia del `InPort`.

Puesto que todo sistema tiene un `SystemBinding` que es quien debería gestionar los *bindings*, tiene sentido encapsular toda la lógica de los *bindings* en dicho gestor, aumentando así la independencia del resto de *elementos*. Por ello se va a hacer un breve paréntesis en la explicación de la modificación del ciclo de ejecución de los *bindings* para explicar como se ha modificado el

`SystemBinding` para este propósito, pues la modificación del ciclo de ejecución está íntimamente relacionada con la estructura interna del `SystemBinding` o Gestor de Bindings.

Como se ha comentado en el Capítulo 3, las peticiones que lleguen a los puertos de un sistema pueden ser tratadas por los aspectos emergentes del sistema o por los elementos arquitectónicos internos a él. Aunque conceptualmente los aspectos emergentes del sistema se encuentran encapsulados dentro de un componente emergente que se comunica a través de *bindings* con los puertos externos del sistema y a través de *attachments* con el resto de elementos internos, en la implementación se ha optado por simplificar.

Pues bien, se ha eliminado la referencia al `ComponentBindingServer` del `InPort` del sistema, de forma que la comprobación de si el servició lo servirá un aspecto del sistema o un aspecto de un elemento arquitectónico interno se hará en el propio sistema. De hecho, al igual que en el caso de los componentes y los conectores, el `InPort` redirige sus peticiones a las cola del elemento arquitectónico (ver Figura 35). Una vez en el hilo de ejecución del sistema, cuando se desencole una petición se comprobará si la petición la sirve un aspecto del sistema o un elemento interno. Esta comprobación se hace mediante la existencia de un delegado en la petición encolada en la cola del sistema, si existe, dicha petición pasará a encolarse en la cola del aspecto. En caso contrario, la petición se pasará al `SystemBinding` que ya se encargará de encontrarle destinatario.

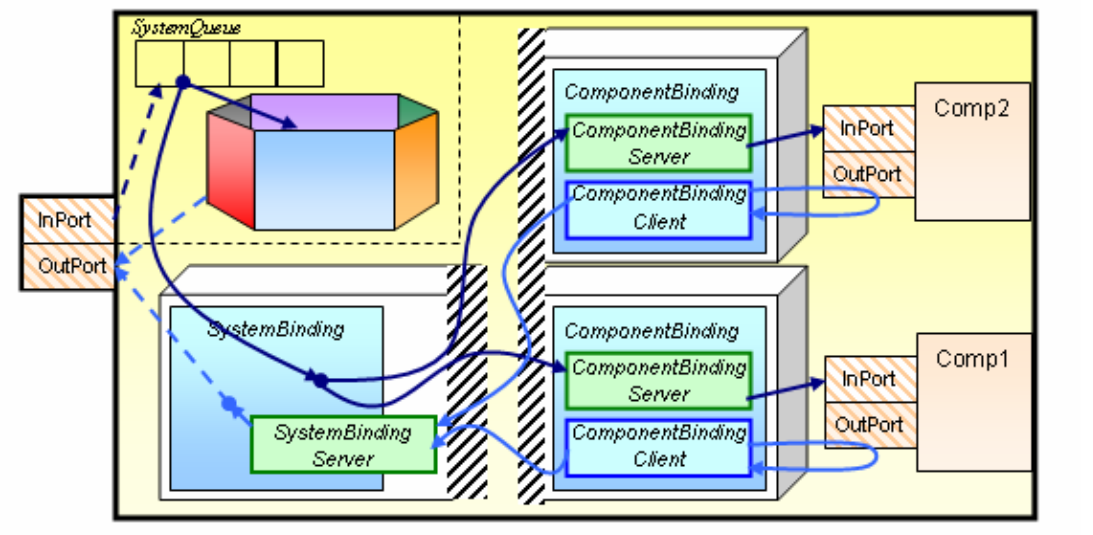



**Figura 35: Esquema de redirección de servicios del sistema**

Mediante esta modificación el `InPort` gana independencia y el `SystemBinding` gana protagonismo, no obstante se añade cierto sobre coste, ya que, a parte de la comprobación al desencolar, el `SystemBinding` ha de localizar el `ComponentBinding` hacia el elemento arquitectónico que sirve la petición. Un puerto puede tener más de un *binding* asociado, por lo que el `SystemBinding` ha de recorrer toda la lista de `ComponentBindings` para localizar todos los *bindings* conectados al puerto por el que venga la petición. Para optimizar la búsqueda

de *bindings* para un determinado puerto, se ha añadido una lista de puertos en el `SystemBinding` llamada `PortList`. Esta lista de puertos se ha implementado en la clase `PortBindingListCollection` (ver Figura 36).

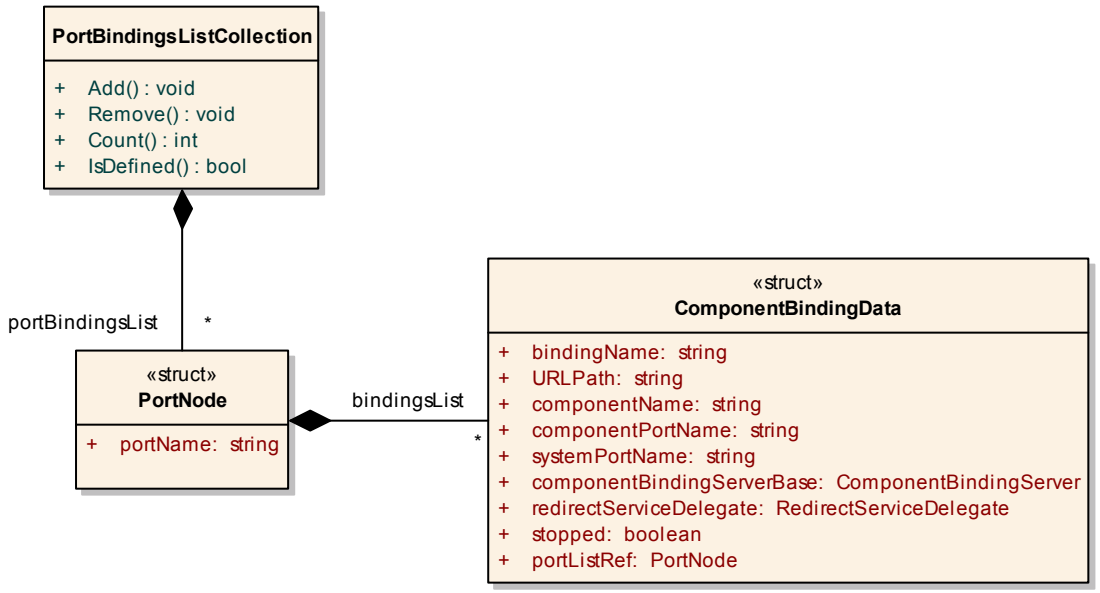



**Figura 36: Clase PortBindingsListCollection**

La clase `PortBindingsListCollection` es una lista formada por nodos de tipo `PortNode`. Estos nodos contienen el nombre del puerto y otro *array* llamado `bindingsList` que contiene la lista de `ComponentBindingData` de los *bindings* asociados a ese puerto.

Se han implementado métodos para añadir y eliminar *bindings* de la lista, que serán invocados al mismo tiempo que se invoquen los métodos `Add()` o `Remove()` de `componentBindingsList` del gestor de *bindings*, respectivamente. Estos métodos liberan totalmente al gestor de *bindings* de la lógica interna de la estructura; el método `Add()` añade un nuevo *binding* a un puerto en la lista, y en el caso de no existir una entrada en la lista para ese puerto, crea una entrada para éste, como se muestra a continuación:

El método `Remove()` elimina un *binding* de la lista de *bindings* del puerto al que pertenece. Es por ello que primero se ha de obtener el puerto al que pertenece, que se extrae de la cadena que conforma el nombre del *binding*. Tras eso, se localiza la lista de *bindings* del puerto y se elimina. Si ese es el único *binding* en la lista del puerto, se elimina el puerto de la lista, pues no hay ningún *binding* asociado a ese puerto. El método se muestra a continuación:

Una vez aplicados estos cambios los *bindings*, y en particular el gestor de *bindings*, están preparados para implementar la lógica de inició (detención, aborto) cíclica para liberar al *middleware* de dicha responsabilidad.

Al igual que en los *attachments*, la idea reside en que, una vez iniciado el proceso, cada parte de la estructura *binding* realice las acciones pertinentes

para su reanudación (detención, aborto) y avisé al siguiente parte. De nuevo la explicación se centrará en el ciclo de reanudación del *binding* (Figura 37), siendo fácilmente aplicable a los otros casos (detención y aborto).

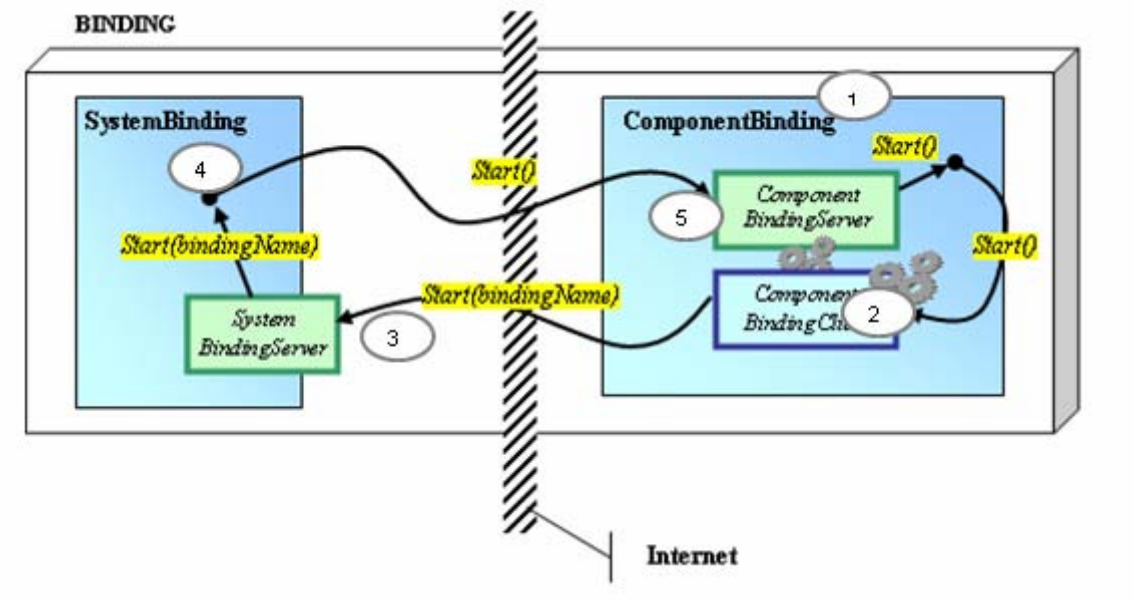



**Figura 37: Ciclo de ejecución de los *bindings* modificado**

Si se empieza por el `ComponentBinding` (1), cuando se le invoque el servicio `Start()` se comprobará que el *flag* `executing` esté puesto a falso, indicando así que esa parte del *binding* está parada. Si esto se cumple, le pasará a su parte cliente una referencia al `SystemBindingServer` con el que se comunicará. Esta referencia se obtiene con el nuevo método `ConnectToBindingServer()`, que devuelve una referencia al `SystemBindingServer` en función de si es remoto o local. Una vez hecho esto, se inicia el hilo del cliente.

Ya en el contexto de la parte cliente (2), el método `Start()` lo primero que hace es registrase como *listener* de los `OutPorts` del elemento arquitectónico para recibir una referencia a la cola del `OutPort`. En el paso anterior se le ha actualizado el *proxy* a la parte servidora del `SystemBinding`, así que se llamará a su método `Start()` (3). Nótese que se le ha de pasar el nombre del *binding* como parámetro, esto es debido a que como se ha comentado, el `SystemBinding` es una única instancia que se conecta con todos los `ComponentsBindings`. Mediante el parámetro, `SystemBinding` identifica al `ComponentBinding` que ha iniciado el proceso.

En la parte servidora del gestor de *bindings* (3), tras registrar el *binding* con el `OutPort` del sistema, se pasa la invocación `Start(bindingName)` para arriba, es decir, hacia el `SystemBinding`, haciendo uso de la propiedad `parent`. De nuevo se pasa como parámetro el nombre del `ComponentBinding`.

El `SystemBinding` (4), al recibir la llamada a su método `Start(bindingName)` realiza los pasos necesarios para reenviarla a la parte servidora del `ComponentBinding` adecuado, es decir, el que ha recibido por parámetro. Los pasos que sigue son los siguientes: Localiza en su lista de *bindings* el *binding* que se

la pasa, en caso de no encontrarlo lanzaría una excepción. Si el *binding* está detenido, cosa que comprueba gracias a un *flag* en el `ComponentBindingData` que le devuelve la lista, obtiene la referencia al `ComponentBindingServer`. Si la referencia es remota, registrará el `SystemBinding` para ser accedido remotamente (en el caso que no esté registrado, esta comprobación ya la hace el propio método `Register()`, también añadido a la clase, al igual que `UnRegister()`). A continuación, actualiza el delegado al `RedirectService()` del `ComponentBindingServer` que se guarda en la estructura de datos del *binding*, así como el valor del *flag* `stopped`. Finalmente invoca al método `Start()` del servidor del *binding* del componente.

Ya en la última parte del ciclo (5), el `ComponentBindingServer` se registra con los puertos de entrada del componente e invoca al `Start()` del `ComponentBinding` mediante su propiedad `parent`. De esta forma se ha llegado al punto de inicio (1). Al ejecutar de nuevo el `Start()` del `ComponentBinding` la ejecución terminará al comprobar el *flag* `executing`, ya que en la primera pasada por el método se ha puesto a cierto.

En el diagrama de secuencia de la Figura 38 se muestra el funcionamiento de este ciclo de puesta en marcha junto a las entidades implicadas.

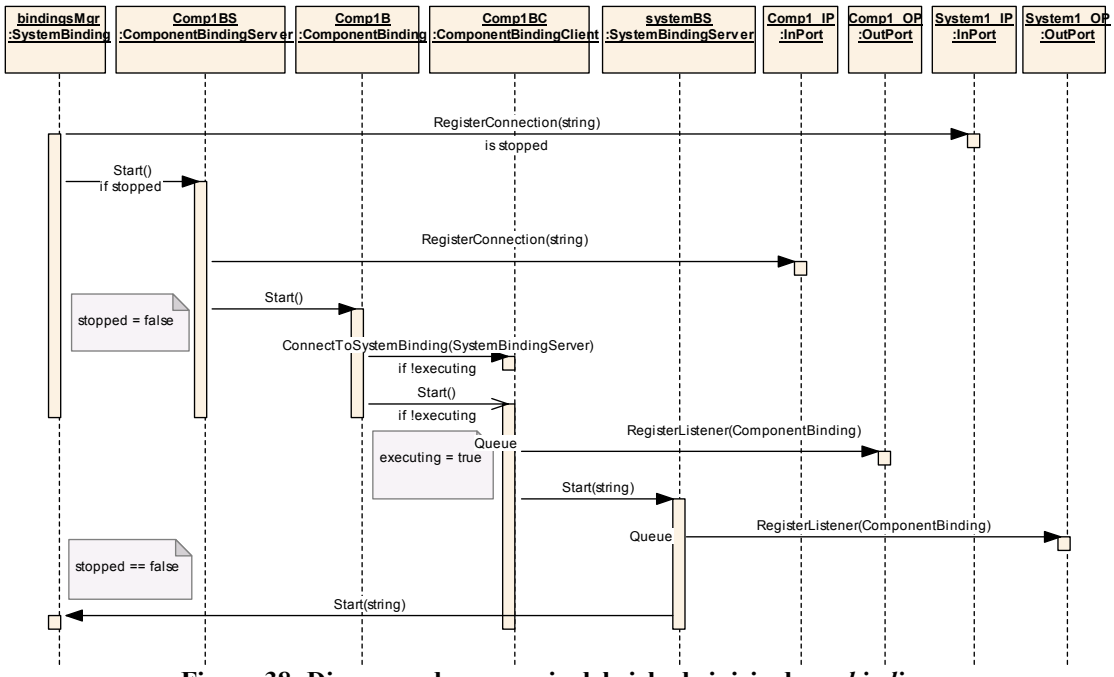



**Figura 38: Diagrama de secuencia del ciclo de inicio de un *binding***

De forma similar, se han modificado los métodos para la detención y aborto de `SystemBinding` y `ComponentBinding` para que se adapten al nuevo ciclo de ejecución. Una de las características importantes del ciclo en cuestión es el hecho que, independientemente de si el proceso se inicia desde el gestor de *bindings* o desde un `ComponentBinding`, la estructura circular del proceso hace que éste se complete correctamente.

### 4.1.3. Modificación de los hilos de escucha de attachments y bindings

Cada `OutPort` tiene un `ListenerPort` por cada *attachment* o *binding* conectado a él. Cada uno de esos `ListenerPorts` encapsula información sobre el *attachment* o *binding* al que está asociado y una cola. Cuando un *attachment* (*binding*) se asocia a un puerto, se crea un `ListenerPort` y se le devuelve una referencia a la cola de ese `ListenerPort` al *attachment* (*binding*): Esa será la cola de la que va a escuchar peticiones. De esa forma, en realidad el `OutPort` tiene una cola por cada canal de comunicación. Cada vez que una petición llega al puerto para que salga por él, éste la encola en las colas de cada uno de los `ListenerPorts` que tiene. Las partes cliente de los `Attachment` y de los `ComponentBinding` tienen un hilo que comprueba periódicamente la cola de un `ListenerPort`. Cuando esta cola está vacía, el hilo se duerme durante un tiempo. Al despertar comprueba de nuevo el estado de la cola, y así sucesivamente.

El diseño explicado tiene algunos defectos, porque aunque exista un canal de comunicación definido, ya sea mediante un *attachment* o un *binding*, éste no tiene que estar continuamente en uso. Por un lado, si el valor del tiempo de espera entre comprobación y comprobación es muy alto las peticiones pueden esperar un tiempo en la cola de los `ListenerPorts` hasta que el hilo de escucha del `AttachmentClient` (`ComponentBindingClient`) despierte y lo procese, lo cual ralentiza la comunicación. Por otro lado, si el valor es muy bajo, el hilo se está durmiendo y despertando continuamente, aún cuando no hay peticiones que servir, con lo cual se sobrecarga el sistema en algo que no está aportando ningún beneficio.

En esta sección se va a presentar como se ha cambiado este mecanismo para que el hilo del `AttachmentClient` y el `ComponentBindingClient` esten suspendidos mientras no haya peticiones en la cola del `ListenerPort`. De forma que cuando el componente encole una petición se despierte el hilo para que las procese. De esa forma, el hilo sólo trabajará cuando haya trabajo y sólo consumirá recursos cuando sea necesario consumirlos.

Para despertar el hilo suspendido, hay que tener en cuenta una serie de consideraciones. Por un lado, el `OutPort` no tiene ninguna referencia al *attachment*, pero sí el `ListenerPort` que guarda el nombre del *attachment* registrado. Por otro lado, hay que tener en cuenta que un `OutPort` puede estar registrado tanto con *attachments* como con *bindings*, por ello necesitamos un mecanismo independiente del tipo de elemento con el que se conecta.

Lo primero ha sido resuelto haciendo que el `ListenerPort` guarde directamente la referencia al *attachment* en lugar del nombre mientras que lo segundo se ha solucionado mediante la aplicación de patrones de software. En concreto, se ha añadido el concepto de suscriptor, tal y como muestra el patrón de software *Observer* [Gamm95]. Un suscriptor será un elemento (*binding* o *attachment*) escuchando a la cola de un `OutPort` determinado. Cuando se encole algo en el `OutPort`, éste lo notificará a todos los suscriptores para que tomen las

medidas oportunas. Así, los *attachments* y *bindings* suscritos a un `OutPort` recibirán una llamada cada vez que éste encole un elemento en la cola del `ListenerPort` que están escuchando. Esta llamada será usada por los suscriptores para despertar al hilo suspendido.

Para incorporar el concepto de suscriptor, se ha creado una nueva interfaz `ISubscriber` que define el método `getRequest()`, que será invocado por el `OutPort` cada vez que encole una petición. En cierto modo, para un suscriptor, la invocación de este método se puede ver como la recepción de un evento provocado por la inserción de una petición en la cola con la que está suscrito. Este método será el encargado de despertar el hilo en el caso que esté suspendido. Además, esta interfaz también puede definir métodos comunes a las clases `Attachment` y `ComponentBinding`, como `Start()`, `Stop()` y `Abort()` (que deberán ser renombrados de esta forma, ya que actualmente se llaman `AttachmentStart`, `AttachmentStop`, etc.). Así pues, estas dos clases implementarán la interfaz `ISubscriber`. Así mismo, una interfaz ISubscriberDataTransfer será usada para abstraer el comportamiento de las clases DataTransfer de *attachments* y *bindings*, clases encargadas de encapsular la información de los *attachments* y *bindings* respectivamente para la movilidad. En la Figura 39 se muestra las interfaces `ISubscriber` e `ISubscriberDataTransfer`.

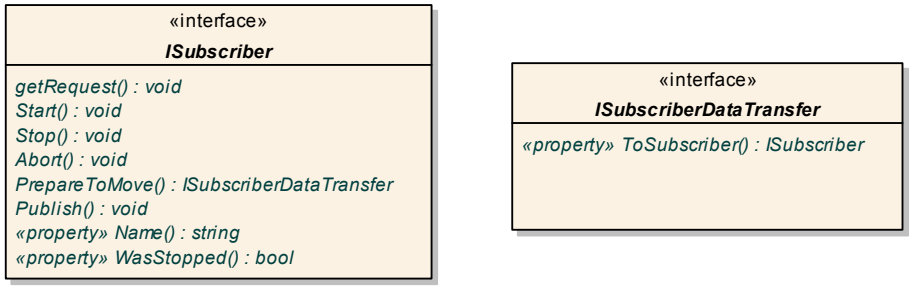



**Figura 39: Interfaces ISubscriber e ISubscriberDataTransfer**

Puesto que se ha modificado la clase `ListenerPort` para que guarde una referencia al *attachment* o *binding* al que está conectado, se ha modificado el constructor y algunos métodos, así como todas las llamadas a éstos adaptándolos a la nueva situación. Se ha cambiado también el nombre de la lista `AttachmentRegistered` del `OutPort` a `subscribersList`, pues la lista contendrá referencias a los `listenerPorts` de los suscriptores, que no serán únicamente *attachments*. Cada vez que el método `AddRequest()` del `OutPort` añada una petición en la cola de un suscriptor, se hará una llamada a su método `getRequest()` que redirige la llamada al método `ResumeAttachmentThread()` para comprobar si el hilo está suspendido y en dicho caso, despertarlo.

En la Figura 40 se muestra el diagrama de clases con las relaciones entre las clases implicadas.

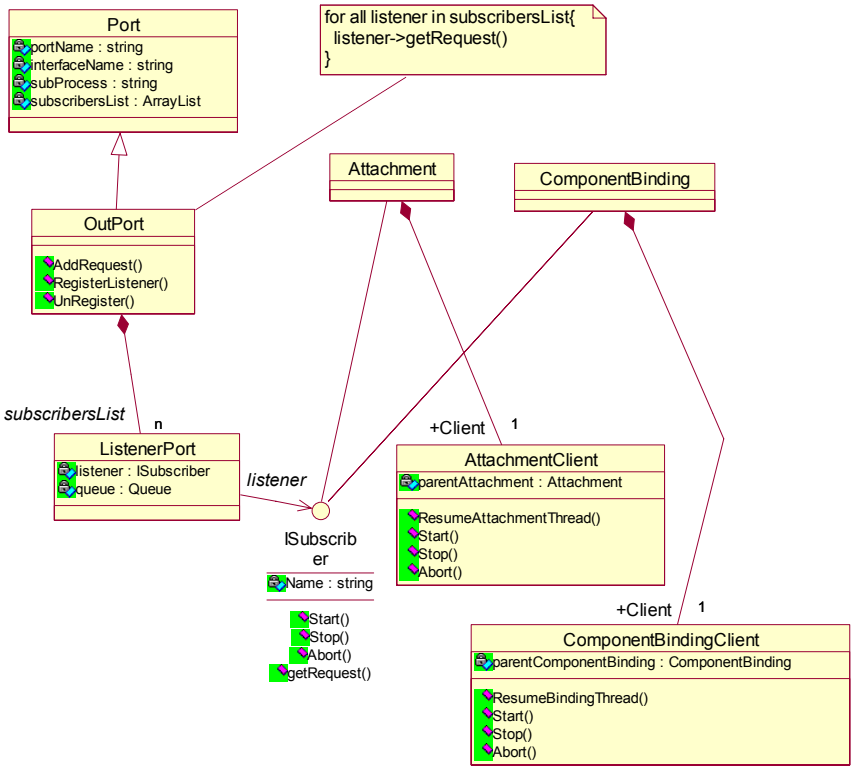



**Figura 40: Diagrama de clases tras la incorporación de ISubscriber**

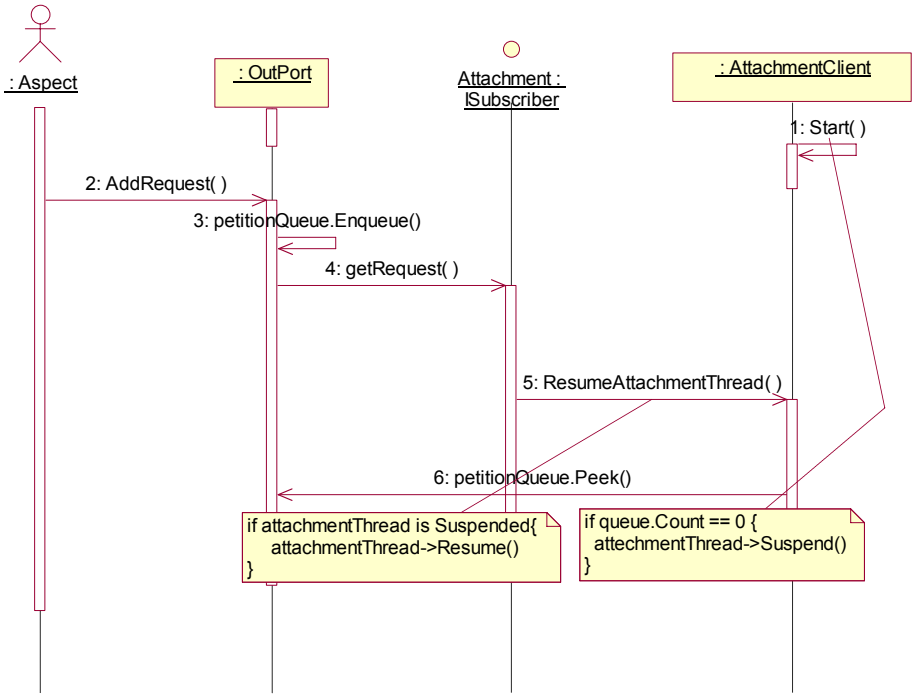



**Figura 41: Diagrama de secuencia de un *attachment* suscrito a un OutPort al que se encola una petición**

A modo de ejemplo, se muestra en la Figura 41 el diagrama de secuencia para el caso concreto de un *attachment* suscrito a un `OutPort`. El aspecto iniciará la interacción añadiendo una petición en la cola del `OutPort`.

Cuando el hilo de la parte cliente se inicia mediante la llamada al método `Start()` (1) del `AttachmentClientBase` comprobará la cola del `ListenerPort` que tiene asignado. Si ésta está vacía el hilo se suspenderá. Por otro lado, un Aspecto puede requerir de un servicio externo, por lo que transmite la petición al `OutPort` mediante `AddRequest()` (2). El `OutPort` encolará dicha petición en cada una de las colas donde escuchan los elementos suscritos a él (3) y enviara la señal `getRequest()` a estos elementos para notificarles que ha modificado la cola en la que escuchan (4). Obsérvese como la llamada la hacen a través de la interfaz `ISubscriber`. La interfaz ya se encarga de llamar a la implementación del método adecuada (la del *attachment* o la del *binding*) para despertar el hilo de escucha (5). En el método se hará la comprobación de si el hilo está suspendido, y en caso afirmativo, lo despertará. En cualquiera de los dos casos, el hilo de escucha cogerá el siguiente elemento de la cola (6).

### 4.1.4. Movilidad simultánea de subscribers

Los nuevos ciclos de ejecución de *attachments* y *bindings* explicados en las secciones 4.1.1 y 4.1.2 respectivamente presentan una limitación: No era posible mover los dos elementos conectados por un *attachment* (*binding*) a la vez. Si eso ocurriese, la notificación de la nueva ubicación del *attachment* no sería efectiva, pues ésta se basa en la referencia que guarda a la parte servidora de su pareja. Si la pareja también se mueve, la referencia se queda obsoleta y no sería posible localizar directamente a la pareja.

Una vez presentado el concepto de *subscriber* es posible abordar el problema de forma global para *attachments* y *bindings*. Gracias a la interfaz `ISubscriber` es posible trabajar con *attachments* y *bindings* de forma transparente, lo cual ayuda a simplificar la implementación; a nivel de *middleware* ya no es necesario distinguir entre gestión de *attachments* y *bindings*, ahora lo que se gestiona son los *subscribers*.

Una primera aproximación para tratar los problemas que presentaba esta limitación fue el restringir la movilidad de un elemento arquitectónico mientras se mueven otros elementos conectados a él. Cuando un elemento arquitectónico se iba a mover, sus *subscribers* eran notificados de este hecho. Éstos a su vez, se lo notificaban a su pareja, por lo que, hasta que se recompusiese el canal de comunicación tras la movilidad, no debían moverse. Esta restricción por tanto, establece un orden secuencial en la movilidad. Este orden secuencial no obedece a las necesidades del elemento arquitectónico, sino a las del *attachment*. Por lo que, con todo, no parece una buena opción.

Como se comentó en el Capítulo 2, la comunicación distribuida en ProActive [Bau00] es proporcionada por *forwarders*. Los *forwarders* son unas entidades que se crean en las máquinas origen tras la movilidad de un componente, en los que se guarda información sobre la nueva ubicación del componente. De esa forma, las peticiones que se hagan a ese componente

procedentes los elementos arquitectónicos que no hayan actualizado la ubicación del componente tras la movilidad, son redirigidas por el *forwarder* a la ubicación correcta. Esta petición redirigida puede encontrarse con otro *forwarder*, si el componente se ha vuelto a mover, formándose así una cadena de *forwarders* que redirigen la petición hasta llegar a destino.

En PRISMANET se ha optado por resolver el problema de la movilidad simultánea de *subscribers* de una forma similar. Puesto que la limitación la provoca el hecho que no sea posible notificar la nueva ubicación a la otra parte del *subscriber* si se ha movido también, lo que se hará es establecer un mecanismo por medio que avisará a ambas partes de la nueva ubicación de la pareja.

Para ello, se ha establecido una lista en la capa *Element Management Layer* que guardará información sobre los *subscribers* que se mueven de forma simultánea (`subscribersInMovement`). Cuando la capa *Distribution Services* esta preparando para mover un *subscriber* que está parado, agregará una entrada a esta lista con el nombre del *subscriber* y la nueva ubicación a la que se va a mover. Si el *subscriber* estaba parado (flag `WasStopped`) es porque la pareja se estaba moviendo. Cuando la pareja finalice la movilidad, intentará obtener una referencia remota a la parte servidora del *subscriber*. Como el *subscriber* se ha movido, este proceso provocará un error. Al capturar el error, la pareja preguntará al *middleware* en que se encontraba la pareja por la información de su nueva ubicación. Esta consulta se hace a través de la capa *Middleware2Middleware*. El *middleware* que contenía el *subscriber* consulta la información sobre la pareja en su lista de `subscribersInMovement` y le devolverá la nueva ubicación del *subscriber* para que le notifique correctamente su nueva ubicación.

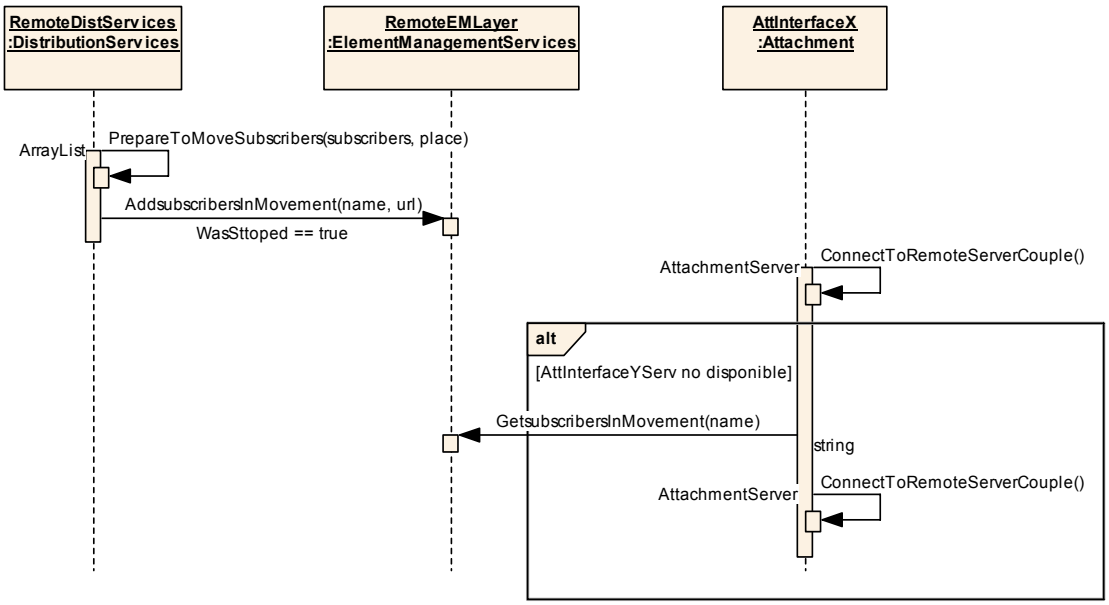



**Figura 42: Diagrama de secuencia en el que se muestra el mecanismo de los subscribers en movimiento**

En la Figura 42 se muestra un diagrama de secuencia donde un *attachment* intenta reciber un error al intentar localizar a la parte servidora del otro lado (`ConnectToRemoteServerCouple`) y hace uso de la información contenida en la lista `subscribersToMove` del *middleware* remoto para localizar a la pareja.

De esta forma se consigue que dos *attachments* se muevan simultáneamente, junto a los elementos que conectan, y puedan restablecer el canal de comunicación tras la movilidad siguiendo el modelo de ejecución propuesto.

## 4.2. Gestión del entorno distribuido de PRISMANET

Como se ha comentado en el Capítulo 4, el entorno de ejecución de PRISMA está formado por un grupo de *middlewares* distribuidos que trabajan de manera conjunta. Para el correcto funcionamiento de los elementos arquitectónicos que se están ejecutando sobre un *middleware*, todos los *middlewares* que forman el entorno de ejecución, han de conocerse entre si para realizar tareas colaborativas. Para ello, cada *middleware* PRISMA tiene que disponer de una lista actualizada con elementos que representen a los diferentes nodos donde se esta ejecutando un servidor *PRISMA-Server*. Además, es necesario que todos los *middlewares* que trabajen de forma conjunta, dispongan de los mismos elementos en la lista, de forma que, aunque cada Middleware tenga su propia lista, se pueda hablar en términos generales de una única lista virtual de *middlewares* para el entorno de ejecución.

En esta sección, se presentará cómo se ha dotado al *middleware* de esta funcionalidad, ampliando así la información que tiene sobre el entorno, facilitando así una explotación mayor de las características distribuidas del modelo. Por ejemplo, el que los *middlewares* que conforman el entorno de ejecución PRISMA se conozcan entre sí, proporciona una situación óptima para implementar un DNS distribuido, como se verá al final de la sección.

Por ello, el objetivo que se persigue, y que se abordará en la presente sección, es dotar al *middleware* de un mecanismo, una lista de *middlewares* conocidos, para que sea consciente de su entorno, siendo capaz de detectar caídas y recuperaciones de los otros *middlewares* y reflejarlos en dicha lista en tiempo real. Se persigue pues, que todos los *middlewares* se conozcan entre si, es decir, tengan los mismos elementos en la lista y que los cambios que se produzcan en el entorno sean reflejados en todas las listas de los *middlewares*, como si de una única lista se tratase.

### 4.2.1. La Lista de Middlewares Conocidos

Esta ultima afirmación, requiere una explicación más detallada. Un *middleware* PRISMA esta compuesto por una serie de capas. Entre estas capas

se encuentran la capa `DNSManagementLayer` y la capa `DistributionManagementLayer`. Es en este ámbito donde entra la Lista de Middlewares Conocidos. La lista es necesaria para la resolución de nombres requerida por la capa `DNSManagementLayer` proporcionando un dominio de búsqueda para las peticiones de resolución de nombres que ésta reciba. Además, es necesaria para proporcionar un entorno distribuido de ejecución para la capa `DistributionManagementLayer`, que proporciona la movilidad de los elementos arquitectónicos.

Como se ha comentado, el listado ha de tener los mismos elementos en todos los *middlewares* del entorno y, en un nivel de abstracción superior, representará al entorno de ejecución, es decir, el mundo en el que pueden existir los elementos arquitectónicos de una determinada configuración PRISMA. Es por lo que se puede ver cada una de las Listas de Middlewares Conocidos que contiene cada *middleware*, como una lista virtual distribuida única que contiene todos los *middlewares* y a la que todos los *middlewares* acceden.

Dicho esto, es necesario remarcar la importancia de la consistencia de la lista y su actualización a nivel de nodo ante los cambios que se produzcan, ya sean activos o pasivos.

Se entiende por cambio activo la modificación de la lista en un *middleware* por petición expresa del usuario; puede que haya un nuevo nodo ejecutando un servidor PRISMA que se quiera añadir al entorno de ejecución o por el contrario, puede que ya no le interese que un nodo forme parte de dicho entorno. El usuario añadirá o eliminará la referencia al *middleware* de la lista, por ejemplo a través de la consola y este cambio deberá propagarse a las listas del resto de *middlewares*.

Los cambios pasivos en la lista, representan los cambios no decididos por el usuario, y por tanto no realizados por él. Por ejemplo, cuando un *middleware* deja de responder porque la máquina sobre la que se está ejecutando ha caído por cualquier motivo, o un *middleware* que se había dado por perdido vuelve a estar operativo, por ejemplo, tras reiniciar la máquina en la que se encontraba. Estos cambios pasivos también han de reflejarse en las listas eliminando o añadiendo a los *middlewares* que los provoquen.

### 4.2.2. La capa de Gestión de la Lista de Middlewares

Para la gestión de dicho listado, se ha añadido una nueva capa al *middleware*, la capa de Gestión de la Lista de Middlewares (*MiddlewaresListManagementLayer)* (Figura 43), que será la encargada de mantener la lista con un estado consistente con la situación actual y con el resto de *middlewares*. Esta nueva capa, además, implementa los mecanismos necesarios para la detección de cambios pasivos.

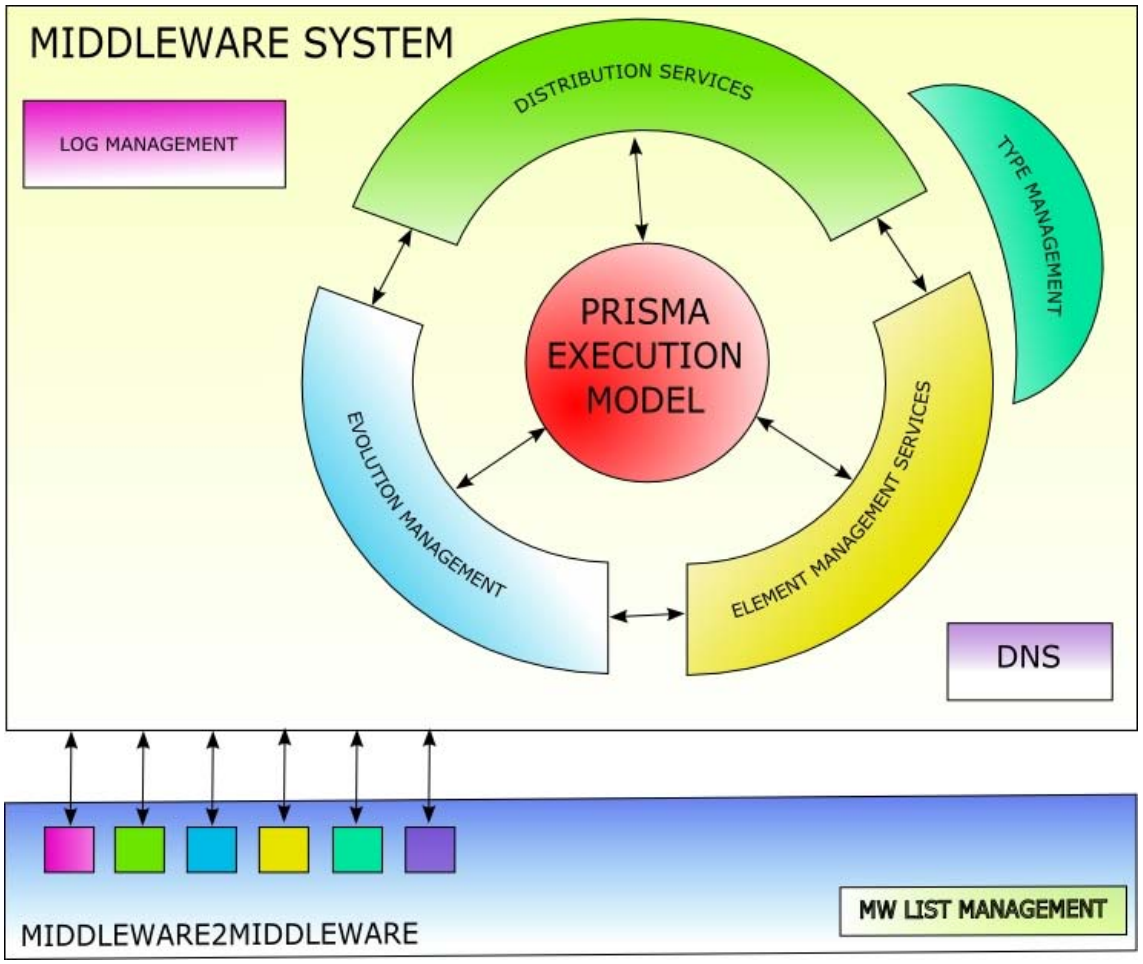



**Figura 43: Diagrama de capas del middleware PRISMA tras la incorporación de MiddlewareListManagement**

Como se puede apreciar en la Figura 43, la capa se añade conceptualmente dentro de la capa `Middleware2Middleware`, recuérdese que esta era la encargada de la comunicación entre *middlewares*. Al ser la capa que gestiona la Lista de Middlewares Conocidos tiene sentido que se encuentre ubicada en la capa centrada en la detección del estado del resto de *middlewares* .

#### *4.2.2.1. Tipos de fallos*

Ábrase en este punto un paréntesis para hablar de los diferentes tipos de fallos en las técnicas tolerantes a fallos. En [Tre04] se distinguen tres modelos de fallos:

- Byzantine Fault: Los nodos fallidos continúan funcionando e interactuando con el resto del sistema, emitiendo salidas incorrectas.

- Fail-stop Fault: Los nodos fallidos dejan de emitir resultados y de interactuar con el resto de nodos.

- Fail-stutter Fault: Incluye los supuestos del modelo Fallo-Parada y añade los fallos de rendimiento, es decir, aquellos casos en que el nodo sigue funcionando pero con un rendimiento inferior al esperado. Este modelo pretende ser un término medio entre los dos modelos anteriores, aunque, por otro lado no está ampliamente aceptado.

Los cambios pasivos, y en concreto la eliminación pasiva, constituye un claro ejemplo de fallo-parada (*Fail-Stop Fault*), al tratarse de una situación en que el nodo deja de responder. La detección de estos fallos se realizará por monitorización de latidos (*heartbeat*) [Kal99]. Cada *middleware* emitirá un latido cada determinado intervalo de tiempo (*frame*) y un *middleware* especial recopilará estos latidos y los monitorizará para determinar, en un instante dado, que *middlewares* de la lista siguen respondiendo y cuales no. De esta forma, un Middleware, y en concreto su `MiddlewaresListManagementLayer` puede adoptar dos roles: Puede enviar latidos o puede monitorizarlos.

#### *4.2.2.2. Líder y Notificadores*

En la implementación se ha decidido que un `MiddlewaresListManagementLayer` que emite latidos adopta un rol de Notificador (*Notifier*), debido a que emite notificaciones periódicas de su estado. Por otro lado, aquel `MiddlewaresListManagementLayer` que monitoriza estas notificaciones o latidos adopta un rol de Líder (*Leader*). Así pues, las notificaciones o latidos no son más que llamadas periódicas por parte del Notificador a un método del Líder (ver Figura 35). El Middleware que ejerce de Líder no necesita hacer notificaciones, su misma existencia garantiza su funcionamiento.

Como se puede apreciar esta solución centraliza en un *middleware* la gestión del entorno distribuido, no obstante esto no va a suponer un problema. Como se va a ver a continuación, se ha implementado un mecanismo para que en caso de fallar el *middleware* que ejerce el rol de Líder, automáticamente otro *middleware* adopte ese papel de Líder, de forma que siempre hay un, y solo uno, Líder.

Además el centralizar la lógica de la gestión de la Lista de Middlewares aporta la ventaja que dicha gestión no se ve perjudicada por la diferencia entre los relojes de diferentes máquinas distribuidas al estar trabajando con marcas de tiempo. De esta forma el único reloj que cuenta es el del Líder: Él es el que anota el instante de la última notificación de determinado *middleware* y él es el que comprueba si se ha vencido el *timeout* en función del instante actual.

El *middleware* cuya capa de Gestión de la Lista de Middlewares ejerza de Notificador, tendrá una referencia al Líder y un hilo que periódicamente irá haciendo llamadas al método `ImAlive` del *middleware* Líder. El método recibe como parámetro el **LOC** que representa al *middleware* que le llama, para que de esa forma el Líder reconozca cual es el *middleware* que le acaba de notificar.


**Figura 44: La clase MiddlewareListManagementServices**

Por otro lado, el *middleware* cuya capa Gestión de la Lista de Middlewares ejerza de Líder escuchará las notificaciones que le irán llegando del resto de Middlewares y anotará, para cada una, el Middleware de procedencia y el instante de tiempo mediante un *timestamp*, reescribiendo el valor que anotó en la última notificación (ver Figura 45). Si recibe notificación de un *middleware* que no tiene en la lista, lo marcará de una forma especial para añadirlo a la Lista de Middleware Conocidos, como se verá más adelante. De esta forma, nuevos *middlewares* se agregan al juego del Líder-Notificador.

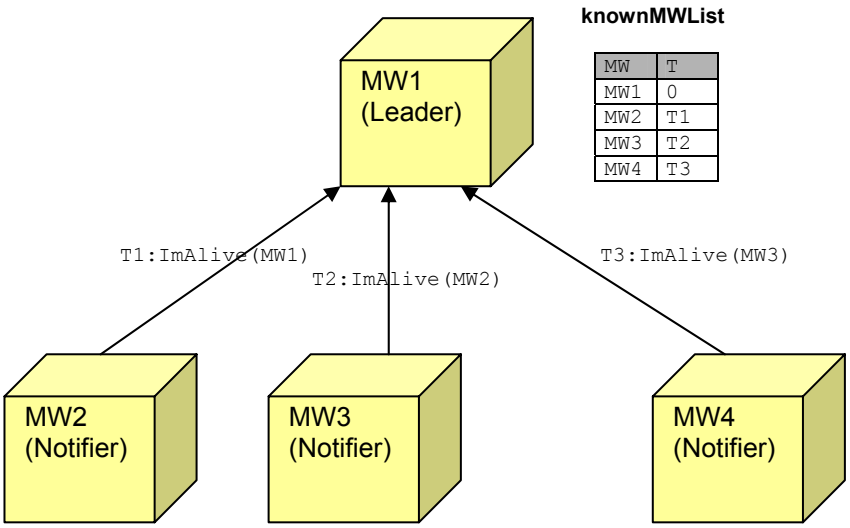



**Figura 45: Interacción Líder-Notificadores**

#### *4.2.2.3. Detección de fallos de los Notificadores*

En el Líder, un hilo irá recorriendo la estructura donde se anote la información sobre los componentes del entorno de ejecución, comprobando para cada *middleware* si la diferencia entre el tiempo actual y si la marca de tiempo de la última notificación es mayor que determinado valor de tiempo establecido para dar un *middleware* por caído. Dicho valor se establece por el usuario y estará en función del tipo de red sobre el que se comuniquen los diferentes nodos. No interesa ni un valor muy pequeño, para evitar middlewares que se den por caídos sólo porque se haya perdido un mensaje de notificación, ni demasiado altos, para evitar una detección tardía de un fallo. Como trabajo futuro se podría considerar adaptar la clase para que estableciese el valor en función de los tiempos de respuesta que recibiese de los Notificadores, de esa forma dicho valor se adaptaría dinámicamente al medio liberando de esa responsabilidad al usuario.

Si la diferencia entre el instante de tiempo actual y el último valor anotado es mayor al valor del *timeout* establecido, se añadirá el *middleware* a una Lista de Middlewares Eliminados que, una vez concluido el proceso de comprobación, se enviará a todos los Notificadores para que se actualicen.

Algo similar ocurrirá para los Middlewares que se hayan recuperado, pues el hilo detectará que están marcados de una forma especial, y los añadirá a una Lista de Middlewares Añadidos que también será enviada a los Notificadores una vez concluida la comprobación. Una vez enviadas las listas de *middlewares* eliminados y añadidos, el hilo del Líder se dormirá por un intervalo de tiempo para después volver a iniciar la comprobación.

En la figura Figura 46 se muestra un diagrama de secuencia del funcionamiento de los nodos ante la caída de un *middleware*. Cuando el Líder detecta que alguno de los Notificadores deja de enviarle latidos, elimina el LOC

que le representa de la Lista de Middlewares Conocidos y lo añade a la Lista de Middlewares Eliminados. A continuación, realiza una difusión de dicha lista entre los *middlewares* que queden mediante el método `BroadcastDeletedList`.

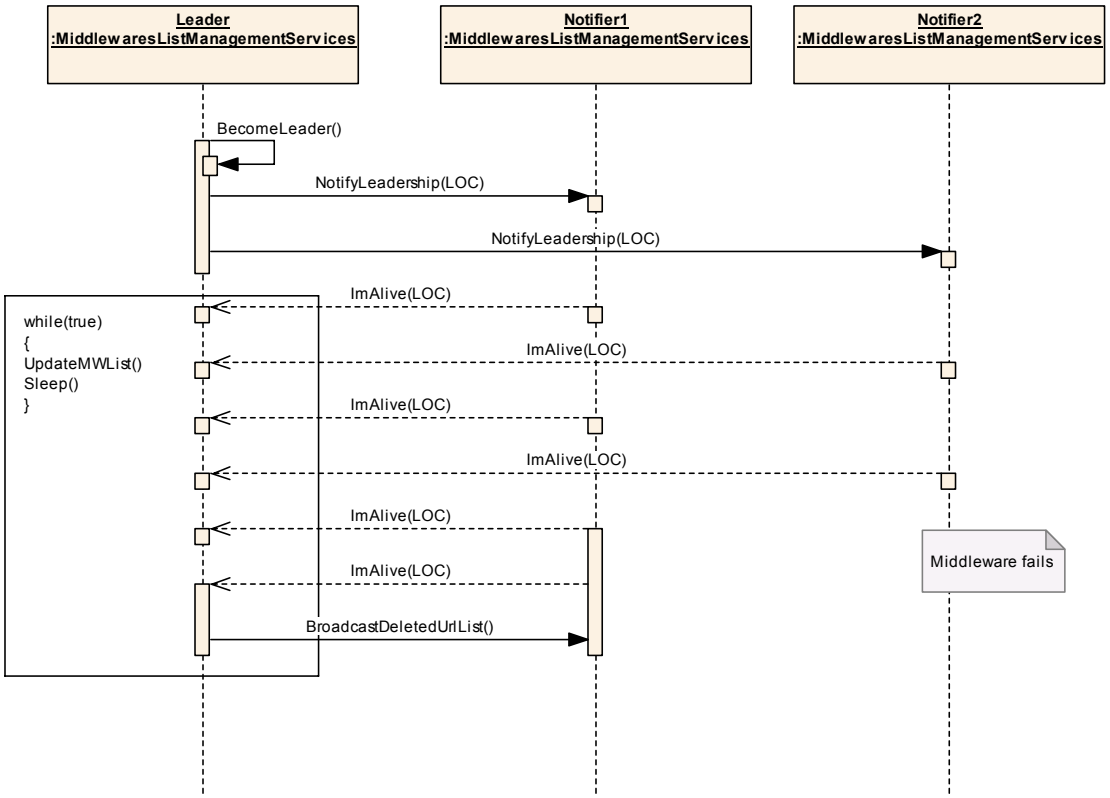



**Figura 46: Diagrama de secuencia del comportamiento cuando un middleware deja de responder**

#### *4.2.2.4. Recuperación de un Notificador caído*

Si un Notificador caído vuelve a ponerse en ejecución, puede ser deseable que se añada pasivamente a la Lista de *middlewares* Conocidos. Para ello se ha creado un mecanismo mediante el cual, un *middleware* guarda el estado de la Lista de *middlewares* Conocidos en memoria secundaria mientras está en ejecución. De esta forma, si la ejecución se interrumpe debido a un fallo, en disco se mantendrá el último estado conocido de la lista. En caso de una finalización correcta, el fichero se vaciará.

Cuando el Gestor de la Lista de Middlewares se pone en funcionamiento, antes de decidir si ha de actuar como Notificador o como Líder, comprueba el fichero de disco donde se guarda la lista. Si este fichero no está vacío, entiende que se acaba de recuperar de un fallo y preguntará secuencialmente a los *middlewares* que obtenga del fichero cual es el actual líder. Si alguno de los *middlewares* aún está en funcionamiento, le devolverá la URL del Líder, por lo que el Gestor de la Lista de Middlewares recuperado se convertirá en Notificador, notificando al Líder que le han devuelto.

En el caso que ninguno de los *middlewares* que obtenga de la lista, es decir los *middlewares* que estaban en funcionamiento la última vez que estuvo en ejecución, responda, el Gestor de Lista de Middlewares no tiene ninguna forma de obtener la información sobre el actual líder automáticamente. Si eso ocurre, se convertiría él mismo en Líder, pues en ese momento no tiene noticia de ningún otro *middleware* en ejecución. En el caso que hubiesen otros *middlewares* en ejecución diferentes a los que tenía en su lista, para que el *middleware* recuperado se uniese a ellos, el usuario debería indicar al verdadero Líder que el *middleware* se ha puesto en funcionamiento. El Líder añadirá de nuevo el *middleware* a su lista y le enviará un mensaje notificándoselo. Este mensaje también degradará al *middleware* recién recuperado al rol de *Notifier* y actualizará su Lista de Middlewares Conocidos. No tiene sentido que hayan dos líderes.

En la Tabla 15 se muestra un breve fragmento de pseudocódigo en el que se muestra como se realiza la inicialización de un *Notifier*: Como se ha comentado, obtiene la lista de middlewares conocidos del disco y se les pregunta uno a uno por el Líder. Si no encuentra ningún *middleware* que le pueda proporcionar esa información entre los que formaban su antiguo entorno de ejecución, él mismo se convertirá en Líder.

```
LastSessionList = ReadMWFile();
If LastSessionList has elements then
       foreach middleware in LastSessionList
              try
                     myLeader = middleware.GetLeader()
              catch
                     next
       endforeach
endif
if MmyLeader is null then BecomeLeader()
```

**Tabla 15: Fragmento de pseudocódigo de la nicialización de un middleware**

#### *4.2.2.5. Cambios activos*

En lo que respecta a los cambios activos, solo se podrán hacer a través de la consola del *middleware*, es decir, del formulario con el que el usuario interactúa con el entorno de ejecución. Al añadir o eliminar un *middleware* bajo demanda, la consola del *middleware* redirige la petición a la capa de Gestión de Lista de Middlewares del *middleware* que se ejecuta en la máquina en que se encuentra. En función de si dicho *middleware* es Líder o no, notificará este cambio a todos los *middlewares* o le reenviará la petición al Líder, respectivamente. El mismo funcionamiento se aplica para el caso de la eliminación activa.

#### *4.2.2.6. Funcionamiento ante el fallo del Líder*

Se ha explicado como se actualiza las Listas de Middlewares Conocidos para que sean consistentes a los cambios activos y pasivos, ahora se verá que ocurre cuando el *middleware* que falla es el que está ejerciendo de Líder.

La eliminación activa de un *middleware* que es el Líder no está permitida en la actual implementación. No obstante, esto es algo que se debería considerar en futuras versiones, pues puede ser interesante para el usuario dar de baja un *middleware*, aunque este sea el Líder, pasando a ejercer esta función alguno de los Notificadores o incluso, que por determinadas razones, interese que determinada máquina ejerza de Líder, con lo que el usuario pueda seleccionar o cambiar en tiempo de ejecución que *middleware* adoptará el papel de Líder, relevando así al que lo estuviera representando en ese momento, que pasaría a ser un Notificador.

Inicialmente, el *middleware* que asume el papel de Líder es el que aparece el primero en la Lista de Middlewares Conocidos. Esto es posible gracias a que los *middlewares* conocidos se guardan en una lista ordenada, en C# una `SortedList` [Ro04], lo que hace que todos los *middlewares* tengan el mismo orden en su lista, al tener los mismos elementos: Un orden alfabético en función de la URL de la máquina. Cuando el Líder falla, los Notificadores lo detectan al intentar notificar su estado (obtienen una excepción de *Remoting*) y toman las medidas pertinentes para designar al nuevo líder: Eliminan al antiguo líder de su Lista de Middlewares Conocidos y eligen como Líder al nuevo primer *middleware* de la lista. Si un Notificador pasa a ser el primero de la lista al eliminar al Líder caído, abortará su hilo de Notificador y comenzará su hilo de Líder; habrá adoptado el rol de nuevo Líder. El resto de Notificadores, también le tendrán como primero de la lista, y por tanto comenzarán a enviarle *heartbeats* al nuevo Líder.

En la Figura 47 se muestra el diagrama de secuencia que representa la recuperación del sistema de actualización ante la caída de un líder. Como se observa, *MW2* y *MW3* son dos *middlewares* que adoptan el papel de Notificador y *MW1* el de Líder. Por ello *MW2* y *MW3* envían notificaciones periódicas a *MW1*. Cuando *MW1* falla, *MW2* y *MW3* reciben un error al intentar notificar con el Líder, por lo que lo eliminan de la lista y proclamen nuevo Líder al primer elemento, esto es, por orden alfabético, *MW2*. Éste, al proclamarse nuevo Líder, anuncia a todos los elementos de su Lista de Middlewares Conocidos este hecho, para que a partir de ese momento, dirijan sus notificaciones hacia él. También aprovecha para pasarles a todos su lista, para, de esta forma, sincronizarla con el resto, algo que no está de más, sobretodo teniendo en cuenta que ha caído un Líder.

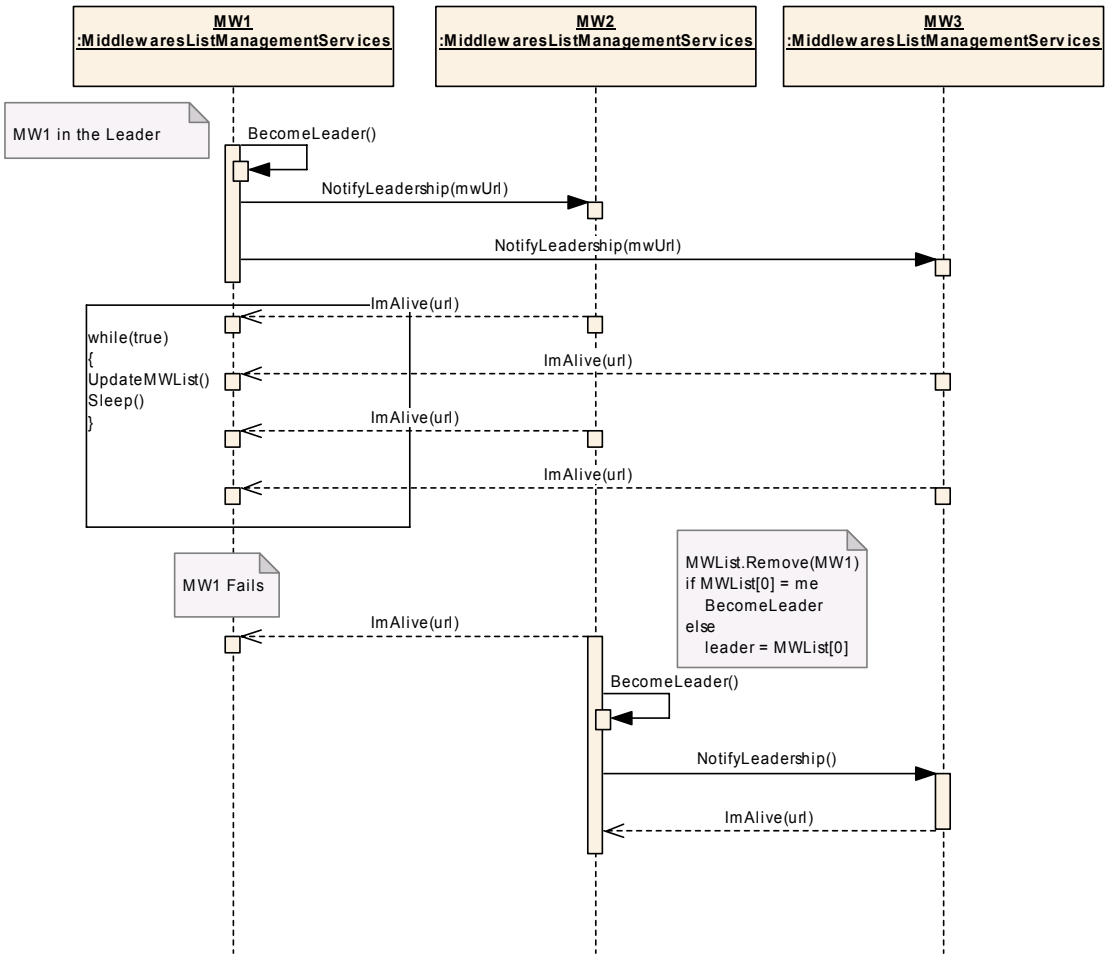



**Figura 47: Diagrama de secuencia que representa la recuperación del sistema de actualización ante la caída de un líder.**

Llegados a este punto se puede afirmar que se ha dotado al *middleware* PRISMA de un mecanismo para que conozca al resto de *middlewares* que componen el entorno de ejecución y sea capaz de reflejar los cambios que en él se produzcan, ya sean activos o pasivos. Gracias a este mecanismo, el *middleware* conoce los nodos entre los que pueden estar distribuidos los elementos arquitectónicos de la configuración que esté ejecutando en determinado momento, la cual cosa da pie, por ejemplo, para construir un DNS distribuido.

#### *4.2.2.7. Representación gráfica*

Para que el usuario sea capaz de aplicar cambios activos a la Lista de Middlewares, o simplemente para que pueda supervisar cual es el estado del entorno de ejecución PRISMA, se ha añadido a la interfaz de la consola PRISMA una sección para gestionar la lista.

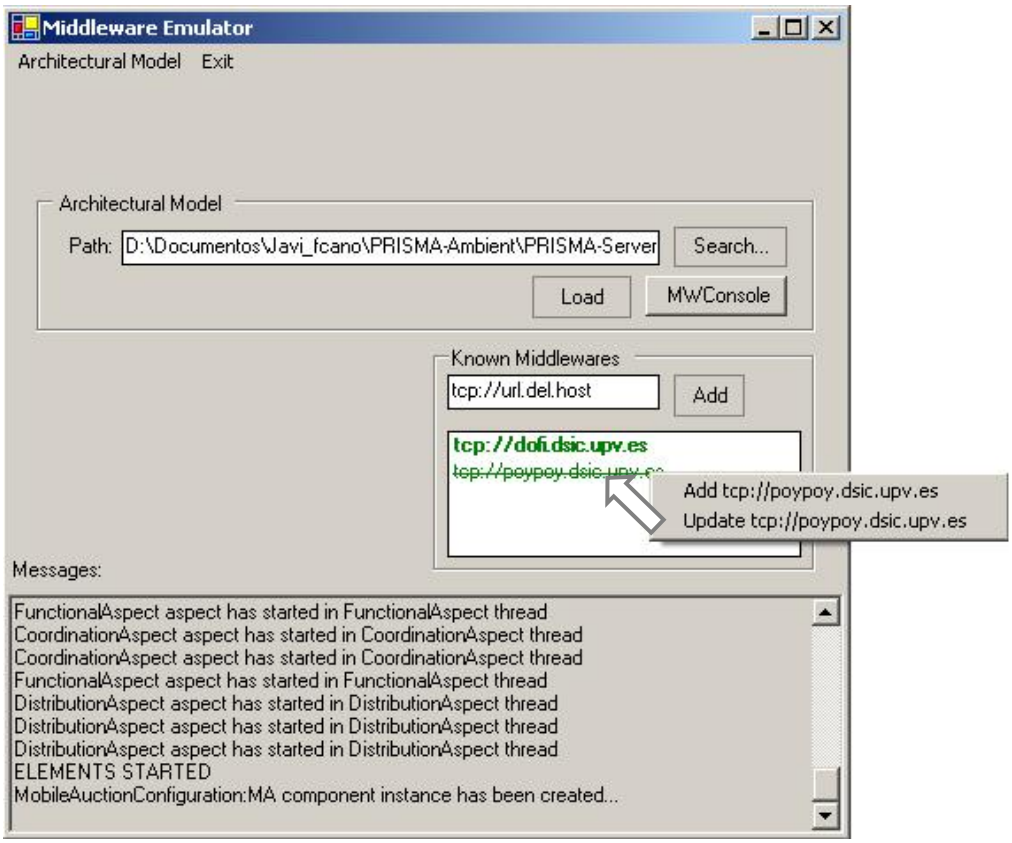



**Figura 48: Panel Known Middlewares en el Middleware Emulator**

Se trata de un panel en el que se muestran los *middlewares* implicados en el entorno de ejecución. En la lista se muestran en color verde los *middlewares* en ejecución y en color rojo los *middlewares* que han dejado de responder. El elemento en negrita representa al middleware que se encuentra en la misma máquina que el *Middleware Emulator* y con el que interactúa directamente. También incorpora un menú contextual para cada elemento de la lista mediante el que se puede eliminar o refrescar. Si se elimina un middleware que está en ejecución, aparecerá en la lista tachado; el panel continuará detectando su estado, pero los elementos arquitectónicos no interactuarán con él. De esa forma se puede volver a añadir en el futuro haciendo uso del menú contextual, pues la opción *Delete* se cambia por *Add* para los elementos que se hayan eliminado manualmente. En la Tabla 16 se muestra el código de representación empleado para la interacción con el usuario.

| Representación | Significado |
|---|---|
| tcp://url.del.host | Host con un middleware PRISMANET en ejecución |
| ~~tcp://url.del.host~~ | Host en el que se ejecutaba un middleware PRISMANET que ya no está en funcionamiento |
| tcp://url.del.host | Host con un middleware PRISMANET en ejecución, pero que se ha eliminado manualmente |
| ~~tcp://url.del.host~~ | Host en el que se ejecutaba un middleware PRISMANET que ya no está en funcionamiento y que se ha eliminado, o se eliminó manualmente |

| | |
|---|---|
| tcp://url.del.host | Host que se comunica directamente con el Middleware Emulator. |

**Tabla 16: Representación de los middlewares en el Panel de Middewares Conocidos**

Para que el usuario pueda añadir nuevos *middlewares* se añade un campo de texto con un botón *Add* en el que se escribirá la URL del *middleware* a añadir. La información reflejada en el panel se extrae de la Lista de Middlewares Conocidos, por lo que el usuario es notificado de los cambios ocurridos en el entorno de ejecución.

## 4.3. Implementación de un DNS distribuido

Como se comentó en el Capítulo 3, en la sección en la que detallaba la arquitectura de PRISMANET, el *middleware* tiene una capa de DNS a la cual pregunta la ubicación de los elementos arquitectónicos. Por otro lado, hay ejecutándose un servidor DNS en una ubicación conocida, y la capa de DNS de los diferentes *middlewares* se conectan a ella para añadir, borrar o consultar ubicaciones de componentes, conectores o sistemas (Figura 49).

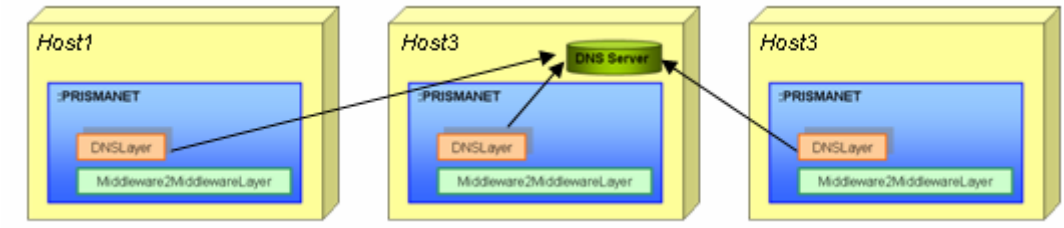



**Figura 49: DNS-Server centralizado**

Como se puede intuir, esta solución no es la más óptima. Su naturaleza centralizada hace que si la máquina en que se está ejecutando, por lo que sea, falla y, por tanto, se pierdan todos los registros. En las secciones siguientes se muestra una solución óptima.

El DNS es una herramienta muy útil para el *middleware*, por lo que es necesario que su funcionalidad se proporcione de forma consistente. A continuación se explican algunos de los casos en los que se hace uso de la capa DNS para obtener información de la ubicación de elementos arquitectónicos:

### Movilidad

- Cuando se mueve un componente se accede al *DNSLayer* del *middleware* origen para desregistrarlo (`Move`) y al del *middleware* destino para registrarlo (`TransferComponent`)

- Para determinar si un componente es distribuido, se usa el *DNSLayer* para localizar el *middleware* en el que está y llamar al

método `IsDistributedThisComponent` del *DistributionLayer* de dicho *middleware*

**Gestión de elementos arquitectónicos**

- Cuando se crea un componente nuevo, se utiliza el *DNSLayer* para comprobar si ya existe algún componente con ese nombre.

- Cuando se quiere eliminar un componente en un *middleware*, y el componente no se encuentra en el propio middleware, se obtiene del *DNSLayer* su ubicación y se reenvía la petición de eliminar.

- Lo mismo ocurre cuando se quiere iniciar o detener un componente; si el *middleware* no tiene ese componente, le reenvía la petición al *middleware* que lo tenga, previa consulta al *DNSLayer* para obtener su localización.

- Para obtener la interfaz de un puerto concreto de un determinado componente, se utiliza la capa de DNS ubicar el componente.

- Para obtener la localización de las dos partes de un *attachment*, se extraen los nombres de los componentes del nombre del *attachment* y se pregunta al *DNSLayer*.

- Para eliminar un `ComponentBinding`, si no lo tiene en local, el *middleware* pregunta al *DNSLayer* para redirigir la llamada.

Como se puede observar el uso del *DNSLayer*, y por tanto, del *DNS-Server* es bastante requerido por las diferentes capas del *middleware*, por lo que surge la necesidad de darle una implementación no centralizada más fiable ante la ocurrencia de fallos en los *middlewares*.

### 4.3.1. Solución descentralizada

Una primera aproximación aprovecha la incorporación de la Lista de Middlewares Conocidos para descentralizar la solución. En esta aproximación, el *DNS Server* se ejecuta en cada nodo y solo guarda la información relativa a ese nodo (ver Figura 50). De esa forma, el único que añade, borra o consulta ubicaciones es la capa DNS del *middleware* que se está ejecutando en dicho nodo. Así, por ejemplo, cuando un elemento arquitectónico se mueve de un middleware a otro, el *middleware* origen lo borra de su *DNS-Server* y el *middleware* destino lo añade al suyo.

Cuando un *middleware* busca información sobre la ubicación de, por ejemplo, un componente, éste reenvía la petición a través de su capa DNS al *DNS Server* local. Si el *DNS Server* local no contiene información de ese componente (la petición retorna una cadena vacía) la capa de DNS recorre la

lista de middlewares conocidos y, a través de la capa `Middleware2Middleware` pregunta al *DNSLayer* de cada *middleware* en busca del componente. Si tras preguntar a todos los *middlewares* no ha localizado el componente, el *DNSLayer* original, el que ha iniciado la búsqueda, lanza una excepción.

Esto es posible gracias a que hay dos métodos para acceder al *DNS Server*: El primero, que es con el que se pregunta al *DNSLayer* del *middleware* `GetLocationOfComponent`, buscará el componente en todos los DNS Servers asociados a todos los *DNSLayer* de todos los *middlewares*. El segundo, `GetURL`, publicado a los demás *middlewares* mediante la capa `Middleware2Middleware` únicamente pregunta al DNSServer del middleware al que pertenece por el componente. Es decir, el método `GetLocationOfComponents` utiliza la capa `Middleware2Middleware` para preguntar a los *middlewares* conocidos si tienen el componente.

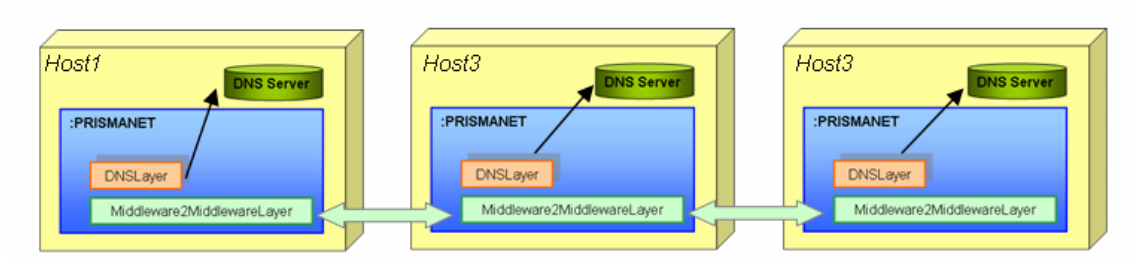



**Figura 50: DNS-Server descentralizado**

Esta solución solventa el problema de la tolerancia a fallos, mayor inconveniente de la solución centralizada, pues si falla un *middleware*, tanto los elementos arquitectónicos que se están ejecutando en él, como su DNS dejan de estar accesibles, de forma que, el resto de *middlewares* puede seguir funcionando, eso si, sin tener acceso a los elementos arquitectónicos que se encontraban en el *middleware* caído, cosa que por otro lado, tampoco es importante porque esos elementos arquitectónicos ya no están en ejecución.

También cabe comentar que como inconveniente añadido, la nueva solución tiene un tiempo de resolución más alto: Si el elemento no es local la resolución ya no es directa, supone interrogar a otros *middlewares* a través de la red, con lo que el proceso es más costoso temporalmente. Para optimizar el proceso de resolución se ha incorporado una caché de resoluciones. De esa forma el proceso de localización se optimizaría considerablemente.

Ahora bien, esta nueva solución descentralizada sigue arrastrando algunos inconvenientes que ya se tenían con el caso centralizado: Se está guardando dos veces la misma información, ya que la lista de componentes de un DNS es la misma que la lista de componentes que guarda la *ElementManagementLayer*. Además de la redundancia, esto hace que, al trabajar con listas diferentes que han de tener los mismos elementos, se puedan alcanzar estados inestables si se produjese una desincronización entre ambas listas. Esto no era tan palpable en la solución centralizada, ya que sólo un subconjunto de las entradas del DNS eran redundantes: Aquellas que

representaban a elementos ubicados en el mismo *middleware* que el DNS. A continuación se explica que medidas se han tomado para resolver este problema.

### 4.3.2. Resolución mediante la Capa de Gestión de Elementos

Puesto que en esta nueva solución se tiene la misma información en los dos sitios y esa información representa lo mismo, una segunda aproximación propuesta parte precisamente de eliminar uno de esos sitios.

Como cada *middleware* tiene su propia lista de elementos arquitectónicos locales, a la que solo él accede, no es necesario tener un programa exclusivo que ejerza de servidor. Un objeto interno al *DNSLayer* puede guardar la lista de los componentes que se ejecutan en el *middleware*, para ser accedido a través del *DNSLayer*, ya sea por el propio *middleware* o por otro, a través del *Middleware2MiddlewareLayer*. Es más, puesto que la lista de componentes que contendrá este objeto interno, será la misma que la que contiene el *ElementmanagementLayer*, se ha optado por utilizar esta lista para resolver las peticiones al *DNSLayer*. Por ello, el *DNSLayer* preguntará al *ElementManagementLayer* por los componentes. En concreto hará uso del método público `GetComponent` que ofrece el *ElementManagementLayer* (ver Figura 51).

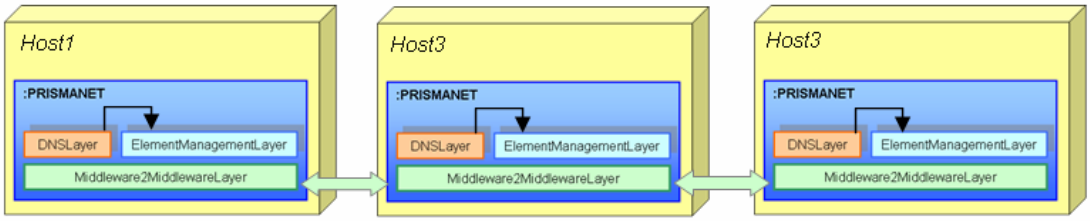



**Figura 51: El DNSLayer interroga directamente al ElementManagementLayer**

El cambio conceptual que ha supuesto la implementación del DNS distribuido ha supuesto eliminar bastantes líneas de código. Por un lado, la clase *DNSServer* que representaba al DNS ya no se utiliza más, como tampoco lo hace *DNSLoader*, la clase que la ponía en ejecución. Por otro lado, gran parte de los métodos de la *DNSLayer*, como los que gestionaban la lista de componentes (`Add`, `Remove`, `Update`) ya no son necesarios, y se han podido eliminar, al igual que las llamadas a estos métodos que se realizaban en las diferentes capas del *middleware* (al añadir, eliminar o mover componentes, por ejemplo). Esto es debido a que al acceder la *DNSLayer* a la lista de componentes de *ElementManagement*, la actualización de esta lista se hace transparente para ella, dejando esa tarea a la capa de gestión de elementos.

Además, se ha añadido una caché en cada *DNSLayer* donde se guardarán las últimas resoluciones realizadas, para de esta forma agilizar el proceso. Para

representar la caché se ha utilizado una tabla *hash*, en la que se introducirán los pares (nombre de componente, URL) indexados por el nombre del componente (su código *hash*). Esta tabla será consultada previamente a realizar la búsqueda global a través de los *middlewares* conocidos y será actualizada tras realizar una búsqueda de este tipo. Puesto que la aparición de un componente en la lista no garantiza que la URL a la que esté relacionado sea la correcta (por ejemplo el componente se puede haber movido desde la inserción del dató en la caché del *middleware* actual), cuando se obtenga un dato de esta lista se comprobará que es correcto, verificándolo directamente con la URL resuelta por la caché. Como tarea a realizar en futuras versiones se podría considerar un mecanismo que, cuando un elemento se mueva, deje en el *DNSLayer* información sobre a que máquina se ha movido para redirigir a los que pregunten por él en el futuro. Seria similar al que se ha explicado en la sección 4.1.4 para permitir que dos *subscribers* se muevan simultáneamente, pero enfocado tanto a los *subscribers* como a los elementos arquitectónicos. Este mecanismo debería actualizar la caché tras una redirección para evitar generar cadenas de redirección muy largas, lo que puede ser problemático [Bau00]. De esa forma, si se resuelve la petición por caché, y el elemento a resolver se ha movido desde la última petición, el *DNSLayer* puede obtener semidirectamente la ubicación siguiendo las redirecciones.

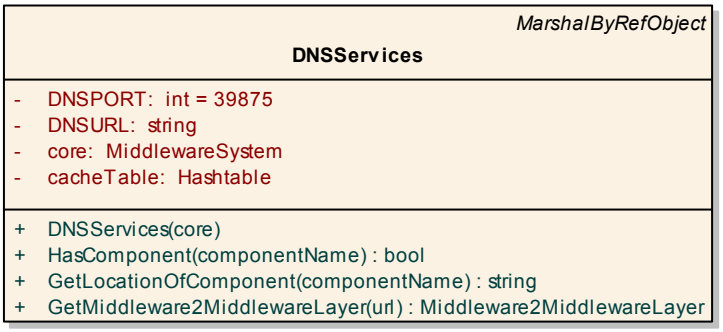



**Figura 52: Clase DNSServices**

Tras aplicar los cambios se eliminan todas las operaciones relativas al mantenimiento del DNS (añadir componente, eliminar componente...) de los métodos de creación, borrado y movilidad de componentes, pues esas tareas ya las realiza implícitamente el *ElementManagementLayer*. De hecho, a parte del constructor, tan solo han quedado dos métodos relevantes en la clase *DNSServices*: `HasComponent` y `GetLocationOfComponent` (ver Figura 52).

Por un lado `HasComponent`, método que devuelve un booleano que indica si el *middleware* tiene el elemento cuyo nombre se pasa como parámetro. En el cuerpo del método tan solo se hace una llamada al método público de mismo nombre de la *ElementManagementLayer* que busca el componente en la lista de componentes de la capa.

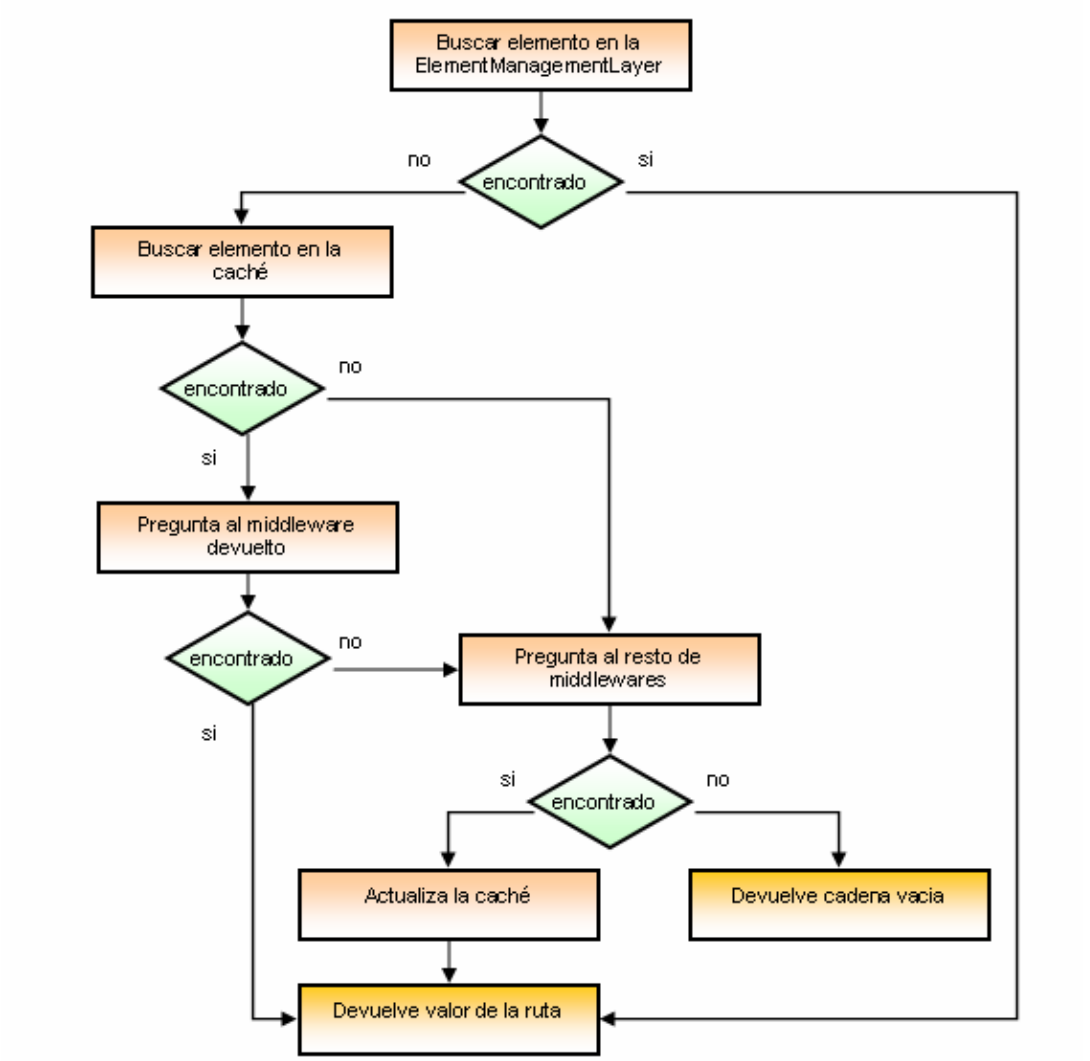



**Figura 53: Diagrama de flujo del proceso de resolución**

Por otro, `GetLocationOfComponent`, método que encapsula toda la lógica de localización de componentes entre todos los *middlewares* que conforman el entorno de ejecución de PRISMA. El método sigue los siguientes pasos para encontrar la ubicación de un componente (ver Figura 53):

1. Comprueba si el elemento está en el mismo *middleware*. En caso afirmativo va al paso 4.
2. Busca el componente en la caché de componentes.
    a. Si lo encuentra, llama al *middleware* ubicado en la URL obtenida para preguntarle por el componente, por si este se hubiese movido. Si lo encuentra va al paso 4.
3. Busca el componente en el resto de *middlewares*, para ello, obtiene un subconjunto de la lista de middlewares conocidos, donde no se encuentra el *middleware* en el que se está ejecutando la petición (donde ya se ha buscado en el paso 1) ni el *middleware* donde apuntaba la caché (en caso que el componente estuviera en la caché ) .

   a. Si lo encuentra, obtiene la ruta del *middleware* que ha respondido positivamente y actualiza la caché.
   b. Si no lo encuentra iguala la ruta del *middleware* a la cadena vacía y, en caso de estar el componente en la caché, lo elimina.
4. Devuelve la ruta del *middleware* donde se encuentre el componente (la cadena vacía en caso de no haberlo encontrado)

De esta forma se ha distribuido el DNS entre los diferentes *middlewares* que componen el entorno de ejecución y se han proporcionado medios para optimizar la localización de de los elementos arquitectónicos. La lógica ha sido encapsulada en la capa *DNSLayer*, que con la ayuda de la capa *ElementManagementLayer* y la *MiddlewareListManagement*, es capaz de resolver la ubicación de cualquier elemento arquitectónicos en ejecución en el entorno de ejecución PRISMA. En la Figura 54 se muestra como ha quedado la clase *MiddlewareSystem* tras la aplicación de los cambios.

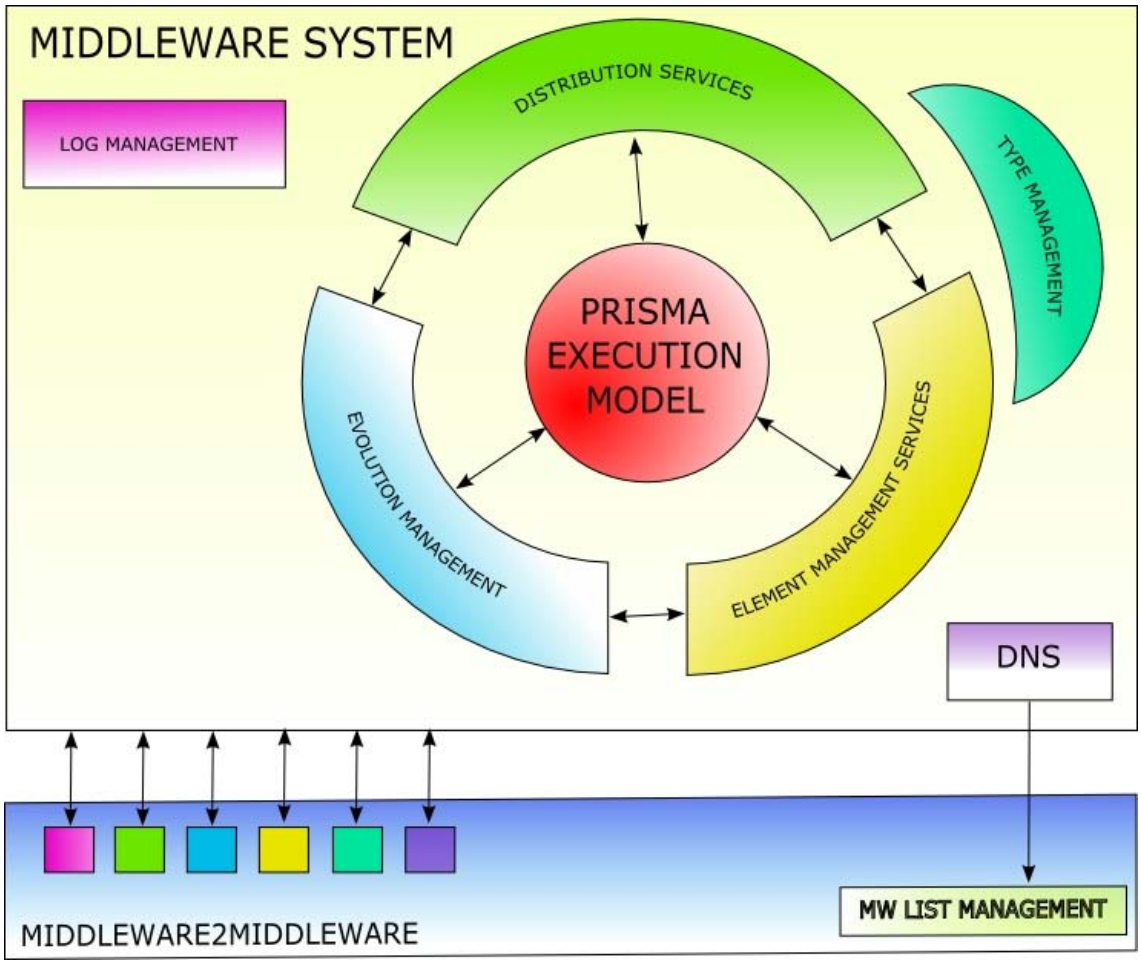



**Figura 54: Diagrama de capas del middleware PRISMA tras la incorporación del DNS distribuido**

## 4.4. Transacciones en el modelo distribuido de PRISMA

En la presente sección se explicará como se ha dotado al modelo de ejecución PRISMA de un mecanismo para poder ejecutar servicios de forma transaccional. Pero antes de entrar directamente a la exposición de como todo esto se ha llevado a cabo, se presenta una subsección para introducir brevemente algunos conceptos básicos de los sistemas transaccionales.

### 4.4.1. Introducción a las transacciones

Para el correcto funcionamiento de los sistemas de información es necesario proporcionarles un mecanismo para que la ejecución de un conjunto de servicios se comporte como una única unidad de trabajo. Esto es, proporcionar un comportamiento transaccional [Gray93] bajo el cual, o se ejecutan correctamente todos los servicios involucrados, o no se ejecuta ninguno. Las transacciones se describen comúnmente mediante las propiedades *Atomicity*, *Consistency*, *Isolation and Durability* (ACID). A continuación se explican estas propiedades:

- **Atomicidad (*Atomicity*):** Se refiere al hecho de que una transacción se trata como una unidad de operación. Por lo tanto, o todas las acciones de la transacción se realizan o ninguna de ellas se lleva a cabo. La atomicidad requiere que si una transacción se interrumpe por un fallo, sus resultados parciales deben ser deshechos. La actividad referente a preservar la atomicidad de transacciones en presencia de abortos debido a errores de entrada, sobrecarga del sistema o interbloqueos se le llama *recuperación de transacciones*. La actividad de asegurar la atomicidad en presencia de caídas del sistema se le llama *recuperación de caídas*.

- **Consistencia (*Consistency*):** La consistencia de una transacción es simplemente su corrección. En otras palabras, una transacción es un programa correcto que lleva al sistema de un estado consistente a otro con esta misma característica.

- **Aislamiento (*Isolation*):** Una transacción en ejecución no puede revelar sus resultados a otras operaciones y/o transacciones concurrentes antes de su confirmación. Es más, si varias transacciones se ejecutan concurrentemente, los resultados deben ser los mismos que si ellas se hubieran ejecutado de manera secuencial (seriabilidad).

- **Durabilidad (*Durability*):** Es la propiedad de las transacciones que asegura que una vez que una transacción se confirma, sus resultados son permanentes

A propósito de la propiedad de aislamiento se pueden producir diversos fenómenos en cuanto a la lectura de datos afectados por otras transacciones que han de prevenirse convenientemente:

- **Lectura sucia (*Dirty read*):** Se da cuando una transacción lee datos modificados por una transacción concurrente que todavía no ha sido confirmada.

- **Lectura no repetible (*Nonrepeatable read*):** Fenómeno que se produce cuando, durante el transcurso de una transacción, se lee un dato que ha sido previamente leído y que ha sido modificado por otra transacción que ha sido confirmada desde la última lectura.

- **Lectura fantasma (*Phantom read*):** Una transacción vuelve a ejecutar un servicio obteniendo un resultado diferente debido a la finalización y confirmación de otra transacción.

Las transacciones pueden ser de varios tipos. Aun cuando los problemas fundamentales son los mismos para los diferentes tipos, los algoritmos y técnicas que se usan para tratarlos pueden ser considerablemente diferentes. Las transacciones pueden ser agrupadas a lo largo de las siguientes dimensiones:

- **Áreas de aplicación:** En primer lugar, las transacciones se pueden ejecutar en aplicaciones distribuidas o no distribuidas. Las transacciones distribuidas son aquellas que operan con datos distribuidos. Por otro lado, dado que los resultados de una transacción que realiza un *commit* son durables, la única forma de deshacer los efectos de una transacción con *commit* es mediante otra transacción. A este tipo de transacciones se les conoce como transacciones compensatorias. Finalmente, en ambientes heterogéneos se presentan transacciones heterogéneas sobre los datos.

- **Tiempo de duración:** Tomando en cuenta el tiempo que transcurre desde que se inicia una transacción hasta que se realiza un *commit* o se aborta, las transacciones pueden ser de tipo *batch* o en línea. Estas se pueden diferenciar también como transacciones de corta y larga vida. Las transacciones en línea se caracterizan por tiempos de respuesta muy cortos y por acceder a una porción relativamente pequeña del sistema en cuestión. Por otro lado, las transacciones de tipo *batch* toman tiempos relativamente largos y acceden grandes porciones del sistema.

- **Estructura**: Considerando la estructura que puede tener una transacción se examinan dos aspectos: si una transacción puede contener a su vez subtransacciones o el orden de las acciones de lectura y escritura dentro de una transacción.

Respecto a la estructura de las transacciones, las transacciones planas consisten en una secuencia de operaciones primitivas. En las transacciones anidadas las operaciones de una transacción pueden ser así mismo transacciones.

Una transacción anidada dentro de otra transacción conserva las mismas propiedades que la de sus padres, esto implica, que puede contener así mismo transacciones dentro de ella. Existen restricciones obvias en una transacción anidada: debe empezar después que su padre y debe terminar antes que él. Más aún, el *commit* de una subtransacción es condicional al *commit* de su padre, en otras palabras: Si el padre de una o varias transacciones aborta, las subtransacciones hijas también serán abortadas.

Las transacciones anidadas proporcionan un nivel más alto de granuralidad a las transacciones. Ya que debido a que una transacción puede estar formada por varias transacciones, es posible ampliar la modularidad del proceso transaccional. Así, por ejemplo, es posible recuperarse de fallos de manera independiente de cada subtransacción. Esto limita el daño a una parte más pequeña de la transacción, haciendo que el coste de la recuperación sea menor.

Las transacciones también pueden ser catalogadas en función del orden de las lecturas y escrituras. Si las acciones de lectura y escritura pueden ser mezcladas sin ninguna restricción, entonces, a este tipo de transacciones se les conoce como generales. En contraste, si se restringe o impone que un dato deber ser leído antes de que pueda ser escrito entonces se tendrán transacciones restringidas. Si las transacciones son restringidas a que todas las acciones de lectura se realicen antes de las acciones de escritura entonces se les conoce como de dos pasos. Finalmente, existe un modelo de acción para transacciones restringidas en donde se aplica aún más la restricción de que cada par <read,write> tiene que ser ejecutado de manera atómica.

En el anexo 2 se han descrito mecanismos que proporciona la plataforma .NET, sobre la que se desarrolla el proyecto, para dar soporte a los procesos transaccionales. No obstante, debido a la complejidad del modelo PRISMA, la aplicación de dichos mecanismos supondría grandes modificaciones en el *middleware*, pues las soluciones proporcionadas por .NET se centran en programación orientada a componentes COM, mientras que el *middleware* se ha implementado sin usar esta tecnología. Por otro lado, las soluciones existentes no tienen noción de aspecto, cosa que es fundamental para PRISMA. Es por todo ello que se ha optado por una solución ad hoc, eso sí, basándose en los conceptos de gestión de transacciones que presentan las otras alternativas, pero orientando la solución a PRISMA, o en términos más generales, al DSOA y al DSBC.

### 4.4.2. Transacciones en PRISMA

En PRISMA las transacciones describen un conjunto de operaciones que se han de ejecutar cumpliendo las propiedades ACID. En el LDA-OA las transacciones se definen en la sección *Transactions* usando la siguiente sintaxis:

```
transactionalService(Type1 param1, Type2 param2 ... TypeN paramN):
      TRANSACTIONALSERVICE = {condition1} Service1?(param1)→STEP2;
      STEP2= {condition2} Interface1.Service2!(param2)→STEP3;
      (...)
      STEPN= {condition3} ServiceN?(paramN);
```

Tal y como se muestra en el fragmento de pseudocódigo, una transacción tiene un nombre (`transactionalService`) y un conjunto de parámetros. Estos parámetros se usarán en el interior de la transacción para invocar los diferentes servicios que la componen. A continuación se describe cada uno de los pasos, o servicios que ha de ejecutar la transacción, recibiendo el primero de ellos el nombre de la transacción. Los servicios que se ejecutan dentro de la transacción pueden llevar una condición asociada, de forma que, si no se cumpliese se debería abortar la transacción. También se pueden introducir bloques *If-Then-Else* para realizar acciones alternativas en función de determinada condición. Por otro lado los servicios que se pueden invocar pueden ser públicos o privados. Los servicios públicos irán precedidos de la interfaz que los define, como es el caso de Service2. En cualquier caso, se indica el tipo de canal, esto es, si es de entrada o de salida mediante los símbolos `?` y `!` respectivamente tras el nombre del servicio.

#### *4.4.2.1. Transacciones Homogéneas e Heterogéneas*

Como se puede intuir, los servicios implicados en una transacción pueden recibir semántica en aspectos diferentes al que define la transacción. Estas son las denominadas transacciones heterogéneas, frente a las transacciones homogéneas, que solo definen llamadas a servicios ofrecidos por el mismo aspecto que define la transacción. Del mismo modo, en una transacción se pueden requerir los servicios ubicados en otros aspectos del mismo componente a través de los *weavings* definidos. Esto último se podría considerar como un caso especial de transacciones heterogéneas, ya que afecta a más de un aspecto.

En definidas cuentas, una transacción ha de poder extenderse tanto a través de los aspectos de un componente como a través de diversos componentes de una arquitectura. Esto es importante, puesto que en función del alcance de dicha transacción, los cambios que se deben confirmar o cancelar afectarán a más de un elemento que pueden además ser distribuidos. Como se verá en las siguientes secciones, estos cambios pueden afectar al estado del componente o componentes implicados o al estado de la arquitectura. La transacción va a determinar cuales son los cambios que se han de confirmar o

cancelar en los elementos implicados. En ese sentido se ha adoptado el concepto de contexto transaccional para definir el límite de una transacción. Así, toda transacción estará ligada a un contexto transaccional que la representará más allá de los límites del *middleware* en el que se ha iniciado, como se verá más adelante.

#### *4.4.2.2. Transacciones anidadas*

Otro aspecto a tener en consideración es el funcionamiento frente a las transacciones anidadas. Como se ha comentado previamente, una transacción anidada es aquella en que alguno de las operaciones de la transacción es a su vez otra transacción que puede también ser anidada. De esa forma se crea una jerarquía transaccional que se podría representar en forma de árbol donde la raíz o *root* sea la transacción que ha iniciado en primera instancia, es decir, aquella bajo la que se han invocado todas las subtransacciones o transacciones anidadas.

En la Figura 55 se muestra una transacción anidada (TX1) en la que se definen dos subtransacciones, TX2 y TX5, respectivamente. TX2, a su vez, también está anidada y desde ella se invoca a las transacciones TX3 y TX4. Junto al esquema de llamadas, en el que se ve como van invocando secuencialmente las diferentes transacciones, se muestra un árbol que representa la jerarquía.

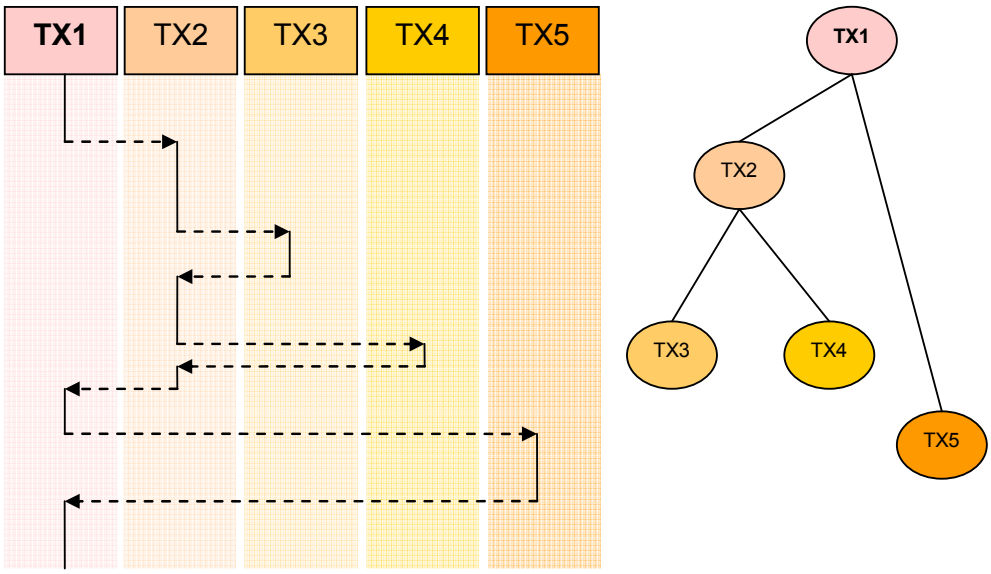



**Figura 55: Transacciones anidadas**

Cuando en una transacción anidada alguna de sus transacciones falla y ese fallo no es controlado, se desharán todas las acciones realizadas por la transacción y por lo tanto todas las transacciones que cuelgan del subárbol del que dicha transacción es raíz (subtransacciones). No obstante el *commit* de una

transacción que no se encuentra en el nivel máximo de la jerarquía, no se puede considerar un *commit* total en el sentido habitual, ya que, como se ha descrito, dependerá de lo que ocurra en las transacciones que están por encima de ella. De todas formas, este *commit* indicará que los datos modificados por la transacción confirmada serán accesibles por la transacción padre y las transacciones de su entorno. Se puede ver el *commit* de una subtransacción como un *commit* relativo o parcial. El único *commit* absoluto será el que provoque la transacción *root*. Siguiendo con el ejemplo de la figura Figura 55, si la transacción TX3 se confirma, tanto TX2 como TX4 tendrán acceso a sus resultados. Si durante la ejecución de TX4 se produce un fallo, el aborto de la subtransacción, en principio no afectará a TX3, sin embargo, si fuese TX2 la que falla, los cambios de TX3 y TX4 deberán ser cancelados, se hayan confirmado o no. Evidentemente, el fallo de TX4 supondrá un fallo en TX2, pues es su transacción padre, por lo que el fallo, indirectamente si afecta a TX3. No obstante ha de contemplar la posibilidad que TX2 podría estar preparada para esperar ese fallo y tratarlo de alguna forma, evitando así el fallo recursivo de toda la transacción. De esa forma, el fallo de TX4 seria capturado y tratado por TX2 de forma que no afectase a la subtransacción confirmada TX3, ni se propagase hasta TX1. Esto, en todo caso, debería ser considerado por el diseñador de la transacción y especificado explícitamente ya que, por defecto, si falla una subtransacción, el fallo se propaga a toda la estructura transaccional en la que se encuentra.

#### *4.4.2.3. Modelo de ejecución*

Cuando se está ejecutando una transacción durante la cual se requiere la ejecución de un servicio ubicado en otro aspecto, ya sea del mismo componente o no, los cambios producidos por la ejecución de ese servicio externo en el estado del aspecto en el que se ejecuta han de poder deshacerse en caso de que la transacción en la que participa sea abortada. El aspecto destino ha de ser informado de que la ejecución del servicio pertenece a una transacción, y que por tanto ha de guardar su estado antes de comenzar. Para garantizar el aislamiento de la transacción, el aspecto se ha de bloquear hasta que termine la transacción, porque la ejecución de servicios, antes de que termine la transacción, pueden volver a modificar el estado, u obtener resultados no confirmados.

La opción del bloqueo supone una primera aproximación al problema. Es evidente que mediante esta solución, si bien se simplifica el proceso, se introduce una limitación en la concurrencia entre transacciones. Por ejemplo, se puede pensar en una transacción larga: Todos los aspectos involucrados en dicha transacción se bloquearán hasta su finalización, por lo que pondrán en cola el resto de peticiones, que durante la ejecución de la transacción se realicen a los aspectos procedentes de otros contextos, no pertenecientes a la transacción que está ejecutando.

Esta limitación se tiene en cuenta, pero se ha dejado para futuras versiones la incorporación de sistemas más eficientes para garantizar el aislamiento de la transacción. Estas soluciones pasarán por marcar los atributos de un aspecto que se vean modificados por la transacción, para evitar el bloqueo total del aspecto. Pero, como se ha dicho, la finalidad de la presente sección es introducir las transacciones distribuidas en el *middleware*, dejando la optimización del modelo de ejecución de las transacciones para trabajos futuros. Así pues, se continuará la explicación usando la solución del bloqueo de aspectos como primera aproximación para las transacciones en PRISMANET.

Pero el bloqueo tampoco presenta una solución definitiva por ejemplo el caso en que en una transacción se requieran dos servicios de un aspecto que no es el que ha iniciado la transacción. Cuando se ejecute el primer servicio, el aspecto en que se ejecute salvará el estado, realizará las acciones con las que se da semántica el servicio y se bloqueará a la espera de una orden de *commit* o de *rollback*. No obstante, antes que esas órdenes, llegará la segunda petición de servicio perteneciente a la transacción, que se quedará para siempre bloqueado en la cola del proceso, provocando así un interbloqueo.

Para evitar este problema, se ha optado por proporcionar un mecanismo de desencolado por prioridades en los aspectos implicados en una transacción que únicamente procese peticiones de servicio relacionadas con la transacción en la que se está ejecutando. Esto se verá con detalle más adelante en la presente sección.

Para las transacciones entre elementos arquitectónicos, además, hay que proporcionar un mecanismo para que el aspecto originador de la transacción, es decir, aquel en el que está definida la transacción, esté informado de los posibles fallos que se produzcan en los aspectos locales o remotos. Para obtener esta información sobre el estado de la transacción, se ha introducido un mediador que gestionará el contexto transaccional. Como el contexto de la transacción puede ir más allá del límite marcado por el *middleware* en que se ejecuta, se ha introducido este mediador en un nivel más alto: será una capa del *middleware*.

En el caso de que una transacción falle, no sólo hay que recuperar el estado previo de los aspectos cuyos métodos estén implicados en la transacción, ya que, como resultado de la ejecución de determinados servicios proporcionados por los aspectos, también es posible modificar la arquitectura PRISMA. Un ejemplo de esto seria una transacción que tuviese implicado una operación de movilidad de un componente; la ejecución de dicho servicio implicaría la creación del componente en el *middleware* destino, así como la actualización de *attachments*, *bindings* etc. Si la transacción fallase posteriormente a la ejecución de dicho servicio, sería necesario devolver el estado de la arquitectura a su estado anterior.

Del párrafo anterior se puede deducir que las transacciones en PRISMA se pueden definir a dos niveles: Transacciones a nivel de elementos

arquitectónicos y a nivel de arquitectura. Por un lado, las transacciones a nivel de elementos arquitectónicos, contemplan todos los cambios acaecidos en los aspectos durante el transcurso de la transacción. Por otro lado, las transacciones a nivel de arquitectura, tienen en cuenta las modificaciones del estado de la arquitectura provocadas por la ejecución de los servicios implicados en una transacción como es la movilidad de un componente.

Cada uno de estos tipos de transacciones serán explicados detalladamente en secciones posteriores, pero antes es necesario introducir más en detalle otro concepto: La Capa de Gestión de Transacciones o *Transaction Manager*.

#### *4.4.2.4. La capa Transaction Manager*

Como se ha comentado, la ejecución de una transacción puede ampliar sus límites más allá de un aspecto, invocando servicios de otros aspectos que pueden estar ubicados en el mismo elemento arquitectónico o en otros, que a su vez pueden estar ejecutándose en el mismo *middleware* o en otro. Como se puede ver, para que esto funcione correctamente, es necesario relacionar todos los aspectos implicados de una transacción de alguna forma para garantizar que la atomicidad de la transacción se extienda más allá de los límites marcados por el *middleware*. Esto se resuelve mediante la utilización de contextos transaccionales.

El contexto transaccional define los límites de una transacción. Todos los aspectos en los que se ejecuten servicios invocados dentro de una transacción, esto es, invocándolos desde un lugar alcanzado por el código marcado como transaccional, han de ser avisados de ese hecho. De esa forma, estos aspectos, una vez hayan ejecutado un servicio para la transacción, sólo aceptarán peticiones pertenecientes al mismo contexto transaccional que, en cierta manera, les ha bloqueado. Así se evita que desde otros lugares se pueda acceder a propiedades no confirmadas, garantizando así el aislamiento de la transacción. Por ello, es necesario que los aspectos conozcan el contexto transaccional del cual provienen las peticiones de servicios.

Para proporcionar esta funcionalidad se han tenido que modificar el sistema de encolado y de envío de mensajes para añadir un atributo `ContextID`, que contendrá un valor identificador del contexto transaccional. En el caso que la petición no pertenezca a ningún contexto, se ha decidido usar como valor de contexto transaccional la cadena vacía. Cada vez que un aspecto procese un elemento de su cola que tenga el valor del `ContextID` definido, realizará una serie de acciones adicionales para gestionar la ejecución dentro de los límites de la transacción que representa el identificador.

Como se ha comentado anteriormente, debido a la naturaleza distribuida de las transacciones, para la gestión del contexto transaccional se ha añadido una nueva capa al *middleware*: La capa `TransactionManager`. En la Figura 56 se

muestra el esquema de las capas del *Middleware System* actualizada con la incorporación del *Transaction Manager*.

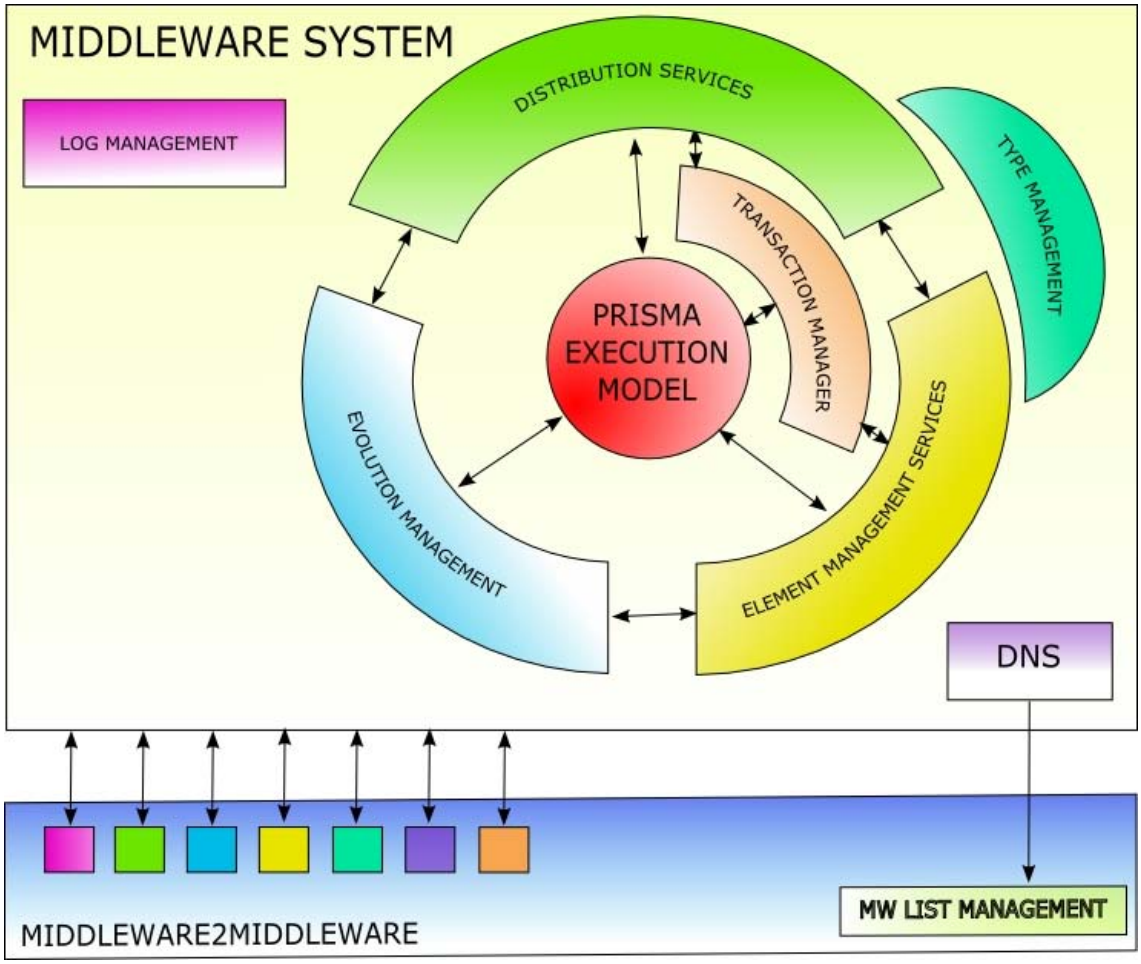



**Figura 56: La capa TransactionManager**

Como se puede apreciar en la figura, la capa de gestión de transacciones se sitúa entre el modelo de ejecución PRISMA y la capa de Servicios de Distribución y la de gestión de elementos. Esto es debido a que el gestor de transacciones actúa directamente con el modelo de ejecución

Cuando se comienza la ejecución de una transacción en un aspecto, se notifica al *middleware*, que añade una entrada en la lista interna del *Transaction Manager*. El *Transaction Manager* contiene una lista de contextos transaccionales en la que guarda la información relativa a las transacciones que se están ejecutando en el *middleware*. Los elementos de la lista son objetos de la clase `TransactionalContext`. En la Figura 57 se muestra un diagrama de clases en el que se puede apreciar cual es la relación entre las distintas clases.

La clase `TransactionManager` guarda un número de secuencia autoincrementable (`sequentialCount`), una referencia al *Middleware System* (`core`) y una lista de contextos transaccionales (`contextList`), donde guardará información sobre los contextos transaccionales en los que se encuentra involucrado dicho *middleware*. En el campo privado de la clase `LogginFile` se

guardará la ruta al fichero de *Log* de la transacción, pero eso se explicará más adelante.


**Figura 57: Las clases Transaction Manager y Transactional Context**

La clase `TransactionalContext` contiene un identificador de contexto (`contextID`) que es una cadena de texto formada por la URL del *middleware* donde se ha originado la transacción que ha dado lugar al contexto junto con un número autoincrementable. La propiedad `consistence` es la encargada de mantener información sobre los posibles fallos parciales de la transacción; por defecto tiene un valor cierto y se pondrá a falso en el momento que se produzca alguna excepción durante el transcurso de la transacción. Esta propiedad está inspirada en el *bit consistence* que se usa en la bibliografía de transacciones. La propiedad `done` indica el estado del contexto de la transacción, indicando cuando la transacción ha finalizado. Esta propiedad está basada en el *bit* del

mismo nombre usado en la bibliografía de transacciones. Además, como se pueden dar transacciones anidadas, que a su vez crearán nuevos contextos transaccionales, la clase `TransactionalContext` tiene un campo `parentID`, por defecto la cadena vacía, en el que guarda información referente al identificador del contexto transaccional bajo el cual se ha iniciado la subtransacción. Relacionado con esto está el campo `nestingLevel`, mediante el cual se especifica el nivel de anidamiento o profundidad del contexto transaccional respecto al contexto raíz, siendo 0 el nivel por defecto. En este punto cabe recordar el hecho que durante la ejecución de una transacción se puede crear una jerarquía de transacciones, es decir, transacciones anidadas que se van creando dentro de la transacción que a su vez pueden crear nuevas transacciones anidadas. Para dar soporte a esto se ha definido el concepto de contexto transaccional raíz o *root*. El *root* es la transacción que se encuentra más arriba en la jerarquía, y conocer su identidad será útil (`rootID`), como se verá más adelante, para evitar interbloqueos entre contextos transaccionales anidados. Con todo ello, el formato de la cadena que representa el `ContextID` tiene la siguiente forma:

```
contextID = seqNum{rootURL}[url.del.host]
```

Siendo `seqNum` un número entero secuencial gestionado por el `TransactionManager`, `url.del.host` la URL del middleware en que se crea y `rootURL` la URL del *middleware* donde se creó el contexto transaccional *root*.

Mediante una enumeración (`state`) se definen los diferentes estados en los que se puede encontrar un contexto transaccional. Una transacción puede estar en 5 posibles estados:

- `ACTIVE` : Transacción iniciada y en ejecución.
- `PRECOMMITTED` : Transacción finalizada, pero que al pertenecer a una jerarquía de transacciones debe esperar a que terminen todas las transacciones para confirmar el *commit* o, por el contrario, realizar un *rollback*.
- `PREROLLBACK` : Transacción finalizada, pero que debe esperar a que la transacción *root* confirme el *rollback* en el orden adecuado.
- `COMMITTED` : Transacción finalizada correctamente.
- `ROLLBACKED` : Transacción finalizada con fallo, que ha revertido los aspectos involucrados a su estado inicial.

De nuevo, el campo `architechturalOperations` hace referencia al proceso de anotación de acciones en un *Log*, proceso que se explicará más adelante.

Los `TransactionalContext` se pueden describir desde dos puntos de vista. Por un lado, un contexto transaccional ubicado en el *Transaction Manager* del *middleware* en que se ha creado y que es capaz de ampliar sus límites a otros

middlewares, es decir, un originador. Por otro lado, cada uno de esos *middlewares* alcanzados por un contexto transaccional definido en otro *middleware*, es decir, un involucrado. Las necesidades de un contexto transaccional en el *middleware* originador y en un *middleware* involucrado son diferentes; el primero necesita saber cual es su alcance, para poder avisar al resto de *middlewares* involucrados cuando la transacción termine, mientras que el segundo, tan solo necesita conocer al originador de la transacción, para informarle si algo ha ido mal durante la ejecución de la transacción en dicho *middleware*.

Por ello se ha especializado la funcionalidad del `TransactionalContext` en dos clases: `TransactionalContextInvolved` y `TransactionalContextOriginator`. En función del rol que desempeñe el *middleware* en que se añada un contexto transaccional a su *Transaction Manager*, se creará una instancia de uno o del otro.

Un `TransactionalContextOriginator` tiene una lista (`involvedList`) de middlewares involucrados en el contexto transaccional, esto es, aquellos cuyos componentes se haya extendido el contexto de la transacción. Esta lista será útil a la hora de confirmar o abortar la transacción por parte del originador, ya que tendrá una lista de los *middlewares* a los que ha de avisar. Además cuenta con un contador autoincrementable para nombrar a las transacciones anidadas (`sequencialCount`), de las cuales guardará identificación en la lista `childTransactions`.

Un `TransactionalContextInvolved` representa un contexto transaccional en un *middleware* que no es el originador de la transacción, por tanto, tan solo guarda la URL del originador (`originator`).

Gracias a este diseño es posible extender los límites de una transacción más allá de un *middleware*. Pero si se recuerda, se habían definido las transacciones en dos niveles: El nivel de elemento arquitectónico y el nivel de arquitectura, en función de si sólo modificaban el estado de los elementos arquitectónicos o también el estado de la arquitectura, respectivamente. Con la estructura de *Transaction Manager* ya se puede abordar la solución para la incorporación de comportamiento transaccional a cada uno de ellos. A continuación se describen más detalladamente estos dos niveles y se comentan las soluciones que se han adoptado haciendo uso del Transaction Manager.

#### *4.4.2.5. Transacciones a nivel de elementos arquitectónicos*

Para la incorporación del funcionamiento transaccional a nivel de elementos arquitectónicos, hay que tener en cuenta que, el estado de los elementos arquitectónicos lo proporciona el estado de los aspectos que le proporcionan la semántica. Por tanto, lo que se persigue es la forma de guardar el estado de los aspectos en un momento dado para, en caso que falle la

transacción, volver a recuperar ese estado, previo a las modificaciones realizadas por la transacción.

Se ha decidido usar el patrón Memento [Gamm95] como mecanismo para guardar el estado de un aspecto. El patrón Memento guarda el estado de un objeto en un instante dado para su recuperación posterior. En este caso, el objeto Memento contendría el estado interno del aspecto que ejecute el servicio dentro del contexto transaccional. Si la transacción es finalmente anulada, se recuperará el estado guardado en el objeto, si se confirma, se destruirá el Memento. De esta forma se ha obtenido un mecanismo de recuperación de estado interno al componente, totalmente transparente al *middleware* y que no viola la encapsulación de los datos del aspecto. En el anexo 1 se explica con más detalle el funcionamiento del patrón Memento.

Para la aplicación del patrón, se ha modificado la clase padre `AspectBase` (Figura 58), añadiéndole un nuevo objeto agregado de tipo `StateCareTaker` que será el encargado de encapsular al objeto Memento y la lógica para su creación y su recuperación y/o destrucción.

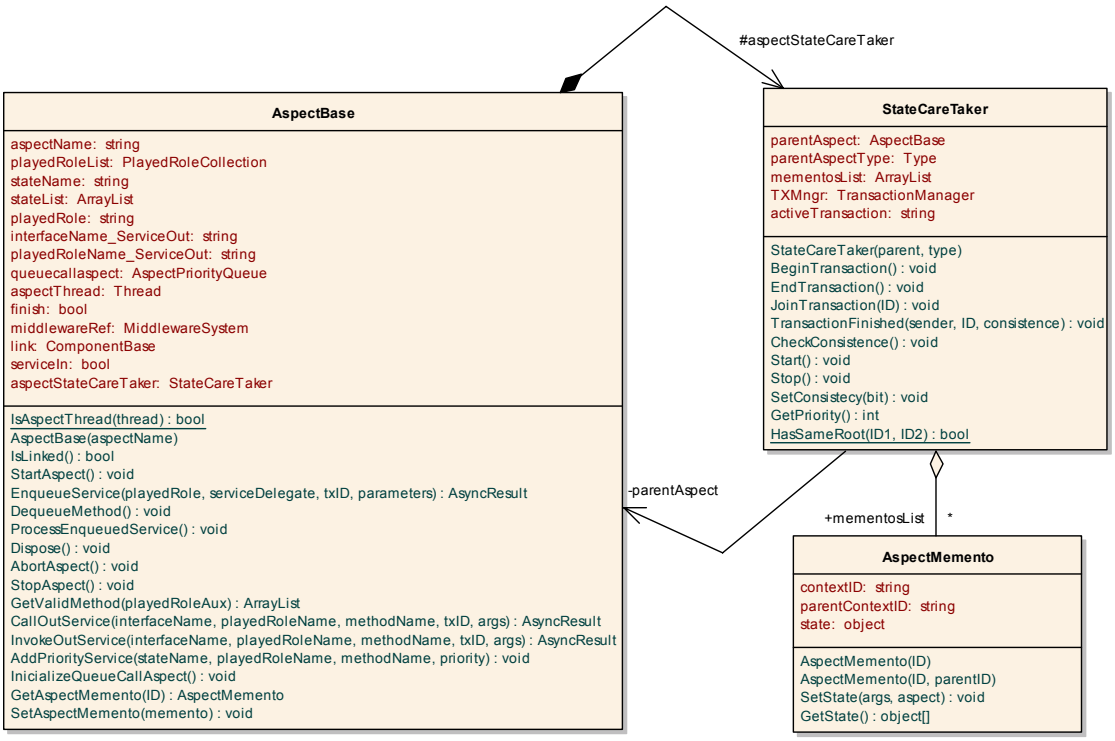



**Figura 58: El StateCareTaker y el AspectMenento**

El `StateCareTaker` tiene una referencia al aspecto al que pertenece (`parentAspect`) y al *Transaction Manager* (`TXMngr`) del *middleware* en el que se está ejecutando, pues él va a ser el intermediario entre el aspecto y el `TransactionManager` durante el proceso transaccional. Cabe remarcar que el hecho que el `aspectCareTaker` tenga una referencia a una capa del *middleware* requiere que dicha referencia se actualice si el elemento arquitectónico que contiene el aspecto se mueve a otro middleware. Para subsanar este hecho se ha delegado

la lógica de actualización de referencias al método `Start()` que, junto al `Stop()`, se ejecutarán en los procesos de inicio y detención del aspecto, respectivamente.

Otros campos importantes del `StateCareTaker` son el `mementosList`, una lista con los diferentes Mementos que guardan el estado del aspecto para los diferentes contextos transaccionales. Como se ha visto un aspecto puede entrar a formar parte de diversas transacciones siempre y cuando estén vinculadas mediante una relación de anidamiento. Cada uno de los Mementos guardará el estado en el momento que ha entrado el aspecto a formar parte de dicha transacción. Cada Memento pertenecerá pues a un contexto transaccional diferente y solo uno de esos contextos transaccionales estará siendo servido por el aspecto: Aquel indicado en el campo `activeTransaction`.

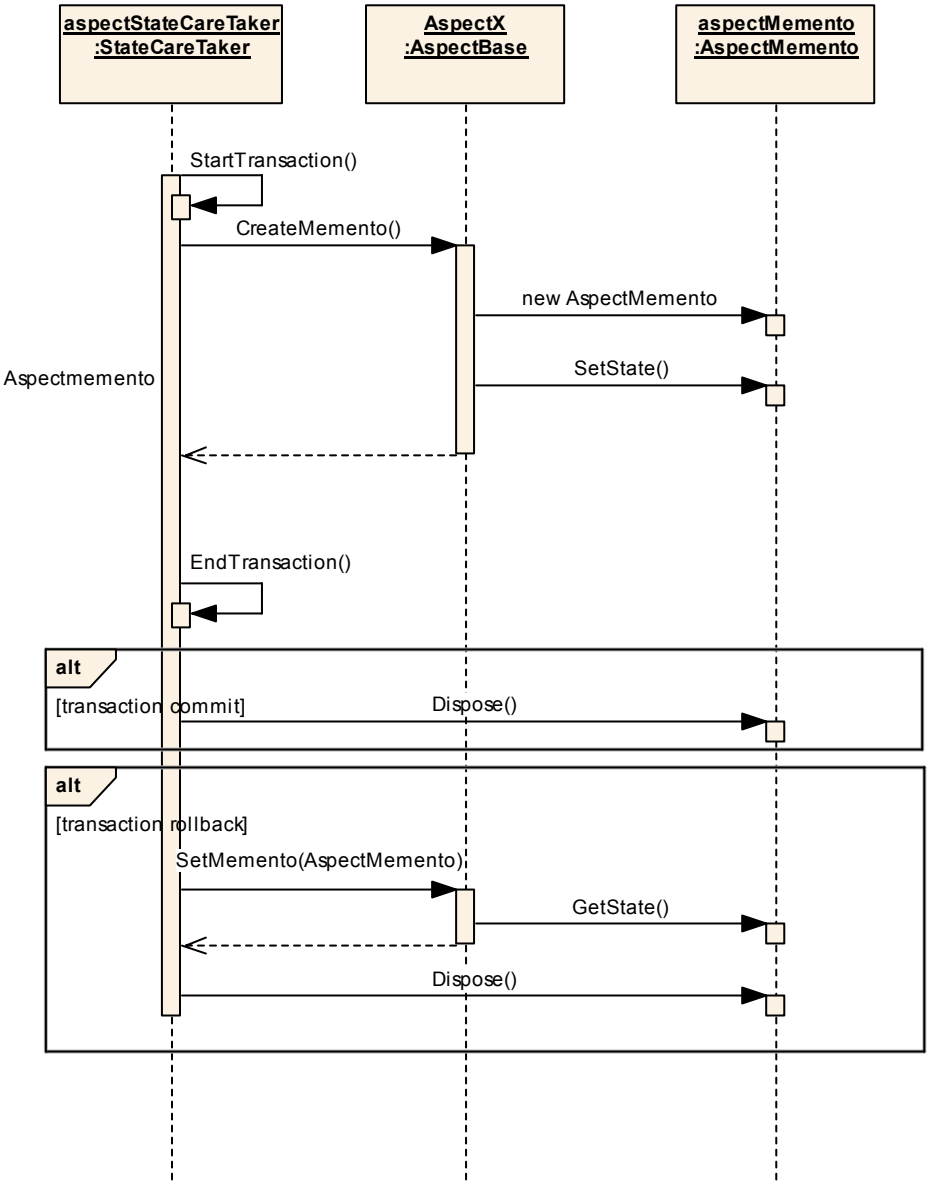



**Figura 59: Funcionamiento del AspectMemento**

El estado del aspecto se guarda en el `AspectMemento` dentro de un *array* multidimensional (`state`) que es rellenado en tiempo de ejecución utilizando los operadores de reflexión proporcionados por el espacio de nombres *System.Reflection*. De eso se ocupará el método `CreateMemento()`, que devolverá el objeto `AspectMemento` instanciado con el estado del aspecto en que es invocado. Así, en el *array* multidimensional que represente el estado, habrá una fila para cada atributo del aspecto; cada fila del *array* contendrá el trío (nombre, tipo, valor) del atributo guardado. Únicamente se guardarán aquellos atributos específicos del tipo de aspecto, ya que son los que potencialmente serán modificados durante la ejecución de servicios durante la transacción. En la Figura 59 se muestra el diagrama de secuencia asociado donde se muestra la interacción entre las diferentes clases al iniciar la transacción.

Mediante esta estructura el aspecto tiene todo lo que necesita para ejecutar servicios en un contexto transaccional: Por un lado el `aspectStateCareTaker` le abstrae de la lógica de iniciar una transacción así como de la comunicación con el *Transaction Manager*. Por otro lado, los métodos `SetAspectMemento` y `GetAspectMemento` le proporcionan la lógica para instanciar el Memento y obtener su funcionalidad manteniendo la encapsulación de los datos.

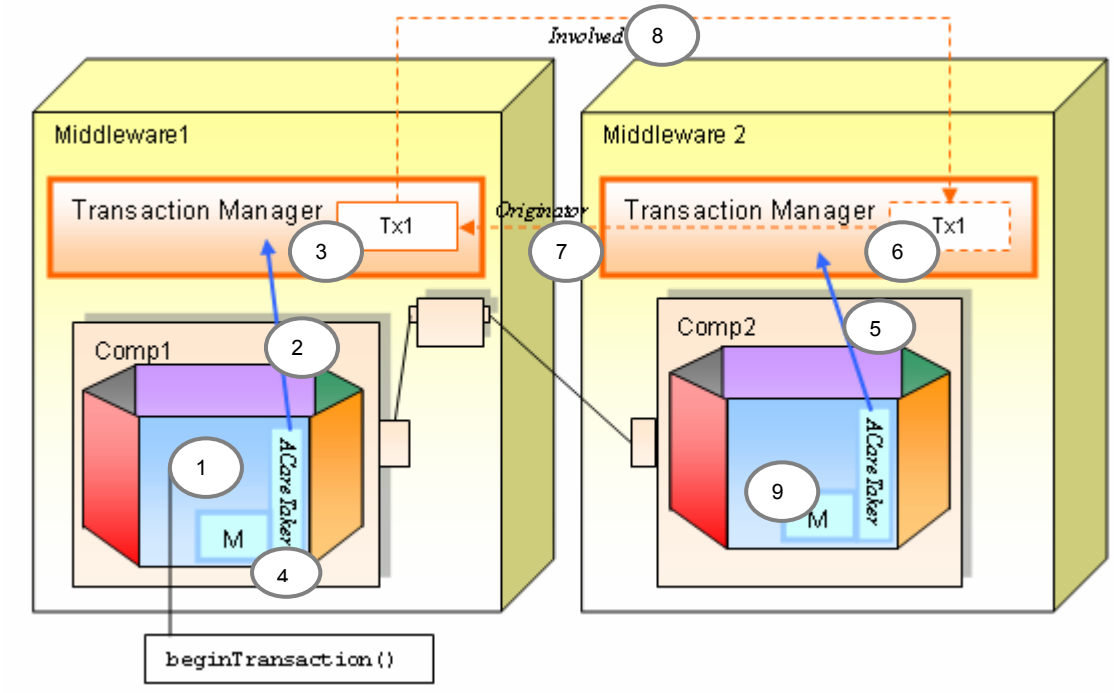



**Figura 60: Funcionamiento del Transaction Manager**

El funcionamiento transaccional se muestra en la Figura 60. Cuando en un aspecto se ejecuta una transacción (1), un fragmento de código avisa al *middleware*, en concreto a la capa `TransactionManager`, que una nueva transacción va a dar comienzo (2). El encargado de realizar las gestiones implicadas en este proceso para el aspecto es el `aspectStatecareTaker`. El *Transaction Manager* crea un nuevo contexto transaccional

(`TransactionalContext`) (3), cuyo originador será el *middleware* sobre el que se esté ejecutando, y lo añade a su lista. Tras la creación, devuelve al aspecto el identificador del nuevo contexto creado para que lo guarde en su `aspectStatecareTaker`. Tras recibir el identificador de contexto transaccional, el `aspectStatecareTaker` se encarga de generar el objeto Memento que guardará el estado actual del aspecto (4). Una vez hecho esto comenzará la ejecución de las operaciones dentro de la transacción, es decir la invocación de servicios en el contexto de la transacción. Como el aspecto tiene el identificador del contexto transaccional, cada vez que solicite la ejecución de un servicio externo al aspecto, encapsulará con la petición el identificador del contexto. Cuando un aspecto, al desencolar una petición de servicio de su cola, encuentre un identificador de contexto distinto de la cadena vacía, se preparará para entrar a formar parte del contexto transaccional: A través de su `aspectStatecareTaker`, antes de crear el Memento que encapsulará el estado del aspecto, informará al *Transaction Manager* de que se ha de unir a la transacción (`JoinTransactionalContext`) proporcionándole el identificador de la transacción (5). Si el *Transaction Manager* no tiene añadido en su lista dicho identificador, lo añadirá (6), extrayendo del identificador de contexto la URL del originador (7). Como el *middleware* no es el originador, enviará un mensaje al *middleware* origen para indicarle que se ha involucrado en la transacción. El *middleware* origen, tras recibir el mensaje, añadirá la URL a su `involvedList` (8). Una vez hecho eso, el nuevo aspecto pasa a formar parte del contexto, el Memento se crea (9) y se comienza la ejecución del servicio. Todos los aspectos cuya ejecución entre a formar parte de un contexto transaccional quedarán bloqueados para atender únicamente las peticiones de servicios que lleguen a través de ese contexto transaccional hasta que se confirme o se aborte la transacción a la que representa.

Respecto al comportamiento del bloqueo, cuando un aspecto se está ejecutando en el contexto de una transacción, solo procesará peticiones pertenecientes a dicha transacción. De esta forma, se garantiza el aislamiento de la operación, ya que hasta que no se confirme la transacción, los datos modificados solo serán accesibles por la propia transacción. Mientras tanto, el resto de peticiones permanecerán en la cola del aspecto respetando el orden de llegada. No obstante, puede que durante el transcurso de la transacción, una nueva petición sea encolada en el aspecto y en ese caso, es necesario procesarla para no incurrir en un interbloqueo. Puede que entre el instante en que el aspecto ha entrado a formar parte de la transacción, es decir, ha procesado el primer servicio perteneciente a la transacción, y el instante en que se vuelve a encolar una petición de servicio que provenga del mismo contexto transaccional, otros aspectos externos al contexto transaccional hayan requerido servicios del aspecto, por lo que estarán esperando en la cola para ser servidos una vez termine la transacción, por delante de la nueva petición de servicio. Como esta petición de servicio sí pertenece al contexto transaccional y por tanto, sí que puede (y debe, si no se quiere provocar un interbloqueo) ser servida por el aspecto, es necesario dar más prioridad a esa petición para ser procesada inmediatamente. Para ello se ha utilizado el mecanismo de cola de prioridad implementado en el aspecto: Cuando se encole una petición de

servicio en un aspecto que pertenece a la transacción activa del aspecto, se le asignará una prioridad negativa en función de su nivel de anidamiento. Así, a mayor nivel de anidamiento más prioridad de desencolado. Cuando se trate de una transacción anidada se podrá servir, sin esperar al resto de peticiones encoladas previamente se procesen, en las colas de los aspectos cuya transacción activa tenga el mismo contexto transaccional raíz.

Una transacción anidada PRISMA se ejecuta de forma secuencial, por tanto en un instante dado tan solo puede llegar una petición de servicio a un elemento bloqueado por una transacción proveniente de la misma transacción.

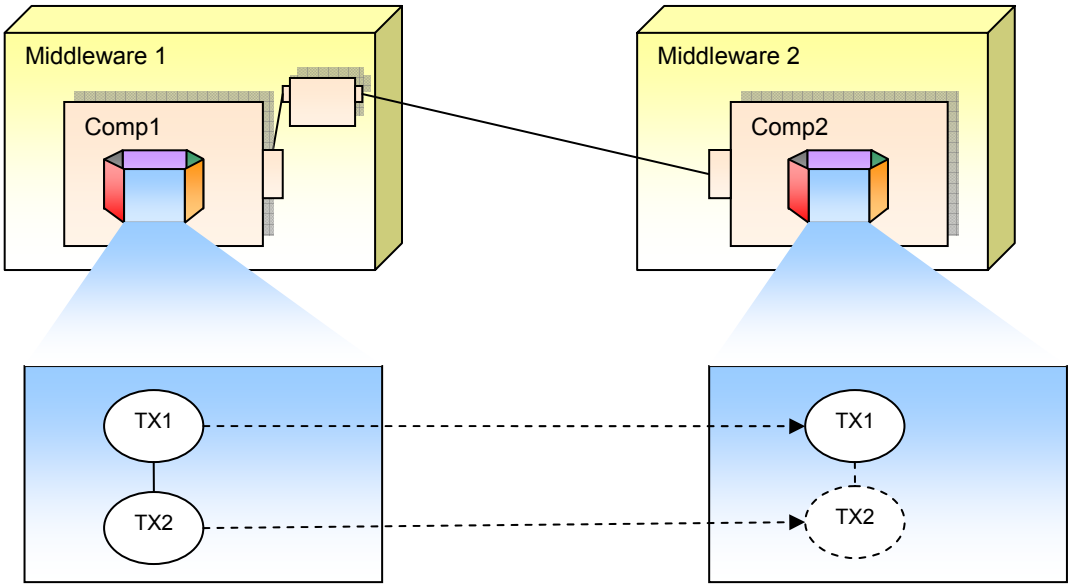



**Figura 61: Ejemplo de interacción con transacciones anidadas**

En la Figura 61 se muestra una configuración formada por dos componentes ubicados en diferentes *middlewares* conectados mediante un conector. En el componente Comp1 se inicia una transacción y una de las operaciones internas a ella invoca un servicio ofertado por un aspecto del componente Comp2. En ese momento el aspecto de Comp2 pasa a atender únicamente las peticiones que vengan etiquetadas por la transacción TX1, pasando a ser esta la transacción activa al aspecto. En un instante posterior, tras terminar la ejecución en Comp2, Comp1 inicia una transacción anidada a la transacción TX1. Esta nueva transacción TX2 tiene como transacción padre TX1, que a su vez es la transacción raíz. Si desde el interior de la transacción anidada TX2 se invoca de nuevo un servicio al mismo aspecto de Comp2 que lo había hecho TX1, esta debe de ser atendida para no provocar un interbloqueo. El aspecto de Comp2 está bloqueado para servir peticiones que vengan a través de TX1, como la nueva petición viene etiquetada por la transacción TX2, el aspecto ha de ser capaz de determinar si esa transacción está relacionada jerárquicamente con TX1, y en ese caso, encolarla con prioridad especial para ser atendida y poner a TX2 como transacción activa. Esto será posible gracias a

que la prioridad va en función del nivel de anidamiento, como TX2 tiene un nivel de anidamiento mayor que TX1, esta tendrá más prioridad.

De está forma, y gracias a la intermediación del *Transaction Manager* podemos extender los límites de un contexto a través de todo el entorno de ejecución PRISMA. Si alguno de los servicios ejecutados en el contexto lanza una excepción se informará al *Transaction Manager* que pondrá el *bit* `consistence` a falso, para indicar que algo ha fallado durante el curso de la transacción. Cuando un *Transaction Manager* pone un *bit* `consistence` a falso de una transacción que lo ha originado, inmediatamente avisa de este hecho al originador, quien a su vez también pondrá su *bit* `consistence` a falso. Esto será detectado al ejecutar la siguiente instrucción del código marcado como transaccional mediante una comprobación, lanzando una excepción que provocará el *rollback* de la transacción. De esta manera se evita continuar ejecutando la transacción cuando ya se sabe que ha fallado.

A continuación, se muestra una plantilla de cómo se transforma, haciendo uso de los mecanismos presentados, una transacción PRISMA a la implementación de PRISMANET en C#. El código debería insertarse en la clase que defina el aspecto en el lugar destinado para las transacciones. En la Tabla 17 se muestra la plantilla de generación de código C# a partir de la especificación de las transacciones en PRISMA LDA-OA.

**PRISMA**

```
transactionalService(Type1 param1, Type2 param2 ... TypeN paramN):
TRANSACTIONALSERVICE = {condition1} Service1?(param1)→STEP2;
STEP2= {condition2} Interface1.Service2!(param2)→STEP3;
(...)
STEPN= {condition3} ServiceN?(paramN);
```

**PSEUDOCÓDIGO C#**

```
aspectStateCareTaker.BeginTransaction()
Try
{
      Service1();
      aspectStateCareTaker.CheckConsistence()
      Service2()
      aspectStateCareTaker.CheckConsistence()
(...)
      ServiceN();
      aspectStateCareTaker.CheckConsistence()

}
Catch
{
      aspectStateCareTaker.SetConsistence(false);
}
Finally
{
       aspectStateCareTaker.EndTransaction()
}
```

**Tabla 17: Plantilla de generación de código C# para las transacciones**

Toda la funcionalidad transaccional, es decir, los métodos implicados en la ejecución de la transacción, serán proporcionados por el `StateCareTaker` del aspecto, quien interactuará con el *Transaction Manager* para gestionar los contextos transaccionales. Como se puede observar, la transacción está encapsulada en un bloque *try-catch-finally*, de forma que delante de un fallo se interrumpa la ejecución para tomar las medidas oportunas.

Inicialmente, se invocará a `BeginTransaction` que obtendrá un identificador de transacción del *Transaction Manager*, que almacenará en el `StateCareTaker` como `ActiveTransaction` y continuará la ejecución del aspecto bajo ese contexto. En caso de que el aspecto ya se encontrase ejecutándose bajo un contexto transaccional, el nuevo contexto transaccional será hijo del contexto actual, siguiendo la estructura jerárquica explicada con anterioridad para las transacciones anidadas.

Una vez obtenido el identificador de contexto, se entra en el bloque *try-catch-finally* donde se invocan los servicios implicados en la transacción. Cada vez que termina un servicio se verifica el estado de la transacción mediante el método del `aspectStateCareTaker` que preguntará por la consistencia de la transacción al *Transaction Manager*. De esta forma, se evita el continuar ejecutando la transacción cuando algún método de la transacción ha fallado. Esto es especialmente útil en las transacciones heterogéneas, en que servicios invocados por la transacción pueden estar ubicados en otro *middleware* y los fallos allí acaecidos se propagan al *middleware* originador mediante la modificación del *bit* de consistencia del contexto transaccional. Si en alguna de las comprobaciones se detecta que el *bit* de consistencia está puesto a falso, se lanzará una excepción que interrumpirá la ejecución de la transacción. De igual manera, cuando alguno de los servicios falle, la excepción se capturará y se tratará poniendo el *bit* de consistencia a falso.

Finalmente, se invocará al método `EndTransaction`, que en función del estado del *bit* de consistencia de la transacción realizará un *rollback* o un *commit*. La petición será propagada por el AspectStateCareTaker hasta el *Transaction Manager*. A continuación se muestra un fragmento de pseudocódigo que muestra el funcionamiento interno del método en el *Transaction Manager*:

```
procedure EndTransaction(string ContextID)
      context = contextList.getContext(ContextID)
      If Context.consistence = false
      Foreach mw in Context.InvolvedList
            Mw.rollback(ContextID)
      Endfor
      Else
      Foreach mw in Context.InvolvedList
            If Mw.Consistence != true
                  Context.consistence = false
                  Break
            endif
      Endfor
      If Context.consistence = false
```

```
            Foreach mw in Context.InvolvedList
                  Mw.rollback(ContextID)
            Endfor
      Else
                  Foreach mw in Context.InvolvedList
                  Mw.commit(ContextID)
            Endfor
      Endif
endProc
```

Es decir, primero localiza el contexto transaccional en la lista de contextos y seguidamente comprueba su *bit* de consistencia. Si el *bit* está puesto a falso, la transacción se ha de abortar, por ello envía un *rollback* con el identificador del contexto a todos los middlewares involucrados en la transacción. Si el *bit* está puesto a cierto, antes de dar por finalizada la transacción se volverá a comprobar en cada uno de los *middlewares* involucrados el *bit* de consistencia de los respectivos `TransactionalContextInvolved`. Si no se localiza ningún error, se enviará a todos los middlewares involucrados un *commit*, para que lo notifiquen a los aspectos que en ellos se ejecutan y que han entrado a formar parte de la transacción.

Cuando un `TransactionContext` recibe una orden de *commit* o de *rollback* asociado a un contexto transaccional, éste actúa sobre los *bits* `consistence` y `done` de dicho contexto de la siguiente forma:

| | Bit Consistence | Bit Done |
|---|---|---|
| **Commit** | true | true |
| **Rollback** | false | true |

Una vez el *bit* `done` se ha puesto a cierto en un `TransactionContext` se ha de avisar a todos los aspectos ubicados en el *middleware* que se encuentran ejecutándose en ese contexto transaccional recién finalizado. Esto se hace mediante el lanzamiento de un evento por parte del *Transaction Manager*. Los aspectos involucrados en la transacción, o más concretamente, el objeto `aspectStateCareTaker` del objeto Memento, han de estar suscritos a dicho evento. Por ello se han de definir mecanismos para cancelar la suscripción de todos los aspectos de un componente cuando, dentro de los límites de la transacción, este requiera moverse a otro *middleware*. Una vez en el destino, los aspectos implicados en la transacción deberán suscribirse en el nuevo *Transaction Manager* que, de no tener ya un componente implicado en el contexto transaccional, añadirá una entrada para el identificador de contexto. Cuando se capture el evento, el `aspectStateCareTaker` comprobará el *bit* `consistence`, si está a falso se ha abortado la transacción, por lo que restaura el estado guardado en el objeto Memento, elimina el objeto Memento y le indica al aspecto que puede reanudar su funcionamiento normal. Cuando el *bit* `consistence` está a cierto, la transacción se ha confirmado, por lo que elimina el objeto Memento e indica al aspecto que la transacción ha terminado. Una vez lanzado el evento, el *Transaction Manager* elimina el `TransactionContext` de la lista.

Para deshacer el estado del aspecto desde el momento que se inició la transacción, se obtiene el estado anterior del `AspectMemento` mediante el método `SetAspectMemento`. El método recibe el `AspectMemento` como parámetro y recupera su estado, asignando cada elemento del estado guardado en el *array* a los campos del aspecto.

#### *4.4.2.6. Transacciones a nivel de arquitectura*

Como se ha comentado anteriormente, las operaciones que se ejecutan dentro de una transacción no sólo modifican el estado de los aspectos implicados (y por tanto de los elementos arquitectónicos a los que dan semántica) sino que pueden modificar la arquitectura PRISMA. Un buen ejemplo de esto es una operación de movilidad, en la que, a través del aspecto de distribución, un componente cambia su localización. Este movimiento supone la destrucción de un componente y su creación en destino, como la modificación de todos los *bindings* y *attachments* que tenga asociados. Si esto se realiza dentro de los límites de una transacción, se ha de proporcionar un mecanismo para que todos los cambios comentados puedan deshacerse en caso de *Rollback*.

Como se ha visto, con las transacciones a nivel de aspectos la solución adoptada supone crear un objeto que encapsulase los campos del aspecto que representaban su estado interno. En este caso la solución no es tan simple, ya que en el estado de una arquitectura influyen muchos factores, tales como el estado de los componentes, conectores y sistemas que forman la arquitectura, como las gestiones internas de las diferentes capas del *middleware*. En ultima instancia, hacer una instantánea del estado de la arquitectura sería equiparable a guardar una copia del estado de la memoria principal de la máquina en que se está ejecutando, y esto, teniendo en cuenta que las arquitecturas pueden estar distribuidas entre diversas máquinas, supone a la postre una solución compleja y costosa.

En lugar de eso se ha optado por una solución más acorde con la problemática. Puesto que para modificar el estado de una arquitectura es necesario invocar una serie de servicios del *middleware*, para deshacer esos cambios basta con invocar el servicio o conjunto de servicios que realice la acción inversa en el orden adecuado. Es decir, si por ejemplo durante la ejecución de una transacción se ejecuta el método `Move` de un componente indicándole su nueva localización, el *middleware* se encargará de crear esa instancia del componente en la localización indicada realizando las modificaciones pertinentes a *attachments* y *bindings* asociados a dicho componente. Pues bien, si esa operación tuviese que deshacerse, el *middleware* destino debería invocar el `Move` del componente pasándole como parámetro la localización en la que se encontraba inicialmente. En este caso la operación inversa a `Move(LOC2)` en `LOC1` es `Move(LOC1)` en `LOC2`.

Para la implementación de un mecanismo que de soporte a la idea antes mostrada se ha pensado en el *Logical Logging* utilizado en otros sistemas transaccionales. En el campo de las bases de datos, los *Logs* representan un historial de las acciones realizadas por el sistema gestor de bases de datos [Moss87].. Normalmente los *logs* tienen un número de secuencia o *Log Sequence Number* (*LSN*) que ejerce de identificador para cada registro del *log*. Mediante el número de secuencia previo o *Previous Log Sequence Number* (*Prev LSN*), cada registro tiene un enlace al registro anterior. Finalmente cada registro es identificado por el ID de la transacción que ha escrito el registro.

La transformación del concepto al *middleware* PRISMA se explica a continuación: Existirá un *log* por cada contexto transaccional. Este *log* será distribuido entre los diferentes *middlewares* involucrados en el contexto transaccional. Cada entrada del *log* tendrá un número de secuencia similar al *LSN* iniciado por el *middleware* originador y una referencia a la localización del número de secuencia previo (*PrevLSN*), que al ser el *log* distribuido, podría encontrarse en otro *middleware*. El registro lo completará la descripción de la operación inversa, con la información necesaria para invocar el método. En este caso no es necesario indicar la transacción a la que pertenece la operación, pues cada transacción, representada en el middleware por un TransactionalContext, tendrá su propio *Log* y será mediante el *PrevLSN* como se enlazarán operaciones inversas a las realizadas bajo un contexto transaccional a las realizadas en las transacciones anidadas a ese contexto. Por ello, la forma que tendrán será la siguiente:

```
numSeq{url.del.host}
```

Por ejemplo `5{host01.dsic.upv.es}`. De esa forma se puede extraer del número de secuencia la ubicación de su predecesor y sabiendo eso y la identidad del *root* resulta sencillo recorrer la lista de forma inversa.

Cuando se habla de operación inversa se hace referencia a la operación o conjunto de operaciones que ejecutadas en determinado orden vienen a contrarrestar el cambio producido en la arquitectura por la operación que deshacen. Por ello es necesario identificar todas las operaciones que se pueden ejecutar en el marco de una transacción y que modifican el estado de la arquitectura, y para cada una de ellas definir su operación inversa. Puesto que esta modificación siempre se hace a nivel de *middleware*, se ha de añadir a la funcionalidad de cada servicio del *middleware* las operaciones a realizar para invertir la operación de forma que sea el mismo método quien diese esa información al *Transaction Manager*. Así la información para la operación inversa se proporciona por el mismo método que se ha de invertir.

Por ejemplo, en la Tabla 18 se muestra un ejemplo de anotado de operación inversa para una operación de creación de un nuevo componente en el *middleware*. Al detectarse que se está invocando el servicio desde el ámbito de una transacción se procede al anotado. La operación inversa será eliminar ese componente creado en el mismo *middleware*. Para ello se anota la cadena

que representaría la invocación de esa operación. Esa cadena, junto al resto de operaciones se compilará a una clase dinámica mediante CodeDom y se pondrá en ejecución, pasándole en cada caso una referencia a `core`, es decir al `MiddlewareSystem`.

```
private void CreateNewComponent(Type componentType,
                                string componentName,
                                params object[] args,
                                string txID)
{

(...)

if (txID.Length > 0)
// this operation belongs to a transaction
// invert operation -> RemoveComponent
string operation = "core.ElementManagementLayer.RemoveComponent(\"" +
componentName +"\",true");";
// add the operation to the transactional context using its txID
core.TransactionMngr.AddArchitecturalOperation(txID, operation);
}

(...)

}
```

**Tabla 18: Ejemplo de anotado de operación inversa en la implementación de PRISMANET**

Por otro lado, el *Transaction Manager* redirige cada mensaje con la información de la operación inversa al *Transactional Context* asociado para que lo anote en su *log*. De esa forma cada contexto transaccional tiene un fichero en disco asociado donde va anotando las operaciones que debería ejecutar, en caso de fallar la transacción, para dejar la arquitectura en el mismo estado que se encontraba antes de iniciar la transacción. Si la transacción raíz se confirma, es decir, si no se ha producido ningún error ni en la transacción ni en sus subtransacciones, el *log* podrá ser eliminado al mismo tiempo que el contexto transaccional es borrado de las listas de contextos transaccionales de los diferentes *Transaction Managers* involucrados. Si se produce algún error, cuando el *middleware* originador de la transacción tenga constancia de ello, iniciará las gestiones para ejecutar en orden inverso las operaciones anotadas en el *log*. Esto lo realizará únicamente un contexto transaccional *root*, es decir, aquel que está en la parte más alta de la jerarquía, ya que dado un error se han de deshacer todos los cambios realizados en la arquitectura, incluso los pertenecientes a transacciones internas que ya se hayan confirmado.

Como se puede ver, de nuevo el *Transactional Context* que ejerce de *root* realiza un importante papel en el proceso. De hecho, este contexto transaccional ubicado en la parte más alta de la jerarquía, es quien gestiona la asignación del *PrevLSN*, para que, en todo momento, tanto los *middlewares* involucrados como las transacciones anidadas que se creen, puedan preguntar cual es la operación inversa previa a la que están escribiendo. Por ello, en caso de fallo, el *PrevLSN* que contenga el *root* será la primera operación a ejecutar para comenzar a deshacer los cambios.

El funcionamiento del *Log* merece una explicación detallada. Puesto que las transacciones pueden extenderse a través del entorno de ejecución PRISMA, las operaciones que modifiquen la arquitectura pueden ocurrir en cualquiera de los *middlewares* que alcance el contexto transaccional. Por tanto es necesario que el *log* sea distribuido. Para ello se ha relacionado el *log* con los contextos transaccionales, pues estos tienen una instancia en cada *middleware* que alcanzan. Cuando la ejecución de una transacción abandona un *middleware*, las operaciones que ha anotado en el *log* del `TransactionalContext` ubicado en ese *middleware* serán como un bloque de operaciones que se han de ejecutar en orden inverso. No obstante, tras anotar operaciones inversas en el *log* del `TransactionalContext` del *middleware* al que se ha extendido la transacción, la ejecución vuelve al *middleware* originario. De nuevo anotará operaciones en el *log* del `TransactionalContext`. Las nuevas operaciones se anotarán a continuación del bloque anterior, pero no se pueden considerar del mismo bloque porque, entre la última operación anotada en ese *log* y las nuevas operaciones, se han anotado operaciones en *logs* en otros *middlewares*. Si se pretende ejecutar las operaciones en un orden correcto hay que tener en cuenta esto.

Por ello se agrupan las operaciones que van seguidas en bloques de operaciones. Para relacionar los diferentes bloques distribuidos, lo que es necesario para recuperar las operaciones en el orden correcto, se usa el *PrevLSN* que enlaza las operaciones en el ámbito distribuido. De esa forma, mientras el *PrevLSN* de una operación apunte al mismo *middleware* se estará recorriendo de forma inversa un mismo bloque de operaciones. En el momento que apunte a otro *middleware*, se estará saltando a un bloque de operaciones nuevo.

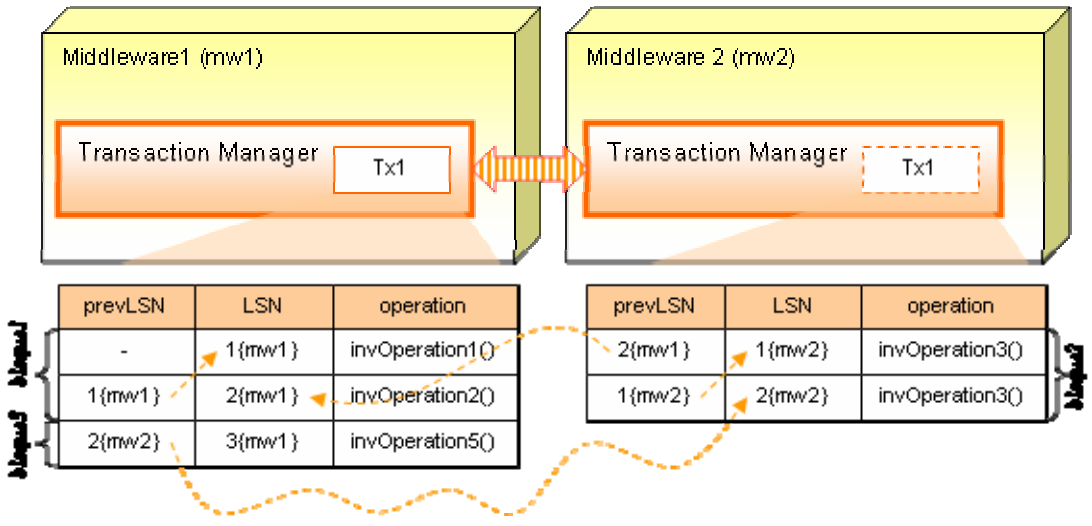



**Figura 62: Ejemplo de funcionamiento del Log**

En la Figura 62 se muestra la interacción entre los *Transaction Managers* de dos *middlewares* a la hora de crear el *Log*. Si durante la transacción *Tx1* se ejecuta una operación que modifica el estado de la arquitectura cuya operación

inversa es `invOperation1()`, esto será notificado al gestor de transacciones que creará una anotación en el *log* transaccional. Como se ha explicado anteriormente, esto se captura en el momento que se ejecuta bajo un contexto transaccional un servicio del *middleware* que modifica la arquitectura. En este caso, el *Transaction Manager* genera un *Log Sequence Number* (*LSN*) para la anotación, y al ser la primera, deja vacío el campo *Previous Log Sequence Number* (*PrevLSN*). La transacción seguirá ejecutándose pudiendo crear nuevas anotaciones en el *Log* si se ejecutasen nuevos servicios en el *middleware* que modificase la arquitectura. En el ejemplo, `invOperation2()` es la operación inversa a una nueva operación ejecutada que puede modificar el estado de la arquitectura. El *Transaction Manager* le crea un nuevo *LSN* y lo anota, ahora si, junto al *PrevLSN*, es decir el *LSN* de la anotación previa.

Si en este punto el límite de la transacción se extendiese a otro *middleware*, y de nuevo realizase acciones que modifican el estado de la arquitectura desde allí, las operaciones inversas de esas acciones se anotarían en el *Log* del `TransactionalContext` representante de la transacción en aquel *middleware*. El `TransactionManager` remoto crearía un *LSN* y preguntaría al *middleware* del que proviene la transacción por el *PrevLSN*. El *Transaction Manager* del *middleware* originario le devuelve el *PrevLSN* que tiene anotado y le pregunta por el *LSN* que va a anotar el *middleware* remoto: Este será el nuevo *PrevLSN*. Se habrá anotado `invOperation3()`. Cuando el *middleware* remoto haga otra anotación, el proceso será similar, generará el *LSN*, obtendrá el *PrevLSN* del originador y le proporcionará el nuevo valor de *LSN*. De esta forma, la operación `invOperation4()` se habrá añadido al *log*.

De nuevo, la transacción vuelve al *middleware* originador. Cuando haga una anotación volverá a generar un *LSN*, y como está en el *middleware* originador, obtendrá directamente el *PrevSLN*. De esta forma se habrá anotado `invOperation5()`.

Ahora bien, para conseguir deshacer los cambios en la arquitectura, deberían ejecutarse las operaciones inversas en el orden inverso al que se han anotado. Por ello no habría más que recorrer de forma inversa las anotaciones siguiendo el *PrevSLN* y empezando por el que tiene el *Transaction Manager* del *middleware* originador. Mediante estos pasos se ejecutarán las operaciones por bloques en cada uno de los *middlewares*. En el ejemplo de la Figura 62 se han marcado tres bloques, uno por cada salto de la transacción de un *middleware* a otro. En el proceso de recuperación del estado de la arquitectura, primero se ejecutarán las operaciones inversas del bloque 3 en el *middleware* originador, después las operaciones inversas del bloque 2 en el *middleware* remoto y finalmente, las del bloque 1, de nuevo en el *middleware* originador.

Para dar soporte a esta funcionalidad se han añadido las clases `TransactionArchitechturalOperations` para representar el *Log*, la clase `OperationLogItem`, para representar una operación inversa y la clase estática `ArchitecturalOperationsUndoer` para la ejecución de las operaciones inversas. Estas clases están relacionadas con los campos de la clase `TransactionManager` y

de `TransactionalContext` que se mostraron en secciones anteriores. En la Figura 63 se muestran las clases `TransactionArchitecturalOperations`, `OperationLogItem` y `ArchitecturalOperationsUndoer`.

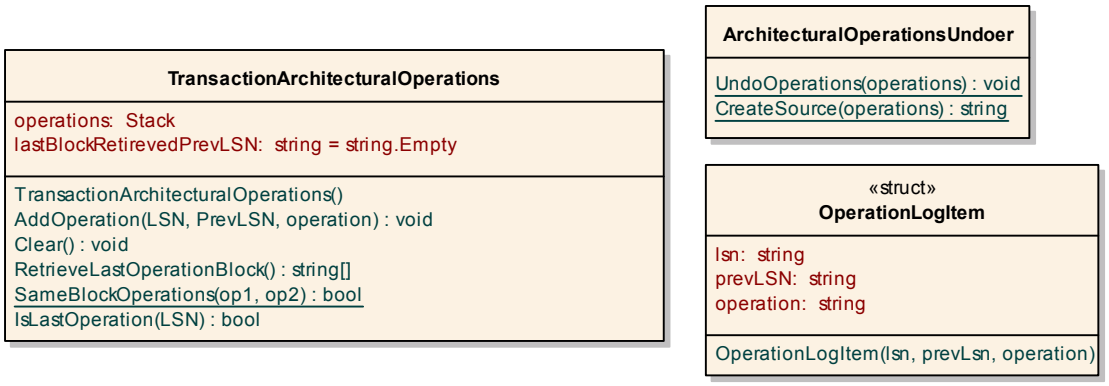



**Figura 63: Clases que implementan el Log**

La clase `OperationLogItem` representa una operación inversa. Tiene un *Log Sequence Number* (`lsn`), un *Previous Log Sequence Number* (`presLSN`) para apuntar a la operación anterior y una cadena de texto que contendrá la operación. Se ha utilizado una cadena de texto para representar la operación porque se usarán los mecanismos de generación de código en tiempo de ejecución que proporciona .NET, en concreto, la tecnología *CodeDOM* [Scott03].

La clase `TransactionArchitecturalOperations` es la que representa el Log de operaciones. Contiene una pila (`operations`) en la que se irán añadiendo los `OperationLogItem`. También guarda información sobre el *PrevSLN* que hace referencia al bloque de operaciones anterior.

Finalmente, la clase `ArchitechturalOperationsUndoer` es una clase estática que encapsula la lógica de generación de código en tiempo de ejecución. Utiliza el compilador interno de *CodeDOM* para generar una clase dinámica que contendrá las invocaciones a los diferentes servicios del *middleware* extraídas en el orden adecuada del *Log* para realizar las operaciones inversas necesarias para recuperar el estado de la arquitectura. El poner en ejecución la clase dinámica que generará será equivalente a ejecutar una por una, en el orden correcto, las operaciones inversas.

CAPÍTULO V

# AMBIENT-PRISMA

## Contenidos del capítulo

# AMBIENT-PRISMA

Hasta este punto se ha presentado cómo el modelo PRISMA da soporte a la distribución y la movilidad de los elementos arquitectónicos. Como se ha visto, la distribución es soportada mediante el uso de canales de comunicación (*attachments* y *bindings*) y el uso de aspectos de distribución.

Respecto al uso de canales de comunicación, los *attachments* y los *bindings* almacenan no solo las referencias de los elementos arquitectónicos que conectan, sino la localización de dichos elementos. Gracias a eso se preserva la reutilización de los elementos arquitectónicos y al mismo tiempo se les proporciona una comunicación distribuida, de forma que, la ubicación local o remota de los elementos arquitectónicos con los que están conectados es totalmente transparente a ellos.

Puesto que en PRISMA la semántica de los elementos arquitectónicos es proporcionada por aspectos que capturan la funcionalidad específica de un determinado *concern*, tiene sentido que las características distribuidas sigan el mismo planteamiento. El Aspecto de Distribución especifica las características y estrategias relacionadas con el comportamiento distribuido de un elemento arquitectónico PRISMA, especificando las características de distribución como es la localización del elemento y siendo el encargado de indicar cuando dicho elemento necesita ser movido.

No obstante, un modelo que incluya una primitiva explícita para especificar localizaciones, esto es, lugares limitados donde los elementos arquitectónicos se ejecuten, y que además sea capaz de organizar dichas localizaciones en una jerarquía proporcionando un soporte para la movilidad de los elementos arquitectónicos a través de los límites definidos es mucho más rico. Por ello se ha combinado la expresividad del modelo PRISMA con el Cálculo de Ambientes [Car98], obteniendo así un nuevo modelo fruto de la unión: El Ambient-PRISMA.

## 5.1. Ampliación del modelo PRISMA

Ambient-PRISMA [Ali06][Ali06b][Ali06c] es la combinación de los conceptos del Cálculo de Ambientes con una Arquitectura Orientada a Aspectos como es PRISMA para permitir la descripción de características de distribución y movilidad de forma independiente de plataforma. Con el objetivo de que los

elementos arquitectónicos PRISMA puedan hacer uso del concepto de ambiente del Cálculo de Ambientes y que el Cálculo de Ambientes pueda describir las arquitecturas distribuidas y móviles haciendo uso del desarrollo software orientado a aspectos y el desarrollo software basado en componentes, dicho concepto ha sido introducido en el metamodelo de PRISMA. Para ello, se han identificado las correspondencias entre el metamodelo de Cálculo de Ambientes y el metamodelo de PRISMA y el concepto de ambiente se ha añadido como ciudadano de primer orden del metamodelo PRISMA.

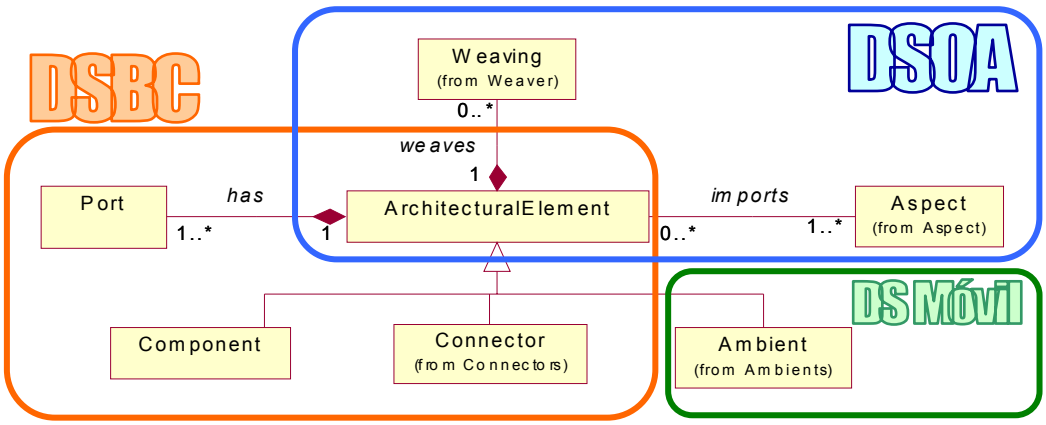



**Figura 64: Ambiente en el metamodelo PRISMA**

En [Car98] se comenta que un ambiente puede ser visto como un componente software que proporciona movilidad y que posee una identidad propia en tiempo de ejecución, por lo que puede ser mantenible. En el modelo PRISMA, esto concuerda con la definición de un elemento arquitectónico PRISMA. Dichos elementos arquitectónicos pueden ser de dos tipos: componentes (elementos que ejecutan procesos computacionales) o conectores (elementos que coordinan dichos procesos). Sin embargo, un ambiente no puede definirse como un componente, conector, o una extensión de éstos, ya que su semántica es distinta. Los ambientes definen los límites donde se ubican los procesos computacionales (componentes), pero no son procesos computacionales, ni se encargan de coordinar su ejecución. Por estos motivos, el concepto de ambiente ha sido introducido en el metamodelo PRISMA como un nuevo tipo de elemento arquitectónico (ver Figura 64) (homogéneo con los anteriores) que proporciona las características de movilidad, comunicación y replicación a los elementos arquitectónicos distribuidos, de acuerdo con el Cálculo de Ambientes.

Un ambiente puede contener una colección de agentes locales (procesos computacionales) y de otros subambientes [Car98]. En Ambient-PRISMA, los agentes locales corresponden a componentes coordinados a través de conectores, de forma que los elementos arquitectónicos se ubican en Ambientes, que son los que les proporcionan localización. Además, la incorporación del concepto de subambiente en Ambient-PRISMA permite modelar la jerarquía de sistemas distribuidos y móviles. De este modo, se definen los ambientes en

Ambient-PRISMA como elementos arquitectónicos complejos que permiten representar adecuadamente la ubicación de componentes y conectores.

Como se ha mostrado en la Figura 64, un ambiente hereda todas las características de un elemento arquitectónico, esto es, su vista DSBC y su vista DSOA y conforma lo que se podría denominar el Desarrollo Software Móvil (DSM). A continuación, de forma similar a como se hizo en el Capítulo 3 se presentarán las novedades del modelo desde una visión orientada a aspectos (DSOA) y desde una visión orientada a componentes (DSBC).

### 5.1.1. Visión Orientada a Aspectos

Por otro lado, la vista DSOA de un ambiente define el conjunto de aspectos que, sincronizados mediante los *weavings*, definen su semántica. El ambiente usa diferentes aspectos para especificar los servicios que ofrece y requiere. Al proporcionar características de movilidad y distribución a los elementos localizados en su frontera, el elemento arquitectónico ambiente ha de importar una serie de aspectos en los que se expresarán la funcionalidad de sus *concerns*. Las características de movilidad vendrán ofrecidas por un Aspecto de Movilidad, con el que el ambiente proporcionará servicios de movilidad a sus elementos internos. Las características de comunicación distribuida las proporcionará un Aspecto de Coordinación, que coordinará el envío de mensajes desde los elementos ubicados en el interior del ambiente con los que están fuera y viceversa. Finalmente, un Aspecto de Distribución será el encargado de proporcionar una semántica de distribución al ambiente, pues el mismo también ha de tener una localización, puesto que un ambiente puede estar en una jerarquía de ambientes.

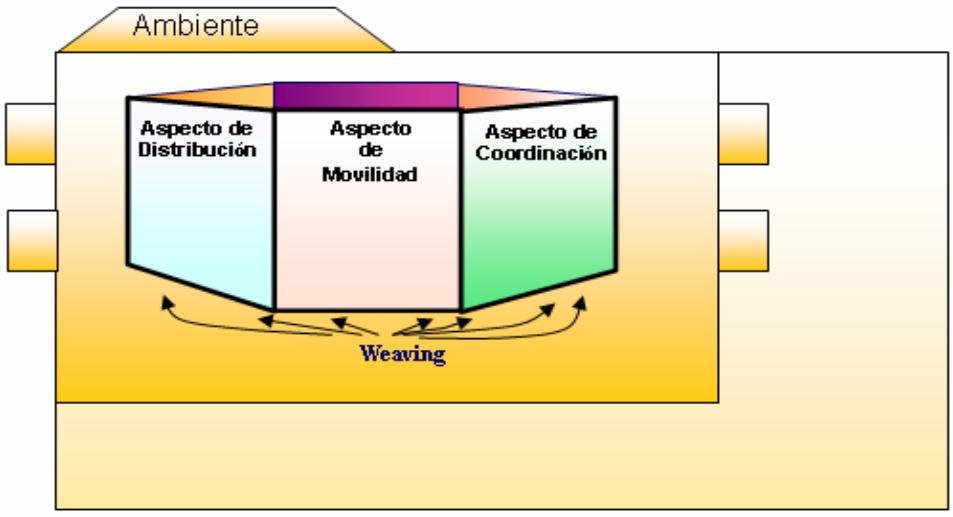



**Figura 65: Visión orientada a aspectos de un ambiente**

En la Figura 65 se muestra los tres aspectos fundamentales de todo ambiente. Puesto que estos aspectos son los que dotan al elemento arquitectónico su semántica, es necesario que todos los ambientes los importen. Es más, en el caso del Aspecto de Movilidad y el Aspecto de coordinación, la

semántica que han de proporcionar es siempre la misma, con lo cual se harán uso de aspectos genéricos que se importarán para todos los ambientes. A continuación se detallan cada uno de los aspectos que han de tener todos los ambientes.

#### *5.1.1.1. Aspecto de Movilidad del Ambiente*

Debido a que los ambientes son los que permiten la movilidad, todos ellos importan un Aspecto de Movilidad que proporciona servicios de movilidad a los elementos arquitectónicos a los que proporciona su localización. Estos servicios son los que proporcionan las *capabilities* del Cálculo de Ambientes. A continuación, se listan las funcionalidades del Aspecto de Movilidad:

- Permite crear subambientes, es decir, definir un nuevo límite computacional alrededor de un conjunto de elementos arquitectónicos ubicados en su interior.

- Ofrece a los subambientes ubicados en su interior el servicio *exit* si necesitan salir de él (Especificación de la *capability exit* del Cálculo de Ambientes).

- Ofrece, mediante el servicio *enter* la entrada de un ambiente en otro ambiente del mismo nivel jerárquico (Especificación de la *capability enter* del Cálculo de Ambientes).

- Permite eliminar un subambiente sin destruir los elementos arquitectónicos que lo constituyen mediante el servicio *ope*n. Como resultado, los elementos arquitectónicos del subambiente destruido pasan a formar parte del ambiente padre (Especificación de la *capability open* del Cálculo de Ambientes).

- Permite replicar subambientes mediante el servicio *replicate* (Especificación de la *capability replicate* del Cálculo de Ambientes).

La separación de la movilidad del resto de *concerns* a través de un aspecto mejora su mantenibilidad y reutilización, ya que no se encuentra entremezclada con el resto de *concerns* del ambiente. Además, el aspecto de movilidad es un aspecto genérico que es reutilizado por todos los ambientes PRISMA. Por lo tanto, los ambientes son definidos importando dicho aspecto de movilidad y adaptándolo a las necesidades del sistema de información mediante *weavings*. Por ejemplo, un ambiente puede necesitar definir movilidad con políticas de seguridad para los procesos que contiene. Por tanto, el ambiente importaría el aspecto de movilidad genérico, un aspecto de seguridad específico, y se definirían una serie de *weavings* para entretejer ambas funcionalidades.

El Aspecto de Movilidad del Ambiente un aspecto genérico que ofrece la semántica de la movilidad haciendo uso de las *capabilities* del Cálculo de

Ambientes y que será importado por defecto por todos los ambientes, pues proporcionará una semántica fundamental al elemento arquitectónico. Por esta razón el aspecto se ha predefinido. Esto es así, porque la semántica de movilidad que ofrece el elemento arquitectónico ambiente a los elementos a los que proporciona localización es proporcionada por este aspecto. Puesto que esta semántica es la misma para todos los ambientes, es posible especificar esta funcionalidad en un aspecto que será el que importen todos los ambientes. La especificación del aspecto predefinido se verá más adelante en este capítulo.

#### *5.1.1.2. Aspecto de Coordinación del Ambiente*

El Aspecto de Coordinación de un Ambiente es necesario para coordinar sus elementos arquitectónicos con el exterior. Este aspecto recibirá llamadas desde elementos arquitectónicos externos al ambiente, es decir, llamadas distribuidas y las redirige a los elementos arquitectónicos correspondientes del ambiente. Del mismo modo, recibe llamadas de los elementos arquitectónicos ubicados en el interior del ambiente y las redirige hacia el exterior.

De esta forma, el Aspecto de Coordinación proporciona un mecanismo de comunicación distribuida mediante la cual los elementos arquitectónicos ubicados en el interior de un ambiente pueden comunicarse con otros elementos arquitectónicos ubicados más allá de los límites del propio ambiente. Esto es así para, formalizar usando primitivas PRISMA las comunicaciones que traspasan los límites del ambiente. Gracias a eso, la comunicación distribuida respeta la jerarquía de ambientes ya que se propaga a través de los Aspectos de Coordinación de los ambientes que haya de traspasar.

Al pasar todas las peticiones de todos los elementos internos y externos al ambiente por el mismo aspecto, se ha establecido una interfaz genérica que abstrae al ambiente de tener que controlar las interfaces de sus elementos internos. Esta interfaz permite la redirección de las peticiones de servicios, vengan de la interfaz que vengan y, al mismo tiempo, al tratarse de una interfaz concreta, da la posibilidad de definir *weavings* entre sus servicios y los de otros aspectos del ambiente.

Esto permite que se pueda agregar semántica a la comunicación distribuida mediante la importación de nuevos aspectos. Por ejemplo, si se quiere que las comunicaciones entre determinado ambiente y el exterior estén sujetas a unas políticas de seguridad, bastaría con importar un aspecto que las gestionase y entretejer los servicios de este aspecto con el aspecto de coordinación que especifica esta interfaz de comunicación genérica.

Este aspecto, al igual que el de Movilidad, es genérico, de forma que se reutiliza en todos los ambientes.

#### *5.1.1.3. Aspecto de Distribución del Ambiente*

El Aspecto de Distribución de un ambiente almacena el nombre de su ambiente padre, es decir, de aquel en que está contenido según la jerarquía de ambientes. Un ambiente Móvil es un Ambiente que requiere servicios de movilidad de su ambiente padre; cuando un ambiente es móvil, su ambiente padre puede ser cambiado.

El Aspecto de Distribución, aunque es obligatorio en todo ambiente, no está definido de forma genérica, permitiendo de esa forma al usuario definir sus propias directivas de Distribución en función del sistema de información. No obstante, es necesario que el aspecto de distribución guarde el nombre del Ambiente Padre y proporcione un servicio `GetParent` para que otros aspectos, como el de Movilidad, lo puedan consultar mediante *weavings*.

### 5.1.2. Visión Basada en Ccomponentes

La vista del DSBC de un ambiente lo describe como una caja negra la cual se comunica con los otros elementos arquitectónicos mediante el uso de puertos que envían y reciben solicitudes de servicios. Para la comunicación entre elementos arquitectónicos se mantiene el concepto de *attachment* de PRISMA, de forma que cada *attachment* se define conectando dos puertos de diferentes elementos arquitectónicos. Como se puede ver, esta es la manera mediante la que los componentes, conectores y subambientes ubicados en un ambiente se comunican con él ambiente padre (ver Figura 66) para solicitar sus servicios. El ambiente será el que proporcione a los elementos arquitectónicos ubicados en su interior los servicios de movilidad y de distribución.

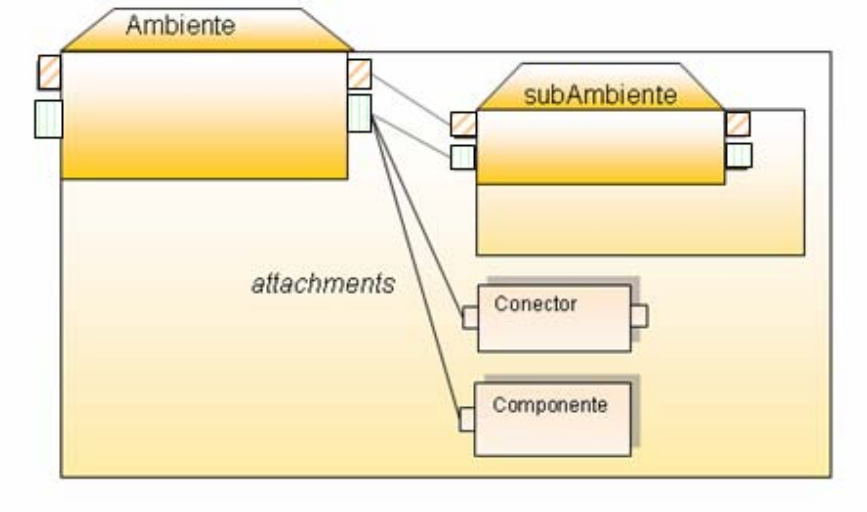



**Figura 66: Visión DSBC de un ambiente**

Cuando se ha hablado de la visión orientada a Aspectos se han presentado los aspectos genéricos que importa todo ambiente. Estos aspectos (Movilidad y Coordinación) ofrecen una serie de servicios a través de puertos. De la misma

forma que dichos aspectos se importan por defecto, es necesario que los puertos que necesita esos aspectos para comunicarse con los elementos arquitectónicos ubicados dentro y fuera del ambiente se creen.

Así pues se definen un total de cuatro puertos: dos para el Aspecto de Movilidad y dos para el Aspecto de Coordinación. Los puertos del aspecto de Movilidad, uno será para ofrecer (requerir para el caso de subambientes) las *capabilities* a los elementos ubicados en el interior del ambiente (*played role* `INTERIOR`) y otro para ofrecer o requerir la ejecución de *capabilities* al ambiente padre (*played role* `EXTERIOR`). Los puertos del Aspecto de Coordinación, son los que ofrecen la comunicación distribuida, y al igual que los del Aspecto de Movilidad, se necesita uno para los elementos internos y otro para los externos, cada cual con *played roles* diferentes.

Por ejemplo, en la Figura 66, el *Componente, Conector y el subAmbiente* pueden solicitar servicios de elementos arquitectónicos distribuidos gracias a que están conectados al puerto del Aspecto de Coordinación del Ambiente. Además, el *subAmbiente* puede solicitar al *Ambiente* servicios de movilidad porque esta conectado con el puerto de los capabilities.

Así gracias a esos puertos los ambientes ofrecen servicios de movilidad y distribución mediante *attachments* conectados a los elementos internos que los vayan a solicitar.

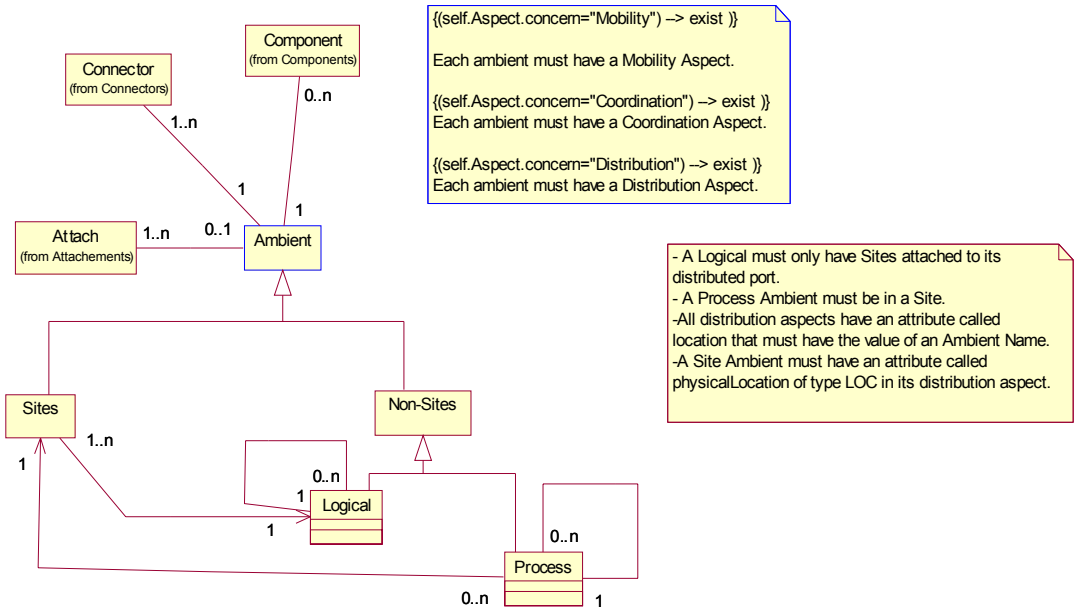



**Figura 67: El paquete Ambient**

En la (Figura 67) se muestra el paquete de ambientes introducido en el metamodelo PRISMA, que define las relaciones y restricciones respecto a otros conceptos del metamodelo. También se puede observar como se definen diferentes tipos de ambientes. Estos tipos modelan los diferentes tipos de

localización que puede representar un ambiente, cada tipo con sus propias restricciones. Hay tres tipos de ambientes:

- **Ambiente *Site*:** Los ambientes *Site* representan un dispositivo, es decir, algo que tiene una dirección física. Un ordenador o un PDA son ejemplos de entidades que se pueden modelar mediante un *Site*. Los ambientes Site sólo pueden contener ambientes *Process*, componentes y conectores.

- **Ambiente *Process*:** Representan una colección de componentes. Su utilidad puede variar en función del dominio: Pueden servir para agrupar un conjunto de elementos arquitectónicos que están sujetos a unas políticas de seguridad específica, o simplemente que se han de mover de forma conjunta. Los ambientes *Process* pueden albergar otros subambientes *Process*, componentes y conectores.

- **Ambiente *Logical*:** Son ambientes virtuales que agrupan una serie de ambientes *Site* o otros ambientes *Logical*. Gracias a ellos es posible modelar redes o espacios virtuales en los que los *Sites* se ubican. El modelar una jerarquía de ambientes *Logical* puede representar la jerarquía de una *Local Area Network* (LAN) para formar una *Wide Area Network* (WAN). En función del diseñador se pueden definir más o menos jerarquías de ambientes *Logical*, pero en PRISMA siempre hay un ambiente *Logical* por defecto que engloba al resto de ambientes, es decir, que representa al *Root*.

Es decir, las combinaciones que se pueden dar son de ambientes *Logical* que contienen otros ambientes *Logical*, que a su vez contienen *Sites*, que solo pueden contener ambientes *Process*, además de componentes y conectores, y estos, pueden contener otros ambientes *Process* en su interior o componentes y conectores

En la Figura 68 se muestra un ejemplo de como se ha modelado usando los diferentes tipos de ambientes dos WANs formadas por LANs, que a su vez están formadas por máquinas, algunas de ellas con carpetas y subcarpetas. En la figura están señalizados los diferentes tipos de ambientes que se han usado en cada caso.

## 5.2. Ampliación del Lenguaje de Definición de Arquitecturas Orientado a Aspectos

Como se ha mostrado en el Capítulo 3, PRISMA proporciona un Lenguaje de definición de Arquitecturas Orientado a Aspectos (LDA-OA) para especificar la definición semántica y sintáctica del modelo. Con la incorporación de los nuevos conceptos del Ambient-PRISMA, el lenguaje se ve enriquecido por las nuevas primitivas.

Así el LDA-OA de Ambient-PRISMA incorpora primitivas para la definición de los nuevos elementos arquitectónicos, es decir los tipos de ambientes que se han descrito en la sección anterior. Estas primitivas permitirán al diseñador especificar localizaciones haciendo uso de los tipos ambiente.

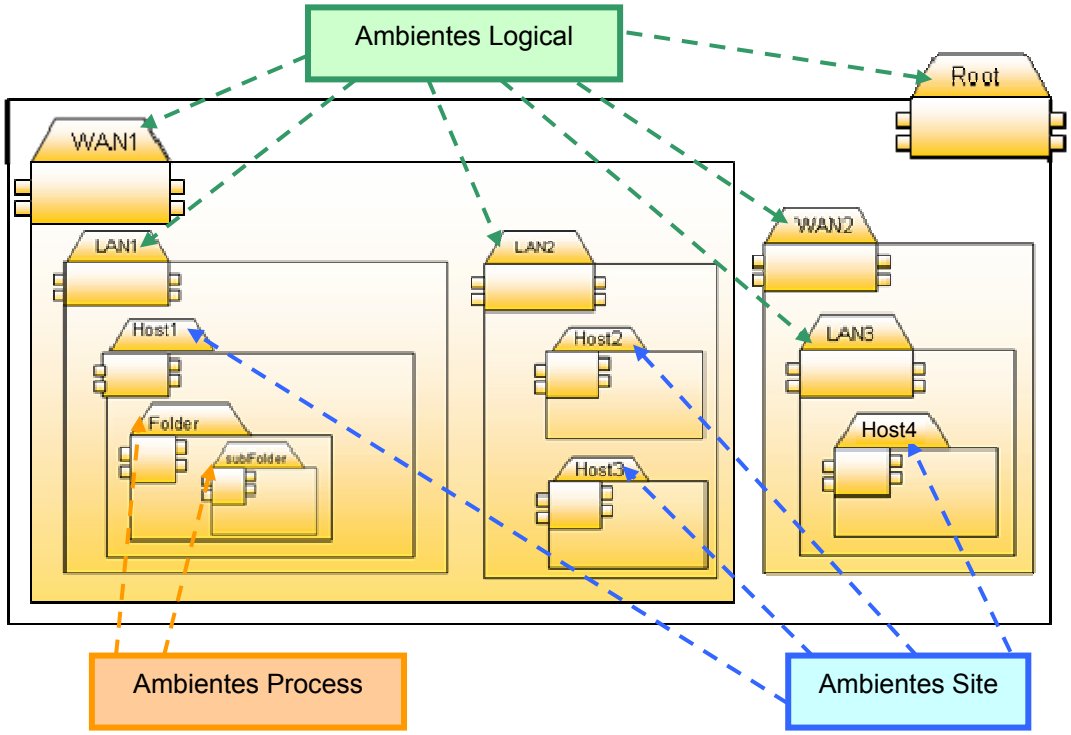



**Figura 68: Ejemplo de tipos de ambientes**

.

En la presente sección se presentarán las especificaciones de los aspectos genéricos usando la plantilla de especificación de aspectos de PRISMA y, posteriormente, las plantillas LDA-OA para la especificación de los ambientes donde son importados dichos aspectos genéricos. La sección está organizada de la siguiente forma: Primero se mostrarán los cambios en el nivel de definición de tipos y después se mostrará como se trasladan estos cambios al nivel de configuración.

### 5.2.1. Nivel de definición de tipos

En esta sección se muestra cómo la incorporación de las nuevas construcciones y de los aspectos predefinidos afectan al LDA-OA PRISMA, que tendrá que proporcionar primitivas para la especificación de éstos.

#### *5.2.1.1. Aspectos genéricos y obligatorios*

En primer lugar se describirá la especificación de los aspectos genéricos que todo ambiente ha de tener, es decir, son obligatorios. Estos aspectos son el Aspecto de Movilidad del Ambiente y el Aspecto de Coordinación del Ambiente.

### o Aspecto de Movilidad del Ambiente

A continuación se mostrará la especificación del Aspecto de Movilidad (`Mobile`) que importarán todos los ambientes. El aspecto especifica la interfaz `ICapability`, la cual proporciona los servicios que se han incorporado para poder proporcionar los capabilities del Cálculo de Ambientes (`enter`, `exit`) junto a nuevos servicios que son necesarios para dotar al aspecto de la funcionalidad necesaria (`startMovement`, `finishMovement`, `accept`, `changeLocation`) (ver Tabla 19). La capability replicate también ha sido incorporada al modelo [Ali06b], pero en la presente sección el discurso se centrará en las capabilities únicamente de movilidad.

```
Interface ICapability

   startMovement(input Name:string, output CommunicationList[]:
                 Attachment);
   exit (Name: string, Parent:string );
   enter (Name: string, NewAmbient: string);
   finishMovement(input Name, input CommunicationList[]: Attachment);
   changeLocation(Name: string, NewLocation: string);
   accept(input NewAmbient: string, input Caller:string,
               input Child:string, input Type: string,
               input AttachmentsList[]:Attachments,
               output AcceptanceOK: boolean);

End_Interface ICapability
```

**Tabla 19: Interfaz ICapability**

La especificación del Aspecto de Movilidad genérico de los ambientes, `Mobile`, se muestra en la Tabla 20..

```
1  Mobility Aspect Mobile using ICapability
     …. …
   Services
       begin();
        in startMovement(input Name:string,
                         output CommunicationList[]: Attachment);
        in finishMovement(input Name:string,
                          input CommunicationList[]: Attachment);
       end;

2  TRANSACTIONS in exit(Requested: string, Ambient: string):
     EXIT := out isChild(input Requested, output isChildOK)
                 →EXIT1;
     EXIT1::= {isChildOK==true & self.Name==Ambient}
               getParent(output Parent)→EXIT2;
     EXIT2::= moving (input Requested, input Parent);

3  TRANSACTIONS in enter(Requested: string, NewAmbient: string):
     ENTER := out areChildren(input Requested, input Ambient,
                              output areChildrenOK)→ENTER1;
     ENTER1::= {areChildrenOK==true} in moving(Requested, NewAmbient);

4  TRANSACTIONS in moving(Requested: string, NewAmbient: string):
     MOVING := out checkTypeAmbient( input Requested, output
                  isTypeAmbient) → MOVING1;
     MOVING1::=  if(isTypeAmbient==false)then
```

```
                createAmbientFor ( input Requested,
                                   output RequestedAmbient) → MOVING2
            else MOVING2 ;
    MOVING2::= out movingInf(input Requested, input self.Name,
                             input Requested, output Type,
                             output AttachmentList[])→ MOVING3;
    MOVING3::= out accept(input NewAmbient, input Type,
                          input Requested, input AttachmentsList[],
                          output AcceptanceOK)→ MOVING4;
    MOVING4::= {Acceptance0K==true}
               out modifyAttachment(Requested)→MOVING5;
    MOVING5::= out removeAttachments(Requested)→MOVING6;
    MOVING6::= out removeChild(Requested);

5   TRANSACTIONS in accept(input NewAmbient:string, input Caller: string,
                           input Child:string, input Type:string,
                           input AttachmentsList[]:Attachments,
                           output AcceptanceOK: boolean):
      ACCEPT::= out addChild(input Type, input Child,
                             input AttachmentsList[], output AddedOK)
                  {AcceptanceOK==AddedOK}→ ACCEPT1;
      ACCEPT1::=out changeLocation(Child, self.Name);

6   Preconditions
          in accept(input NewAmbient, input Caller,
                    input Child, input Type,
                    input AttachmentsList[],
                    output AcceptanceOK)
             if
               {self.Name==NewAmbient};

7   Played_Role
      INTERIOR for ICapability::= accept?(input NewAmbient,
                                          input Caller,
                                          input Child,
                                          input Type,
                                          input AttachmentsList[],
                                          output AcceptanceOK)
                                  +
                                  accept!(input NewAmbient,
                                          input Caller,
                                          input Child,
                                          input Type,
                                          input AttachmentsList[],
                                          output AcceptanceOK)
                                  +
                                  enter?(input Requested,
                                         input NewAmbient)
                                  +
                                  exit?(input Requested,
                                        input Ambient)
                                  +
                                  startMovement?(input Name,
                                                 output CommunicationList[])
                                  +
                                  finishMovement?(input Name,
                                                  input CommunicationList[])
                                  +
                                  changeLocation!(input Child,
                                                  input self.Name);

8   EXTERIOR for ICapability::= accept?(input NewAmbient, input Caller,
                                        input Child, input Type,
                                        input AttachmentsList[],
                                        output AcceptanceOK)
                                +
                                accept!(input NewAmbient, input Caller,
                                        input Child, input Type,
```

```
                                        input AttachmentsList[],
                                        output AcceptanceOK) ;

    …    …
End_Mobility Aspect Mobile
```

**Tabla 20: Aspecto de Movilidad Mobile**

Los servicios `startMovement` y `finishMovement` (Tabla 20, sección 1) son servicios auxiliares que utiliza el aspecto para realizar acciones especiales al inicio y al final de un proceso de movilidad. El servicio `StartMovement` es invocado por un elemento arquitectónico para avisar al ambiente que va a empezar a invocar una cadena de *capabilities*. El servicio `FinishMovement` es invocado por un elemento arquitectónico para indicar al ambiente que éste ha terminado de invocar a su cadena de *capabilities*, es decir, ha finalizado el proceso de movilidad. De esta forma los ambientes disponen de servicios para sincronizar el inicio y el final de un proceso de movilidad.

La transacción `exit(Requested: string, Ambient: string)` (Tabla 20, sección 2) es la encargada de procesar las peticiones de salida por parte de los elementos arquitectónicos. Para ello se comprueba que el elemento que requiere la petición se encuentra entre los elementos ubicados en el ambiente. Si esto es así, el ambiente comprueba si el `exit` va dirigido a él. De cumplirse las condiciones, se obtiene el nombre del Ambiente Padre( esto es posible gracias que el servicio `getParent()` tiene un *weaving* con el servicio `getLocation()` del aspecto de distribución), es decir el que será la nueva ubicación del elemento arquitectónico tras la movilidad y se procesaría la movilidad mediante el servicio transaccional `moving`.

La transacción `enter(Requested: string, NewAmbient: string)` (Tabla 20, sección 3) es la encargada de procesar las peticiones de entrada por parte de un elemento arquitectónico que necesite la entrada a un subambiente del ambiente. Para ello comprueba que ambos, tanto el elemento que solicita la entrada en el ambiente, como el ambiente destino de la movilidad son hijos del ambiente que recibe la petición. Si eso se cumple, de nuevo, se procesaría la movilidad mediante una llamada a `moving`.

La transacción `moving(Requested: string, NewAmbient: string)` (Tabla 20, sección 4) es la que alberga prácticamente toda la lógica de la movilidad en el ambiente origen de ésta. Inicialmente comprueba que el elemento arquitectónico que se desea mover es un ambiente. Esto es debido a que toda movilidad estará encapsulada en un ambiente. Por ello, si el tipo no es un ambiente, se creará un nuevo ambiente alrededor del elemento móvil para proceder a su movimiento. Una vez comprobado esto, se obtienen los canales de comunicación conectados al subambiente que se encapsulan en una lista para poder regenerarlos en destino. Con toda la información recopilada se invoca al método `accept` del ambiente destino. El ambiente destino de la movilidad será el Ambiente Padre en caso de un `exit` o un ambiente hijo en el caso de un `enter`. En cualquier caso, el ambiente que está procesando la orden estará conectado mediante los puertos del aspecto de coordinación a ambos. Si el ambiente

destino acepta la entrada del elemento arquitectónico, el ambiente origen concluye el proceso de la movilidad mediante la modificación de los *attachments*, con lo que reconstruirá los canales de comunicación. Finalmente, el ambiente eliminará el elemento arquitectónico y los *attachments* asociados de su lista de elementos.

Mediante la transacción `accept` (Tabla 20, sección 5), un ambiente ubicará un nuevo elemento arquitectónico. Esta transacción añade el elemento a su lista de elementos, conectándolo convenientemente gracias a la lista de canales de comunicación que recibe como parámetro y finalmente solita el servicio `changeLocation` del nuevo elemento, para que este actualice su ubicación en el Aspecto de Distribución. Este servicio transaccional alberga la lógica de la movilidad en el ambiente destino de ésta. Dicha transacción tiene asociada una precondición (Tabla 20, sección 6), según la cual, el aspecto solo ha de ejecutar dicho servicio transaccional si va dirigido a él.

Finalmente se describen los *played roles* `INTERIOR` (Tabla 20, sección 7) y `EXTERIOR` (Tabla 20, sección 8). Mediante el *played role* `INTERIOR` el ambiente ofrece las *capabilities* a los elementos ubicados en su interior. Es importante subrayar en este punto que, aunque ofrezca todas las *capabilities* existen una restricción para que los elementos arquitectónicos puedan solicitarlas. De hecho, un elemento arquitectónico no puede solicitar un `enter` o un `exit` por si mismo, sino que toda petición de `enter` y exits ha de estar entre un `StartMovement` y un `FinishMovement`. Esta restricción se aplica a nivel de especificación en los aspectos de distribución de los elementos móviles. Del mismo modo, no se puede solicitar un `accept` directamente a un ambiente; esta petición ha de venir de la ejecución de un `enter` o un `exit`.

Por otro lado, el *played role* `EXTERIOR` ofrece las *capabilities* a los elementos ubicados en el exterior del ambiente. Después de explicar las restricciones a que están sujetas la solicitud de *capabilities* se puede entender porqué para los elementos ubicados en el exterior, el ambiente tan solo ofrece el servicio `accept`. Es decir, el ambiente padre tan solo usará ese puerto con *played role* `EXTERIOR` para enviar al ambiente los `accepts` de los elementos arquitectónicos que quieran ubicarse en su interior. En ningún caso a través de ese *played role* un elemento arquitectónico le va a solicitar otra *capability* de movilidad, ya que el ambiente no proporciona semántica de movilidad a los elementos ubicados en su exterior.

### o Aspecto de Coordinación del Ambiente

En esta sección se muestra la especificación LDA-OA del aspecto de Coordinación del Ambiente. Este aspecto es importado por todos los ambientes en una especificación genérica, al igual que el de Movilidad. De nuevo, esta especificación no podrá ser modificada por el usuario y su semántica tan solo será ampliable mediante otros aspectos entretejidos mediante *weavings*.

El aspecto de Coordinación del Ambiente (`ACoordination`) implementa la interfaz `ICall` que es una interfaz de llamadas a método genéricas. La idea reside en que exista un método que represente de forma genérica a todas las llamadas a métodos, para de esa forma poder procesar las peticiones de forma transparente a la interfaz a la que pertenezcan. En la Tabla 21 se muestra la interfaz `ICall`.

```
Interface ICall

   call(input MethodName:string, input paramsList[]: Parameters,
        input paramType[]:bool);

End_Interface ICall
```

**Tabla 21: Interfaz ICall**

Como se puede ver en la signatura del método `call`, el primer parámetro hace referencia al nombre del servicio que se quiere invocar y en el siguiente se introduce un array con los argumentos del método. Puesto que los parámetros en los servicios pueden ser de entrada o salida, es decir, *input* y *output*, respectivamente, es necesario añadir un tercer parámetro que sea un *array* con la información sobre el tipo de los parámetros. Este *array* tendrá las mismas posiciones que la lista de parámetros del método original y para cada una de esas posiciones indicará con un valor booleano si el parámetro es de entrada (*true*) o si es de salida (*false*). Esta lista permite reconstruir la llamada original tras salir del Aspecto de Coordinación. De esa forma, se puede adaptar a todo tipo de servicios PRISMA, sea cual sea su signatura. En la Tabla 22 se muestra la especificaión del aspecto `ACoordination`.

```
1  Coordination Aspect ACoordination using ICall

2  Services
      begin();
      in/out call(input MethodName:string,
                  input paramsList[]: Parameters,
                  input paramType[]: bool);
      end;

3  Played_Role
     INTERIOR for ICall::= call?(input MethodName, input paramsList[], input
   paramType[])
                            +
                            call!(input MethodName, input paramsList[], input
   paramType[];

4  EXTERIOR for ICall::= call?(input MethodName, input paramsList[], input
   paramType[])
                            +
                            call!(input MethodName, input paramsList[],
                                  input paramType[]: bool);
5  Protocol
     ACOOR:= begin→ ACOOR1;
     ACOOR1:= ACOOR2+ACOOR3+ACOOR4+ACOOR5+ end;
     ACOOR2:=INTERIOR.call?(input MethodName, input paramsList[], input
   paramType[]) →
              EXTERIOR.call!(input MethodName, input paramsList[], input
   paramType[]);
```

```
   ACOOR3:=INTERIOR.call!(input MethodName, input paramsList[], input
paramType[])→
          EXTERIOR.call?(input MethodName, input paramsList[], input
paramType[]);
 ACOOR4:= EXTERIOR.call!(input MethodName, input paramsList[], input
paramType[]) →
          INTERIOR.call?(input MethodName, input paramsList[], input
paramType[]);
 ACOOR4:= EXTERIOR.call?(input MethodName, input paramsList[], input
paramType[]) →
          INTERIOR.call!(input MethodName, input paramsList[], input
paramType[]);

End_Coordination Aspect ACoordination
```

**Tabla 22: Aspecto de Coordinación del Ambiente ACoordination**

La especificación del aspecto (Tabla 22, sección 1) define el servicio de entrada/salida `call` con los parámetros antes explicados (Tabla 22, sección 2). Del mismo modo se definen dos *played roles*, uno para la comunicación con los elementos ubicados en el interior del ambiente (`INTERIOR`)( Tabla 22, sección 3) y otro para la comunicación con los elementos ubicados en exterior del ambiente (`EXTERIOR`)( Tabla 22, sección 3). Finalmente se describe el protocolo (Tabla 22, sección 4) en el que se orquesta el comportamiento del Aspecto de Coordinación; los flujos de entrada que entran por un *played role* son transformados en flujos de salida por el otro y viceversa.

#### *5.2.1.2. Aspectos no genéricos y obligatorios: Aspecto de Distribución*

El Aspecto de Distribución es un aspecto que todo ambiente ha de importar. Pero a diferencia del Aspecto de Coordinación y el de Movilidad no se especifica de forma genérica, pudiendo el diseñador adaptarlo a sus necesidades.

Como se ha comentado, la semántica del Aspecto de Distribución en el Ambiente es darle una ubicación al propio ambiente en la jerarquía de ambientes. Todos los ambientes están ubicados dentro de otro ambiente, siguiendo las restricciones del modelo. Por ello, los ambientes tendrán una referencia a su ambiente padre, es decir aquel en el que se ubican, mediante el campo `location` del aspecto de distribución. Este campo ha de ser consultable mediante un método con el cual se pueden definir *weavings* desde otros aspectos.

Hay una excepción a la afirmación de que todos los ambientes han de tener un padre, puesto que a la hora de modelar hay que poner un límite al detalle de la jerarquía de localizaciones; al final se llegará a un punto en que el diseñador ya no vea necesario seguir modelando ambientes. Por ello, el ambiente que está en la parte más alta de la jerarquía de ambientes, es decir el *Root*, es el único que no está ubicado en un ambiente. Visto desde un punto de vista arquitectónico, un ambiente de tipo *Logical* es el único que se puede acoger a esta excepción, ya que los ambientes *Process* están obligados a

pertenecer a un ambiente *Site* o a otro *Process* y los ambientes *Site* han de pertenecer a un *Logical*.

A modo de ejemplo, la Tabla 23 se muestra la especificación de un Aspecto de Distribución sencillo, con la funcionalidad mínima para albergar la información sobre la localización del ambiente, para lo cual importa la interfaz `IGetLocation`:

```
1   Interface IGetLocation
       getLocation(output Location: string);
    End_Interface IGetLocation

2   Distribution Aspect ADist using IGetLocation

3   Attributes
      Constant
      location : string NOT NULL;
4     [physicalLocation: LOC;]

5   Services
      begin(input ParentAmbient: string, input PhysicalLocation: LOC)
         Valuations
          [begin (ParentAmbient[,PhysicalLocation])]
                              location := ParentAmbient,
                              [physicalLocation:=PhysicalLocation;]
      in getLocation( output Location:string)
         Valuations
             [getLocation(output Location)] Location := location;
      end;

6   Protocol
      DIST:= begin→ DIST1;
      DIST1:= getLocation(Location)+ end;

    End_Distribution Aspect ADist
```

**Tabla 23: Especificación de un aspecto de Distribución**

Tras importar la interfaz `IGetLocation` (Tabla 23, sección 1) en la cabecera del aspecto (Tabla 23, sección 2), el aspecto define sus atributos. Como se puede observar el aspecto tiene un campo constante no nulo en el que almacena la información sobre la ubicación (Tabla 23, sección 3), es decir, el ambiente padre. Puesto que el campo es no nulo, el ambiente *Root* tendrá un valor nulo que indicará que es la parte más alta de la jerarquía.

Un caso especial se da en los ambientes de tipo *Site*, pues han de incorporar un campo especial llamado `physicalLocation` (4) en el que se indique la localización física a la que representan. Este campo tan solo tiene sentido en este tipo de ambientes, pues como se ha explicado, representan dispositivos físicos que tienen una localización en la red. Para el tipo del campo se ha utilizado el tipo `LOC`, que como se comentó, representa una localización válida en el entorno de ejecución.

A continuación se definen los servicios, donde se da semántica al servicio `GetLocation` (Tabla 23, sección 5) y los protocolos (Tabla 23, sección 6). La especificación puede ampliarse en función de las necesidades del usuario.

#### *5.2.1.3. Ambientes*

A parte de los aspectos genéricos que ha de incorporar todo ambiente, ha sido necesario incorporar la plantilla de especificación LDA-OA para la descripción de los ambientes. Esta plantilla ha de permitir al diseñador definir el tipo de ambiente que quiere especificar (Tabla 24 sección 1) y , al igual que en el caso de los componentes y conectores, la importación de aspectos, definición de *weavings* y creación de puertos. Todo esto teniendo en cuenta los aspectos genéricos, obligatorios y sus puertos, que siempre han de crearse.

A continuación se lista la plantilla de especificación de ambientes en el LDA-OA de Ambient-PRISMA:

```
1   Ambient_[ Site | Process | Logical]_type <nombre_Ambiente>

2   Mobility Aspect Import Mobile;
    Coordination Aspect Import ACoordination;

3   Distribution Aspect Import <nombre_aspecto_dist>;

4   <tipo_aspectok> Aspect Import <nombre_aspectok>;

5   Weaving
      <nombre_aspecto_dist>.getLocation(Location) instead
                                                    Mobile.getParent(Parent);

6   <nombre_aspectok>.<nombre_serviciok>
      <operador_weaving>
        <nombre_aspecton>.<nombre_servicion>;
    End_Weaving;

7   Port
         InCapabilitiesPort: ICapability
             Played_Role Mobile.Interior;
         ECapabilitiesPort: ICapability
             Played_Role Mobile.Exterior;
         EServicesPort: ICall
             Played_Role ACoordination.Exterior;
         InServicesPort: ICall
             Played_Role ACoordination.Interior;

8        <nombrei> : <interfazi>
              Played_Role <nombre_aspecto>.<nom_playedRole_del_aspecto>;
    End_Port;

9   Initialize
      New(<argumentos>) {
        <nombre_aspectoi>.begin(<args_constructori>);
      }
    End_Initialize;

    Destruction
      Destroy() {
        <nombre_aspectoi>.end(<args_destructori>);
      }
    End_Destruction;
```

```
End_Ambient_[ Site | Process | Logical ]_type <nombre_Ambiente>;
```

**Tabla 24: Plantilla de ambiente**

Es decir, inicialmente se indica el tipo de ambiente que se va a especificar (Tabla 24, sección 1) y se indica el nombre. A continuación se importan los aspectos obligatorios. Nótese como para los aspectos genéricos ya se proporciona el nombre del aspecto (Tabla 24, sección 2), mientras que para el caso del Aspecto de distribución (Tabla 24, sección 3), aspecto obligatorio pero no predefinido, se incluye la línea de importación dejando el nombre para la decisión del diseñador. Es posible importar más tipos de aspectos (Tabla 24, sección 4).

A continuación se definen los *weavings* entre los aspectos, al igual que ocurría en el caso de los componentes y los conectores. En este punto cabe destacar que es necesaria la definición de un *weaving* especial, un *weaving* que entreteja el aspecto genérico de Movilidad y el aspecto de Distribución. El Aspecto de Movilidad genérico tiene un servicio para consultar la ubicación del ambiente (`GetParent`). Este servicio ha de entretejerse con el aspecto de distribución, que es quien tiene esta información, en concreto con el servicio `GetLocation`. Por eso es obligatorio que el aspecto de Distribución de un Ambiente implemente la interfaz `IGetLocation`, como se ha explicado en la sección 5.1.1.3. Así, se define el *weaving* entre estos dos aspectos de manera predeterminada (Tabla 24, sección 5). Al igual que se pueden importar más tipos de aspectos, la plantilla permite definir más *weavings* (Tabla 24, sección 6):

También cabe remarcar la sección `Ports`, en la que se definen los puertos. Como se ve, por defecto se crean cuatro puertos, correspondientes al *played role* `INTERNO` y `EXTERNO` de los dos aspectos predefinidos, es decir, dos puertos para el Aspecto de Movilidad y dos puertos para el Aspecto de Coordinación del Ambiente (Tabla 24, sección 7). A partir de ahí es posible definir nuevos puertos en función de los aspectos importados previamente (Tabla 24, sección 8).

Finalmente se especifican las operaciones de inicialización y destrucción de los aspectos, al igual que se hace para el resto de elementos arquitectónicos

### 5.2.2. Nivel de configuración

Finalmente, ya solo quedaría ver como afectan los cambios al nivel de configuración. Puesto que se han definido primitivas nuevas en el LDA-OA, es posible instanciar nuevos tipos para ponerlos en ejecución. También se ve afectado el orden de creación de las intancias, ya que, como la instanciación de los aspectos de distribución de los elementos arquitectónicos requieren el nombre de una instancia de ambiente, es necesario crear los ambientes

previamente. Ocurre algo similar a lo que ocurría con los LOC en el nivel de configuración de PRISMA.

Otra característica a remarcar en este nivel es el hecho que, debido a la necesidad de definir un ambiente root que englobe todo el entorno de ejecución, es necesario instanciar ese ambiente en todo caso. Se tratará de un ambiente *Logical* que se instanciará el primero, pues los siguientes ambientes se definirán jerárquicamente debajo de él y, por tanto, necesitarán una referencia.

En la Tabla 25 se muestra la plantilla de una configuración arquitectónica.

```
1  Architectural_Model_Configuration <nombre_configuracion> =
2  New <nombre_modelo>{
3    Root = new Ambient_type_Logical() ;
4    <nom_inst_ambiente>= new <nom_ambiente> (<args>);
5    <nom_inst_comp>= new <nom_componente> (<args>) ;
     <nom_inst_con>= new <nom_conector> (<args>) ;
6  <nom_inst_att>= new <nom_attachment> (<args>) ;
   }
```

**Tabla 25: Plantilla de Configuración Arquitectónica**

Como se puede observar, al igual que ocurre en PRISMA; se define el nombre de la configuración (Tabla 25, sección 1) y se instancia el modelo (Tabla 25, sección 2). Para ello, se empieza instanciando el ambiente *Logical Root* (Tabla 25, sección 3). A partir de él se instanciarán tantos subambientes como el diseñador estime necesario para definir su modelo (Tabla 25, sección 4). Cada instancia recibirá un nombre (<*nom_instancia_ambiente*>). Es necesario remarcar que los ambientes se han de definir en orden, ya que los subambientes requieren una referencia al padre que debe estar previamente creado. Una vez instanciados los ambientes, se intancian el resto de los elementos arquitectónicos (Tabla 25, sección 5). Para cada uno de los tipos de elementos arquitectónicos se van creando las instancias, a las cuales se les da un nombre (<*nom_instancia_tipo*>) y a las que se localizarán mediante los ambientes previamente creados.

Finalmente se instancián los *attachments* entre los elementos arquitectónicos (Tabla 25, sección 6). Los *attachments* representan los canales de comunicación, por ello, al definirse en base a los elementos arquitectónicos que comunican es necesario que se intancien después de éstos.

Para simplificar la tarea del diseñador, para la instanciación de los *attachments* no es necesario representar la jerarquía de ambientes, pues con la información introducida previamente (instanciación de los elementos

arquitectónicos que se van a conectar y los ambientes a que pertenecen) ya se tiene la información necesaria para establecer esta comunicación automáticamente, es decir, crear el camino de *attachments* entre los diferentes puertos del ambiente que tienen un *played role* con los aspectos de coordinación de los diferentes ambientes que han de atravesar el canal de comunicación.

A continuación se muestra la instanciación de un modelo arquitectónico simple, utilizando el ejemplo del Sistema Bancario, esta vez con ambientes. Se supone que existen dos ubicaciones físicas en las que pueden ejecutarse los tipos. Estas ubicaciones se representarán mediante ambientes *Site* a los que se asociarán los tipos, en función de donde se vayan a ejecutar. Estos ambientes son instanciados mediante instancias de la clase LOC, definidas previamente, tal y como se muestra a continuación:

```
Architectural_Model_Configuration BankConf =

New SimpleBankSystem
{
        IP1 = new LOC(ip.del.host.1)
        IP2 = new LOC(ip.del.host.2)

        ROOT = new Root () ;

        SITE1 = new AmbientSite(ROOT, IP1);
        SITE2 = new AmbientSite(ROOT, IP2)


        ACCOUNT1 = new Account("000001", SITE2);
        ACCOUNT2 = new Account("000002", SITE2);
        CONNECTOR = new CnctrAccount(SITE1);

        ATT1 = new AccountOpAtt(ACCOUNT1,Operations_port,
                                AccountOp_port, CONNECTOR);
        ATT2 = new AccountOpAtt(ACCOUNT2,Operations_port,
                               AccountOp_port, CONNECTOR);
};
```

**Tabla 26: Ejemplo de configuración arquitectónica con ambientes**

Se han instanciado dos ambientes *Site*, uno que contiene las cuentas *Account1* y *Account2* y otro con el conector *Connector*. Ambos ambientes están ubicados en el ambiente *Logical* Root (ver Figura 69). Como se puede observar, pese a que los canales de comunicación instanciados están formados por varios tramos entre los diferentes ambientes, a la hora de definirlos tan solo se han definido los elementos arquitectónicos que conectan y a través de que puertos.

Automatizando el proceso de creación de los canales de comunicación simplifica la tarea al diseñador. Esto ha sido posible gracias a la utilización de aspectos predefinidos con sus puertos también predefinidos. De no ser predefinidos no se podría generar un camino a través de la jerarquía de ambientes entre dos puertos de diferentes elementos arquitectónicos, tan solo conociéndolos a ellos, pues seria necesario conocer los puertos que usaría cada ambiente por el que pasa el canal de comunicación para la comunicación

distribuida, cosa que no ocurre si estos puertos son genéricos y comunes a todos los ambientes.

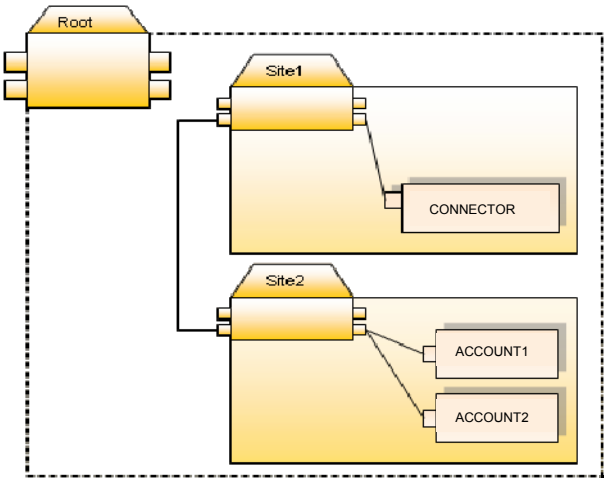



**Figura 69: Modelo del Sistema Bancario haciendo uso de ambientes**

También se puede observar en la configuración de la Tabla 26 cómo al instanciar los ambientes *Site* se les ha pasado el valor del `physicalLocation` con una instancia de LOC previamente creada.

## 5.3. Caso de estudio

En la presente sección se presentará la especificación de un caso de estudio haciendo uso del LDA-OA en el que se pondrán de manifiesto las novedades incorporadas por el Ambient-PRISMA para especificar un sistema distribuido con elementos arquitectónicos móviles. El caso de estudio utilizado es del área de los agentes móviles.

Los Agentes Móviles [Chess95] son componentes que pueden moverse desde una máquina a otra para desempeñar una serie de tareas de forma autónoma. Este tipo de agentes resuelven algunos de los problemas que son encontrados en los sistemas distribuidos. El tráfico de red se reduce ya que el agente se mueve a la máquina destino a desempeñar sus tareas de forma local en lugar de pasar información a través de la red. Además, en el caso de pérdidas de conexión con determinado nodo de la red ya no supone un problema, pues el agente realiza sus tareas en local y transmitirá sus resultados cuando la conexión sea restablecida.

Son muchas las aplicaciones en las que se pueden aplicar los agentes móviles. En la presente sección se presentará la especificación PRISMA de una aplicación de comercio electrónico: Un sitio de subastas electrónicas donde son subastados productos.

### 5.3.1. Agentes Móviles en una Subasta Electrónica

Para la presentación del caso de estudio del Sitio de Subastas se especificará la funcionalidad para este tipo de sitios: Un cliente del sitio de subastas puede estar interesado en comprar un producto específico con determinadas características a un precio máximo. Para mantener un seguimiento de las diferentes subastas que tienen lugar en el sitio de subastas, el cliente designa dos agentes móviles para encargarse de la elección del producto y su compra enviándolos al sitio de subastas. Ya en el sitio de subastas, los dos agentes trabajarán de forma colaborativa para actuar en nombre del cliente y realizar la compra. Una vez realizada la compra, los agentes volverán al sitio del cliente.

En el caso de estudio, el cliente designa dos agentes: El *Purchaser* y el *Collector*. Como el *Purchaser* y el *Collector* necesitan trabajar de forma colaborativa, un conector llamado *AgentCnctr* se encargará de conectarlos. El cliente, representado por el componente software *Customer*, localizará estos elementos arquitectónicos en un ambiente *AmbientProcess* llamado *AgentsAP* al que se conectará mediante un conector *AgentCustCnctr*. Inicialmente el *Customer* y el *AgentsAP* están ubicados en *ClientSite*, que representará el sitio del cliente.

Por el otro lado, en el sitio de la subasta ,*AuctionSite*, hay un componente *Auction* que representa la subasta. *AuctionCnctr* es un conector que coordina *Auction* con el resto de elementos arquitectónicos. Como *AuctionCnctr* puede comunicarse con elementos arquitectónicos distribuidos, tiene un *attachment* con el ambiente *AuctionSite*. En la figura Figura 70 se muestra la configuración inicial del caso de estudio.

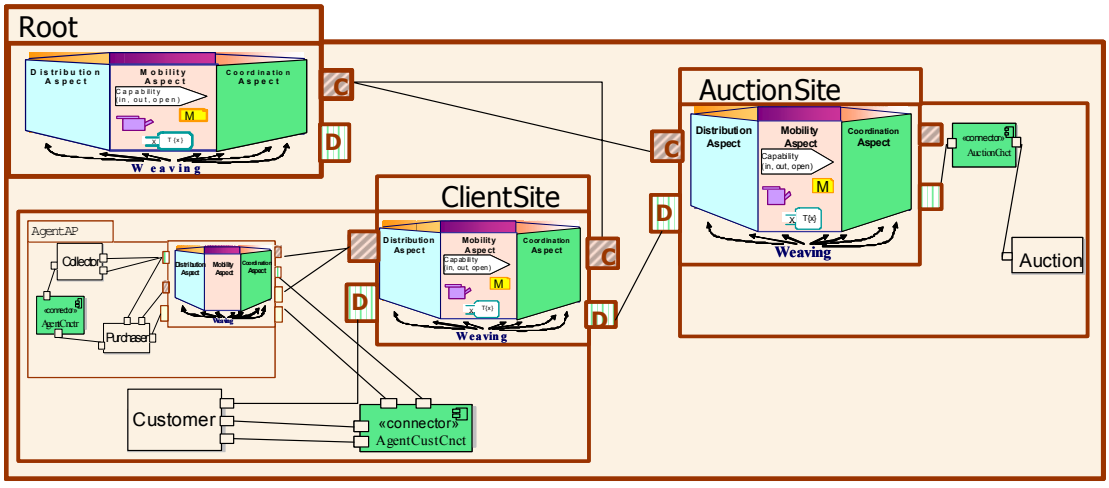



**Figura 70: Configuración inicial de la arquitectura del caso de estudio**

La movilidad del *AgentAP* se iniciará cuando el componente *Customer* solicite su cambio de localización hacia *AuctionSite*. El ambiente *AgentsAP* recibirá la petición de movilidad a través de sus puertos (en concreto, a través de un puerto llamado *EMovilityPort*) y esto provocará la ejecución del servicio `move` del Aspecto de Distribución que importa el ambiente. El método `move` del

aspecto, especificará que el ambiente ha de salir de su ambiente padre y hará después entrara al AuctionSite mediante las operaciones propias del Ambient-PRISMA, es decir, mediante un `exit` y un `enter` respectivamente. Estos servicios serán servidos por el ambiente padre del *AgentAP*. Nótese como, para poder proporcionar estos servicios una vez *AgentAP* se encuentra fuera de *ClientSite* es necesario que exista un ambiente padre a éste, que se denominará *Root* y será al que el subambiente móvil solicitará las *capabilities* en ese estado de transición durante la movilidad.

A continuación se mostrará la especificación en el LDA-OA del caso de estudio. La idea es remarcar la naturaleza distribuida del ejemplo y de como se ha modelado haciendo uso de las primitivas del Ambient-PRISMA.

### 5.3.2. Especificación del caso de estudio

En esta subsección se mostrarán los diferentes tipos Ambient-PRISMA que se han definido para llevar a cabo la especificación del caso de estudio. La sección está organizada en diferentes subsecciones en las que se explican los tipos de elementos arquitectónicos. En cada sección se hallará la definición del elemento arquitectonico y de los tipos de aspectos que importa. Los aspectos que se reutilizan en diferentes elementos arquitectónicos sólo se han definido una vez junto al primer elemento arquitectónico que los importa.

#### *5.3.2.1. Componentes*

En primer lugar se mostrarán las especificaciones relativas a las interfaces y aspectos que dan semántica a dichas interfaces que son necesarios para la definición de los diferentes componentes de la configuración arquitectónica.

- **Customer**

Interfaces, aspectos y la especificación del componente Customer, que representa al cliente de la subasta electrónica.

**Interfaces IMobility, ICustAuct y ICustAgent**

Los aspectos del componente `Customer` importarán las siguientes interfaces:

```
Interface IMobility
 move(NewAmbient: string);
End_Interface IMobility

Interface ICustAuct
 bid(ProductID: string, amount: real);
 notifyBuy (Buy : bool);
End_Interface ICustAuct
```

```
Interface ICustAgent
 changeProductDescription(input ProductDescr: string);
 changeMaxBidQuantity(input NewMaxBidQuantity: real);
End_Interface ICustAgent
```

**Tabla 27: Interfaces IMobility, ICustAuct y ICustAgent**

### Aspecto de Distribución CustDist

El Aspecto de Distribución del `Customer` (`CustDist`) describe cómo el `Customer` puede provocar el movimiento de los Agente mediante un servicio de salida. El servicio **out** `move(NewAmbient: string )`. El `Customer` tiene un valor para su localización constante, esto es, no es un elemento móvil.

```
Distribution Aspect CustDist using IMobility
 Attributes
  Constant
  location : string NOT NULL;
 Services
  begin(input ParentAmbient: string)
     Valuations
      [begin (ParentAmbient)] location := ParentAmbient;
  out move (NewAmbient:loc );
  end;

 Played_Roles
  moveAgent for IMobility ::= move!(NewAmbient);
 Protocol
  CUSTDIST:= begin→ CUSTDIST1;
  CUSTDIST1:= moveAgent.move!(NewAmbient)+ end;

End_Distribution Aspect CustDist
```

**Tabla 28: Aspecto de Distribución CustDist**

### Aspecto Funcional CustFunc

El Aspecto Funcional del `Customer` (`CustFunc`) almacena la información acerca del cliente además de permitir pujar y pagar por un producto en la subasta. También permite modificar la descripción del producto y el valor máximo de puja que ha indicado en el agente.

```
Functional Aspect CustFunct using ICustAuc, ICustAgent

 Attributes
  Variables
  userName : string;
  password: string;
  address: string;
  telephone: string;
  credit: real;
  buy: bool;

 Services
  begin;
  changePassword( NewPassword:string)
     Valuations
```

```
        [changePassword( NewPassword)] password := NewPassword;
   changeAddress( NewAddress:string)
      Valuations
        [changePassword(NewAddress)] address := NewAddress;
   changeTelephone( NewTelephone:string)
      Valuations
        [changeTelephone( NewTelephone)] telephone := NewTelephone;
   addCredit(Amount: real)
      Valuations
        [addCredit(Amount)] credit:= credit+Amount;
   in pay(Amount: real)
      Valuations
        [pay(Amount)] credit:= credit-Amount;
   in notifyBuy(input Buy : bool)
     Valuations
        [notifyBuy(Buy)] buy:= Buy;

   out bid( input ProductID: string, input Quantity: real);
   out changeProductDescription(input ProductDescr: string);
   out changeMaxBidQuantity(input NewMaxBidQuantity: real);
   end;

  trigger
   in pay(Amount) when {Buy== true};

  Played_Roles
   CUSTAUCT for ICustAuc ::= bid!(ProductID, Quantity)+ notifyBuy?(Buy)
   CUSTAgent for ICustAgent ::=changeProductDescription!( ProductDescr)
                               +
                              changeMaxBidQuantity!(NewMaxBidQuantity);

Protocol
  CUSTFunct ::= begin→ CUSTFunct1;
  CUSTFunct1 ::= CUSTMODIFY + BIDING + end;
  CUSTMODIFY ::=(changePassword?(NewPassword)
                +
                changeAddress?(NewAddress)
                +
                chageTelephone?(NewTelephone)
                +
                addCredit?(Amount)
                +
                 CUSTAgent.changeProductDescription!( ProductDescr)
                 +
                 CUSTAgent. changeMaxBidQuantity!(NewMaxBidQuantity))
                → CUSTFunct1;
  BIDING ::= CUSTAUCT.bid!(ProductID, Quantity)→
                          ((BIDING
                            +
                           (CUSTAUCT.notifyBuy?(Buy)→pay(Amount))
                            → CUSTFunct1;

End_Functional Aspect CustFunct
```

**Tabla 29: Aspecto Funcional CustFunc**

### Componente Customer

El componente `Customer` importa un aspecto funcional y un aspecto de distribución. El componente tiene tres puertos: El puerto `MovePort` para enviar la orden de movilidad a los agentes, el `AgentsPort` para enviar mensajes a los agentes y el `AuctionPort`, para enviar mensajes a la subasta.

```
Component_type Customer
```

```
 Import Distribution Aspect CustDist;
 Import Functional Aspect CustFunct;

 Ports
     MovePort: IMobility, Played_Role CustDist.moveAgent;
     AgentsPort: IAgents Played_Role CustFunct.CUSTAgent;
     AuctionPort: IAuction Played_Role CustFunct.CUSTAUCT;
 End_Ports

End Component_type Customer;
```

**Tabla 30: Componente Customer**

### o Collector

Interfaces, aspectos y la especificación del componente Collector, agente encargado de recopilar información sobre subastas en curso.

#### Interfaces ICustAgent, ICollAuct y ICollPur

Los aspectos del componente `Collector` importarán las siguientes interfaces:

```
No distribution or mobile interfaces

Interface ICustAgent
 changeProductDescription(ProductDescr: string);
 changeMaxBidQuantity(NewMaxBidQuantity: real);
End_Interface ICustAgent

Interface ICollAuct
 bid(input ProductID: string, input Quantity: real);
 lookforProduct(input Description, output ProductID, output CurrentBid);
 notifyBuy(Buy : bool);
End_Interface ICollAuct

Interface ICollPur
 puchase(input ProductID: string, input Quantity: real);
End_Interface ICollPur
```

**Tabla 31: Interfaces ICustAgent, ICollAuct y ICollPur**

#### Aspecto de Distribución Dist

El aspecto de distribución del `Collector` sólo especifica su localización. El `Collector` tiene una localización constante porque su ubicación no va a cambiar. Esto es porque siempre va a estar dentro del mismo ambiente.

```
Distribution Aspect Dist

Attributes
 Constant
   location : string NOT NULL;

Services
```

```
    begin(input ParentAmbient: string)
       Valuations
        [begin (ParentAmbient)] location := ParentAmbient;
    end;

  Protocol
    DIST:= begin→ DIST1;
    DIST1:= end;
  End_Distribution Aspect Dist
```

**Tabla 32: Aspecto de Distribución Dist**

### Aspecto Funcional CollectFunct

El Aspecto Funcional del `Collector` contiene la descripción del producto a comprar y el máximo valor de puja, así como servicios para modificar estos valores. También tiene información sobre el estado de la compra, que será modificada mediante el servicio `in notifyBuy(input Buy : bool)`. El aspecto requerirá servicios de puja siempre y cuando el valor no supere al valor máximo impuesto (`bid`), obtención de información sobre productos en subasta (`lookForProduct`) y de compra a otros elementos arquitectónicos (`purchase`).

```
  Functional Aspect CollectFunct using ICollPur, ICollAuct, ICustAgent

   Attributes
    Variables
    maxBidPrice: real NOT NULL;
    productDescrip: string NOT NULL;
    currentBidAmount: real;
    productID: string;
    buy: bool;

   Services
    begin(input MaxBidPrice: string, ProductDescrip: string)
       Valuations
        [begin (MaxBidPrice, ProductDescrip)]
                    maxBidPrice:=MaxBidPrice &
                    productDescrip:= ProductDescrip ;

    in changeProductDescription(input ProductDescr: string)
      Valuations
           [changeProductDescription(ProductDescr)]
                                    productDescrip:= ProductDescr;
    in changeMaxBidQuantity(input NewMaxBidQuantity: real)
      Valuations
           [changeMaxBidQuantity(NewMaxBidQuantity)]
                                    maxBidPrice:= NewMaxBidQuantity;
    out lookforProduct(input Description, output ProductID,
                                       output CurrentBidAmount)
       Valuations
           [lookforProduct(input Description, ProductID, CurrentBidAmount)]
                                    Description:=productDescrip &
                                    productID:= ProductID &
                                    currentBidAmount:= CurrentBidAmount;
    out bid( input ProductID: string, input Quantity: real)
     Valuations
           [bid(ProductID, Quantity)] ProductID:= productID &
                                     Quantity:= maxBidPrice ;
    in notifyBuy(input Buy : bool)
      Valuations
           [notifyBuy(Buy)] buy:= Buy;
```

```
     out purchase(input ProductID, input Quantity: real);
     end;

    Preconditions
     out bid(ProductID, Quantity) if {currentBidAmount<= maxBidPrice};

    trigger
     out purchase(ProductID, Quantity) if {currentBidAmount<= maxBidPrice};

    Played_Roles
     COLLPUR for ICollPur::= purchase!(ProductID,Quantity);
     COLLAUCT for ICollAuct ::= lookforProduct!(Description,
                                                ProductID,
                                                CurrentBid)
                                →
                                bid!(ProductID, Quantity:real )
                                →
                                notifyBuy?(Buy);
     CUSTAgent for ICustAgent ::= changeProductDescription?(ProductID,
                                                            ProductDescr)
                                  +
                                  changeMaxBidQuantity?(NewMaxBidQuantity);

    Protocol
     COLLECTFUNCT ::= begin→ COLLECTFUNCT1;
     COLLECTFUNCT1 ::= CHANGEINF + BIDINGAUCT + end;
     CHANGEINF ::= CUSTAgent.changeProductDescription?(ProductID,
                                                       ProductDescr)
                   +
                   CUSTAgent.changeMaxBidQuantity?(NewMaxBidQuantity)
                   → COLLECTFUNCT1;
     BIDINGAUCT ::= (lookforProduct!(Description, ProductID,CurrentBid)→
                           bid!(ProductID,Quantity)→
                           notifyBuy?(Buy) →
                           COllPUR.purchase!(ProductID, Quantity))
                     →COLLECTFUNCT1;

End_Functional Aspect CollectFunct
```

**Tabla 33: Aspecto Funcional CollectFunct**

### Componente Collector

El componente `Collector` importa un aspecto funcional y un aspecto de distribución. Así mismo incorpora tres puertos, uno para comunicarse con `Purchaser`, otro para comunicarse con `Customer` y otro para comunicarse con `Auction`.

```
Component_type Collector

 Import Distribution Aspect Dist;
 Import Functional Aspect CollectorFunct;

Ports
     CustAgentPort: ICustAgent Played_Role CollectFunct.CUSTAgent;
     COLLAUCTPort: ICollAuct Played_Role CollectFunct.COLLAUCT;
     CollPurPort: ICollPur Played_Role CollectFunct.COLLPUR
End_Ports

End Component_type Customer;
```

**Tabla 34: Componente Collector**

## o Purchaser

Interfaces, aspectos y la especificación del componente Purchaser, que representa un agente encargado de realizar las compras de productos en la subasta.

### Interface IPurAuct, IMobility, ICollPur

La siguiente interface se importará por los aspectos del componente `Purchaser`:

```
Interface IMobility
 move(NewAmbient: string);
End_Interface IMobility

Interface ICollPur
 purchase(input ProductID: string, input Quantity: real);
End_Interface ICollPur

Interface IPurAuct
 registerForPayment(input UserName:string,input Password:string,input
ProductID: string, output Register:bool)
End_Interface IPurAuct
```

**Tabla 35: Interface IPurAuct, IMobility y ICollPur**

### Aspecto de Distribución PurDist

El Aspecto de Distribución del `Purchaser` define que un `Purchaser` puede tener dos tipos de comportamientos moviles: uno es salir del `AgentAP` para realizar la compra y volver a entrar cuando termine y otro que indica que el `Purchaser` provoca la movilidad del `AgentAP` para volver al `ClientSite` cuando termine su labor. Por la primera razón el PurDist especifica dos transacciones `exiting` y `entering` para invocar los servicios de su ambiente padre para realizar la movilidad.

```
Distribution Aspect PurDist using IMobility, ICapability
 Attributes
  location : string NOT NULL;
  clientLocation:string;
  originAmbient: string;

 Services
  begin(input ParentAmbient: string)
     Valuations
      [begin (ParentAmbient)] location := ParentAmbient;
  in changeLocation(Name: String, NewLocation:loc)
        Valuations
          [changeLocation()] location:=NewLocation;

  out startMovement(input Name:string,
                    output CommunicationList[]: Attachment)
     Valuations
         [startMovement(Name, CommunicationList[])] Name:= self.Name;
```

```
      out exit (Name: string, Parent:string )
         Valuations
             [exit(Name, Parent)] Name:= self.Name &
                                  Parent:=originAmbient;
      out enter (Name: string, NewAmbient: string);
         Valuations
             [exit(Name, NewAmbient)] Name:= self.Name &
                                          NewAmbient:=originAmbient;

      out finishMovement(input Name, input CommunicationList[]);
         Valuations
             [finishMovement(Name, CommunicationList[])] Name:= self.Name;
      out move(NewAmbient:string)
         Valuations
             [move(NewAmbientLocation)] NewAmbient:= clientLocation;
      end;

     Transactions exiting(Name:string, Parent:string)
      EXITING::= out startMovement(input Name, output CommunicationList[])
                 →EXITING1;
      EXITING1::= out exit(Name,Parent)→ EXITING2;
      EXITING2::= out finishMovement(input Name, input CommunicationList[]);

     Transactions entering(Name:string, NewAmbient:string)
      ENTERING ::= out startMovement(input Name,
                                     output CommunicationList[])→ENTERING1;
      ENTERING1 ::= out enter(Name,Parent)→ ENTERING2;
      ENTERING2 ::= out finishMovement(input Name, input CommunicationList[]);

     Preconditions
        in  changeLocation(Name, NewLocation)
                                     if {self.Name==Name};

     Played_Roles
      moveAgent for IMobility ::= move!(NewAmbient);
      CapParent for ICapability ::=startMovement!(Name, CommunicationList)→
                                   exit!(Name,Parent)→
                                   finishMovement!(Name,CommunicationList) →
                                   changeLocation?(Name, NewLocation)→
                                   startMovement!(Name, CommunicationList)→
                                   enter!(Name, NewAmbient)→
                                   finishMovement!(Name,CommunicationList) →
                                   changeLocation?(Name, NewLocation);

     Protocol
      PURDIST:= begin→ PURDIST1;
      PURDIST1:=  (exiting(Name, Parent)→
                  changeLocation?(Name, NewLocation)→
                  entering(Name,NewAmbient)→
                  changeLocation?(Name, NewLocation))→
                  moveAgent.move!(NewAmbient))
                  +
                  end;

   End_Distribution Aspect PurDist
```

**Tabla 36: Aspecto de Distribución PurDist**

### Aspecto Funcional PurchFunct

El Aspecto Funcional del `Purchaser` tiene información sobre el cliente, su nombre, su *password* y su crédito. Implementa un servicio para pagar por un producto (`pay`), una vez ha ganado la subasta. Este servicio es invocado desde

una transacción (purchase), según la cual, primero utiliza la información que tiene del usuario para acreditarse (registerForPayment) y posteriormente paga el precio.

```
Functional Aspect PurchFunct using ICollPur, IPurAuct

 Attributes
  Variables
  userName: string NOT NULL;
  password: string NOT NULL;
  credit: real NOT NULL;

 Services
    begin(input UserName: string, input Password: string,
                                  input Credit: real)
     Valuations
      [begin (UserName, ProductDescrip)]
                  userName:= UserName & password:= Password &
                                        credit:= Credit;
  out registerForPayment( input UserName: string, input Password: string,
                          input ProductID: string, output Register:bool);
  in pay(input Quantity)
    Valuations
      [pay (Quantity)] credit:= credit-Quantity;
  end;

 Played_Roles
  COLLPUR for ICollPur ::= purchase?(input ProductID, input Quantity);
  PURAUC for IPurAuct ::=  registerForPayment!( userName, password,
                                                ProductID, Register);

 Transactions in purchase(input ProductID, input Quantity);
   PAY::= registerForPayment!( userName, password, ProductID, Register)
          →PAY1;
   PAY1::= {Register==true}pay(input Quantity);

 Protocol
  PurchFunct ::= begin→ PurchFunct1;
  PurchFunct1 ::= COLLPUR.purchase?(input ProductID, input Quantity) + end;


End_Functional Aspect PurchFunct
```

**Tabla 37: Aspecto Funcional PurchFunct**

### Componente Purchaser

El componente Purchaser importa un aspecto de distribución y un aspecto funcional. Así pues incorpora una serie de puertos para comunicarse con el Collector, con Auction, con los puertos ICapability y IMobility del AgentAP. Ademas, define dos *weavings* para indicar desde el aspecto funcional cuando es necesario la salida del ambiente AgentAP y la entrada a el.

```
Component_type Purchaser

 Import Distribution Aspect PurDist;
 Import Functional Aspect PurchFunct;
```

```
    Ports
        PURAUCPort: IPurAuct Played_Role PurchFunct.PURAUC;
        COLLPURPort: ICollPur Played_Role PurchFunct.COLLPUR;
        MoveAgentPort: IMobility Played_Role PurDist.moveAgent;
        CapParentPort: ICapability Played_Role PurDist.CapParent;
    End_Ports

    Weavings
     PurchDist.exiting(Name,Parent) before PurchFunct.purchase(input ProductID,
   input Quantity);
     PurDist.entering(Name, NewAmbient) after
                                  PurchFunct.pay(ProductID,Quantity);

End Component_type Purchaser;
```

**Tabla 38: Componente Purchaser**

## o Auction

Interfaces, aspectos y la especificación del componente Auction, que representa a la subasta electrónica.

### Interface IAuct

La siguiente interfaz será importada por los aspectos del componente `Auction`:

```
Interface IAuct
 bid(input ProductID: string, input Quantity: real);
 lookforProduct(input Description: string, output ProductID: string ,
                                           output CurrentBid:  real);
 notifyBuy(Buy : bool);
 registerForPayment( input UserName: string, input Password: string,
                     input ProductID: string, output Register: bool);
End_Interface IAuct
```

**Tabla 39: Interface IAuct**

### Aspecto de Distribución Dist

El aspecto de distribución del Auction es el mismo que el del componente Collector (ver Tabla 32). Este es un claro ejemplo en que un mismo aspecto de distribución puede ser reutilizable en diferentes elementos arquitectónicos.

### Aspecto Funcional AuctFunct

El Aspecto Funcional del `Auction` guarda información sobre el producto y el estado actual de su subasta. Además proporciona servicios para la puja (`bid`), la búsqueda del producto (`lookForProduct`) y para el registro del cliente para proceder con el pago (`registerForPayment`).

```
Functional Aspect AuctFunct using IAuct
```

```
Attributes
 Variables
  productID: string NOT NULL;
  productDescription: string NOT NULL;
  currentBid: string NOT NULL;
  username: string NOT NULL;
  password: string NOT NULL;

Services
 begin;
 in registerForPayment( input UserName, input Password,
                        input ProductID, output Register)
  Valuations
    {username==UserName & password==password & productID==ProductID}
        [registerForPayment(UserName, Password, ProductID, Register)]
             Register= true ;
 in bid(input ProductID: string, input Quantity: real)
  Valuations
    {ProductID== productID}
     [bid(input ProductID, input Quantity)]
                currentBid:= Quantity;
 in lookforProduct(input Description:string, output ProductID:string,
                   output CurrentBid:real)
   Valuations
      [lookforProduct(Description, ProductID, CurrentBid)]
          ProductID==productID & CurrentBid== currentBid;
 out notifyBuy(input Buy : bool)
   Valuations
     [notifyBuy(Buy)] Buy== true;
 end;

Played_Roles
 AUCT for IAuct ::= lookforProduct!( Description, ProductID, CurrentBid)
                    →
                    bid?( ProductID, Quantity)
                    →
                    notifyBuy!( Buy)
                    →
                    registerForPayment?( UserName,
                                         Password,
                                         ProductID,
                                         Register);

Protocol
 AUCTFunct ::= begin→ AUCTFunct1;
 AUCTFunct1 ::= lookforProduct!( Description,
                                 ProductID,
                                 CurrentBid)
                →
                bid?( ProductID, Quantity)
                →
                notifyBuy!( Buy)
                →
                registerForPayment?( UserName,
                                     Password,
                                     ProductID,
                                     Register) + end;

End_Functional Aspect AuctFunct
```

**Tabla 40: Aspecto Funcional AuctFunct**

**Componente Auction**

El componente `Auction` importa un aspecto de distribución (el mismo que el componente `Collector`) y un aspecto funcional. Incorpora un puerto mediante el cual se comunicará con el resto de elementos arquitectónicos.

```
Component_type Auction
 Import Distribution Aspect Dist;
 Import Functional Aspect AucFunct;
 Ports
     AuctionPort: IAuct Played_Role AuctFunct.AUCT;
 End_Ports
End Component_type Auction;
```

**Tabla 41: Componente Auction**

#### *5.3.2.2. Conectores*

De forma similar a como se ha hecho para los Componentas, a continuación se lista las especificaciones relativas a los conectores.Todos los conectores importan el mismo aspecto de distribución que los componentes *Auction* y *Collector*.

### o AgentCnctr

Aspecto y especificación del conector que coordina la comunicación entre el *Purchaser* y el *Collector*.

**Aspecto de Coordinación AgentCnctrCoordinator**

El Aspecto de Coordinación en este componente actúa como un repetidor de mensajes, redirigiendo mensajes entrantes por un *played role* a mensajes salientes por otro.

```
Coordination Aspect AgentCnctrCoor using ICollPur

  Services
    begin;
    in/out purchase(input ProductID: string, input Quantity: real);
    end;

  Played_Role
    COLLPUR for ICollPur =
    (
      ( purchase?( ProductID,Quantity )
       →
       ( purchase?( ProductID,Quantity ) )
    );
      PURCOLL for ICollPur =
    (
      ( purchase?( ProductID,Quantity )
       →
       ( purchase?( ProductID,Quantity ) )
```

```
    );

  Protocol
    AGENTCNCTRCOOR = begin.COORD;
    COORD =
    (
      ( COLLPUR.purchase?( ProductID,Quantity ) →
        PURCOLL.purchase!( ProductID,Quantity ) ).COOR
    +
    ( PURCOLL.purchase?( ProductID,Quantity ) →
        COLLPUR.purchase!( ProductID,Quantity ) ).COOR
    +
        end
    );

End_Coordination Aspect AgentCnctrCoor;
```

**Tabla 42: Aspecto de Coordinación AgentCnctrCoordinator**

### Conector AgentCnctr

El conector `AgentCnctr` importa un aspecto de distribución y un aspecto de coordinación. Incorpora dos puertos para comunicar al `Collector` y al `Purchaser`.

```
Connector_type AgentCnctr
 Import Distribution Aspect Dist;
 Import Coordination Aspect AgentCnctrCoor;
 Ports
     CollectorPort: ICollPur Played_Role AgentConctrCoor.COLLPUR;
     PurchaserPort: ICollPur Played_Role AgentConctrCoor.PURCOLL;
 End_Ports
End Connector_type AgentCnctr;
```

**Tabla 43: Conector AgentCnctr**

## o AgentCustCnctr

Aspecto y especificación del conector que coordina la comunicación entre el *Customer* y el *AgentsAP*.

### Aspecto de Coordinación AgentCustCnctr

El Aspecto de Coordinación en este componente, de nuevo actúa como un repetidor de mensajes, redirigiendo mensajes entrantes por un *played role* a mensajes salientes por otro.

```
Coordination Aspect AgentCustCnctrCoor using ICustAgent, IMobility

  Services
    begin;
      in/out purchase(changeProductDescription(input ProductDescr: string);
      in/out changeMaxBidQuantity(input NewMaxBidQuantity: real);
      in/out move(NewAmbient: string);
    end;
```

```
Played_Role
  CUSTAGENT for ICustAgent =
  (
    (changeProductDescription?( ProductDescr)
     →
     changeProductDescription!( ProductDescr) )
     +
     (changeMaxBidQuantity?( NewMaxBidQuantity)
     →
     changeMaxBidQuantity!( NewMaxBidQuantity) )
  );
   AGENTCUST for ICustAgent =
  (
    (changeProductDescription?( ProductDescr)
     →
     changeProductDescription!( ProductDescr) )
     +
     (changeMaxBidQuantity?( NewMaxBidQuantity)
     →
     changeMaxBidQuantity!( NewMaxBidQuantity) )
  );
       AGENTCUSTMOB for IMobility =
  (
    move?( NewAmbient)
     →
     move!( NewAmbient)
  );
   CUSTAGENTMOB for IMobility =
  (
    move?( NewAmbient)
     →
     move!( NewAmbient)
  );

Protocol
  AGENTCUSTCNCTRCOOR = begin.COORD;
  COORD =
  (
    (CUSTAGENT.changeProductDescription?( ProductDescr ) →
      AGENTCUST.changeProductDescription!( ProductDescr ) ).COOR
  +
     (AGENTCUST.changeProductDescription?( ProductDescr ) →
      CUSTAGENT.changeProductDescription!( ProductDescr ) ).COOR
  +
     (AGENTCUSTMOB.move?( NewAmbient) →
      CUSTAGENTMOB.move!( NewAmbient) ).COOR
  +
      (CUSTAGENTMOB.move?( NewAmbient) →
      AGENTCUSTMOB.move!( NewAmbient) ).COOR
   +
     end
  );

End_Coordination Aspect AgentCustCnctrCoor;
```

**Tabla 44: Aspecto de Coordinación AgentCustCnctr**

### Conector AgentCustCnctr

El conector `AgentCustCnctr` importa un aspecto de distribución y un aspecto de coordinación. Incorpora cuatro puertos para satisfacer la comunicación entre `Customer` y `AgentsAP`.

```
Connector_type AgentCustCnctr

 Import Distribution Aspect Dist;
 Import Coordination Aspect AgentCustCnctrCoor;

 Ports
     CustAgentPort: IMobility, Played_Role AgentCustCnctrCoor.CustAgent;
     AgentCustPort: IMobility, Played_Role AgentCustCnctrCoor.Agentcust;
     CustAgentMobPort: IMobility, Played_Role
                 AgentCustCnctrCoor.CustAgentMob;
     AgentCustMobPort: IMobility, Played_Role
                 AgentCustCnctrCoor.AgentCustMob;
End_Ports

End Connector_type AgentCustCnctr;
```

**Tabla 45: Conector AgentCustCnctr**

### ○ AuctionCnctr

Aspecto y especificación del conector que coordina la comunicación entre el *Auction* y el *Customer*.

#### Aspecto de Coordinación AuctionCnctrCoor

De forma similar a los casos anteriores, el Aspecto de Coordinación en este componente, actúa como un repetidor de mensajes, redirigiendo mensajes entrantes por un *played role* a mensajes salientes por otro.

```
Coordination Aspect AuctionCnctrCoor using IAuct

 Services
  begin;
      in/out bid(input ProductID: string, input Quantity: real);
      in/out lookforProduct(input Description: string,
                            output ProductID: string ,
                            output CurrentBid:  real);
      in/out notifyBuy(Buy : bool);
      in/out registerForPayment( input UserName: string,
                                 input Password: string,
                                 input ProductID: string,
                                 output Register: bool);
  end;

 Played_Role
  AUCTCUST for IAuct =
  (
    (bid?( ProductID,Quantity )
     →
     (bid!( ProductID,Quantity ) )
     +
    (lookforProduct?( Description, ProductID, CurrentBid )
     →
     (lookforProduct!( Description, ProductID, CurrentBid) )
     +
     (notifyBuy?( Buy )
     →
     (notifyBuy!( Buy ) )
     +
```

```
        (registerForPayment?( UserName, Password, ProductID, Register)
        →
        (registerForPayment!( UserName, Password, ProductID, Register) )
    );

        CUSTAUCT for IAuct =
    (
      (bid?( ProductID,Quantity )
       →
       (bid!( ProductID,Quantity ) )
       +
       (notifyBuy?( Buy )
       →
       (notifyBuy!( Buy ) )
    );


  Protocol
    AUCTIONCNCTRCOOR = begin.COORD;
    COORD =
    (
      (AUCTCUST.bid?( ProductID,Quantity ) →
       CUSTAUCT.bid!( ProductID,Quantity ) ).COOR
    +
      (AUCTCUST.lookforProduct?( Description, ProductID, CurrentBid ) →
       CUSTAUCT.lookforProduct!( Description, ProductID, CurrentBid)).COOR
     +
       (AUCTCUST.notifyBuy?( Buy ) )→
       CUSTAUCT.notifyBuy!( Buy )).COOR
     +
       (AUCTCUST.registerForPayment?( UserName, Password, ProductID,
Register) →
       CUSTAUCT.registerForPayment!( UserName, Password, ProductID, Register)
).COOR
     +
       (CUSTAUCT.bid?( ProductID,Quantity ) )→
       AUCTCUST.bid!( ProductID,Quantity )  ).COOR
     +
       (CUSTAUCT.notifyBuy?( Buy ) )→
       AUCTCUST.notifyBuy!( Buy ) ).COOR
    +
       end
    );

End_Coordination Aspect AuctionCnctrCoor;
```

**Tabla 46: Aspecto de Coordinación AuctionCnctrCoor**

### AuctionCnctr Connector

El conector `AuctionCnctr` importa un aspecto de distribución y un aspecto de coordinación. Mediante los puertos que incorpora conecta al `Auction` con el resto de elementos arquitectónicos a través de su interfaz `IAuct`.

```
Connector_type AuctionCnctr

 Import Distribution Aspect Dist;
 Import Coordination Aspect AuctionCnctrCoor;

 Ports
     AuctionPortCUST: IAuction, Played_Role AuctionCnctrCoor.CustAuct;
     AuctionPortAUCT: IAuction, Played_Role AuctionCnctrCoor.AuctCust;
 End_Ports
```

```
End Connector_type AuctionCnctr;
```

**Tabla 47: AuctionCnctr Connector**

#### *5.3.2.3. Ambientes*

A continuación se listan las especificaciones en LDA-OA para los ambientes del caso de estudio.

- **AgentAP**

Ambiente *Process* que ubica al *Collector* y al *Purchaser*, y es el que se mueve a través de la red para realizar sus tareas en el Site de la subasta, para volver al *Site* del cliente con los resultados.

**Aspecto de Distribución AAPDist**

El Aspecto de Distribución del `AgentAP` (`AAPDist`) se moverá cuando el `Customer` y el `Purchaser` así lo soliciten. Por este motivo, el aspecto de distribución tiene dos played_roles de la interfaz `IMobility`. Un *played_role* para recibir la solicitud del `Customer` y la otra del `Purchaser`. El movimiento viene definido por salir del ambiente padre y entrar al ambiente que le indiquen.

```
Distribution Aspect AAPDist using IMobility, ICapability

 Attributes
  location : string NOT NULL;

 Services
   in changeLocation(Name: String, NewLocation:loc)
        Valuations
          [changeLocation()] location:=NewLocation;
   getLocation( output Location:loc)
        Valuations
          [getLocation()] Location:=location;
   out startMovement(input Name:string,
                     output CommunicationList[]: Attachment)
     Valuations
          [startMovement(Name, CommunicationList[])] Name:= self.Name;
   out exit (Name: string, Parent:string )
     Valuations
          [exit(Name, Parent)] Name:= self.Name & Parent:=location;

   out finishMovement(input Name, input CommunicationList[]);
     Valuations
          [finishMovement(Name, CommunicationList[])] Name:= self.Name;

   out enter (Name: string, NewAmbient: string);
     Valuations
          [exit(Name, NewAmbient)] Name:= self.Name

  Transactions in move(NewAmbient:string)
      move::= out startMovement(input Name, output CommunicationList[])
```

```
                → MOVE1;
        MOVE1::= out exit(Name,Parent)→ MOVE2;
        MOVE2::= out enter(Name, NewAmbient)→MOVE3;
        MOVE3::= out finishMovement(input Name, input CommunicationList[]);


     Preconditions
         in  changeLocation(Name, NewLocation)
                                  if {self.Name==Name};

     Played_Roles
       moveCust for IMobility ::= move?(NewAmbient);
       movePurch for IMobility ::= move?(NewAmbient);
       CapParent for ICapability ::=startMovement!(Name, CommunicationList)→
                                  exit!(Name,Parent)→
                                  enter!(Name, NewAmbient)→
                                  finishMovement!(Name,CommunicationList)→
                                  changeLocation?(Name, NewLocation);

   Protocol
    AAPDIST:= begin→ AAPDIST1;
    AAPDIST1:= getLocation(Location)+ (move?(NewAmbient)→
               CapParent.changeLocation?(Name, NewLocation))
               + end;

End_Distribution Aspect AAPDist
```

**Tabla 48: Aspecto de Distribución AAPDist**

### AmbientProcess AgentAP

El `AmbientProcess AgentAP` importa un aspecto de movilidad, un aspecto de distribución y un aspecto de coordinación. Nótese como los aspectos de Coordinación (`ACoordination`) y Movilidad (`Mobile`) no han sido definidos en la presente sección, puesto que son aspectos predefinidos que no se pueden modificar en la especificación. Además, el ambiente incorpora una serie de puertos, algunos de ellos predefinidos (`InCapabilitiesPort, ECapabilitiesPort, EServicesPort, InServicesPort`) y otros añadidos para interactuar con los nuevos aspectos. También se definen los *weavings* asociados a los aspectos importados.

```
Ambient_Process type AgentAP

 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect AAPDist;

 Ports
     DCapabilitiesPort: ICapability, Played_Role AAPDist.CapParent;
     EMobilityPort: IMobility Played_Role AAPDist.moveCust;
     InMobilityPort: IMobility Played_Role AAPDist.movePurch;
     InCapabilitiesPort: ICapability Played_Role Mobile.Parent;
     ECapabilitiesPort: ICapability Played_Role Mobile.Child;
     EServicesPort: ICall Played_Role ACoordination.Client;
     InServicesPort: ICall Played_Role ACoordination.Server ;
 End_Ports

Weavings
     APPDist.getLocation(Location) instead
                                Mobile.getParent(Parent);
```

```
End Ambient_Process type AgentAP;
```

**Tabla 49: AmbientProcess AgentAP**

### o HostSite

Ambiente *Site* que representa una localización física, una máquina en la que se pueden ejecutar los elementos arquitectónicos.

#### Aspecto de Distribución HostDist

El aspecto de distribución del ambiente *Site* ha de tener el atributo physicallocation que representa al dispositivo físico al que representa mediante una instancia del tipo LOC:

```
Distribution Aspect HostDist using IGetLocation

Attributes
  Constant
  location : string NOT NULL;
  physicalLocation: LOC NOT NULL;

Services
  begin(input ParentAmbient: string, input PhysicalLocation: LOC)
     Valuations
      [begin (ParentAmbient,PhysicalLocation)]
                          location := ParentAmbient,
                          physicalLocation:=PhysicalLocation;
  in getLocation( output Location:string)
     Valuations
         [getLocation(output Location)] Location := location;
  end;

Protocol
  DIST:= begin→ DIST1;
  DIST1:= getLocation(Location)+ end;

End_Distribution Aspect HostDist;
```

**Tabla 50: Aspecto de Distribución HostDist**

#### Ambiente Site HostSite

El ambiente ClientSite de nuevo importa los aspectos predefinidos y un aspecto de distribución. Incorpora los puertos predefinidos de los ambientes y define los *weavings* entre los aspectos.

```
Ambient_Site type HostSite

 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect HostDist;

 Ports
```

```
        InCapabilitiesPort: ICapability Played_Role Mobile.Parent;
        ECapabilitiesPort: ICapability Played_Role Mobile.Child;
        EServicesPort: ICall Played_Role ACoordination.Client;
        InServicesPort: ICall Played_Role ACoordination.Server ;
    End_Ports

    Weavings
        ADist.getLocation(Location) instead
                                  Mobile.getParent(Parent);

    End Ambient_Site type ClientSite;
```

**Tabla 51: Ambiente Site HostSite**

- **Root**

Ambiente *Logical* que representa el entrono de ejecución, es decir, el conjunto de máquinas entre als que tiene lugar la interacción.

#### Aspecto de Distribución RootDist

El ambiente *Logical Root* es el que está situado en la parte más alta de la jerarquía, por tanto el mismo no está ubicado en ningún ambiente y esto se ha de reflejar en su campo `location`. Se ha decidido que en este caso el campo `location` tiene un valor nulo. Pese a todo, el aspecto sigue implementando la interfaz `IGetLocation` para seguir conforme a las restricciones del modelo. No obstante, esta petición no debería hacerse nunca, ya que al ser él la parte más alta de la jerarquía nunca se va a requerir el valor de su padre para solicitarle un `accept`.

```
Distribution Aspect RootDist using IGetLocation

Attributes
  Constant
  location : string;

Services
  begin()
     Valuations
      [begin ()]
                           location := null,
   in getLocation( output Location:string)
     Valuations
         [getLocation(output Location)] Location := location;
  end;

Protocol
  DIST:= begin→ DIST1;
  DIST1:= getLocation(Location)+ end;

End_Distribution Aspect RootDist
```

**Tabla 52: Aspecto de Distribución RootDist**

**Ambiente Logical Root**

El ambiente `Root` de nuevo importa los aspectos predefinidos y un aspecto de distribución. Incorpora los puertos predefinidos de los ambientes y define los *weavings* entre los aspectos.

```
Ambient_Logical type Root

 Import Mobility Aspect Mobile;
 Import Coordination Aspect ACoordination;
 Import Distribution Aspect ADist;

 Ports
     InCapabilitiesPort: ICapability Played_Role Mobile.Parent;
     ECapabilitiesPort: ICapability Played_Role Mobile.Child;
     EServicesPort: ICall Played_Role ACoordination.Client;
     InServicesPort: ICall Played_Role ACoordination.Server ;
 End_Ports

 Weavings
     Dist.getLocation(Location) instead
                                Mobile.getParent(Parent);

End Ambient_Logical type Root;
```

**Tabla 53: Ambiente Logical Root**

### 5.3.3. Configuración

En la sección anterior se han definido todos los tipos necesarios para el modelo. Ya sólo quedaría instanciar dichos tipos para lograr la configuración mostrada en la Figura 70. Esto, como ya se ha visto, se realizará en el nivel de configuración, instanciando un nuevo modelo.

La instanciación debe seguir el siguiente orden: Inicialmente se instancian los ambientes, después el resto de elementos arquitectónicos, indicando, en cada caso, en que ambiente se ejecutarán. Finalmente se crearían los canales de comunicación para interconectar a los elementos arquitectónicos mediante *attachments*.

```
Architectural_Model_Configuration AuctionConf =

New MobileAuction
{
       IP1 = new LOC(ip.del.host.1);
       IP2 = new LOC(ip.del.host.2)

       ROOT = new Root() ;

       CLIENTSITE = new HostSite(ROOT, IP1);
       AUCTIONSITE = new HostSite(ROOT, IP2)

       AAP = new AgentAP(CLIENTSITE);

       CUSTOMER = new Customer(CLIENTSITE);
```

```
            ACCNCTR = new AgentCustCnctr(CLIENTSITE);
            COLL = new Collector(50.0,'MP3 player',AAP);
            PUR = new Purchaser('Juan','4fxvt',100.0,AAP);
            ACNCTR = new AgentCnctr(AAP);
            AUCT = new Auction(AUCTIONSITE)
            AUCNCTR = new AuctionCnctr(AUCTIONSITE);

            ATT1 = new CustCnctrAtt(CUST, AgentsPort,
                                    CustAgentPort, ACNCTR);
            ATT2 = new CustCnctrAttM(CUST, MovePort,
                                    AccountOp_port, ACNCTR);
            ATT3 = new CustAuctCnctrAtt(CUST, AuctionPort,
                                    CustAgentMobPort, ACNCTR);
            ATT4 = new CnctrAgntsAtt(ACCNCTR, AgentCustPort,
                                    EServicesPort, AAP);
            ATT5 = new CnctrAgntsAttM(ACCNCTR, AgentCustMobPort,
                                    EMobilityPort, AAP);
            ATT6 = new AgntsHostAtt(AAP, DCapabilitiesPort,
                                    ICapabilitiesPort, CLIENTSITE);
            ATT7 = new PurAAPAtt(PUR, MoveAgentPort,
                                    InMovilityPort, AAP);
            ATT8 = new PurAucAtt(PUR, PURAUCPort,
                                    AuctionPortCUST, AUCNCTR);
            ATT9 = new PurCnctrAtt(PUR, COLLPURPort,
                                    PurchaserPort, ACNCTR);
            ATT10 = new ColCnctrAtt(COL, CollPurPort,
                                    CollectorPort, ACNCTR);
            ATT11 = new ColAuctAtt(COL, COLLAUCTPort,
                                    AuctionPortCUST, AUCNCTR);
            ATT12 = new ColCustAtt(COL, CustAgentPort,
                                    CustAgentPort, ACCNCTR);
            ATT13 = new AuctCnctrAtt(AUCT, AuctionPort,
                                    AuctionPortAUCT, ACNCTR);

    };
```

**Tabla 54: Configuración de Mobile Action**

Como se puede observar en la instanciación de los *attachments*, no se han creado los *attachments* con los que los componentes móviles se comunican con el ambiente padre para solicitar las *capabilities*. Esto se hará automáticamente, ya que los servicios de la interfaz `ICapability` sólo pueden invocarse al ambiente en el que están ubicados los elementos arquitectónicos. Por ello, cuando se detecta un puerto con la interfaz `ICapability` en el modelo, automáticamente se conecta al puerto con interfaz `ICapability` y *played role* `INTERNO` del Ambiente Padre.

CAPÍTULO VI

# MIDDLEWARE PARA AMBIENT-PRISMA

## Contenidos del capítulo

# MIDDLEWARE PARA AMBIENT-PRISMA

En el presente capítulo se presentarán los cambios que se han añadido al *middleware* PRISMANET para incorporar los conceptos del Ambient-PRISMA. Como resultado de la aplicación de estos cambios se obtendrá una nueva versión del middleware que proporcionará un entorno de ejecución para las arquitecturas modeladas con Ambient-PRISMA. Esta nueva versión del *middleware* se llamará Ambient- PRISMANET.

## 6.1. Incorporación de nuevos tipos

El primer paso para la incorporación de los conceptos del Ambient-PRISMA al *middleware* es añadir soporte para los nuevos tipos. La principal aportación del nuevo metamodelo es el concepto de ambiente y, en concreto, se definen tres tipos de ambientes: ambiente *Site*, ambiente *Process* y ambiente *Logical*. Las diferencias entre los tres tipos ya han sido explicadas en el capítulo anterior, así que el presente se centrará en como se han transformado dichos conceptos al *middleware*.

Los ambientes son nuevos elementos arquitectónicos con los que el usuario puede construir arquitecturas distribuidas, en concreto y como ya se ha comentado, los ambientes serán los encargados de modelar la ubicación de los elementos que contengan. Al tratarse de elementos arquitectónicos PRISMA, los ambientes comparten las características del resto de elementos arquitectónicos. Así pues, los ambientes importan aspectos, como mínimo los aspectos de Movilidad, Coordinación y Distribución, que son obligatorios para todos los ambientes, tienen puertos por donde se comunican (como mínimo tienen cuatro puertos, para ofrecer los servicios de cada uno de los aspectos a los elementos a los que ubican y al resto de elementos) y definen *weavings* entre los aspectos. Además de todo esto, los ambientes también guardan información sobre los elementos arquitectónicos que albergan en su interior.

Siguiendo la estructura de anteriores secciones, para explicar la incorporación de los nuevos elementos arquitectónicos al *middleware,* se partirá por explicar como se han incorporado sus aspectos predefinidos, para, posteriormente explicar como se incorpora el elemento arquitectónico y como se importan dichos aspectos para dar semántica al elemento en cuestión.

### 6.1.1. Aspectos predefinidos de los ambientes

Como se ha comentado en el capítulo 5, todos los ambientes tienen dos aspectos predefinidos mediante los que reciben semántica y proporcionan servicios al resto de elementos arquitectónicos, tanto a los que proporcionan localización como a los que se encuentran en el exterior. En concreto, se trata del Aspecto de Coordinación del Ambiente (`ACoordinationAspect`) y del Aspecto de Movilidad (`MobileAspect`).

#### *6.1.1.1. El Aspecto de Coordinación del Ambiente*

El Aspecto de Coordinación del Ambiente es el encargado de proporcionar una vía de comunicación entre los elementos arquitectónicos que se ubican en el ambiente y el exterior. El ambiente tiene dos puertos predefinidos que están relacionados con este aspecto: Uno para conectar los elementos ubicados en el ambiente y otro para los elementos externos. La labor de coordinación del aspecto se reduce a reenviar las peticiones que le llegan por un puerto al otro y viceversa, de esa forma establece un canal de comunicación a través de los límites del ambiente.

Para diferenciar el papel que ha de desempeñar en función por donde le lleguen las peticiones, se han definido dos *played roles*:

- `INTERNO`: Será el *played role* que desempeñará para los elementos ubicados en el ambiente. Desde el punto de vista de los elementos arquitectónicos ubicados en su interior, el ambiente ejerce un papel servidor en la ejecución de sus peticiones.

- `EXTERNO`: Será el *played role* que desempeñará para los elementos arquitectónicos ubicados en el exterior del ambiente. Para los elementos arquitectónicos ubicados en su interior, el ambiente actúa como cliente, pues a través de ese *played role* les reenvía las peticiones que lleguen desde el exterior del ambiente.

No obstante, este aspecto es un tanto especial. Al tratarse de un aspecto predefinido, no tiene información sobre qué interfaces ha de coordinar. Por otro lado, como el conjunto de elementos arquitectónicos ubicados en el interior de un ambiente puede variar dinámicamente, el definir una interfaz concreta ha de estar sujeto a los cambios que en el ambiente se produzcan, es decir, debería tratarse de una interfaz dinámica.

La implementación de una interfaz dinámica en el aspecto, ya sea obtenida a partir de los *attachments* conectados a los puertos relacionados al aspecto o desde la lista de componentes, y que sea actualizada en función de los movimientos acaecidos en el ambiente resulta difícil de implementar. Hay que tener en cuenta que todo cambio aplicado a la interfaz de un aspecto ha de

actualizar los *weavings* asociados con esa interfaz, con lo cual el proceso puede llegar a ser complicado y costoso.

En lugar de eso, se ha optado por definir una interfaz genérica que será la encargada de pasar mensajes a través del ambiente. Esta interfaz encapsulará las peticiones dentro de un método de invocación genérico (`Call()`) que será con lo que trabajará el aspecto de coordinación. De esa forma, si se tuviese que crear un *weaving* para controlar el tráfico de entrada y salida del ambiente, tan solo habría que definirlo con el método `Call()`.

De esa forma, todas las peticiones que lleguen al Aspecto de Coordinación han de encapsularse en una llamada `Call()` y de la misma forma, todas las peticiones que salgan del aspecto, han de extraerse de una llamada `Call()` y transformarse de nuevo en la llamada original. Esto es posible gracias a una modificación en los puertos predefinidos del ambiente, pero de ello ya se hablará cuando le llegue el turno al elemento arquitectónico, de momento se centrará en los aspectos.

Para la encapsulación de peticiones, el servicio `Call` tiene la signatura que se muestra en la Tabla 55.

```
public interface ICall
        {
                AsyncResult Call(string methodName, object[] args);
        }
```

**Tabla 55: Interface ICall en C#**

El Aspecto de Coordinación del Ambiente implementa la interfaz `ICall` (ver Figura 71) en la cual se define el servicio `Call()`. El servicio proporcionado por el Aspecto de Coordinación del Ambiente, encapsula el nombre del método cuya llamada representa mediante una cadena de caracteres y el conjunto de sus parámetros en un *array* de `objects`. Para obtener de nuevo la llamada original, es decir cuando la petición vaya a ser encolada en los puertos de salida, tan solo habrá que extraer el nombre del servicio y el conjunto de parámetros para llamar al método de encolado.

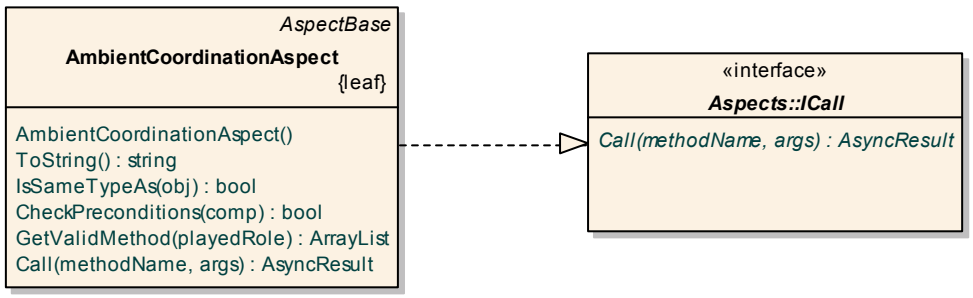



**Figura 71: AmbientCoordinationAspect y Interfaz ICall**

Como se ha comentado, el `AmbientCoordinationAspect` es un aspecto predefinido. Este tipo de aspecto requiere una implementación especial. Un aspecto normal es definido por el usuario, para lo cual hereda de la clase `AspectBase` y define todos los servicios que definan la semántica de ese aspecto [Cos05]. Estos servicios pueden estar relacionados con una interfaz o no. Pues bien, un aspecto predefinido es un aspecto que define una semántica inherente al modelo, de forma que dicha semántica es totalmente transparente a las necesidades del usuario, y por ello no la puede modificar.

Así `AmbientCoordinationAspect` es una clase que hereda de `AspectBase`, como lo haría cualquier otro aspecto, y está marcada como *sealed*, es decir, no es posible heredar de ella. De esta forma no se le permite al usuario modificar o ampliar la semántica de la clase. Si en un momento dado necesitase de una semántica adicional al proceso de comunicación entre ambientes, debería en todo caso, crear un nuevo aspecto para tal efecto y relacionarlo con `AmbientCoordinationAspect` mediante *weavings*. Es importante remarcar que es un aspecto que refleja un comportamiento interno al modelo.

```
[Serializable]
public sealed class AmbientCoordinationAspect:
                                        AspectBase, IAspectType, ICall
{
      public AmbientCoordinationAspect(): base("AmbientCoordination")
      {

            #region Creación de estructura de subprocesos
            // Creación de la estructura de subprocesos para este
            //  aspecto
            PlayedRoleClass playedRoleICallINTERNO =
                              new PlayedRoleClass("INTERNO");
            playedRoleICallCLIENT.AddMethod("Call",true);

            this.playedRoleList.Add(playedRoleICallINTERNO);

            // Asignación de prioridades para los distintos
            // estados- playedrole-servicios
            AddPriorityService("","EXTERNO","Call",10);

            // Creación de la estructura de subprocesos
            // para este acpecto
                  PlayedRoleClass playedRoleICallEXTERNO =
                                    new PlayedRoleClass("EXTERNO");
            playedRoleICallEXTERNO.AddMethod("Call",true);

            this.playedRoleList.Add(playedRoleICallEXTERNO);

            // Asignación de prioridades para los distintos
            // estados-playedrole-servicios
            AddPriorityService("","EXTERNO","Call",10);
            #endregion

      }
(...)
```

**Tabla 56: Código del aspecto ACoordination siguiendo la plantilla de aspectos**

En el fragmento de código de la Tabla 56 se muestra como se ha implementado el aspecto predefinido siguiendo las plantillas de generación de código PRISMA [Ca05] para la implementación en C# de los aspectos.

Como se puede observar, se añaden los *played roles* (`INTERNO` y `EXTERNO`) y para cada uno de ellos se añade el servicio `Call()` con la prioridad por defecto. Estos *played roles* representarán respectivamente las peticiones que se tengan que redirigir desde los elementos arquitectónicos ubicados en el interior del ambiente hacia fuera y las que vengan de fuera, hacia dentro.

Como se ha visto el `Call()` proporciona una interfaz genérica necesaria para la comunicación entre ambientes, pero su genericidad incorpora una serie de problemas. Los aspectos PRISMA tienen una lista de `ValidOperations` en que se especifican las operaciones que pueden servir. Esta lista es propagada a los `InPorts` de los `Ports` asociados a dicho aspecto para formar la lista `methodCollection`, en la que se encuentran los métodos que sirve el aspecto al que están asociados. Esta lista es consultada por la parte cliente de los *attachments* a través de su método `ImplementInterface` en el momento de la creación para establecer la lista de `validMethods`, es decir, de los métodos cuyas peticiones ha de escuchar, descartando el resto de peticiones. Desde el punto de vista PRISMA, la lista es confeccionada en el momento de la creación por el *attachment* y no se verá modificada de forma dinámica.

Esto es necesario para evitar una sobrecarga de los canales de comunicación, ya que en PRISMA cuando hay varios canales de comunicación conectados a un mismo puerto se realiza una difusión. Esta sobrecarga aún es más notable en Ambient-PRISMA, en el Aspecto de Coordinación del Ambiente, pues al condensar en un mismo aspecto una amplia variedad de interfaces, es posible que peticiones de servicios se propaguen por canales que no llevan a ningún servidor de este servicio. En este punto, el punto de vista es diferente, puesto que en Ambient-PRISMA, los servicios ofrecidos por el aspecto de coordinación son dinámicos, pues elementos arquitectónicos que ofrecen servicios entran y salen del ambiente. Con lo cual, el planteamiento de actualización de la lista `validMethods` de los *attachments* es diferente en PRISMA y en Ambient-PRISMA.

Como se ha comentado, en PRISMA esto se evita obteniendo una lista en tiempo de ejecución de los métodos que ofrece el *played role* del aspecto. Esta lista de métodos es guardada en la lista `methodCollection` del `InPort` previa solicitud al método de `AspectBase.GetValidMehods(PlayedRoleClass PlayedRole)` al cual se le pasa un *played role* para que obtenga sus servicios asociados (ver Figura 72). De esa forma, cada vez que el `InPort` reciba una petición comprueba que el servicio requerido está en su lista de servicios. De no ser así, descarta la petición.

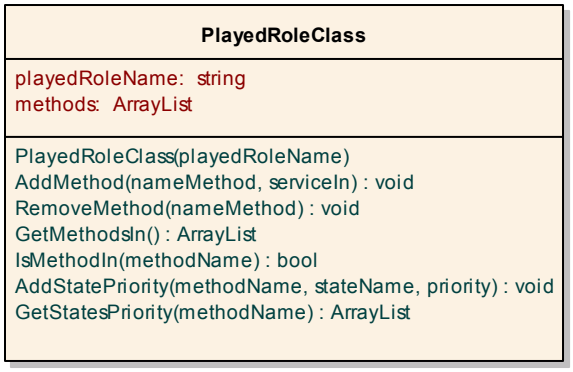



**Figura 72: La clase PlayedRoleClass**

En Ambient-PRISMA, el conjunto de interfaces que tiene el puerto genérico depende de los elementos que albergue en su interior y por tanto, es dinámico. Es necesario modificar la obtención de la lista de métodos válidos en el caso de los Aspectos de Coordinación del Ambiente. En la Figura 71 se muestra la clase perteneciente al aspecto `AmbientCoordinationAspect` en la que se muestra, entre otras cosas, que se ha sobrescrito el comportamiento de `GetValidMethods()`. En el caso de los ambientes, el método `GetValidMethods()` preguntará la interfaz al ambiente que, en función del estado inicial devolverá la lista de interfaces que se pueden acceder en el interior del ambiente.

La idea es que cada vez que se añade algún elemento arquitectónico a un ambiente, ya sea al crear la configuración inicial o tras mover un elemento arquitectónico se actualiza la lista de elementos del ambiente, cosa que desencadenará la actualización de la lista `validMethods` de los *attachments*. Los servicios encargados de actualizar la lista del ambiente, también se encargan de actualizar la lista del `InPort` del puerto para la comunicación distribuida. Cuando el `InPort` actualiza su lista `methodCollection` con los nuevos métodos, avisa a todos los *subscribers* conectados a él de este hecho, para que también ellos actualicen sus listas, pues son estos los que descartan los mensajes pertenecientes a interfaces que no conectan.

Del mismo modo, cuando un elemento es eliminado de un ambiente los servicios que sirve son eliminados de la lista del ambiente, del `InPort` del puerto genérico y, finalmente, de los *subscribers* conectados al él. Para evitar que durante este proceso se eliminen servicios de la interfaz que son servidos por el elemento arquitectónico que sale y por otro elemento arquitectónico que continua en el ambiente, cada servicio de la lista de servicios está representado por su nombre y por el del elemento que lo sirve. De esa forma sólo se eliminarán aquellas entradas pertenecientes a los servicios que servía el elemento arquitectónico que ha salido del ambiente.

En la Tabla 57 se muestra un listado de código con los servicios `AddServices()` y `RemoveServices()` del `InPort`. Estos servicios serán invocados por el ambiente para mantener actualizada la lista de métodos del puerto de comunicación genérica.

```
public void AddServices(ICollection services)
{
        ArrayList add = new ArrayList();
        // update this inport's list
        foreach (MethodStructure service in services)
        {
                if (methodsCollection.Contains(service) == false)
                {
                        methodsCollection.Add(service);
                        add.Add(service.methodName);
                }
        }
        // update subscribers
        foreach (string attName in this.SubscribersList)
        {
                Attachment att =
MiddlewareSystem.GetInstance().ElementManagementLayer.GetAttachment(attName);
                att.Client.AddServices(add);
        }
}

public void RemoveServices (ICollection services)
{
        ArrayList remove = new ArrayList();
        // update this inport's list
        foreach (MethodStructure service in services)
        {
                methodsCollection.Remove(service);
                remove.Add(service.methodName);
        }
        // update subscribers
        foreach (string attName in this.SubscribersList)
        {
                Attachment att =
MiddlewareSystem.GetInstance().ElementManagementLayer.GetAttachment(attName);
                att.Client.RemoveServices(remove);
        }
}
```

**Tabla 57: AddServices() y RemoveServices() del InPort**

#### *6.1.1.2. Aspecto de Movilidad*

El otro aspecto predefinido en los ambientes es el Aspecto de Movilidad. Este aspecto es el encargado de dotar al ambiente de la semántica para gestionar la localización de sus elementos. Puesto que su semántica está tan ligada a los ambientes se restringe su importación a este tipo de elementos arquitectónicos, ya que no tiene sentido que otros elementos arquitectónicos importen esta semántica. Mediante los servicios de este aspecto, el ambiente proporcionará las *capabilities* del Cálculo de Ambientes [Car98] al resto de elementos arquitectónicos. El aspecto implementa la interfaz `ICapability`.

En la presente versión del *middleware* sólo se da soporte para las *capabilities* propias de la movilidad esto es, `Enter()`, `Exit()`, `StartMovement`, `FinishMovement` y `Accept()`. La funcionalidad de estos servicios se explica a continuación:

- `Enter()`: Un elemento arquitectónico solicita al ambiente que va a entrar en un ambiente hijo suyo.

- `Exit()`: Un elemento arquitectónico le indica al ambiente que va a salir de él.

- `Accept()`: Un ambiente le solicita a otro la entrada de un elemento arquitectónico.

- `StartMovement()`: Un ambiente realiza las operaciones previas a la movilidad de un elemento arquitectónico.

- `FinishMovement()`: Un ambiente realiza las operaciones posteriores a la movilidad de un elemento arquitectónico.

Mediante estas operaciones se modela el comportamiento móvil de los elementos arquitectónicos y están implementadas en la clase `MobilityAspect`, de la misma forma que se ha hecho para el Aspecto de Coordinación.

Como ocurría en el caso del Aspecto de Coordinación del Ambiente, el tipo del Aspecto de Movilidad no puede ser heredado para ampliar su funcionalidad, así que, de nuevo, está implementado en una clase *sealed*.

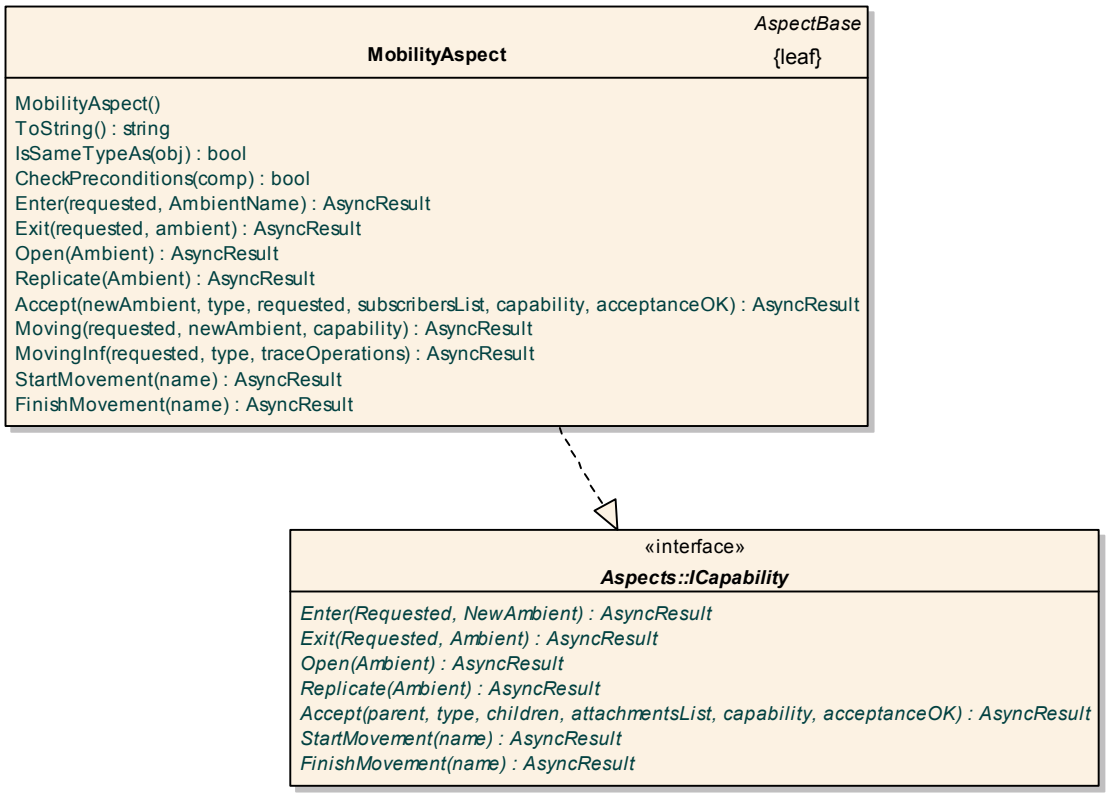



**Figura 73: La clase MobilityAspect y la interfaz ICapability**

Como se puede ver en la Figura 73, el Aspecto de Movilidad predefinido ofrece los servicios de movilidad a los elementos arquitectónicos basados en las *capabilities* del Cálculo de Ambientes. A continuación se explica, paso a paso, como han sido transformados a la implementación cada uno de los servicios del

Aspecto de Movilidad. Para ello hay que tener presente la especificación LDA-OA del `MobilityAspect` que se ha mostrado en el capítulo anterior.

- El servicio `StartMovement(name)` se encarga de gestionar las tareas necesarias para preparar el elemento para mover dado su nombre. Básicamente su función es preparar los canales de comunicación conectados a dicho elemento para la movilidad. Debido a la naturaleza intrínseca del servicio, delega toda su funcionalidad al ambiente al que está enlazado. El funcionamiento interno de este método se comentará más adelante en este capítulo, cuando se hable de los canales de comunicación en Ambient-PRISMA.

- El servicio transaccional `Enter(requested, newAmbient)` comprueba que ambos elementos arquitectónicos están localizados en el ambiente que procesa la *capability*. Para ello se hace uso del enlace al elemento arquitectónico que contiene el aspecto, como se ha comentado un ambiente y se le pregunta directamente si contiene los elementos. Si esto se cumple, se invocará al servicio `Moving()` para que comience el proceso de movilidad. Si no se cumple, una excepción será lanzada. El servicio `Enter()` está implementado según la plantilla de implementación de transacciones presentada en 4.4.2, por lo que el lanzamiento de una excepción en cualquier punto del proceso supondrá abortar todos los cambios realizados.

- El servicio transaccional `Exit(requested, Ambient)` funciona de forma similar al `Enter()`, sólo que haciendo las comprobaciones necesarias para permitir una operación de salida del ambiente. En este caso se trata de comprobar si el elemento arquitectónico que solicita la *capability* es uno de sus hijos, que se hará de forma similar a como se ha hecho para el `Enter()` y comprobará que él es el ambiente al que va dirigida la petición. Esto se hace mediante la consulta del nombre del elemento arquitectónico al que está enlazado, es decir el ambiente al que proporciona la semántica de movilidad. Si las condiciones se cumplen, se obtendrá el destino de la movilidad haciendo uso del servicio `GetParent()` del Aspecto de Distribución, con el que ha de tener un *weaving*. Si no se cumpliesen las condiciones, como pasaba en el caso del `Enter()` se lanzaría una excepción para cancelar el proceso. Una vez obtenido el nombre del padre, es decir, el destino del elemento que quiere salir, se usará para invocar al servicio `Moving()`.

- Ambos métodos (`Exit()` y `Enter()`) invocan al servicio `Moving(requested, newAmbient)` que es el método que realiza el proceso de la movilidad. Por ello sus dos parámetros son el elemento que solicita moverse y el destino de su movilidad. El servicio transaccional `Moving()` solicita al servicio `MovingInf` información sobre el elemento que se va a mover. `MovingInf`, de nuevo, obtendrá esa información preguntándosela al ambiente mediante la

propiedad del aspecto `link`. Una vez obtenida la información, el servicio invoca al `Accept()` del ambiente destino, tras comprobar si dicho ambiente es un padre o un hijo, ya que el *played role* para enviar la petición cambia, siendo `EXTERNO` para el caso del ambiente padre o `INTERNO` para el caso del ambiente hijo. Una vez dicho ambiente le notifique si ha aceptado o no la petición (`AcceptanceOK`), el ambiente realiza un postproceso tras la movilidad del elemento arquitectónico para actualizar los canales de comunicación, o lanza una excepción, en el caso que el ambiente destino no acepte la petición. Las peculiaridades del postproceso comentado se explicarán más adelante en este capítulo, en la sección dedicada a la comunicación distribuida.

- El servicio transaccional `Accept(newAmbient, Type, requested, subscribersList, capability, out acceptanceOK)` es el encargado de recibir al elemento arquitectónico en el ambiente destino. Comprueba que la petición vaya dirigida a él, y de ser así invoca al ambiente al que está enlazado el aspecto para indicarle que agregue el nuevo elemento pasándole la información sobre éste que ha recibido como parámetros. Una vez el ambiente ha incorporado el nuevo elemento arquitectónico que, entre otras cosas, ha creado un *attachment* a su puerto `ICapability`, el aspecto de Movilidad lanza una petición de `ChangeLocation()` hacia el elemento arquitectónico para que éste actualice su aspecto de Distribución con la nueva ubicación.

- Finalmente, el Aspecto de Movilidad implementa el servicio `FinishMovement(name)`, que se encarga, dado un elemento arquitectónico ubicado en el ambiente de realizar las tareas necesarias para que los canales de comunicación relacionados con él se pongan en funcionamiento, dando así por finalizada la movilidad. De nuevo, el funcionamiento interno de este servicio se explicará en la sección de comunicación distribuida, más alante en este capítulo.

Como se puede advertir, el funcionamiento de los servicios ofrecidos por el Aspecto de Movilidad genérico están íntimamente relacionados con el ambiente. Por eso es una restricción que este aspecto sólo se pueda importar a un elemento arquitectónico ambiente.

### 6.1.2. La clase Ambient

En las secciones anteriores se ha mostrado como se han implementado los aspectos genéricos que proporcionan la semántica a los ambientes. A continuación, se explicará como se ha incorporado el concepto de ambiente al *middleware*, haciendo uso de dichos aspectos. Como se explicó en el capítulo anterior, un ambiente es un elemento arquitectónico del metamodelo de Ambient-PRISMA, por lo que sus características arquitectónicas (vista interna

y externa) son similares al resto de elementos arquitectónicos. Es por esa razón por la que se ha optado por introducir la clase `Ambient` como otra clase heredada de la clase que representa a los elementos arquitectónicos, esto es, la clase `ComponentBase`.

No obstante, como se presentó en el capítulo anterior, existen diferentes tipos de ambientes, en función del tipo de localización que pretendan modelar, cada tipo con unas restricciones especiales que lo caracterizan. Estos tipos son ambientes *Site*, *Process* y *Logical*.

Los ambientes *Site*, como se ha explicado, representan un dispositivo físico que proporciona un entorno de ejecución para los elementos arquitectónicos que tiene en su interior. En Ambient-PRISMANET esa descripción coincide con la de *middleware*. Por tanto, todo *middleware* está representado por un ambiente *Site*, que será el intermediario entre los elementos arquitectónicos que se ejecuten en el *middleware* y las capas internas a éste. Por tanto, un ambiente *Site* está ligado a un *middleware* y viceversa. Por esa razón, no tiene sentido que un ambiente *Site* se mueva a otro *middleware*, ya que, a parte que un ambiente *Site* no puede entrar en otro *Site*, en el momento que se instancia, recibe como parámetro una referencia al *middleware* al que representa y este valor ya no se podrá cambiar. Esta referencia se denomina localización física (*physical location*). Lo que no quiere decir que un *Site* no pueda cambiar el valor de su padre, ya que dentro de la jerarquía de ambientes podrá moverse a través de la jerarquía superior, es decir, la de los ambientes *Logical*. Por ello, para los ambientes *Site* se distinguen dos localizaciones: La localización física (*middleware*) y el ambiente en el que se encuentran.

Los ambientes *Process* son entidades cuya finalidad es agrupar elementos arquitectónicos y proporcionar así una jerarquía de ambientes dentro de un ambiente *Site*. Esta agrupación puede darse por diversas razones, por ejemplo para aplicar unas políticas determinadas de seguridad en cuanto a comunicaciones a un determinado conjunto de elementos, o simplemente, para mover de forma conjunta a un grupo determinado de elementos arquitectónicos. A diferencia de los ambientes *Site*, los ambientes *Process* tan solo tienen una referencia al ambiente padre, es decir, el ambiente *Site* en el que se encuentran.

Los ambientes *Logical* son ambientes virtuales que modelan la agrupación de *Sites*, para de esa forma establecer, por un lado el modelado de todo aquello que esté por encima de los *middlewares*, esto es, topologías de redes y subredes entre los *hosts* en que se ejecutan los diferentes *middlewares* que componen el entorno de ejecución. Además, los ambientes *Logical* son los encargados de proporcionar la movilidad real de los elementos arquitectónicos, es decir, aquella que se da entre *middlewares* diferentes, por lo que hace falta ponerse en contacto con la capa de distribución del *middleware* para serializar los elementos en cuestión.

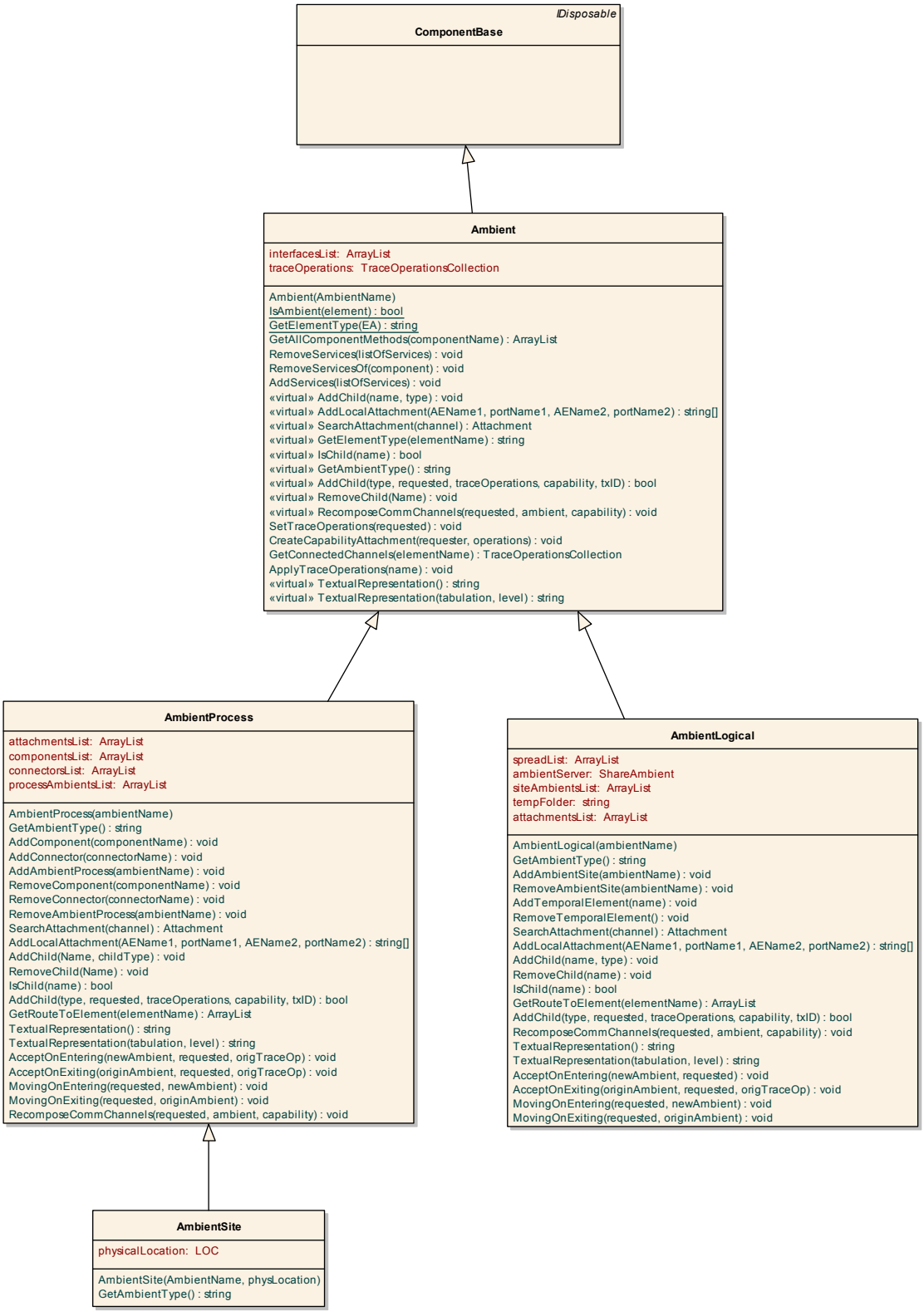



**Figura 74: Diagrama de clases de la clase Ambient**

Como se puede ver, cada tipo de ambiente tiene un propósito determinado, por lo que a la hora de implementarlos es necesario contemplar estas diferencias y restricciones. En la Figura 74 se muestra un diagrama de clases

con las clases necesarias para representar los diferentes tipos de ambientes en el *middleware*.

Como se observa en el diagrama de la Figura 74, se ha optado por separar el comportamiento diferente de los diferentes tipos de ambientes haciendo uso de la herencia. La funcionalidad común a todos los tipos de ambientes se define en la clase `Ambient`. La clase `Ambient`, como todo elemento arquitectónico en PRISMA hereda de `ComponentBase`, clase base de todo elemento arquitectónico que alberga toda la funcionalidad necesaria para la agregación de puertos, aspectos, *weavings*, etc. Al heredar de `ComponentBase`, `Ambient` recibe la funcionalidad necesaria para comportarse como un elemento arquitectónico más.

Además de heredar la funcionalidad necesaria, la clase `Ambient` aporta una funcionalidad característica mediante la cual se diferencia del resto de los elementos arquitectónicos. De esta forma, se definen servicios y campos de clase que dotan a la clase `Ambient` de una entidad propia. Los servicios que se implementan en la clase `Ambient` son comunes a todos los tipos de ambientes. Además, el constructor de la clase `Ambient` incorpora parte de la especificación por defecto del elemento arquitectónico. Esta especificación por defecto importa los aspectos predefinidos y crea los puertos para dichos aspectos. Esto es así porque al ser predefinidos siempre se van a crear. Por tanto, todo ambiente ha de importarlos. En la Tabla 58, se muestra un extracto de código de la clase `Ambient` en la que se muestra como se importan los aspectos predefinidos haciendo uso de `AddAspect()` y como se crean los puertos de entrada y salida del ambiente para cada aspecto.

```
[Serializable]
public class Ambient: ComponentBase
{

(...)

public Ambient(string AmbientName): base(AmbientName)
{

(...)
        /************************************
          •   DEFINICIÓN DE ASPECTOS DEL AMBIENTE *
         ************************************/
        // Creación del aspecto de coordinación del aspecto
        AddAspect(new AmbientCoordinationAspect());
        // Creación del aspecto de movilidad
        AddAspect(new MobilityAspect());

        // Obtenemos referencias a los aspectos
        IAspect ambientCoorAspect =
                        GetAspect(typeof(AmbientCoordinationAspect));
         Iaspect mobAspect =
                        GetAspect(typeof(MobilityAspect));



        /************************************
          •   DEFINICIÓN DE PUERTOS DEL AMBIENTE *
         ************************************/
```

```
            // PUERTOS de ENTRADA
            // Puerto comunicación genérica
             //Comunicación de dentro hacia fuera
            InPorts.Add("IcallPort-InOut", "Icall", "INTERNO");
             //Comunicación de fuera hacia dentro
            InPorts.Add("IcallPort-OutIn", "Icall", "EXTERNO");

             // Puertos para la movilidad
             //Para recibir peticiones del parent
            InPorts.Add("IcapabilityPort-OutIn", "Icapability", "EXTERNO");
            //Para servir capabilities
             InPorts.Add("IcapabilityPort-InOut", "Icapability", "INTERNO");

            // PUERTOS de SALIDA
            // Puerto comunicación generica
            OutPorts.Add("IcallPort-InOut", "Icall", "INTERNO");
            OutPorts.Add("IcallPort-OutIn", "Icall", "EXTERNO");

             // Puerto para la movilidad
             //Para solicitar capabilities al parent
            OutPorts.Add("IcapabilityPort-OutIn", "Icapability", "EXTERNO");
            //Para enviar peticiones a los hijos
            OutPorts.Add("IcapabilityPort-InOut", "Icapability", "INTERNO");

    }
```

**Tabla 58: Extracto de código del constructor de la clase Ambient**

El código de la Tabla 58 sigue la plantilla de generación de código C# desde PRISMA LDA-OA para los elementos arquitectónicos presentado en [Ca05]. No obstante, tal y como esta definida la plantilla en [Ca05] es en la clase final, es decir aquella que hereda del tipo de elemento arquitectónico quien define la importación de aspectos, puertos, *weavings*, etc. Así por ejemplo, un componente *Account* en C# hereda de la clase `ComponentBase`, obteniendo así su funcionalidad y describe los aspectos que importa, los *weavings* entre esos aspectos y los puertos para un *Account*. Como se puede ver en el caso de los ambientes es algo diferente, ya que en la clase base (`Ambient`) ya se define parte de esa importación de aspectos y generación de puertos.

No obstante, en la clase que represente el ambiente concreto, que heredará del tipo de `Ambient` (es decir de las clases `AmbientLogical`, `AmbientProcess` o `AmbientSite`), esta importación de aspectos será ampliada. De hecho ha de ser ampliada, pues es necesario importar el Aspecto de Distribución, que es obligatorio pero no está predefinido. Además, se pueden importar más tipos de aspectos, tantos como el diseñador estime oportuno. De la misma manera, en esas clases que representen ya al tipo de la arquitectura hay que definir obligatoriamente un *weaving*, entre el aspecto de Movilidad y el de distribución, además de los *weavings* que sean necesarios entre los demás tipos de aspectos. De la misma forma, en esa clase final que representará el elemento arquitectónico ambiente podrá definir también nuevos puertos. En definidas cuentas, la plantilla será similar a la del resto de elementos arquitectónicos con la salvedad que, por un lado todo lo relacionado con la importación de los elementos que proporcionan al ambiente su semántica (aspectos predefinidos y puertos) se realiza por defecto en el constructor de la clase base y, por otro lado, es necesario importar en la clase heredada el Aspecto de Distribución y definir un *weaving* con el Aspecto de Movilidad.

Pero, como se ha dicho, entre los diferentes tipos de ambientes existen una serie de restricciones que los caracterizan. Eso provoca que la implementación de determinados servicios en función del tipo de ambiente sea diferente. Por ello se ha decidido utilizar la herencia de la clase `Ambient` para separar los diferentes tipos de ambientes.

Una de las características en que se diferencian los diferentes tipos de ambientes son los tipos de elementos que pueden contener. Los ambientes de tipos *Process* y *Site* pueden ubicar componentes, conectores y otros ambientes *Process*, mientras que los ambientes *Logical* sólo pueden contener otros ambientes *Logical* o ambientes *Site*. Por otro lado, los ambientes *Site* están ligados a una localización física (*physical location*) mientras que los ambientes *Process* no, lo que establece una funcionalidad extra a los ambientes *Site* para gestionar este enlace.

Siguiendo estos planteamientos se ha optado por el diseño de la Figura 74, en el que las clases `AmbientProcess` y `AmbientLogical` heredan la funcionalidad directamente de la clase `Ambient` y sobrescriben servicios para adaptar la funcionalidad a las necesidades de la clase. Por otro lado, la clase `AmbientSite` hereda de `AmbientProcess`, porque si bien su funcionalidad es muy similar, ha de agregar un comportamiento específico para gestionar su enlace con el dispositivo físico.

Nótese que al utilizar la herencia, es posible hacer transparente el tipo del ambiente gracias al polimorfismo de inclusión. De esa forma, se puede garantizar que los aspectos predefinidos en las secciones anteriores puedan interactuar con el ambiente al que están agregados independientemente de cual sea el tipo de este. Esto es interesante, puesto que como se ha explicado, los aspectos predefinidos dotan a los ambientes de las semánticas de movilidad y comunicación distribuida (`MobilityAspect` y `AcoordinationAspect`, respectivamente) y se han de importar a todos los ambientes, sean del tipo que sean.

Esta transparencia respecto al tipo del ambiente propicia una forma de implementar los aspectos de forma genérica y delegar a la clase que represente del tipo de ambiente la funcionalidad interna. Por ejemplo, las peticiones de movilidad entre ambientes *Process* ubicados en un mismo *Site* no conllevan una serialización a través de la red del elemento arquitectónico que las solicita, sin embargo, si dicha movilidad es entre *Sites*, en el proceso irá implícita una serialización del elemento arquitectónico, pues ha de pasar por la red que separa ambos *middlewares* que representan los *Sites*. En el primer caso la movilidad seria orquestada por el ambiente *Process* o *Site* que contuviese los ambientes entre los que se mueve el elemento arquitectónico, mientras que en el segundo caso lo hará el ambiente *Logical* que ubicase ambos *Sites*. En todos los caso, la movilidad se proporcionaría a través del aspecto de Movilidad y las *capabilities* del Cálculo de Ambientes. Todos los ambientes importarán exactamente el mismo Aspecto de Movilidad, pero como se ha visto, la

funcionalidad será diferente. Esto se consigue gracias a la transparencia de tipo, que permite al aspecto solicitar servicios al ambiente al que está agregado para que este, en función de su tipo, actúe de un modo u otro.

A continuación, se abrirá una subsección para cada uno de estos tipos de ambientes en la que se expondrán como se han implementado los servicios que se reemplazarán en las clases heredadas de la clase base `Ambient`.

#### *6.1.2.1. La Clase AmbientProcess*

Como se ha comentado, la clase `AmbientProcess` (ver Figura 74) representa a los ambientes *Process*. Toda especificación de ambientes de este tipo ha de heredar de esta clase. La movilidad que proporciona este tipo de ambiente (para entrar o salir de él) se caracteriza que en ningún caso conlleva una serialización de los elementos arquitectónicos móviles. Esto no quiere decir que un ambiente *Process* no se deba serializar, pues se serializará en caso de moverse entre ambientes *Site*. Pero en el caso de la movilidad que proporciona un ambiente *Process*, la movilidad es siempre local a la máquina en que se está ejecutando. Si un elemento arquitectónico entra en su interior lo hará desde un ambiente de nivel jerárquico superior en el que se encontrará ubicado el ambiente *Process*, que podrá ser un ambiente *Site* u otro ambiente *Process*. Lo mismo ocurrirá si un elemento arquitectónico sale de él, pues las restricciones para la jerarquía de ambientes así lo estipulan.

La clase `AmbientProcess` tiene una lista para cada tipo de elemento arquitectónico que puede albergar. La restricciones del metamodelo limitan que un ambiente *Process* solamente puede proporcionar ubicación a elementos arquitectónicos componente, conector y a otros ambientes *Process*. Por ello, la clase tiene una lista para componentes (`componentsList`), otra para conectores (`connectorsList`) y otra para ambientes *Process* (`processAmbientsList`).

El comportamiento de los servicios heredados de la clase `AmbientProcess` se describe a continuación:

- `AddChild()`: Servicio interno para añadir elementos a las listas de elementos arquitectónicos ubicadois en el componente. Hay una lista para cada tipo de elemento arquitectónico (componente, conector o ambiente *Process*) por lo que el servicio comprueba el tipo del elemento pasado y añade a la lista adecuada.

- `RemoveChild()`: Servicio interno para eliminar un elemento de las listas del ambiente. Localiza al elemento entre todas las listas y lo elimina.

- `IsChild()`: Servicio publico que devuelve un valor booleano que indica si el nombre de elemento arquitectónico pasado pertenece a algún elemento de los ubicados en el interior del ambiente.

- `AddLocalAttachment()`: Servicio que crea un *attachment* en el interior del ambiente en colaboración con la capa del *middleware Element Management Layer*. Este *attachment* puede ser entre elementos arquitectónicos ubicados en su interior, o entre éstos elementos y los puertos del ambiente. En el caso de los ambientes *Process* este *attachment* nunca va a ser distribuido, por lo que es posible saltarse algunas comprobaciones para hacer más eficiente su creación. Este servicio es de utilidad para la recomposición de los canales de comunicación, como se verá más adelante.

- `SearchAttachment()`: Servicio que, busca un *attachment* en el ambiente que represente un determinado canal de comunicación pasado como parámetro. De nuevo, este servicio es de utilidad para la recomposición de los canales de comunicación, como se verá más adelante.

- `GetAmbientType()`: Servicio que devuelve una cadena con el tipo del ambiente. En este caso dicha cadena es "*AmbientProcess*".

- `TextualRepresentation()`: Servicio que devuelve una representación textual de la jerarquía de ambientes por debajo de ese ambiente. Para ello utiliza la representación textual para las primitivas de ambientes empleada por Cardelli [Car98].

Falta por explicar `RecomposeCommChannels` y la versión pública de `AddChild`, pero debido a su íntima relación con el modelo de ejecución de la movilidad se explicarán más adelante, en la sección específica para tal efecto.

#### *6.1.2.2. La Clase AmbientSite*

La clase `AmbientSite` (ver Figura 74) representa a los ambientes *Site*. Toda especificación de ambientes de este tipo ha de heredar de esta clase. Este tipo de ambientes tienen una funcionalidad muy parecida a los ambientes *Process*, pero se caracterizan por representar a un dispositivo físico, por lo que su semántica es diferente. No obstante, los tipos de elementos arquitectónicos que pueden ubicar son los mismos que en el caso de los `AmbientProcess` y por tanto los servicios de gestión de dichas listas los puede heredar directamente.

Puesto que el ambiente *Site* representa un dispositivo físico que se ha identificado con el *middleware* PRISMANET sobre el que se está ejecutando, se ha modificado el constructor para que no sólo reciba el nombre del ambiente, sino que también reciba un `LOC` que le indique la localización del *middleware* al que va a representar. Durante la instanciación, el ambiente *Site* comprueba que la localización representada por ese `LOC` es la misma en la que se está poniendo en ejecución y, de no ser así, lanza una excepción. Esto es así porque

el ambiente *Site* ha de representar al dispositivo en el mismo dispositivo, y no tiene sentido que lo haga remotamente.

Cuando el ambiente *Site* se pone en instancia avisa al dispositivo, es decir, al *middleware* al que representa que él va a ser quien le represente. De esa forma se establece una relación bidireccional entre el ambiente *Site* y el middleware. De esa forma, el middleware puede solicitar servicios al *Site* que lo representa y a su vez, el *Site* conoce la localización (URL) del middleware. Esto es importante, porque de todos los elementos arquitectónicos ubicados dentro del *Site*, él es el único que tiene información sobre la verdadera localización física del *middleware*. El resto de elementos arquitectónicos guardan información sobre el ambiente en el que se encuentran.

Así, se puede afirmar que los ambientes *Site* son como un puente, un vínculo entre la jerarquía de ambientes que ubican y el *middleware* al que representan. Cada vez que un elemento arquitectónico que estaba ubicado dentro de un *Site* traspase las fronteras de éste mediante la petición de *capabilities*, es lo mismo que si ese elemento arquitectónico cambiase el *middleware* en el que se va a ejecutar. Pero eso es transparente al elemento arquitectónico, que tan solo cambiará la referencia a su ambiente padre en su aspecto de distribución. Esta es la verdadera fuerza expresiva de los ambientes *Site*; son los únicos que conocen dónde se están ejecutando realmente y son los que realmente establecen los límites de cada *middleware*.

Por esta razón, no tiene sentido que un ambiente *Site* cambie su localización física en tiempo de ejecución, él representa a uno y solo un middleware. Si ese *middleware* deja de existir, el ambiente *Site* y todo lo que haya en su interior también. Es más, si un ambiente *Site*, durante su instanciación, intenta asociarse a un *middleware* que ya tiene un *Site* asociado, lanzará una excepción: no pueden haber dos *Sites* ejecutándose en el mismo *middleware*.

En lo que respecta a la implementación de la clase `AmbientSite`, por lo que se ha comentado anteriormente, tiene un campo de clase llamado `physicalLocation` de tipo `LOC` en el que guarda el valor de la localización física pasado como parámetro en el constructor. El resto de servicios que heredan los tipos de ambientes de la clase `Ambient` actúan de la misma forma que en `AmbientProcess`, por eso en la Figura 74 hereda directamente de ese tipo de ambiente. La única excepción es la del servicio `GetAmbientType()`, que por motivos evidentes devuelve la cadena "*AmbientSite*".

#### *6.1.2.3. La Clase AmbientLogical*

La última clase de la que queda por hablar de la Figura 74 es la clase `AmbientLogical`, que es la que representa a los ambientes de tipo *Logical*. Estos ambientes son los que se encuentran por encima de los *Sites* y son los que modelan el entorno de ejecución PRISMANET, formado por los diferentes

*middlewares*. Todo elemento arquitectónico que se mueva entre dos *Sites*, o lo que es lo mismo, de un *middleware* a otro, ha de pasar necesariamente por un ambiente *Logical*. No obstante, no puede quedarse en él, ya que los ambientes *Logical* sólo pueden ubicar ambientes *Site* y a otros ambientes *Logical*.

En la implementación, los ambientes *Logical* modelan las estructuras que están por encima de las máquinas que ejecutan el *middleware*. De esa forma es posible modelar una LAN que contienen varias máquinas ejecutando PRISMANET y por encima de ella, una WAN que contiene varias LAN, etc. El nivel de detalle lo determina el diseñador. Lo único que se ha de tener en cuenta es que existirá un ambiente *Logical* llamado *Root* que es el que estará en la parte más alta de la jerarquía y que ubicará al resto de ambientes. Este ambiente tendrá como ambiente padre en su Aspecto de Distribución la cadena vacía.

Para simplificar la implementación, en la primera versión de Ambient-PRISMANET se ha limitado el modelo para que solo soporte un nivel de ambientes *Logical*. Es decir, tan sólo se podrá definir un ambiente *Logical* que englobará a todos los *Sites* que representen a todos los *middlewares* en funcionamiento en el entorno de ejecución. Por tanto, únicamente se definirá el *Root* de la jerarquía de ambientes. Por esta razón, de momento, no se han estudiado la implantación de una jerarquía de ambientes *Logical* dejando esa tarea para futuras versiones.

La implementación de los ambientes *Logical* es algo más compleja que los otros dos tipos de ambientes. Se ha de tener en cuenta que se pretende representar una entidad que engloba varios *Sites*, o lo que es lo mismo, varios *middlewares*, y esta entidad ha de estar en ejecución a la vez en todos los *middlewares* a los que ubica, pues todos le pueden solicitar servicios a través de su puerto `ICapability`.

Una solución basada en la distribución del elemento arquitectónico Ambient *Logical*, es decir, que el ambiente se ejecute simultáneamente en todos los middlewares a los que proporcione ubicación conlleva muchas dificultades. Hasta ahora todo elemento arquitectónico en PRISMA se ejecutaba en un único *middleware*, teniendo la opción de migrar a otro gracias a las características distribuidas del modelo. El *middleware* no está preparado para partir la ejecución de elementos arquitectónicos, más aún, teniendo en cuenta que la semántica de los elementos arquitectónicos viene dada por sus aspectos. Por tanto se ha intentado buscar una aproximación capaz de simular este comportamiento.

Otra opción que se ha considerado es la de utilizar réplicas del Ambient *Logical* que estén en ejecución. De esa forma, el elemento arquitectónico no se distribuye, sino que varias instancias de él le representan en los diferentes *middlewares*. Estos ambientes *Logical*, mediante un canal de comunicación interno (por ejemplo un objeto servidor agregado) podrían intercambiar peticiones de servicios, por ejemplo lanzar una búsqueda distribuida de los

elementos arquitectónicos que contienen, etc. Es decir, múltiples réplicas que trabajan de manera conjunta, como si de una sola entidad se tratase. Para conseguir este propósito, todas las réplicas tendrían una lista de expansión, mediante la cual conocerían al resto de réplicas. Respecto a la comunicación con los elementos internos, todos los ambientes *Site* deberían conectarse a todos los puertos de movilidad de todas las réplicas, de forma que pudiesen recibir peticiones de *capabilities* del Ambiente Padre, que en este caso, puede sería a través de cualquiera de los Aspectos de Movilidad de las réplicas, de ahí la necesidad de conectarlos a todas.

Pero esta aproximación también presenta problemas: por un lado, las necesidades del modelo PRISMANET estipulan que no pueden haber dos elementos arquitectónicos con el mismo nombre, por lo que las réplicas deberían llamarse de forma diferente. Eso conlleva que dos ambientes *Site* hermanos, tuviesen valores diferentes en el campo `parent` de su aspecto de distribución. Por otro lado, el hecho que cada réplica tenga su propio hilo de ejecución, con sus propias colas, hace que el conjunto de ambientes *Logical* actúe de forma poco similar a un verdadero elemento arquitectónico: Se pierde el orden de llegada de peticiones (cada ambiente tendrá el suyo), la ejecución simultánea de servicios en los aspectos, etc. En cualquier caso es una solución que, con un estudio más exhaustivo podría proporcionar una forma de obtener ese comportamiento distribuido del ambiente *Logical*. No obstante, debido a la problemática que se presenta, se ha descartado para su implementación en este proyecto.

Una aproximación más sencilla supone centralizar el funcionamiento del ambiente *Logical*, de forma que únicamente se ejecute en un *middleware*, guardando la información sobre todos los *Sites* a los que proporciona localización. Estos *Sites* se comunicarían con el ambiente mediante *attachments* distribuidos (excepto el *Site* que representase al mismo *middleware* en el que se ejecuta el ambiente *Logical*) conectados a su puerto Icapability. Puede que una idea centralizada no sea la más adecuada para un modelo altamente enfocado a la distribución como es el de Ambient-PRISMA, pero dado que se está presentando una primera versión a la implementación del middleware para Ambient-PRISMA, se dará soporte a esta aproximación, dejando su mejora para futuras versiones.

Como se ha dicho, los ambientes *Logical* proporcionan una visión virtual de la infraestructura que hay por encima de los *middlewares*. En cierta manera son unos ambientes por los que pasarán los elementos arquitectónicos en su migración de un *Site* a otro. Y es durante este paso donde se producirá realmente la serialización. Por ello, son los ambientes Logical los encargados de serializar a los elementos arquitectónicos, puesto que dicha serialización se realizará en un momento entre que el elemento arquitectónico sale del *Site* en el que se encontraba y entra en el *Site* destino.

Esta serialización se hará mediante la capa *DistributionServices* del *middleware*, de forma que, cuando un elemento arquitectónico, tras salir de un

ambiente *Site*, entre en un ambiente *Process*, éste escuchará la siguiente *capability* del elemento arquitectónico (que será un `enter()` a un *Site*) y, no solo enviará el `accept()` al *Site* adecuado, sino que usará la información que tiene sobre él para lanzar una petición a la capa de Servicios de Distribución para que el elemento arquitectónico se mueva al *middleware* en que se encuentra dicho *Site*.

Entrando ya en la implementación de la clase, según las restricciones del metamodelo, un ambiente *Logical* sólo puede contener en su interior otros ambientes *Logical* o ambientes *Site*. Según la limitación impuesta de que sólo puede haber un nivel de ambientes *Logical*, los contenidos de este tipo de ambientes se reducen únicamente a ambientes *Site*. Los ambientes *Site* que se ubican dentro de un ambiente *Logical* se guardarán en la lista `siteAmbientsList`. Pero además, el ambiente *Process* ha de mantener también información sobre los elementos que están de paso. Debido a las restricciones del modelo transaccional, un ambiente sólo podrá servir un proceso de movilidad en un instante de tiempo, recuérdese que estos procesos siempre se definen transaccionalmente. Por ello, es necesario que tenga una lista genérica donde guardar información sobre el elemento que se encuentre de paso. Este elemento puede ser un componente, conector o un ambiente *Process*. Para guardar este valor se define el atributo de campo `tempFolder`, que representa al elemento de paso para el que el ambiente *Logical* es una ubicación temporal.

Por lo que respecta a los comportamientos de los servicios heredados de la clase `Ambient`, al igual que se ha hecho para la clase `AmbientProcess`, se describen a continuación:

- `AddChild()`: Servicio interno para añadir elementos a las listas de elementos arquitectónicos ubicados en el ambiente. Hay una lista para los ambientes *Site*, pues, como se ha explicado, son los únicos elementos que puede albergar.

- `RemoveChild()`: Servicio interno que localiza y elimina un ambiente *Site* de la lista de ambientes *Site*.

- `IsChild()`: Servicio publico que devuelve un valor booleano que indica si el nombre de elemento arquitectónico pasado pertenece a algún ambiente *Site* de los ubicados en el interior del ambiente o si es un elemento de paso.

- `AddLocalAttachment()`: Servicio que crea un *attachment* en el interior del ambiente en colaboración con la capa del *middleware Element Management Layer* asociada al *middleware* en que se esté ejecutando el ambiente. Este *attachment* puede ser local cuando se conecte al puerto de Movilidad de los elementos arquitectónicos de paso por el ambiente o al del ambiente *Site* ubicado en el mismo *middleware*. En otro caso el *attachment* es distribuido. Por esta razón, antes de crear los *attachments* se obtiene la ubicación de los

elementos arquitectónicos que va a conectar haciendo uso de la Capa de DNS para delegar la creación de cada parte del *attachment* al *middleware* en que se encuentren dichos elementos arquitectónicos. Este servicio es de utilidad para la recomposición de los canales de comunicación, como se verá más adelante.

- `SearchAttachment()`: Servicio que, busca un *attachment* en el ambiente que represente un determinado canal de comunicación pasado como parámetro. De nuevo, este servicio es de utilidad para la recomposición de los canales de comunicación, como se verá más adelante.

- `GetAmbientType()`: Servicio que devuelve una cadena con el tipo del ambiente. En este caso dicha cadena es "*AmbientLogical*".

- `TextualRepresentation()`: Servicio que devuelve una representación textual de la jerarquía de ambientes por debajo de ese ambiente. Para ello utiliza la representación textual para las primitivas de ambientes empleada por Cardelli [Car98]. El servicio obtiene la representación textual de los elementos arquitectónicos ubicados directamente en el mismo, y reenvía la petición a los ambientes para obtener la jerarquía completa recursivamente.

## 6.2. Comunicación distribuida en Ambient-PRISMA

Uno de los grandes desafíos de Ambient-PRISMA es proporcionar a los elementos arquitectónicos la manera de comunicarse de forma distribuida, es decir que elementos arquitectónicos en diferentes ambientes puedan comunicarse respetando la jerarquía de ambientes. De esta forma, los elementos arquitectónicos se pueden comunicar de forma distribuida sin que ello conlleve a ejecutar operaciones de movilidad, tal y como simula el *Ambient Calculus* [Car98] la comunicación entre ambientes, donde todas las comunicaciones son locales.

### 6.2.1. Introducción

Los elementos arquitectónicos ubicados en diferentes ambientes se comunican a través de los puertos de sus ambientes. De esa forma se establece un canal de comunicación a través de la jerarquía de ambientes en que se ubican los elementos arquitectónicos que se han de comunicar, a través del cual circularán los mensajes traspasando los límites establecidos por los ambientes.

En concreto, esta dilatación de la comunicación más allá de la frontera del ambiente se proporciona a través de dos puertos genéricos del ambiente (ver puertos D en Figura 75). Dichos puertos están asociados al Aspecto de Coordinación del Ambiente (ver sección 6.1.1.1), que es el encargado de

redirigir las peticiones hacia en el interior o el exterior de las fronteras del ambiente. El aspecto de coordinación redirige las peticiones recibidas desde el puerto `CallPort` con `Played_Role EXTERIOR` hacia el puerto `CallPort` con `Played_Role INTERIOR`. Además, el aspecto de coordinación redirige las peticiones recibidas desde el puerto `CallPort` con `Played_Role INTERIOR` hacia el puerto `CallPort` con `Played_Role EXTERIOR`. La Figura 75 muestra los diferentes elementos que componen el ambiente, acentuando la funcionalidad del puerto de los puertos de comunicación genérica.

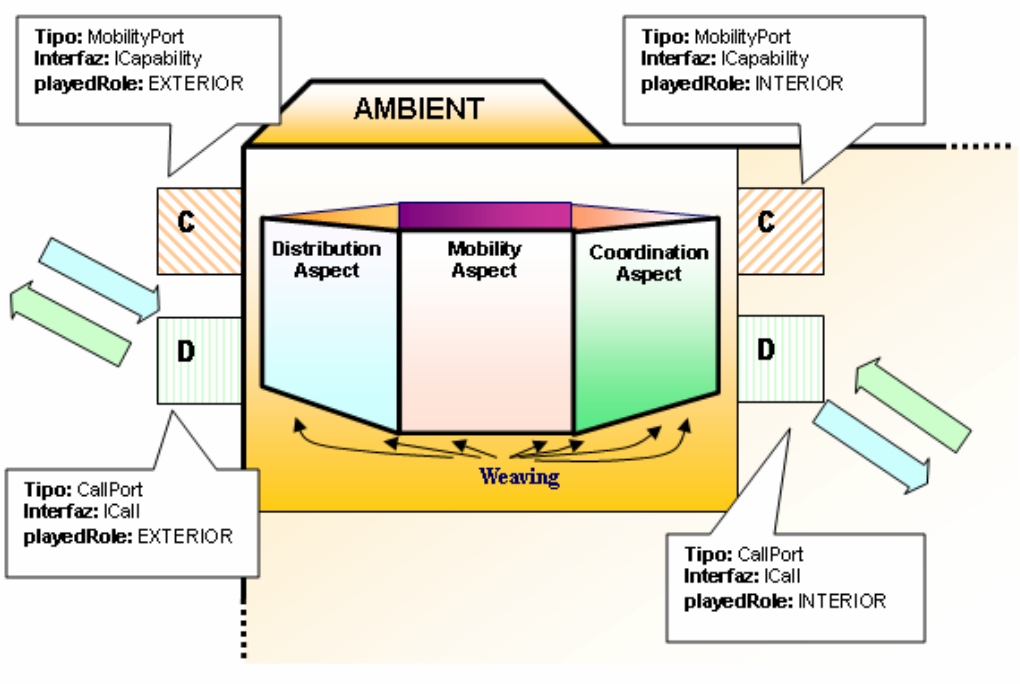
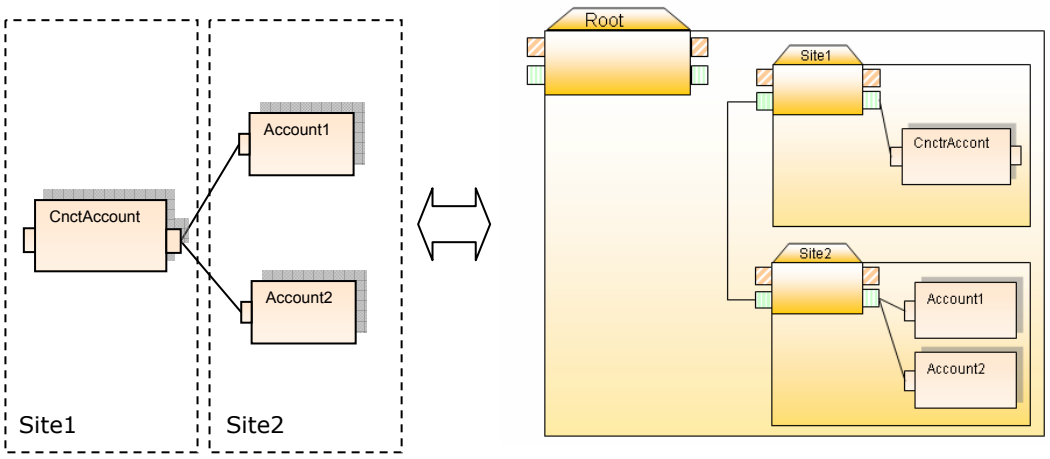



**Figura 75: Representación de los puertos predefinidos de un ambiente.**

Como se puede deducir, la incorporación de canales de comunicación distribuida complica el modelo, en el sentido de que surge la necesidad de generar varios *attachments* para representar un canal de comunicación que en PRISMA se representaba tan solo con uno. Esto no será siempre así, puesto que en las comunicaciones locales (al ambiente) los *attachments* tendrán la misma semántica que tenían en el modelo PRISMA, pero cuando la comunicación va más allá del ámbito local representado por un ambiente, es necesaria la instanciación de más *attachments* para obtener dicha semántica.

En la Figura 76(a) se muestra una configuración del ejemplo de la Entidad Bancaria en donde dos instancias de `Account` (*Account1* y *Account2*) y una instancia `CnctAccount` (*CnctAccount1*) están conectados mediante dos *attachments*. Por otro lado, en la Figura 76(b) se muestra como al ubicar los elementos arquitectónicos en ambientes, los dos *attachments* instanstanciados inicialmente tienen que ser traducidos en una serie de *attachments* entre los elementos y la jerarquía de ambientes para proporcionar la comunicación entre las instancias.




**(a) Visión del usuario** **(b) Visión Ambient PRISMA**

**Figura 76: Dos visiones en la instanciación de los canales de comunicación**

Como se puede apreciar, el modelo se complica considerablemente por ejemplo, el *attachment CnctAccount-↔Account1*, se transforma en el conjunto de *attachments CnctrAccount↔Site1*, *Site1↔Site2*, *Site2↔Account1*. Como se puede imaginar, en un ejemplo más complejo, con una jerarquía de ambientes algo más poblada, este hecho es aún más notable.

No obstante, no es la finalidad de Ambient-PRISMA dificultar la tarea de modelado al usuario, sino más bien aumentar su capacidad de expresividad. Por ello, se ha decidido mantener la instanciación de los *attachments* que componen el canal de comunicación transparente al usuario, quien solo deberá preocuparse de facilitar, a parte de la instanciación de la jerarquía de ambientes y los elementos arquitectónicos, la semántica de la comunicación. Esto es, siguiendo el ejemplo propuesto de la Figura 76, el usuario instanciará la jerarquía de ambientes `Root[ Site1 | Site2 ]`. Tras definir de esa forma las localizaciones donde ocurrirá la computación, el usuario instanciará los elementos arquitectónicos, indicando, para cada uno de ellos, su ubicación a través de su aspecto de distribución. Las instancias *Account1*, *Account2* y *CnctAccount* se pondrán en ejecución dejando la jerarquía de ambientes de la siguiente forma:

```
Root[ Site1 [ CnctAccount ] | Site2 [ Account1 | Account2 ] ]
```

Acto seguido, el usuario creará dos instancias del tipo de *attachment CnctAccount↔Account*, una que conectará el puerto del conector con *Account1* y otra que hará lo propio con *Account2*.

Eso será la instanciación desde el punto de vista del usuario. En realidad, el *middleware* al procesar la creación de los *attachments* se encargará de localizar los elementos arquitectónicos en la jerarquía de ambientes y comunicarlos a través de esta mediante la creación de tantos *attachments* como sea necesario para obtener la semántica de la comunicación expresada por el usuario.

### 6.2.2. Configuración inicial de los canales de comunicación

Como se ha explicado en la sección anterior, el usuario no es consciente de la creación de un canal de comunicación complejo formado por varios *attachments*: Él solamente ha definido un *attachment* que comunica dos elementos arquitectónicos.

En realidad la Capa *Element Management Layer* es la encargada de albergar la lógica de creación de los *attachments* que forman ese canal de comunicación en Ambient-PRISMA. En la anterior implementación de PRISMANET (ver Capítulo 3) esta era la capa encargada de crear los *attachments*, localizando los componentes y reenviando, en caso de *attachments* entre elementos distribuidos, la petición de creación al *middleware* correspondiente. Para dar soporte a los canales de comunicación complejos reutilizando las estructuras creadas para PRISMANET, se ha tenido que modificar la clase *attachment* para aportar a su lógica información sobre los canales de comunicación.

Llegados a este punto cabe hacer un breve paréntesis para establecer el significado que en Ambient-PRISMA se le da al concepto de canal de comunicación. En PRISMA un canal de comunicación simplemente se formaba de un *attachment*, que conectaba un puerto de un conector con el puerto de un componente. En Ambient-PRISMA el canal de comunicación distribuido ya no se forma por un solo *attachment* sino es formado por varios y un *attachment* puede formar parte de varios canales de comunicación. Es importante comprender la diferencia del concepto de *attachment* entre los dos modelos, pues a partir de ahora se hablará de *attachment* para hacer mención a un tramo, que puede ser el único, de un canal de comunicación y el término canal de comunicación se empleará para denotar a lo que en PRISMA se denomina *Attachment*.

En la implementación de Ambient-PRISMA ha sido necesario modificar la clase *attachment* para incorporar la información necesaria para dar soporte a la nueva definición. De esa forma, se ha añadido una colección en la que se añadirán los canales de comunicación que harán uso de ese *attachment* en uno de sus tramos. Efectivamente, un *attachment* de Ambient-PRISMA puede ser usado por más de un canal de comunicación, como por ejemplo, en la Figura 76 (b), el *attachment* que une *Site1* y *Site2* está siendo usado simultáneamente por los canales de comunicación *Accont1*↔*ConnectorAccount* y *Account2*↔*ConnctrAccount*.

La información de los canales de comunicación a los que sirven es fundamental en un *attachment* Ambient-PRISMA, porque son la justificación de su existencia. Como se verá más adelante, los canales de comunicación se reconfigurarán en tiempo de ejecución cuando se produzcan operaciones de movilidad y los *attachments* que en un momento eran un tramo de un canal de comunicación, pueden dejar de serlo tras la movilidad de alguno de los

elementos conectados, o incluso por la reconfiguración de la jerarquía de ambientes que los separa. Cuando un *attachment* no pertenece a ningún canal de comunicación, no tiene sentido que siga en ejecución: Ya que a través de él nadie va a enviar/recibir más peticiones.

Un canal de comunicación es único, ya que no tiene sentido en PRISMA definir dos instancias de *attachments* entre dos elementos arquitectónicos a través de los mismos puertos, y por tanto se representa mediante una cadena de caracteres con el siguiente formato:

*AE1.port1↔AE2.port2*

Siendo *AE1* y *AE2* los nombres de las instancias de los elementos arquitectónicos que conecta y *port1* y *port2* los puertos de las instancias que conecta. Para cada canal de comunicación se genera una cadena única siguiendo el formato expuesto que le servirá de identificación. Para garantizar la unicidad, la creación de este identificador se genera mediante un método estático de `ElementManagementLayer` que obtiene la cadena previa ordenación alfabética de los nombres.

De esta forma, los canales de comunicación están representados por cadenas. En implementaciones anteriores se consideró hacer del canal de comunicación una clase que guardase información sobre todos los *attachments* que lo formaban. Estos *attachments*, a su vez guardaban referencias a la instancia de los canales de comunicación de los que formaban parte. Gracias a estas referencias informaban al canal de comunicación de los cambios en caso de producirse. Así, por ejemplo, si un *attachment* dejaba de formar parte de un canal de comunicación, este hecho era notificado a la instancia del canal de comunicación que daba de baja el *attachment* de su lista, al tiempo que el *attachment* borraba la referencia a ese canal de comunicación.

En la práctica se ha visto que no hace falta guardar la información del canal de comunicación en una instancia de una clase, con el incremento de coste computacional que su gestión supone, por lo que se ha optado por simplificarlo considerablemente, usando tan solo una cadena de texto que represente al canal de comunicación, dejando la gestión de los canales de comunicación sólo a los *attachments*.

Por tanto, se han incorporado a la clase *attachment* métodos para la consulta y modificación de esa colección de canales de comunicación a que puede pertenecer una instancia de *attachment*. De esta forma se puede mantener la información del *attachment* acorde con la configuración de la arquitectura en tiempo de ejecución.

Otro campo que se ha añadido al *attachment* es uno que contempla el rol que desempeña el *attachment* respecto a su pareja. En Ambient-PRISMA se establecen dos tipos de relaciones básicas: relación entre padre e hijo, que es la que tiene todo elemento arquitectónico que, ya sea por movilidad o por

comunicación distribuida, esté conectado con el ambiente que le contiene y relación entre hermanos, que es la que tiene con los otros elementos arquitectónicos ubicados en el mismo ambiente y a los que esté conectado. De esa forma, se ha añadido el campo `Role` al *attachment* que puede adoptar el valor "*parent*" en caso que representen una relación padre-hijo o un valor "*sibling*" en caso de una relación entre hermanos. La necesidad de este campo se justificará más adelante, cuando se explique la regeneración de los canales de comunicación tras la movilidad en tiempo de ejecución.

Una vez explicados los cambios más significativos en la clase `Attachment`, llega el momento de mostrar como se obtiene la configuración inicial de los canales de comunicación.

Como se ha comentado anteriormente, el usuario define la semántica de la comunicación y esta información es recibida por el *middleware* a través de la capa de *Element Management* quien procederá a la generación de los *attachments* necesarios para instanciar ese canal de comunicación. Así pues, se utilizará la capa *DNS* del *middleware* para localizar los elementos arquitectónicos en el entorno de ejecución, es decir, localizará los *middlewares* en que se están ejecutando dichas instancias.

La idea es que cada *middleware*, a través del ambiente *Site* que lo representa en el modelo, obtenga la ruta del elemento arquitectónico a través de la jerarquía de ambientes. Así, el *Site* mirará entre sus listas internas para ver si el elemento arquitectónico es alguno de sus hijos, de no ser así, transferirá esta petición a cada uno de sus subambientes (ambientes *Proccess* que contiene) para que realicen lo mismo. En esta búsqueda *top-down*, cuando se localice el elemento arquitectónico, se terminará la recursión y se devolverá hacia arriba a través de la pila de ejecución creada para el recorrido (un `ArrayList`) donde se irán apuntando los nombres de los ambientes por los que se ha pasado, de forma que cuando la ejecución retorne al ambiente *Site*, en la lista se encuentre la ruta que se ha de seguir para llegar al elemento arquitectónico: los nombres de los ambientes que se han de atravesar. De manera análoga, en el *middleware* del otro elemento arquitectónico que se quiere conectar, si no es la misma, se obtendrá también la ruta.

En la Figura 77 se muestra como el algoritmo de búsqueda se realiza mediante una sucesión de llamadas a los subambientes de *Site1* para la obtención de la ruta del elemento arquitectónico *Account2* en la jerarquía de `Site1[ AP1[ AP3[ ] ] | AP2[ AP4[ ] | Account2 ] ]`.

El algoritmo de obtención de la ruta del canal de comunicación a través de los ambientes puede resultar ineficiente cuando la jerarquía de ambientes está muy poblada, pues es en realidad una búsqueda en profundidad en la rama de un árbol [Cor01]. Se ha propuesto una alternativa más eficiente a este proceso de búsqueda. Si bien la base de este algoritmo es el hecho de que todo ambiente conoce a sus hijos, una buena idea es invertir la búsqueda y que sean los hijos quien conozcan a su padre. Como se tiene el nombre del elemento

arquitectónico a conectar, una vez resuelta su ubicación física mediante el DNS, se obtiene una referencia a él y se le pregunta por su padre. De forma sucesiva, se va obteniendo el nombre del ambiente padre de cada ambiente por el que pasemos hasta llegar al ambiente *Site*, el punto de partida.

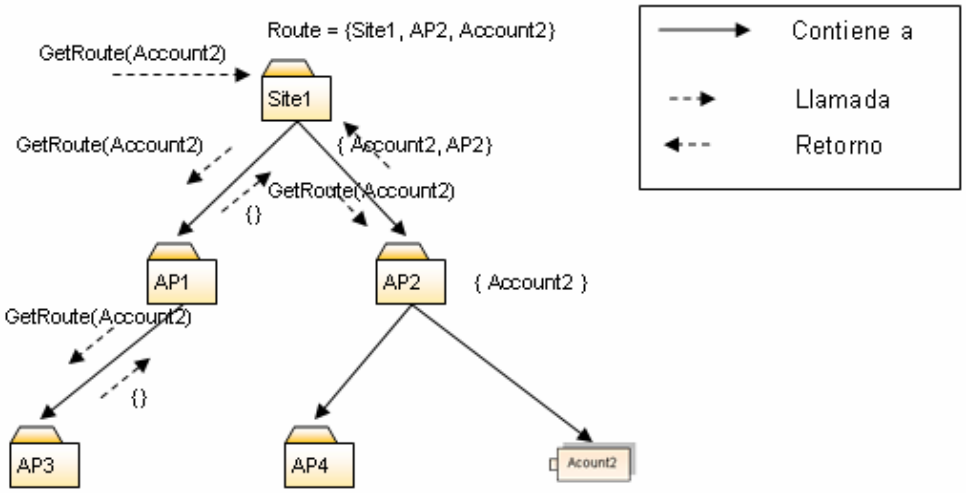



**Figura 77: Algoritmo de obtención de rutas top-down.**

De esa forma, la ruta de ambientes por los que ha de pasar la comunicación distribuida en esa rama de la jerarquía se obtiene a la primera, puesto que cada elemento arquitectónico tan solo tiene un padre, frente a los múltiples hijos que tiene un ambiente, por lo que en cada nodo la búsqueda es directa. En este caso se trata de una búsqueda bottom-up. En la Figura 78 se muestra un esquema con su funcionamiento. Como se puede observar en la figura, el `ElementManagementLayer` llama directamente al elemento arquitectónico y mediante llamadas consecutivas hasta llegar al *Site* donde representa el *middleware* se obtiene la ruta del elemento.

Una vez se ha obtenido la ruta a los elementos, ordenada en una lista donde el primer elemento es el *Site* que representa al middleware es necesario combinar la información. Para ello, se localizan los elementos que tengan en común ambas listas: El último elemento en común será el ambiente en que el *attachment* a crear sea de tipo *sibling*, es decir, una comunicación entre hermanos. De esa forma, partiendo del primer elemento arquitectónico, se recorre la lista al revés, conectando el elemento arquitectónico al puerto genérico con `Played_Role INTERNO` del ambiente inmediatamente superior. Este ambiente inmediatamente superior, es decir, el siguiente elemento de la lista, conectará su puerto `ICallPort` con `Played_Role EXTERNO` al puerto `CallPort` con `Played_Role INTERNO` del que a su vez, es su ambiente inmediatamente superior; de nuevo el siguiente de la lista. Así sucesivamente hasta que se llega al elemento de la lista anterior al elemento común. En este punto hay que hacer un *attachment sibling* (hasta ahora habían sido *attachments* de tipo *parent*) entre los dos elementos anteriores al elemento común de las dos lista, en este caso conectarán los dos puertos `CallPort` de los respectivos ambientes en su `Played_Role EXTERNO`. A partir de ese momento, se pasa a recorrer la lista con la

ruta en el otro *middleware*, desde el elemento que se acaba de conectar, creando *attachments parent* hacia los ambientes hijos definidos en la lista hasta llegar al elemento arquitectónico.

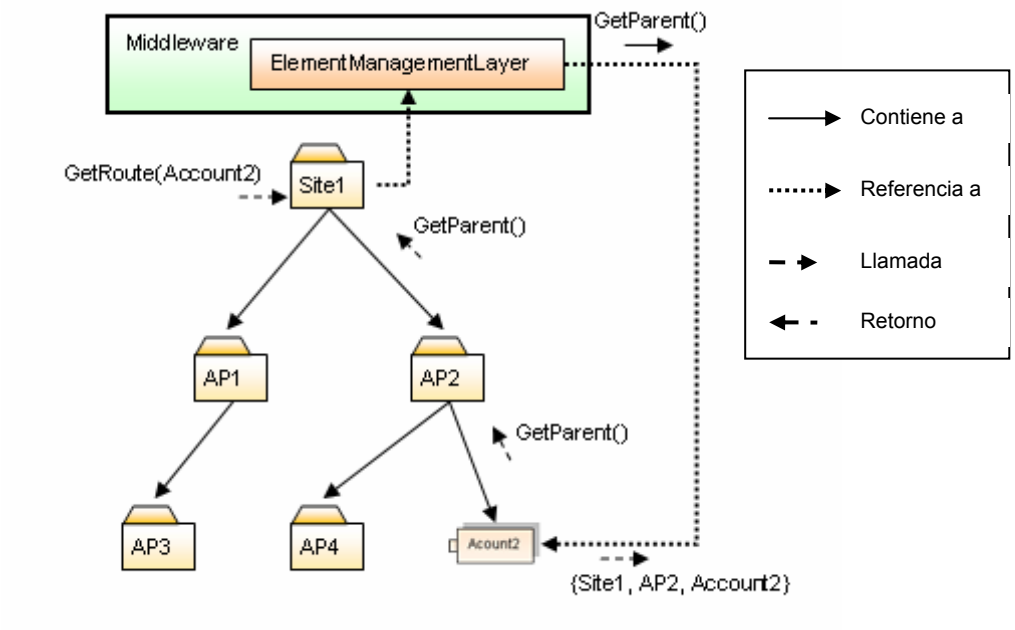



**Figura 78: Algoritmo de obtención de rutas recursivo hacia arriba.**

En la Figura 79 se muestra el esquema de creación de los *attachments* para establecer un canal de comunicación entre el componente `Account2` y el conector `CnctrAccount`. Nótese como se han creado dos *attachments parent* y uno *sibling*.

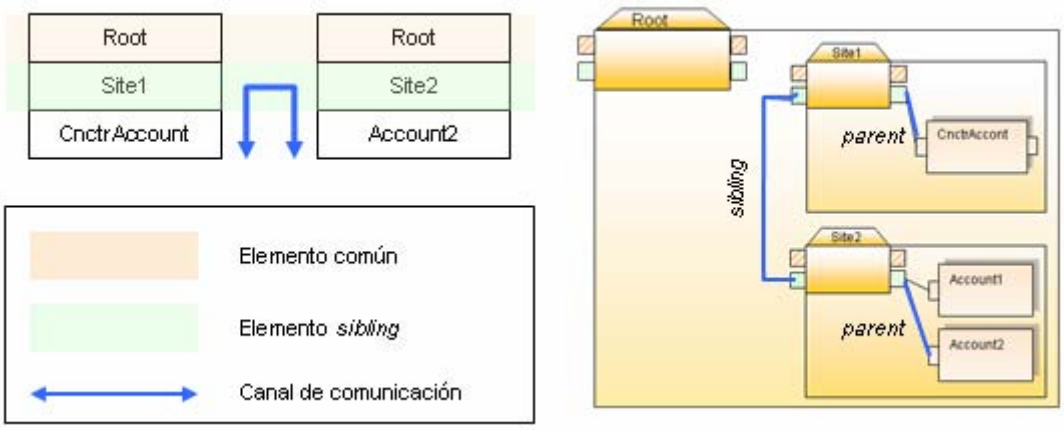



**Figura 79: Esquema de creación del canal de comuniación**

Tras este proceso automático, los *attachments* que conforman configuración inicial de los canales de comunicación han sido creados y las instancias se encuentran en ejecución, siguiendo el modelo de Ambient-PRISMA y con los canales de comunicación expresados por el usuario.

### 6.2.3. Modificación de los canales de comunicación

El tener la configuración inicial del canal de comunicación instanciada no es suficiente, pues como se ha comentado, los *attachments* que componen un canal de comunicación van a modificarse en tiempo de ejecución según las necesidades de los elementos arquitectónicos móviles.

De esa forma, cada vez que un elemento arquitectónico se mueve, el conjunto de *attachments* que forman los canales de comunicación con los que se conecta con el resto de elementos arquitectónicos varía. Durante un proceso de movilidad en Ambient- PRISMA, cuya mínima expresión viene dada por la ejecución de las *capabilities* `enter()` y `exit()`, las instancias de *attachment* son modificadas: se eliminan los canales de comunicación de los *attachments* a los que pertenecían por los que ya no va a pasar los mensajes entre los dos elementos arquitectónicos que lo componían, y son añadidos a otros *attachments* que entran a formar parte del conjunto de *attachments* que representan dicho canal de comunicación. Del mismo modo, si un *attachment* pasa a no formar parte de ningún canal de comunicación, se eliminará, pues su ejecución ya no tiene sentido en el modelo.

Por ello, es necesario proporcionar un mecanismo que, durante las operaciones de movilidad, sea capaz de recomponer los canales de comunicación en función del estado de la arquitectura.

En una implementación previa se consideró que en cada paso (`enter()` o `exit()`) del proceso de movilidad, tanto el ambiente origen como el destino obtuvieran la información de regeneración de los canales de comunicación desde la lista de *attachments* conectados al elemento arquitectónico móvil. En realidad, con las modificaciones que se han aplicado a la clase *attachment*, en cada instancia de *attachment* tenemos la información necesaria para localizar los elementos a los que se ha de conectar el elemento arquitectónico en el ambiente destino. Del mismo modo, con esa información, el ambiente origen, del que ha salido el elemento arquitectónico, se puede obtener la información necesaria para reconfigurar los *attachments* pertenecientes a los canales de comunicación relacionados con el elemento arquitectónico que se está moviendo definidos en su interior, para que apunten a la nueva ubicación de éste, o para que se reconfiguren para mantener la semántica de la comunicación tras los cambios en la jerarquía provocados por dicha movilidad.

En este punto es interesante señalar el hecho que, cuando un elemento sale o entra a un ambiente, mediante la solicitud de los *capabilities* `enter()` y `exit()` respectivamente, a efectos prácticos, la reconfiguración de la arquitectura sólo afecta al ambiente origen y destino, haciendo estos cambios transparentes al resto de la arquitectura. De esa forma, cada paso en el proceso de movilidad solo afecta a los dos ambientes implicados, y al elemento arquitectónico que se mueve, por supuesto. El ambiente origen eliminará de su lista de elementos arquitectónicos al elemento que ha salido de él, ya sea al entrar en otro ambiente de su mismo nivel de jerarquía o saliendo de ese nivel

al inmediatamente superior. El ambiente destino añadirá en su lista de hijos al nuevo elemento arquitectónico que ha incorporado, ya sea porque ha salido de un ambiente hijo o porque ha entrado desde el nivel jerárquico superior. Y, además, el elemento arquitectónico móvil modificará el valor de su localización en su aspecto de distribución: el valor de la localización será el nombre del ambiente destino.

Pues bien, para los canales de comunicación ocurre lo mismo: tan solo se han de reconfigurar los *attachments* implicados en el canal de comunicación relacionado con el elemento arquitectónico móvil en los dos ambientes implicados en la movilidad: El origen y el destino. El resto de *attachments* que forman el canal de comunicación y que se encuentran en ejecución en otros ambientes no se han de modificar, pues mientras se reconfiguren correctamente los *attachments* en los dos ambientes implicados en la movilidad, la semántica de la comunicación se mantiene inalterable.

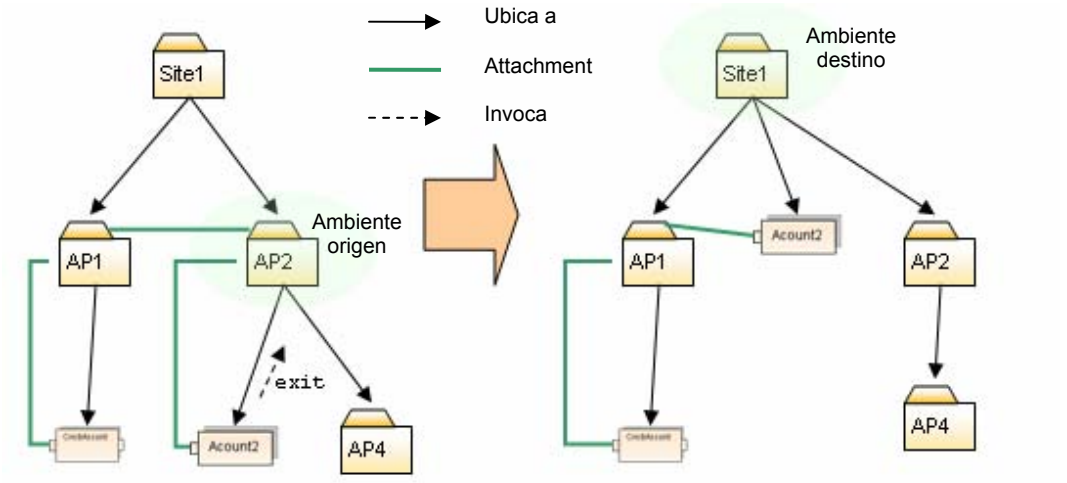



**Figura 80: Reconfiguración de *attachments* tras movilidad**

Como se observa en la Figura 80 el componente *Account2* tiene establecido un canal de comunicación a través de la jerarquía de ambientes con el conector *CnctrAccount*. Cuando *Account2* se mueve, los *attachments* que forman el canal de comunicación se verán modificados sólo en el ambiente origen (*AP2*) y en el ambiente destino (*Site1*). Como se puede ver en la parte derecha de la figura, la que representa el estado tras la movilidad, tras el movimiento de *Account2* se ha eliminado un *attachment* en *AP2,* el que conectaba el ambiente con el componente. En *Site1* se ha eliminado el *attachment* que conectaba *AP1* con *AP2*, o al menos se ha eliminado el canal de comunicación que representaba la comunicación entre el componente y el conector de la lista de canales de comunicación del *attachment*, y se ha creado un *attachment* entre el componente y el ambiente *AP1*. En el resto de ambientes por los que pasa el canal de comunicación no han tenido que modificarse, por lo que, para ellos, la movilidad ha sido transparente.

Así pues, como se ha comentado, gracias a la información de los *attachments* conectados al elemento arquitectónico es posible reconfigurar los *attachments* para mantener el canal de la comunicación tras cada paso de la movilidad. Cada vez que un ambiente reciba una petición de `enter()` o `exit()` a través de su puerto de movilidad, obtendrá la lista de *attachments* que están conectados a ese elemento arquitectónico y se la pasará al ambiente destino a través de `accept()` para que haga las modificaciones pertinentes en sus *attachments* y el origen hará lo propio una vez el elemento arquitectónico haya llegado al ambiente destino.

No obstante, esta aproximación tiene un inconveniente: Se están creando los canales de comunicación en cada paso de la movilidad, aún cuando es muy probable que esos *attachments* que estamos creando se tengan que destruir en el paso siguiente. En concreto, todo *attachment* conectado al elemento que se está moviendo se destruirá en el paso siguiente, en caso de haberlo. La creación de estos *attachments* hacia el elemento arquitectónico es, por otro lado una tarea sin sentido, ya que, a no ser que el elemento arquitectónico haya terminado su movilidad no se van a usar nunca, pues mientras el elemento está en movimiento está parado y no procesará peticiones que no sean las de su aspecto de distribución, mediante las cuales proseguirá sus pasos para completar la movilidad.

Esto no es siempre así, puesto que hay un *attachment* que sí se ha de crear necesariamente, pues de su existencia depende el buen funcionamiento de la movilidad: el *attachment* que comunica el puerto de interfaz `ICapability` del elemento arquitectónico con el ambiente. Este *attachment* se ha de crear cada vez que un elemento arquitectónico móvil entra a un ambiente para de esa manera garantizar que el elemento arquitectónico pueda solicitar la siguiente *capability* que conforma la movilidad. Este enlace se puede considerar un caso especial en la movilidad que se ha de tratar por separado al resto de *attachments*. A partir de ahora se dará por supuesto que en cada paso de la movilidad, este *attachment* se crea.

Por ejemplo, en la Figura 81 se muestra una configuración formada por dos ambientes *Site* (*Site1* y *Site2*) contenidos en un ambiente *Logical* llamado *Root*. En *Site2* hay un ambiente *Process* (*AP1*) que se va a mover de *Site2* a *Site1*. En la parte izquierda de la figura se ve el estado previo a la movilidad: *AP1* Está conectado al puerto ICapability del Aspecto de Movilidad (línea naranja) y al puerto ICall del Aspecto de Coordinación del Ambiente. En la parte derecha de la figura se muestra un paso intermedio al proceso de movilidad desde *Site2* a *Site1*; la situación tras solicitar un `exit()` a *Site2*. Como se puede observar, el *attachment* con el puerto `ICapability` se ha creado con el ambiente *Root*, puesto que es él quien le puede proporcionar el `enter()` a *Site1*, paso necesario para completar la movilidad. Nótese como no se ha recreado el *attachment* con el puerto de comunicación distribuida, pues el proceso de movilidad no ha finalizado y de crearse ese *attachment* se debería destruir en el siguiente paso.

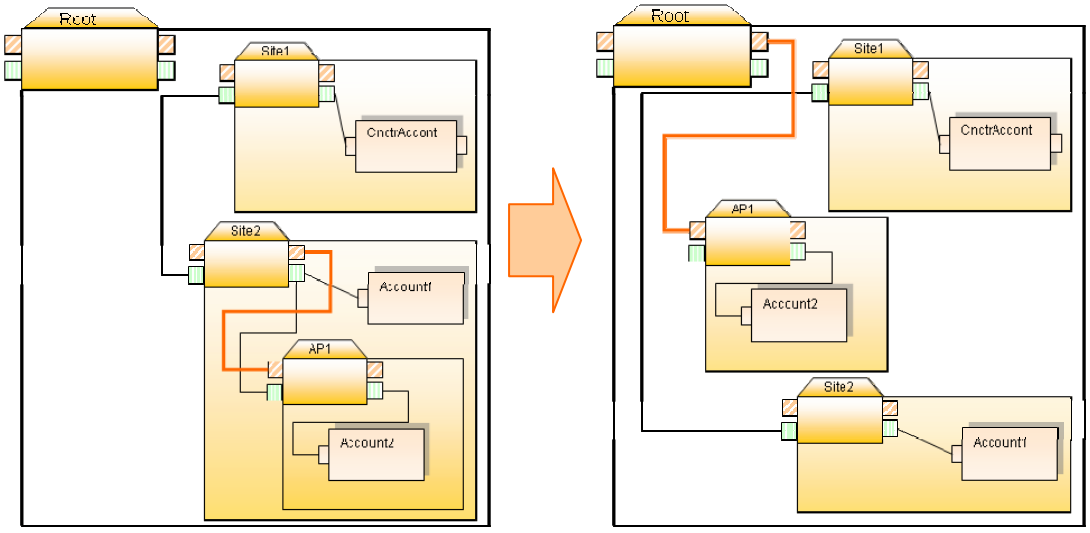



**Figura 81: Attachment con el puerto ICapability**

Por tanto, resulta interesante revisar el mecanismo de regeneración de canales de comunicación tras la movilidad para conseguir una regeneración de los canales de comunicación más eficiente. El problema reside en que, tal y como se ha descrito, la correcta recreación de los canales de comunicación está muy relacionada con la obtención de los *attachments* conectados al elemento arquitectónico móvil, por lo que este método deja de ser valido si estos *attachments* ya no se crean.

Como primera aproximación para la regeneración del canal de comunicación dado este nuevo modelo de movilidad en que no se crean los *attachments* conectados con el elemento arquitectónico en los estados intermediarios, es decir, ambientes intermedios de la movilidad se ha optado por dejar esta recreación hasta el final del proceso de movilidad. De ese modo, cuando el elemento arquitectónico llega al ambiente destino de la movilidad, tras haber traspasado los ambientes intermedios que establecen la jerarquía, se solicita al ambiente que regenere los canales de comunicación. Para ello, es necesario que el ambiente pueda obtener información sobre los canales de comunicación conectados al elemento arquitectónico que se ha movido, para que de esa forma solicite la recreación de estos. El proceso de recreación en este caso es similar al comentado para la configuración inicial: se vuelve a calcular la ruta para cada canal de comunicación y se va notificando a cada ambiente de la ruta los cambios, para que actualice sus ambientes internos, añadiendo nuevos, modificando los existentes, y si se da el caso, eliminando los que ya no están en uso.

Esto supone que en cada uno de los ambientes que ha traspasado el elemento arquitectónico móvil se han de localizar los *attachments* relacionados con canales de comunicación que conectan los puertos del elemento arquitectónico. Como se ha comentado, este proceso se iniciará una vez finalizada la movilidad, por lo que implica que el elemento arquitectónico debe indicar al ambiente en que se encuentra que ha finalizado un proceso de

movilidad. Para ello se hará uso del servicio `FinishMovement` de la interfaz `ICapability`.

No obstante, se ha desestimado la aproximación de realizar ese proceso al final de la movilidad, pues supone un postproceso que puede llegar a ser muy largo y, en ocasiones, problemático, ya que se ha de sincronizar a todos los ambientes por los que ha pasado el elemento arquitectónicos móvil durante el proceso de movilidad. Frente a esto se ha optado por usar una aproximación más elegante, en la que, de nuevo se utilice la información de los *attachments* conectados al elemento móvil para la regeneración del canal de comunicación en el paso siguiente, pero sin la necesidad de generar los *attachments* necesarios en cada paso. Se trata de una solución a medio camino entre las dos aproximaciones, la que recrea el canal de comunicación a cada paso y la que lo recrea al final de la movilidad.

Esto se puede conseguir gracias a una estructura intermedia, que en cada paso guarde la información necesaria sobre los *attachments* que se deberían crear para reconstruir los canales de comunicación conectados al elemento arquitectónico que se mueve, pero sin crearlos. De esa forma, esta estructura se va pasando a través de los ambientes de forma que estos puedan extraer de ella la información que necesitan para construir otra estructura similar, que será la que contenga la información sobre los canales de comunicación en ese ambiente una vez el elemento arquitectónico ha entrado en él.

Por otro lado, como se ha comentado, en cada proceso de movilidad expresado en su mínima expresión, esto es un `enter()` o un `exit()`, tan solo hay dos ambientes implicados, el origen y el destino. El ambiente origen, del que sale el elemento arquitectónico, le pasa al ambiente destino la lista con la información de los *attachments* y el elemento destino, en función los datos que extraiga de esa lista, obtendrá una nueva lista que podrá usar para pasar a otro ambiente en futuros pasos del proceso de movilidad, o, en caso que sea el último paso, para regenerar realmente los canales de comunicación. Pero aún hay más, porque esta lista de operaciones, que guarda la información necesaria sobre qué *attachments* se deberían crear para proporcionar los canales de comunicación necesarios, también es usada por el ambiente origen para reconstruir el canal de comunicación tras la salida del elemento arquitectónico. Es decir, una vez que el elemento destino de la *capability* ha recibido el elemento y ha generado la correspondiente lista de *attachments* que se deberían crear, el ambiente origen utiliza la lista que tenia para reconstruir los canales de comunicación para adaptarse a la nueva configuración alcanzada tras la movilidad. De esta manera, la comunicación se recompone parcialmente a cada paso:

- **Ambiente origen:** Le pasa la lista con la información sobre los *attachments* conectados al elemento arquitectónico que se va a mover al ambiente destino mediante la *capability* `accept()`. Tras la aceptación del elemento arquitectónico en el destino, reconfigura sus *attachments* internos para adaptar los canales de comunicación

a la nueva configuración, es decir, aquella en que el elemento arquitectónicos que se acaba de mover se encuentra en su nueva ubicación. Tras eso, vacía la lista de información de los *attachments*, pues esa información ya no hará falta en ese ambiente

- **Ambiente destino:** en función de la información recibida de los *attachments* conectados al elemento arquitectónico, genera una nueva lista con la información de los *attachments* que se deberían crear para adaptar los canales de comunicación a la nueva configuración de la arquitectura.

Cuando el proceso de movilidad llegue a su final, se invocará el servicio `FinishMovement()`, que tan solo deberá coger la lista de *attachments* que haya generado el ambiente al recibir el elemento arquitectónico en su último movimiento y generar, esta vez sin estructura intermedia, cada uno de los *attachments*.

Como se puede apreciar, tras la ejecución de `FinishMovement()` los canales de comunicación conectados a elemento móvil mantienen la semántica gracias a la reconstrucción parcial que se ha llevado a cabo en cada paso y a la reconstrucción final llevada a cabo por `FinishMovement()` en el ambiente final del movimiento. Desde ese punto de vista, se ha obtenido la configuración deseada sin necesidad de recurrir en ningún momento a ambientes no implicados en la ejecución del paso de movilidad, es decir, que no sean el origen o el destino.

Puesto que en la movilidad se va a trabajar con listas de información de *attachments* en lugar de con *attachments* directamente, surge la necesidad de realizar un preproceso antes de iniciar el proceso de movilidad. Este preproceso será el encargado de generar la primera lista de información de *attachments* conectados al elemento arquitectónico en su ubicación inicial. Esta lista es la que se pasará en el primer `accept()` del proceso al ambiente destino del primer paso de la movilidad para que realice las acciones comentadas anteriormente. Esta funcionalidad se realizará mediante la invocación de la *capability* `StartMovement()`.

Nótese que con la incorporación de esta funcionalidad, el aspecto de distribución del elemento arquitectónico que realiza el proceso de movilidad deberá indicar el comienzo y el fin del movimiento. Esto implica que las solicitudes de los las *capabilities* `exit()` y `enter()` deben siempre ser realizadas dentro de una transacción para que la movilidad sea correcta, y por tanto, se realicen todos sus operaciones relacionadas, como la regeneración de los canales de comunicación. Es necesario que las *capabilities* `enter()` y `exit()` se completen con una llamada al servicio `StartMovement()` y otra al servicio un `FinishMovement()`, perdiendo así la posibilidad de ejecutarse por si solas.

La información de un *attachment* se ha implementado haciendo uso de un *struct* en C# llamado `TraceOperation` (Figura 82). Los ambientes tienen una lista

de elementos `TraceOperation` en la que guardan la información de los *attachments* que están conectados al elemento móvil. La lista se rellenará cuando el aspecto de movilidad del ambiente sirva un `accept()`, es decir, cuando un elemento arquitectónico vaya a entrar en su interior y se vaciará cuando este elemento arquitectónico salga, es decir al finalizar la ejecución del `moving()`.

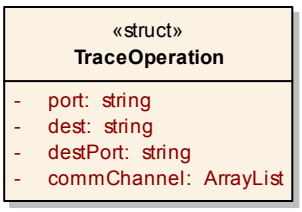



**Figura 82: Estructura TraceOperation**

Cada uno de los elementos `TraceOperations` está relacionado con el elemento arquitectónico que está en movimiento, de hecho, representa a un *attachment* que está conectado a él. La información que guarda sobre este *attachment* es el puerto (`port`) del elemento arquitectónico móvil y el nombre (`dest`) y puerto (`destPort`) del elemento arquitectónico al que conecta el *attachment*. Finalmente, guarda una lista de canales de comunicación con los nombres de los canales de comunicación que están compartiendo este *attachment* como un tramo de ellos.

En la Tabla 59 se muestra la implementación del servicio `StartMovement()` del Aspecto de Movilidad genérico. Como se puede ver delega su funcionalidad al elemento arquitectónico que tiene por debajo, esto es, un ambiente. El servicio `SetTraceOperations()` del ambiente localizará al elemento arquitectónico para el que ha de obtener la información sobre *subscribers* y generará una lista de `TraceOperation` con todos los *attachments*. Esa lista será almacenada en el ambiente, y en sucesivos pasos será requerida por el Aspecto de Movilidad. Una vez guardada la lista de *subscribers* con la que se va a trabajar, ya se pueden eliminar los *subscribers* conectados al elemento arquitectónico; puesto que si se va a mover no tiene sentido que siga recibiendo peticiones. No obstante es importante que en el borrado de *subscribers* no se elimine el *attachment* con el que el elemento arquitectónico se comunica con el puerto `ICapability` del ambiente `parent`.

```
public AsyncResult StartMovement(string name)
{
  //Obtiene los datos de los subscribers conectados al
  //elemento arquitectónico pasado y los guarda en
  //el Ambiante
   ((Ambient)link).SetTraceOperations(name);

  return null;
}
```

**Tabla 59: Implementación del servicio StartMovement en el Aspecto de Movilidad predefinido**

A modo de ejemplo de lo explicado en los anteriores párrafos, se mostrará una traza de la regeneración de los canales de comunicación haciendo uso del caso de estudio propuesto en la sección 5.3. La siguiente traza se centrará en las operaciones relacionadas con la generación de los canales de comunicación.

Se parte de una situación en que se encuentra el modelo en ejecución, con un componente *Customer* y un ambiente *Process AgentsAP* ubicados en el ambiente *Site ClientSite* y un componente *Auction* ubicado en el *Site AuctionSite*. Completan el modelo los componentes *Purchaser* y *Collector*, ubicados en *AgentsAP* y los conectores necesarios.

En un momento dado, el *Customer*, a través del conector *AgentsCustCnctr*, decide mover el ambiente *Process AgentsAP* al *AuctionSite* para que realice las acciones pertinentes para comprar un objeto en una subasta. Para ello hace uso del servicio *move(AuctionSite)* de su aspecto de distribución (*CustDist*). Este servicio lo procesará el aspecto de distribución de *AgentsAP* (*AAPDist*) mediante la transacción que se muestra en la Tabla 60.

**PRISMA LDAOA**

```
Transactions in move(NewAmbient:string)
      move::= out startMovement(input Name, output CommunicationList[])→ MOVE1;
      MOVE1::= out exit()(Name,Parent)→ MOVE2;
      MOVE2::= out enter()(Name, NewAmbient)→MOVE3;
      MOVE3::= out finishMovement(input Name, input CommunicationList[]);
```

**IMPLEMENTACIÓN C#**

```
public AsyncResult Move(string NewAmbient)
{
        // Inicio transacción
        aspectStateCareTaker.BeginTransaction();

        try
        {

                string Name = this.componentLink.componentName;
                string Location = this.location;

                // va a solicitar servicios
                this.ServiceIN = false;

                //MOVE
                StartMovement(Name);
                aspectStateCareTaker.CheckConsistence();

                //MOVE1
                Exit(Name, ref Location);
                aspectStateCareTaker.CheckConsistence();

                //MOVE2
                Enter(Name, NewAmbient);
                aspectStateCareTaker.CheckConsistence();

                // MOVE3
                FinishMovement(Name);
                aspectStateCareTaker.CheckConsistence();
        }
        catch (Exception ex)
        {
                aspectStateCareTaker.SetConsistecy(false);
                throw ex;
        }
        finally
        {
```

```
                    // Fin de la transacción
                    aspectStateCareTaker.EndTransaction();
            }

            return null;
    }
```

**Tabla 60: Transacción que define la movilidad de AgentsAP**

La primera operación `startMovement(AgentsAP)` indicará al ambiente padre que *AgentsAP* va a moverse, por lo que se ha de preparar para tal propósito. Lo que hará el ambiente padre es obtener todos los *subscribers* conectados al elemento arquitectónico y encapsulará la información de cada uno de ellos en un `TraceOperation`. Como resultado, `startMovement()` devolverá un listado con todos los `TraceOperation`, es decir, con la información de todos los *attachments* conectados a *AgentsAP*. Mediante la ejecución de este servicio el ambiente padre rellena el valor del campo `traceOperations` que usará en adelante.

Como el elemento arquitectónico se va a mover, también tiene sentido en este paso eliminar todos los *attachments* conectados a él; a partir de ahora se trabajará con sus listas de *attachments*. No obstante, el *attachment* conectado al puerto `ICapability` del ambiente padre no se borrará. Ya se ha comentado que este *attachment* se trata de una forma especial en la movilidad. En este caso es necesaria su existencia para garantizar que el elemento arquitectónico que se mueve (*AgentsAP*) pueda comunicarse con su ambiente padre (*ClientSite*). De esto también se encarga la *capability* `StartMovement`.

| TO1 | | | |
|---|---|---|---|
| port | dest | destPort | commChannel |
| ECapabilitiesPort | ClientSite | ICapabilitiesPort | AgentsAP ←→ ClientSite[7] |
| EMobilityPort | AgentCustCnctr | AgentCustMobPort | AgentsAP ←→ AgentCustCnctr |
| EServicesPort | AgentCustCnctr | AgentCustPort | AgentCustCnctr←→Purchaser |
| DCapabilitiesPort | ClientSite | ICapabilitiesPort | AgentsAP ←→ ClientSite [8] |

**Tabla 61: Trazas de operaciones ClientSite**

En el siguiente paso (`MOVE1`) el elemento arquitectónico solicita un `exit()` a *ClientSite*. El `exit()` hará las comprobaciones pertinentes y obtendrá de *ClientSite* la listas de trazas de operaciones que se ha creado en el paso anterior. Esta lista será enviada mediante el `accept()` al ambiente padre, es decir, al *Root*. El *Root* al procesar el `accept()` recibirá la lista de trazas de operaciones. Gracias a esa información, el *Root* es capaz de generar otra lista de trazas de operaciones en la que se anotara los *attachments* que se debería de crear para mantener los canales de comunicación tras el `exit()` (`TO2`). Además, el *Root* crea un *attachment* desde su puerto `ICapability` al puerto de comunicación del elemento arquitectónico. Para determinar cual es el puerto de movilidad del elemento arquitectónico el *Root* también hace uso de la lista de

[7] AgentsAP.ECapabilitiesPort ←→ ClientSite.ICapabilitiesPort[7]
[8] AgentsAP .DCapabilitiesPort←→ ClientSite.ICapabilitiesPort [8]

trazas de operaciones que ha recibido: busca en ella el puerto que se conectaba al puerto `ICapability` del ambiente anterior, es decir, *ClientSite*.

| TO2 | | | |
|---|---|---|---|
| port | dest | destPort | commChannel |
| ECapabilitiesPort | Root | ICapabilitiesPort | AgentsAP ←→ Root |
| EMobilityPort | ClientSite | EServices | AgentsAP ←→ AgentCustCnctr |
| EServicesPort | ClientSite | EServices | AgentCustCnctr←→Purchaser |
| DCapabilitiesPort | Root | ICapabilitiesPort | AgentsAP ←→ Root |

**Tabla 62: Trazas de operaciones Root**

Antes de terminar la ejecución del `exit()`, pero una vez *AgentsAP* ya esté en *Root* (ver Figura 71) el *ClientSite* recompondrá los canales de comunicación haciendo uso de las trazas de operaciones que tenia guardadas (`TO1`). Para ello tendrá en cuenta *AgentsAP* ha salido de *ClientSite*, y por tanto hará que todos los *attachments* que se conectaban a él (los dos que tenia con *AgentCustCnctr*) se recreen conectados al puerto de comunicación distribuida del *ClientSite*. Un caso especial de este comportamiento es el que recibe el *attachment* que conectaba a *AgentsAP* con *ClientSite*. Este *attachment* no se ha eliminado en el paso anterior, cuando se creó la lista de trazas de operaciones (`TO1`) en la ejecución de `StartAttachment`. Pues bien, como ahora *AgentsAP* se encuentra en *Root* y ya tiene creado un *attachment* con el puerto `ICapability` de su nuevo ambiente, este *attachment* ya no hace falta. Por tanto, el *attachment* con el puerto `ICapability` de *ClientSite* se elimina. Finalmente, como el *ClientSite* ya no necesita para nada la información contenida en `TO1`, la puede borrar.

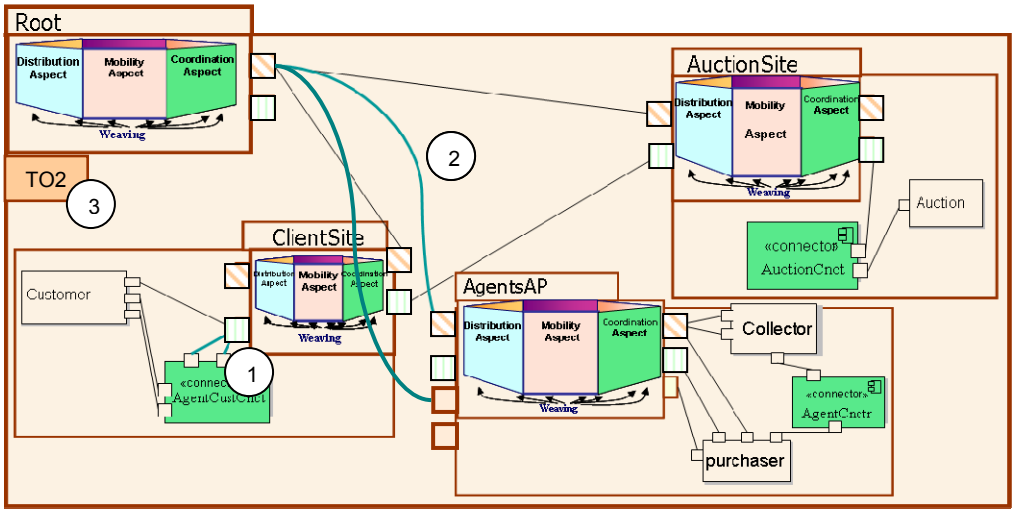



**Figura 83: Configuración de MobileAuction tras el exit() de AgentsAP**

En la Figura 83 se muestra el estado de la configuración tras la salida de *AgentsAP* de *ClientSite*. Como se puede observar, la configuración de los canales de comunicación de *ClientSite* ha cambiado (1) creándose dos nuevos *attachments* (en verde). Por otro lado, *AgentsAP* se encuentra en *Root* con los únicos *attachment* creados al puerto `ICapability` de su Ambiente Padre (2) y con

la información del resto de *attachments* en una lista de trazas de operaciones, `TO2` (3).

En esa situación *AgentsAP* solicita un `enter()` a *Root* (paso `MOVE2`). De la misma manera que en el caso del `exit()`, la petición desencadena en un `accept()` a *AuctionSite* en el que se le pasa la lista de trazas de operaciones `TO2`. De esa forma, *AgentsAP* entra en *AutionSite*. *Root* genera su propia lista de trazas de operaciones (`TO3`) en base a la lista recibida y crea el *attachment* entre su puerto `ICapability` y el puerto de movilidad del *AgentsAP*. Al igual que en el caso anterior obtiene la información sobre ese puerto desde la lista de trazas de operaciones pasada (`TO2`).

| TO3 | | | |
|---|---|---|---|
| port | dest | destPort | commChannel |
| ECapabilitiesPort | AuctionSite | ICapabilitiesPort | AgentsAP ←→ AuctionSite |
| EMobilityPort | AuctionSiite | IServices | AgentsAP ←→ AgentCustCnctr |
| EServicesPort | AuctionSite | IServices | AgentCustCnctr←→Purchaser |
| DCapabilitiesPort | AuctionSite | ICapabilitiesPort | AgentsAP ←→ AuctionSite |

**Tabla 63: Trazas de operaciones AuctionSite**

El ambiente *Root* finalizará el servicio de la *capability* `enter()` regenerando los canales de comunicación en sus límites. Para ello hará uso de la lista de trazas de operaciones `TO2`. Además *Root* eliminará el *attachment* que tenia al puerto de movilidad de *AgentsAP*. En la Figura 84 se muestra el estado de la configuración tras la ejecución de la *capability* `enter()`. *AgentsAP* se encuentra en *AuctionSite* y tiene un *attachment* con su nuevo ambiente (1). En *Root* los canales de comunicación están restablecidos, pues el *attachment* entre los puertos de comunicación distribuida de *ClientSite* y *AuctionSite* proporcionan servicio a los canales de comunicación *Customer*↔*AgentsAP* y *Cuistomer*↔*Auction* (2). Finalmente, *AuctionSite* tiene su propia lista de trazas de operaciones (3), donde guarda la información sobre los *attachments* que se deben crear en su interior para reconstruir el canal de comunicación.

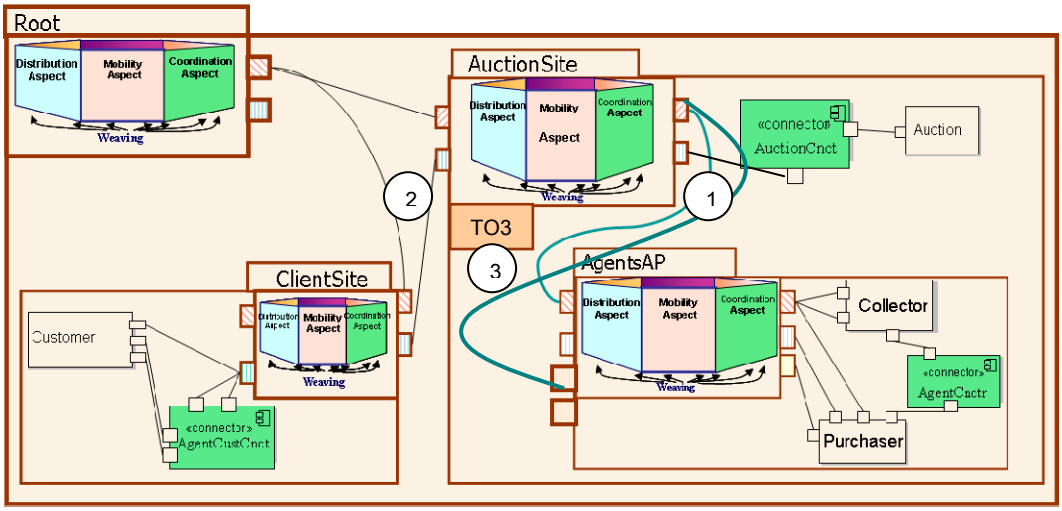


**Figura 84: Configuración de MobileAuction tras el enter() de AgentsAP**

El último paso de la transacción (MOVE3) es indicar que el proceso de movilidad ha finalizado mediante la *capability* `FinishMovement`. *AuctionSite*, al recibir dicha petición del *AgentsAP* obtiene la lista de trazas de operaciones que tiene, TO3, y crea los *attachments* necesarios para restablecer los canales de comunicación. En la Figura 85 se muestran en verde los *attachments* creados, uno con el *Auction* y otro con el puerto genérico de *AuctionSite*, mediante el que se conectará con el *Customer*.

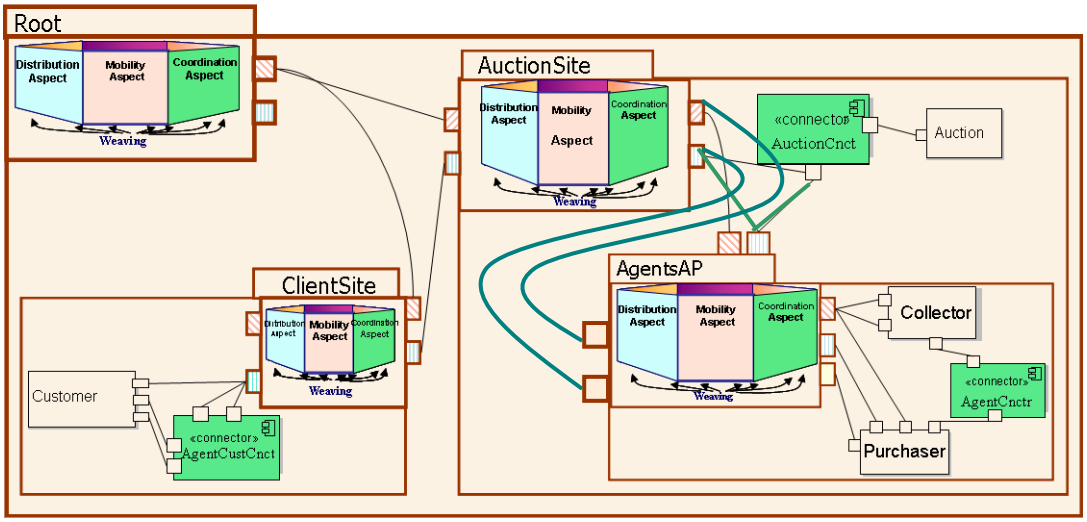



**Figura 85: Configuración de MobileAuction tras el FinishMovement**

Al finalizar la ejecución del servicio, la transacción ha finalizado: *AgentsAP* se ha movido desde *ClientSite* a *AuctionSite* y los *attachments* de toda la arquitectura se han reconfigurado para adaptarse a la nueva situación. Es interesante remarcar el hecho que los canales de comunicación se han ido recomponiendo paso a paso, siempre después que el elemento arquitectónico saliese del ambiente. Con esta reconstrucción parcial se consigue, por un lado que no se tengan que crear a cada paso los *attachments* necesarios, que probablemente se tendrán que borrar en el caso siguiente y, por otro lado, que no sea necesario volver a pasar al final por todos los ambientes por los que ha pasado el elemento arquitectónico recomponiendo los canales.

## 6.3. Modelo de ejecución de la movilidad

En anteriores secciones se han presentado las diferentes entidades que desempeñan un papel importante en el proceso de movilidad, y se ha explicado la gestión de los canales de comunicación. En esta sección se explicará cual es el nuevo modelo de ejecución de la movilidad obtenido de la utilización de ambientes y *capabilities*. La explicación se basará en la descripción de los servicios implicados, sobre todo al nivel del elemento arquitectónico ambiente.

La semántica de la movilidad en Ambient-PRISMA es proporcionada en parte por el Aspecto de Movilidad genérico (*Mobility*) y en parte por el tipo de

ambiente que subyace, es decir que tiene ese aspecto. En anteriores secciones se ha mostrado cómo la implementación de los servicios internos de movilidad ofrecidos por los diferentes tipos de ambientes varía en función del tipo (ver 6.1.2).

Las capabilities del Aspecto de Movilidad se podrían separar en dos grupos. Un primer grupo estaría formado por `startMovement()` y `finishMovement()` son *capabilities* encargadas de preparar al elemento arquitectónico que se va a mover. Por un lado, `startMovement()` propicia un entorno válido para iniciar el proceso de movilidad, transformando los *subscribers* conectados al elemento arquitectónico en una lista de `TraceOperations`. El método `finishMovement()`, por el otro lado, es el encargado de convertir un estado intermedio de la movilidad, en un estado final y válido de la arquitectura, transformando la lista de `TraceOperations` en *subscribers* en ejecución conectados al elemento arquitectónico móvil.

Otro grupo, lo podrían conformar las *capabilities* `enter()` y `exit()`, encargadas de proporcionar la movilidad al elemento arquitectónico. La ejecución de estas *capabilities* siempre resultan en la salida de un elemento arquitectónico de un ambiente origen y su entrada en un ambiente destino (para un `enter()`, el ambiente destino es hijo del ambiente origen y para un `exit()`, el ambiente destino es padre del ambiente origen). Este movimiento común entre dos ambientes se puede ver en la separación de la funcionalidad de los servicios `enter()` y `exit()` en las llamadas a `moving()` y `accept()`.

Mediante el servicio `moving()` se representan los cambios que ocurrirán en el ambiente origen cuando el elemento arquitectónico salga de él. Por eso, `moving()` se ocupa de obtener la información del elemento arquitectónico (nombre, tipo, lista de trazas de operaciones) y de eliminar el elemento arquitectónico de la lista interna del ambiente y de borrar su *attachment* con el puerto `ICapability` cuando la entrada en el ambiente destino se ha hecho efectiva.

El servicio `accept()` encapsula la funcionalidad relacionada con las gestiones necesarias para la recepción del elemento arquitectónico en el ambiente destino. Por ello, `accept()` se encargará de añadir el elemento arquitectónico a la lista interna del ambiente destino así como de procesar la lista de operaciones traza que reciba para generar su propia lista de trazas de operaciones. También se encargará de generar el *attachment* con el puerto `ICapability` del ambiente destino.

Como se puede ver, la funcionalidad de `enter()` y `exit()` está separada en aquello que se realiza en el ambiente origen y aquello que se ejecuta en el ambiente destino (`moving()` y `accept()` respectivamente). Así, pese a que el proceso es iniciado por el Aspecto de Movilidad del Ambiente origen, previa solicitud del elemento arquitectónico conectado mediante un *attachment* a su puerto `ICapability`, esta petición se transforma en un `accept()` que repercute en el ambiente destino. Ésto está en armonía con lo que ya se ha comentado

respecto al hecho que, todo proceso de movilidad entre un origen y un destino está formado por una sucesión de pasos (`enter()` y `exit()`) en los que solo participan dos ambientes: el ambiente origen y ambiente destino (ver Figura 86).

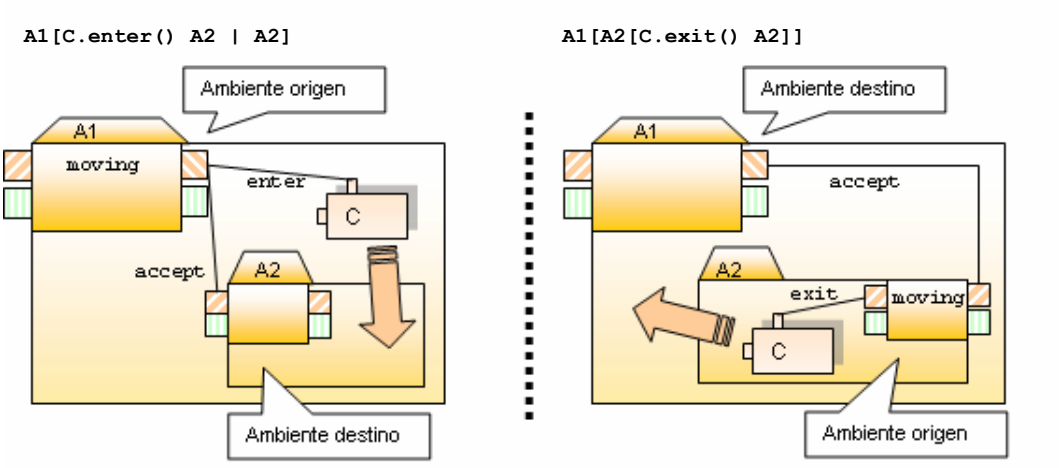



**Figura 86: Esquema de un paso de movilidad**

Pero cabe en este punto dar más detalle a esta ejecución bilateral de la movilidad, porque si bien, en cada uno de los dos ambientes, origen y destino, se realizan las acciones antes mencionadas, el comportamiento varía en función del tipo de ambiente en el que se ejecuten. En la sección 6.1.2 ya se han introducido las diferencias entre los diferentes tipos de ambientes. En este punto, se va a mostrar con más detalle como estas diferencias afectan a la implementación de los servicios internos del elemento arquitectónico ambiente, en cada uno de sus tipos.

### 6.3.1. Servicio Moving

El servicio `moving()` del Aspecto de Movilidad genérico obtiene la información sobre el elemento arquitectónico que ha solicitado la *capability* a través de la que se ha llegado (`enter()` o `exit()`) mediante la invocación del servicio privado del aspecto `movingInf()`. Este servicio interroga al ambiente al que está enlazado por las trazas de operaciones y por el tipo del elemento arquitectónico. Una vez obtenidos dichos datos, el ambiente utiliza el *attachment* con que se establece la comunicación con el ambiente destino para solicitarle su *capability* `accept()`.

Es necesario en este punto señalar como el ambiente origen determina que *attachment* ha de usar para enviar la petición del `accept()`, ya que como se ha explicado hay dos puertos `ICapability`, uno con *played role* `EXTERIOR`, con un *attachment* a su ambiente padre, y otro `INERIOR`, con *attachments* a sus ambientes hijos. En función de si el `moving()` es invocado desde un `exit()` o desde un `enter()` tendrá que usar uno o otro. Si es un `exit()`, deberá solicitar el `accept()` al ambiente padre, mediante el puerto con *played role* `EXTERIOR`. Si el `moving()` ha sido invocado desde un `enter()` el `accept()` se enviará a través del

*played role* `INTERNO`. Por ello, es necesario que `moving()` tenga un parámetro en el que se indique la *capability* desde la que se ha invocado.

Esta diferenciación también se produce en el ambiente subyacente cuando ha de realizar las acciones pertinentes a la ejecución del `moving()`. En concreto, cuando recibe confirmación del `accept()` desde el ambiente destino, el Aspecto de Movilidad invoca al servicio `RemoveChild()` que como se comentó en la sección 6.1.2, es un servicio que actúa de forma diferente según el tipo de ambiente, pues cada uno puede contener tipos de elementos arquitectónicos diferentes. Pero antes de eso, solicita el servicio `RecomposeCommChannels()` del ambiente al que está enlazado, para que recomponga los canales de comunicación, de la forma en que se ha explicado en la sección 6.2. Al igual que ocurre para el `RemoveComponent()`, el servicio actúa de forma diferente según si lo sirve un ambiente *Process*, un ambiente *Site* o un ambiente *Logical*.

#### *6.3.1.1. Ambientes Process y Site*

Los ambientes *Process* y los ambientes *Site*, como se ha comentado en secciones anteriores comparten casi toda la funcionalidad interna. El caso de `RecomposeCommChanels()` ocurre lo mismo. Este servicio, recibe desde el Aspecto de Movilidad que lo invoca la información del elemento arquitectónico que se ha movido, la información del ambiente destino al que se ha movido y la *capability* que se ha ejecutado para provocar ese paso de la movilidad. Además, el ambiente tiene implícita la información del nombre del ambiente origen de la movilidad (él mismo) y las trazas de operaciones que se crearon en torno a los *attachments* conectados al ambiente.

El propósito principal de este servicio es reconstruir los canales de comunicación a la salida del elemento arquitectónico, y como añadido, eliminar el *attachment* al puerto `ICapability` del ambiente origen, pues a esas alturas el elemento arquitectónico ya tiene un *attachment* al puerto `ICapability` del ambiente destino.

La funcionalidad del servicio es diferente para el caso de una *capability* `enter()` y el caso de una *capability* `exit()`, por ello la funcionalidad se ha separado en dos servicios diferentes `MovingOnEnterning()` y `MovingOnExiting()`, que serán invocados en función de si `RecomposeCommChannels()` ha sido invocado tras la ejecución de una *capability* `enter()` o `exit()`, respectivamente.

- `MovingOnEntering(requested, newAmbient)`: El servicio, partiendo de la lista de trazas de operaciones, ha de recomponer la configuración de los canales de comunicación en el ámbito del ambiente origen. Para ello, recorre las trazas de operaciones y crea los *attachments* en función de la información que extraiga de ellas. Si una operación de la traza representa a un *attachment* entre el elemento arquitectónico y otro elemento arquitectónico ubicado en el ambiente origen, se creará un *attachment* entre ese elemento

arquitectónico y el puerto de comunicación genérica del ambiente destino; a partir de ahora, el elemento arquitectónico con el que conectaba será accesible a través de ese puerto. En el caso que el elemento arquitectónico al que estaba conectado fuese el propio ambiente destino, este *attachment* no se creará: ese *attachment* representa que el elemento arquitectónico móvil está en comunicación con uno o varios elementos ubicados en el interior del ambiente destino, por lo que ya no es tarea del ambiente origen recomponer dichos canales de comunicación. Para la creación de los *attachments*, comprueba primero si ya existe un *attachment* que cubra ese canal de comunicación. De ser así, obtiene una referencia a él y añade el canal de comunicación obtenido del `TraceOperation` a dicho *attachment*. En caso contrario, crea un nuevo *attachment* con la información extraída de la lista de trazas de operaciones.

- `MovingOnExiting(requested, newAmbient)`: El servicio recorre la lista de trazas de operaciones y para cada información de un *attachment* conectado entre un elemento interno al elemento arquitectónico móvil, ha de crear un canal de comunicación con el puerto de comunicación genérica con *played role* `INTERNO` del ambiente origen. De la misma forma que en el caso anterior, se comprueba si el *attachment* ya existe, en cuyo caso se le añade el canal de comunicación, o en caso contrario se crea un *attachment* nuevo. A partir de ahora, todos los elementos que se comunicaban en el interior del ambiente origen con el elemento arquitectónico móvil deberán hacerlo a través del ambiente en el que se encuentran, ya que al hacer el `exit()` el elemento arquitectónico se ubicará fuera de las fronteras del ambiente.

#### *6.3.1.2. Ambiente Logical*

Los ambientes *Logical*, como se ha comentado, engloban ambientes *Site*, estableciendo, de esa forma unos límites al entorno de ejecución PRISMA, formado por el trabajo colaborativo de los *middlewares* a los que representan los *Sites*.

Cuando un elemento arquitectónico hace un `exit()` de un ambiente *Site*, este pasa a formar parte del ambiente *Logical*. No obstante, el elemento arquitectónico no puede quedarse definitivamente en esa ubicación, pues este tipo de ambientes solo puede albergar ambientes *Site*, aunque, eso si, proporcionan una vía de paso a los elementos que se mueven de un *Site* a otro. Por ello, y debido a la limitación de un nivel de ambientes *Logical*, no tiene sentido que un ambiente *Logical* proporcione un `startTransaction()` o un `endTransaction()` a través de su Aspecto de Movilidad. Si le es solicitado alguno de esos servicios, el ambiente lanzará una excepción cuando el Aspecto de Movilidad le transmita la petición.

Centrando la discusión en el ambiente *Logical* como una vía de paso de los elementos arquitectónicos en su movilidad de un *Site* a otro es necesario hacer dos consideraciones. Por un lado, que es necesario que el ambiente *Logical* disponga de una forma de albergar elementos de paso, pese a su restricción de únicamente ubicar ambientes *Site* y la otra es que en algún momento, durante ese paso por el ambiente *Logical*, el elemento arquitectónico ha de ser serializado a través de la red (para lo que se utilizarán los mecanismos de serialización que proporciona la capa *Distribution Services* del *middleware*) para ubicarse efectivamente en el *middleware* destino, es decir, en aquel representado por el *Site* al que hace un `enter()`.

La primera consideración, se resuelve con la existencia del campo de la clase `AmbientLogical` llamado `tempFolder`, en el cual se guardará la referencia del elemento arquitectónico de paso. Este campo es accedido por los servicios de búsqueda de elementos como si fuese una lista más de elementos que puede albergar el ambiente *Logical*, aunque en realidad no sea un elemento que pueda ubicar, según las restricciones del metamodelo. No obstante, se sigue ese planteamiento para conseguir la transparencia entre los diferentes tipos de ambientes de la que se ha hablado con anterioridad. Además, este campo solo tendrá elementos durante la ejecución de una transacción (la que define el proceso de movilidad), y en el momento que esta termine, el campo siempre estará vacío, puesto que no se puede terminar la transacción con un elemento arquitectónico que no sea un ambiente *Site* ubicado en el ambiente *Logical* (de hecho, como se acaba de decir, el intento de ejecución de `finishTransaction()` lanzaría una excepción). Es como un estado no valido que se alcanza dentro de los límites de una transacción, pero que no se persiste.

La segunda consideración, presenta algo más de complejidad; puesto que el ambiente *Site* destino no se sabrá hasta que el elemento arquitectónico ejecute el `enter()`, no se podrá hacer nada hasta que comience la ejecución de dicha *capability*. En el momento que el elemento arquitectónico solicite el `enter()`, el Aspecto de Movilidad realizará las tareas pertinentes: hacer las comprobaciones, obtener la información del elemento arquitectónico que se va a mover, enviar el `accept()` al ambiente *Site* destino y, finalmente recomponer los canales de comunicación por mediación del servicio `RecomposeCommChannels()` del Ambient *Logical* subyacente.

Este servicio del `AmbientLogical`, como se ha visto, redirigirá la petición al `MovingOnEntering()` o al `MovingOnExiting()`, en función de la *capability*. En la versión actual, al solo haber un ambiente *Logical*, no tiene sentido la ejecución de `MovingOnExiting()`, pues nadie va a salir de él. En el caso ed `MovingOnEntering()`, el servicio no se dedicará exclusivamente a la recomposición interna del canal de comunicación para adaptarse a los cambios producidos por la salida del elemento arquitectónico, como era en el caso de ambientes *Process* y *Site*. En este caso, además de eso, el

`MovingOnEntering()` ha de mover físicamente, es decir, serializar al elemento arquitectónico con el objetivo que, tras la ejecución del `enter()`, el elemento no sólo se encuentre en las listas del ambiente destino, y que tenga como `location` en su aspecto de Distribución al ambiente destino además de un *attachment* al puerto `ICapability` de éste, sino que se encuentre físicamente ejecutándose en el mismo *middleware* que el ambiente *Site* destino. Este es, al fin y al cabo, el objetivo de la movilidad.

En este punto, el comportamiento del ambiente *Logical* respecto a la ejecución del `moving()` y su relación con la serialización está clara. Pero su implementación resulta más compleja de lo que parece. El problema reside en que si el ambiente *Logical*, por mediación de la capa *Distribution Services*, intenta mover al elemento arquitectónico durante la ejecución del `enter()`, la movilidad nunca será efectiva, produciéndose un interbloqueo. Esto es así, porque al mover un elemento arquitectónico, la capa *Distribution Services* lo detiene, puesto que el *middleware* como ya se ha comentado no proporciona *strong mobility*. Cuando un elemento arquitectónico es detenido en PRISMA, se detienen sus aspectos, y para que ello ocurra, se ha de esperar a que los hilos de ejecución terminen sus respectivas tareas. Por tanto, si el elemento arquitectónico está ejecutando una transacción (la de movilidad), a la cual aún le quedan servicios por procesar (como mínimo uno, el `finishMovement()`) es imposible que se detenga. Lo que quiere decir que el hilo de ejecución del aspecto no terminará hasta que no termine la transacción, y la transacción no terminará hasta que el elemento llegue a su destino, al que, por otro lado, nunca llegará si no termina el hilo de su aspecto y deja que se mueva.

Para atajar este problema directamente, seria necesario el poder partir las transacciones, para lo cual habría que modificar su modelo de ejecución. Por ejemplo, se podría separar la ejecución de cada uno de los estados de una transacción PRISMA a un servicio, de forma que se pudiesen ejecutar de forma modular, y de esa manera, al final de un determinado paso, detener el aspecto para mover al elemento arquitectónico, y en destino reanudar la ejecución por el servicio que correspondiese al servicio siguiente de la transacción. Pero para que esta solución funcionase correctamente, es necesario modificar el gestor de *weavings* para que también pudiese partir su ejecución, de forma que se pudiese detener y reanudar el elemento arquitectónico al tiempo que se está ejecutando un *weaving*. Debido a la complejidad de esta solución, se ha optado dejar las modificaciones pertinentes para futuras versiones en las que se mejore el modelo de movilidad de los ambientes aplicando estos cambios y otros. En esta versión se ha optado por una solución más sencilla que consigue atajar el problema sin realizar grandes cambios en el *middleware*.

Puesto que todos los procesos de movilidad se realizan en el ámbito de una transacción y las transacciones se gestionan desde el gestor de transacciones, la idea reside en apuntar en el contexto transaccional. Para ello se ha añadido una referencia a la estructura `MovingInfo` en los

`TransactionalContext` (ver Figura 87). La estructura contiene información sobre el componente que se ha de mover y la localización a la que se ha de mover. El Aspecto de Movilidad, cuando detecta que se trata del `moving()` de un ambiente *Logical* (mediante el servicio `GetAmbientType()`) avisa de este hecho al *Transaction Maneger* mediante su `aspectCareTaker` y el servicio `SetMovingInfo(txID, targetSite, component)` que encapsula la lógica de localización de contexto transaccional en el que se ha de registrar la movilidad, creación del `LOC` desde el nombre del ambiente destino (haciendo uso de la capa DNS).

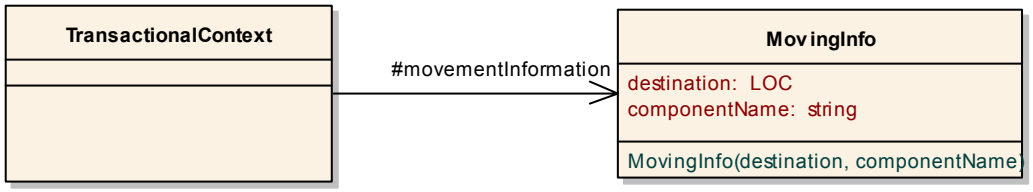



**Figura 87: Estructura MovingInfo**

De esa forma, la transacción proseguirá asumiendo que el elemento arquitectónico está en destino, aunque en realidad no lo esté. La transparencia de la comunicación distribuida mediante *attachments* conseguirá que el elemento arquitectónico pueda seguir enviando *capabilities* a los ambientes padre, aunque no se encuentre físicamente en la misma máquina que ellos. En cierta manera se encontrará virtualmente, ya que para los ambientes por los que pase tras la supuesta serialización, el elemento arquitectónico será uno de sus hijos, y el *attachment* al Ambiente Padre se irá recreando a cada paso. Cuando la transacción que englobe el proceso de movilidad se confirme, el Aspecto del Distribución del *attachment*, mediante su `aspectCareTaker` notificará al *Transaction Manager* que la transacción ha finalizado. El *Transaction Manager* comprobará la consistencia y antes de propagar el lanzado de eventos *commit* en todos los *middlewares* involucrados comprobará si hay pendiente una operación de movilidad. En el caso que la haya, solicitará a la capa *Distribution Services* que la realice. Puesto que el Aspecto ya ha finalizado en ese momento su ejecución de la transacción, podrá ser detenido. La serialización, por tanto se llevará a cabo. En la Figura 86 se muestra un esquema del funcionamiento expuesto.

Con el elemento arquitectónico se moverán los Mementos creados durante la transacción. Una vez llegado a destino, el elemento ya se encuentra en el *middleware* adecuado, con sus *attachments*, anteriormente distribuidos, ahora ya son locales y siguen manteniendo la semántica adecuada. En ese momento, el *Transaction Manager* ya puede propagar el *commit*: la transacción de movilidad ha finalizado.

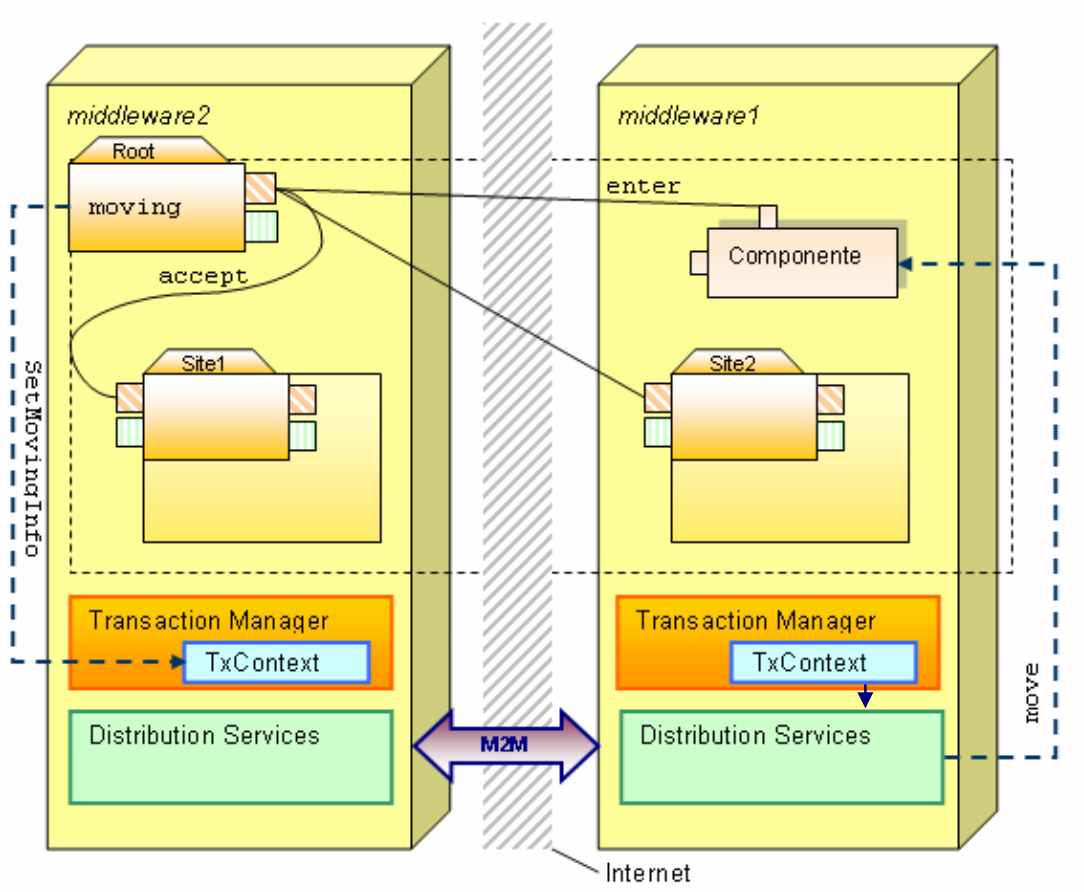



**Figura 88: Serialización en los ambientes Logical**

Nótese como lo que está ocurriendo es que el elemento arquitectónico se marca para moverse, pero no se mueve hasta que no termina la ejecución de los servicios. De esa forma, se van creando los *attachments* con el resto de puertos `ICapability` de los ambientes por los que pase en la máquina destino, como si estuviese en local, y finalmente, tras la ejecución del `finishMovement()`, se crearán todos los *attachments* de nuevo que conectan con el elemento arquitectónico, que serán distribuidos, pues estará en otra máquina. Cuando se mueva físicamente, al final de la transacción, los *attachments* se moverán con él, convirtiéndose en locales. Esta aproximación, además, aporta la ventaja de que si algo va mal, no es necesario deshacer una serialización, puesto que el elemento arquitectónico no se serializa hasta el final.

No obstante, la solución presenta también algunas limitaciones. Por ejemplo, solo es posible definir una serialización (`MovingInfo`) en la capa *Transaction Manager*, por lo que no es posible definir una transacción que suponga una ida y vuelta. Esto no tiene sentido, porque si el movimiento se produce al final y la movilidad requiere moverse a un destino, realizar unas operaciones y volver al inicio, con el modelo actual no se podría detener el aspecto hasta el final de la transacción, por lo que al final ya no se ganaría nada haciendo la movilidad.

En cualquier caso, el propósito de este proyecto es dar un primer paso para sentar las bases de la implementación de Ambient-PRISMA, por lo que es necesario seguir trabajando en esta línea para mejorar los mecanismos de movilidad y hacerlos más eficientes.

A continuación, se explica el funcionamiento del servicio `RecomposeChannels()` en un ambiente *Logical*. Al igual que en el caso de los ambientes *Process* y *Site*, también discrimina la funcionalidad en el caso que se ejecute a expensas de una *capability* `enter()` o una *capability* `exit()`.

- `MovingOnEntering(requested, newAmbient)`: De nuevo partiendo de la lista de trazas de operaciones, el servicio recompone la configuración de los canales de comunicación en el ámbito del ambiente origen. Para ello, recorre las trazas de operaciones y crea los *attachments* en función de la información que extraiga de ellas. En este caso los *attachments* que creará serán siempre entre *Sites*, por lo que comprobará si ya existe un *attachment* establecido entre los *Sites*, en cuyo caso le añadirá la información sobre el canal de comunicación. En caso de no existir ningún canal de comunicación previo, creará el *attachment*. De nuevo, si la información de la traza de operaciones le indica que el *attachment* conectaba el elemento arquitectónico que ha salido con el ambiente al que ha entrado, no se creará ningún *attachment*.

- `MovingOnExiting(requested, originAmbient)`: Puesto que se ha limitado el modelo para que tan solo haya un nivel de ambientes *Logical*, es decir, únicamente un ambiente *Root*, este servicio no se ejecutará nunca, pues nunca un elemento arquitectónico podrá solicitar un `exit()` de un ambiente *Logical*. No obstante se ha implementado, aunque en su cuerpo únicamente lance una excepción, de cara a próximas ampliaciones.

### 6.3.2. Servicio Accept

El servicio `accept()` es el que se encarga de la movilidad en la parte del ambiente destino. Mediante este servicio, un ambiente entra en la transacción que define la movilidad, recibiendo toda la información necesaria para añadir al elemento móvil a su listado.

Este servicio proporcionado por el Aspecto de Movilidad solicita el añadido del elemento arquitectónico al ambiente subyacente mediante el servicio `AddChild()`. Este servicio traspasa la información recibida al ambiente que hay por debajo, de forma que éste recibe el nombre del elemento arquitectónico que ha de añadir, el tipo de dicho elemento arquitectónico así como la lista de trazas de operaciones del ambiente origen. Con esta información el ambiente añade el elemento a su lista interna, de acuerdo con las restricciones del tipo de

ambiente, y genera una nueva lista de trazas de operaciones en función de la recibida.

El servicio también recibe por un parámetro la *capability* mediante la cual ha sido invocado. Esto es así porque en función de si el elemento arquitectónico que está recibiendo fruto de la movilidad tiene como origen un ambiente padre (`enter`) o un ambiente hijo (`exit`) del ambiente destino (el que recibe el `accept`) se ha de actuar de un modo u otro para regenerar la lista de trazas de operaciones. En este proceso de generación de la lista de trazas de operaciones se crean también el *attachment* entre el puerto `ICapability` del ambiente destino y el puerto de movilidad del elemento arquitectónico, cuya identificación se extrae también de la lista de trazas de operaciones.

Además, cuando procesa un `accept()`, el Aspecto de Movilidad del Ambiente se encarga de invocar al servicio `ChangeLocation()` del elemento arquitectónico móvil mediante su recién creado canal de comunicación con el puerto de movilidad del elemento arquitectónico. De esa forma, el elemento arquitectónico actualiza ya el valor del campo `location` de su Aspecto de Distribución a la nueva ubicación.

En cualquier caso, a nivel interno del ambiente que subyace, la lógica de la movilidad la encapsula el servicio `AddChild()`. A continuación se describen las diferencias entre la ejecución del `AddChild()` para los diferentes tipos de ambientes.

#### *6.3.2.1. Ambientes Process y Site*

De nuevo los ambientes *Process* y *Site* comparten la funcionalidad interna en lo que respecta a las tareas necesarias para hacer efectiva la incorporación de un elemento arquitectónico al ambiente en el que se ejecutan. Mediante el servicio `Addchild()`, el Aspecto de Movilidad del Ambiente avisa al ambiente al que está agregado que se ha de añadir un elemento arquitectónico. Mediante esta llamada le proporciona el nombre y tipo del elemento arquitectónico que se va a añadir al ambiente, así como la lista de trazas de operaciones. También, informa de la *capability* a la que responde la invocación. De nuevo, es importante señalar la *capability* que se ha ejecutado para que el ambiente destino (el que se está ejecutando el `accept`) sea capaz de recomponer correctamente la lista de trazas de operaciones.

El propósito fundamental del servicio `AddChild()` es hacer que el elemento arquitectónico móvil se ubique en el ambiente. Para lo cual, añadirá el elemento a la lista adecuada, creará la lista de trazas de operaciones con la que se trabajará en sucesivos pasos de la movilidad y, también muy importante, será el encargado de crear el *attachment* entre el puerto de movilidad del elemento arquitectónico (por el que solicita las *capabilities*) y el puerto `ICapability` del que, a partir de ese momento, será su Ambiente Padre.

Como se ha dicho, esta funcionalidad es ligeramente diferente en función del tipo de movimiento que se está haciendo. Por ello se pasa la información sobre la *capability* que se ha ejecutado y en función de eso, `AddChild()` llama a uno de los siguientes servicios:

- `AcceptOnEntering(newAmbient, requested, origTraceOp)`: El servicio creará una lista de trazas de operaciones en función de la lista de trazas de operaciones recibida del ambiente origen y del hecho que la movilidad ha sido de entrada, es decir, el ambiente origen es su Ambiente Padre. Lo primero que hace es obtener de la lista de trazas de operaciones recibida la información del *attachment* que conecta al elemento arquitectónico móvil con el ambiente origen. De esa elemento extrae la información necesaria (el puerto de movilidad del elemento arquitectónico móvil) para crear el *attachment* con el puerto `ICapability`, por el que a partir de ese momento, el elemento arquitectónico solicitará las *capabilities* al nuevo ambiente. Después de eso, recorre el resto de trazas de operaciones y para cada una de ellas genera una nueva entrada en su lista `traceOperations`, en función del tipo de *attachment* que represente. Si es un *attachment* que conectaba el elemento arquitectónico con un rol *parent*, es decir, con el ambiente origen, creará una anotación para la creación de un *attachment* con el puerto de comunicación genérica del nuevo ambiente. Si representa a un *attachment* con rol *sibling*, quiere decir que conectaba a un elemento arquitectónico ubicado en ambiente origen. Como ambiente destino también está ubicado en el ambiente origen, se comprueba si se trata del ambiente destino. Si no es el ambiente destino, de nuevo se creará una anotación para el puerto de comunicación genérica con *played role* `INTERNO` del nuevo ambiente; el elemento al que conectaba dicho *attachment*. Si se trata del ambiente destino y el puerto es el de comunicación genérica es porque el elemento arquitectónico móvil se ha de comunicar con algún elemento que está en el interior del ambiente destino. En ese caso se localiza el canal de comunicación en el interior del ambiente y se actualiza, para que deje de comunicarse con el elemento arquitectónico ubicado en el exterior; como se ha movido al interior del ambiente, ahora se comunicará con un *attachment* de rol *sibling*, que es anotado en la lista de trazas de operaciones.

- `AcceptOnExitying(originAmbient, requested, origTraceOp)`: Este servicio realiza las acciones relativas a la ubicación de un elemento arquitectónico móvil a un ambiente provocada por la ejecución de una *capability* `exit()`. Es decir, el elemento arquitectónico proviene de un ambiente que es hijo del ambiente destino. De nuevo, lo primero que se hace es crear el *attachment* entre los puertos de movilidad para que el elemento arquitectónico pueda solicitar las *capabilities* al nuevo ambiente. Después, se recorre la lista de información de los *attachments* del ambiente origen que se ha

recibido para anotar las operaciones de creación de *attachments* en la lista `traceOperations` del nuevo ambiente. Para los *attachments* que conectaban al elemento *arquitectónico* con un rol *sibling*, se ha de anotar un *attachment* al puerto de comunicación genérica del ambiente origen con *played role* `EXTERNO`, pues los elementos que se conectaban estarán en su interior. Para el caso de una comunicación de tipo *parent*, el destino de la comunicación se encontrará en el ambiente destino, por lo que se localiza el canal de comunicación en el nuevo ambiente y se anota la creación de un *attachment* entre el elemento arquitectónico móvil y el destino del *attachment* localizado.

#### *6.3.2.2. Ambiente Logical*

Respecto a los ambiente *Logical*, la ejecución del `accept()` supone que un elemento arquitectónico ha salido de un ambiente *Site* y, por tanto, en el paso siguiente va a entrar en otro. Entre los dos pasos, el elemento arquitectónico se va a quedar temporalmente ubicado en el ambiente *Logical*. Como se ha justificado anteriormente, esta estancia no puede ser más que temporal.

La implementación centralizada del Ambient *Logical* no supone ningún inconveniente a esto. El elemento arquitectónico, una vez salga del *Site* que lo ubicaba se quedará físicamente en el mismo *middleware* en el que se encontraba. Eso si, la referencia a su padre tras el paso de movilidad apuntará al ambiente *Logical*, al igual que su *attachment* para la movilidad (ver Figura 89).

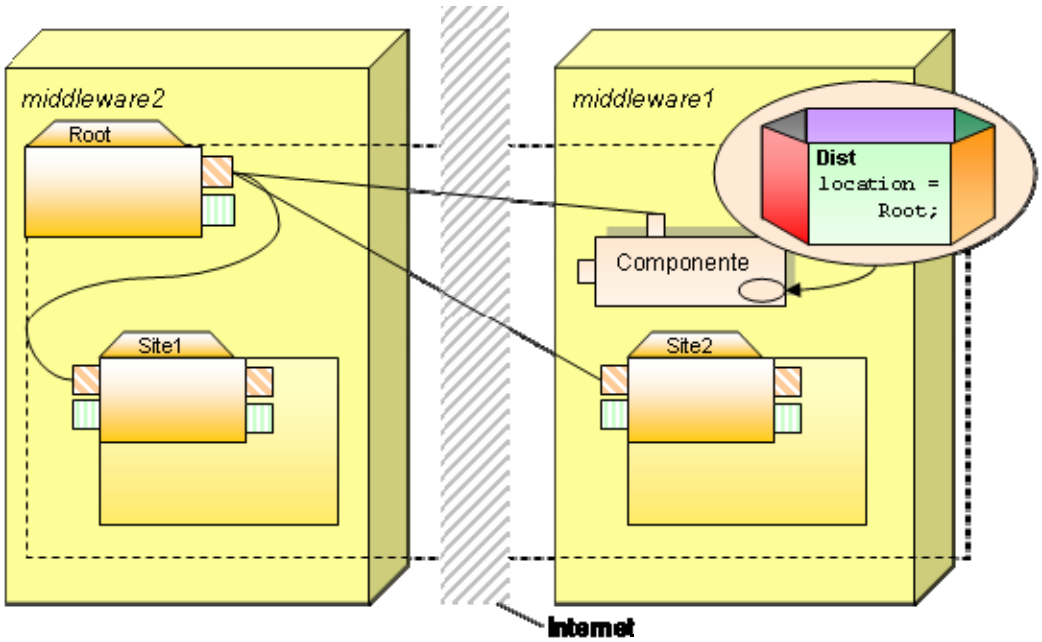



**Figura 89: Componente en un paso intermedio de la movilidad**

En este caso la implementación del `AddChild()` es similar a la de los otros tipos de ambientes: de la misma forma separa su funcionalidad dependiendo del la *capability* solicitada en los servicios `AcceptOnEntering()` y `AcceptOnExiting()`. Como ocurría en el caso del `MovingOnExiting()` para los ambientes *Logical*, la funcionalidad no está implementada puesto que es un caso que nunca se dará, debido a las limitaciones de la primera versión.

En lo que respecta al `AcceptOnEntering()`, la funcionalidad es similar a la expuesta para los ambientes de tipo *Logical* y *Site*. El servicio encapsula la lógica para la creación del *attachment* con el puerto `ICapability` que en este caso, puede ser un *attachment* distribuido (ver Figura 89). Por lo demás, crea la lista de trazas de operaciones siguiendo los mismos planteamientos que en el caso de los *Logical* y *Sites*.

### 6.3.3. Otras consideraciones

Ya se han explicado en detalle los diferentes servicios que entran en juego en un paso de movilidad. El conjunto de todos los pasos de movilidad, incluyendo la llamada a `startMovement()` y a `finishMovement()` conforman el proceso de movilidad completo. Además, se ha detallado la importancia de separar la funcionalidad de los servicios que realizan los ambientes para proporcionar la movilidad en función de la *capability* que se haya solicitado (ver Figura 90).

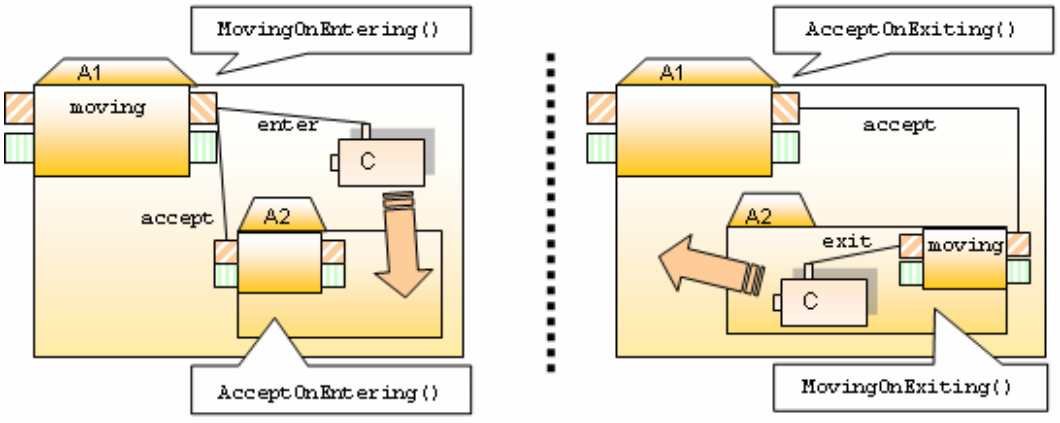



**Figura 90: Servicios implicados en un proceso de movilidad**

Pero, como se ha visto para proporcionar la movilidad entre ambientes *Site* un ambiente *Logical* ha de recurrir a la capa de *Distribution Services*, en concreto a su servicio `Move(component, newLocation)` para que serialice realmente al elemento arquitectónico móvil. El funcionamiento de este servicio ha sido explicado en la sección en la que se mostró el modelo de movilidad de PRISMANET (Capitulo 3). No obstante, el servicio citado, al reutilizarse de su implementación en PRISMANET sólo está preparado para mover elementos arquitectónicos simples, es decir, componentes o conectores.

En Ambient-PRISMA, el concepto de ambiente *Process* es especialmente útil, entre otras cosas, para la movilidad conjunta de todos los elementos arquitectónicos a los que proporciona ubicación. Por ello, es necesario ampliar el servicio `Move()` de la capa *Distribution Services* para que sea capaz de mover ambientes *Process* junto a todos los elementos internos, que a su vez, pueden ser otros ambientes *Process*.

La manera en que se ha abordado esta situación supone una solución sencilla. Puesto que el servicio que serializa los elementos arquitectónicos requiere un nombre de elemento arquitectónico y un destino, basta con comprobar si el nombre pertenece a un ambiente *Process*. Esto lo hará la capa *Distribution Services* en colaboración con la capa *Element Management*. Si es así, la capa *Distribution Services*, obtendrá la lista de los elementos arquitectónicos que están ubicados en el ambiente que se ha de mover. Esto lo obtendrá directamente del ambiente *Process*. Para cada uno de esos elementos se invocará de nuevo al servicio `Move()` de *Distribution Services* pasándole como parámetro el nombre del elemento arquitectónico. Si el elemento arquitectónico es otro ambiente *Process*, el proceso se repetirá para los elementos que tenga ubicados también.

De esa forma, uno a uno, los elementos internos se irán moviendo a la máquina destino. Puesto que la capa de distribución ya se ocupa de gestionar la movilidad de los *attachments*, no es necesario modificar nada en ese sentido: los elementos se irán moviendo uno a uno a la máquina destino junto a sus *attachments* que pasarán de distribuidos a locales a medida que vayan llegando a destino el resto de elementos arquitectónicos. Cuando se hayan movido todos los elementos internos al ambiente *Process*, entonces es cuando se procederá a la serialización del ambiente.

Como se puede ver, la idea es mover de forma secuencial y mediante el `Move()` del *Distribution Services* los elementos internos a un ambiente *Process*, siendo el ambiente el último elemento que se moverá.

CAPÍTULO VII

# CONCLUSIONES Y TRABAJOS FUTUROS

## Contenidos del capítulo

# CONCLUSIONES Y TRABAJOS FUTUROS

En el capítulo actual, se presentan las conclusiones y los trabajos futuros. En la primera sección, la de las conclusiones, se resumirá brevemente cómo se han alcanzado los objetivos del proyecto y cuáles han sido los obstáculos que se han tenido que salvar para su consecución. En la segunda sección, los trabajos futuros, se da una visión de las tareas que se han de implementar en un futuro próximo tomando como punto de partida los resultados obtenidos en el presente trabajo.

## 7.1. Conclusiones

En este proyecto se han llevado a cabo dos tareas bien diferenciadas, pero relacionadas, por lo que se puede afirmar que el proyecto está dividido en dos partes. En una primera parte se ha ampliado el modelo de distribución y movilidad del *middleware* PRISMANET y se le ha dotado de un mecanismo para la ejecución de transacciones. En una segunda parte, se ha propuesto una aproximación a la implementación de las primitivas de movilidad promovidas por el Ambient-PRISMA.

Respecto a la ampliación del modelo de distribución del *middeware*, se han detectado algunas carencias en la implementación previa que comprometían la independencia de las entidades en ejecución. Puesto que el modelo PRISMA se basa en gran medida en la definición de entidades independientes para de esa forma garantizar su reutilización, se estaba perdiendo la trazabilidad de los conceptos del modelo a la implementación. Tras un estudio detallado de la problemática, se ha solventado el problema mediante la utilización de los patrones de diseño. De esa forma se ha conseguido un modelo más eficiente, donde las entidades de comunicación actúan con mayor independencia. La utilización de patrones de diseño ha dado pie a una abstracción para los canales de comunicación en la implementación, por lo que se han podido implementar soluciones globales para los diferentes tipos de canales de comunicación que se pueden definir en PRISMA.

En el desarrollo del proyecto, también se ha estimado necesario un mecanismo para la gestión del entorno distribuido. De forma similar a lo que ocurre en otros *frameworks* distribuidos, era necesario que PRISMANET dispusiese de la forma de conocer los nodos entre los que se propaga una configuración arquitectónica, es decir, el entorno de ejecución. También era

importante proporcionar una manera para añadir nuevos nodos, eliminar existentes o detectar fallos. Para ello, y tras estudiar los conceptos de las técnicas de tolerancia a fallos, se ha diseñado un gestor para el entorno de ejecución, relegado a una capa del *middleware*, que se encargue de mantener una lista de *middlewares* consistente. Gracias a este gestor de la Lista de Middlewares Conocidos, se ha podido distribuir el DNS centralizado y modificarlo para conseguir una versión simplificada del mismo, con la misma funcionalidad y así tener las ventajas de un DNS distribuido con un diseño más sencillo.

Además de las mejoras en el modelo de ejecución distribuido, era necesaria la incorporación de una nueva característica al *middleware* que había quedado pendiente en anteriores implementaciones. Se trata del soporte para transacciones PRISMA. La implementación de dicha característica parte de un estudio detallado de las implicaciones de la ejecución transaccional en el modelo arquitectónico distribuido PRISMA, seguido de un estudio sobre cuáles son las tecnologías que, en este caso, el *framework* .NET, proporcionaba para tal efecto. Debido a las particularidades del modelo PRISMA, y que las soluciones ofrecidas por .NET tenían difícil adaptación a éstas, se ha optado por una solución *ad-hoc*. Esta solución, basándose en las características de otros modelos transaccionales, y usando de nuevo los patrones de diseño, se ha diseñado de forma personalizada y no dependiente de la plataforma (.NET) para el modelo PRISMA. De esa manera, se ha proporcionado un modelo transaccional que trata con aspectos y elementos arquitectónicos, así como con configuraciones arquitectónicas distribuidas, proporcionando una solución global a las necesidades del modelo.

Una vez realizados los cambios descritos en los párrafos anteriores se disponía de un *middleware* preparado para adaptarse a las necesidades de las nuevas primitivas del Ambient-PRISMA. Esta nueva aproximación, basada en primitivas del metamodelo que permiten el modelado de la distribución y movilidad desde las etapas más tempranas del desarrollo, requería un entorno de ejecución altamente distribuido y con soporte a transacciones, por ello, era necesario ampliar PRISMANET antes de abordar la implementación de los conceptos del Ambient-PRISMA. El estudio de esta implementación conformaría la segunda parte del proyecto.

El último capítulo de contenidos del proyecto muestra como se han transformado los conceptos del Ambient-PRISMA a la implementación, añadiendo las nuevas primitivas al *middleware*. En este punto se han planteado algunos problemas cuyo análisis ha sido interesante. Por un lado, ha surgido la necesidad de establecer un mecanismo capaz de automatizar en la medida de lo posible la creación de canales de comunicación, más complejos que en PRISMANET, así como su reconfiguración automática durante la movilidad. La solución ha surgido tras un profundo estudio del modelo Ambient-PRISMA y de las necesidades de comunicación durante la movilidad. Además de ello, se ha presentado el nuevo modelo de ejecución de la movilidad, adaptando el existente en PRISMANET a las nuevas características de movilidad y

distribución de Ambient-PRISMANET. Las diferentes estrategias para resolver estos problemas son explicadas a lo largo del capítulo. Como resultado se ha obtenido una primera versión de la implementación de los conceptos de Ambient-PRISMA en .NET en la que se han abordado algunos de los conceptos más importantes.

Como fruto del trabajo realizado en este proyecto, se ha obtenido las siguientes publicaciones en congresos internacionales:

- Cristóbal Costa, Nour Ali, **Carlos Millán**, Jose A. Carsí, *"Transparent Mobility of Distributed Objects using .NET"*, 4th International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2006.

- Nour Ali, **Carlos Millán**, Isidro Ramos, *" Developing Mobile Ambients using an Aspect-Oriented Software Architectural Model"*, Proceedings of The 8th International Symposium on Distributed Objects and Applications (DOA), LNCS, Springer Verlag, Montpellier, France, Oct 30 - Nov 1, 2006.

## 7.2. Trabajos futuros

Uno de los objetivos del proyecto era dotar al *middleware* PRISMANET de soporte para la ejecución de servicios transaccionales distribuidos. Como resultado, se ha obtenido una primera versión funcional que es capaz de gestionar operaciones transaccionales, anidadas y distribuidas. No obstante, se trata de una primera versión. En futuras versiones se deberá mejorar el modelo de ejecución de las transacciones, adoptando una política que no bloquee totalmente el aspecto, sino los campos involucrados. Del mismo modo, se ha detectado que la movilidad de un elemento arquitectónico durante el transcurso de una transacción presenta algunos problemas. Esto es debido a la combinación del bloqueo transaccional del aspecto con la movilidad débil que proporciona PRISMANET, que requiere la parada del elemento para su movilidad. Esto puede ser atajado mediante una factorización de los servicios transaccionales, de forma que un aspecto pueda interrumpir temporalmente su ejecución en una transacción (para moverse) y continuar la ejecución de la transacción en destino.

Respecto a la versión de Ambient-PRISMANET, aún se encuentra en una fase temprana y no se puede dar el trabajo por concluido. Futuras versiones han de mejorar la eficiencia del prototipo y proponer soluciones para la descentralización del elemento arquitectónico ambiente *Logical*, distribuyendo el elemento arquitectónico y sus aspectos. Además de eso, se deberá ampliar la expresividad de la plataforma de ejecución de modelos eliminando la limitación de la jerarquía de ambientes *Logical*; en futuras versiones se ha de poder poner en ejecución tantos niveles de ambientes *Logical* como el diseñador estime necesario para modelar su sistema.

Finalmente, aunque no menos importante, uno de los trabajos que queda por realizar es el desarrollo de un compilador que transforme las especificaciones Ambient-PRISMA en .NET siguiendo las pautas que se han propuesto en este proyecto. El desarrollo de este compilador será una ampliación del compilador de especificaciones PRISMA de que se dispone, añadiendo los nuevos conceptos a su metamodelo y plantillas de generación de código para la obtención del código *C#* asociado a la especificación.

# BIBLIOGRAFÍA


[Ali03] Ali N.H., Silva J., Jaen J., Ramos I., Carsí J.A., Pérez J. *Mobility and Replicability Patterns in Aspect-oriented Component-Based Software Architectures.* In Proc. of 15th IASTED, Parallel and Distributed Systems, Acta Press, ISBN: 0-88986-392-X, ISSN: 1027-2658, pp. 820-826. Marina del Rey, C.A., USA, Noviembre 2003.

[Ali04] Ali, N., Pérez, J., Ramos, I. *High level Specification of Distributed and Mobile Information Systems*. In Proc. of the Second International Symposium on Innovation in Information & Communication Technology (ISSICT2004), 21-22 Abril, 2004, Amman, Jordan, 2004.

[Ali05] Ali, N., Ramos, I., Carsí, J.A. *A Conceptual Model for Distributed Aspect Oriented Software* Architectures, International Conference on Information Technology (ITCC 2005), IEEE Computer Society, ISBN 0-7695-2315-3, April 2005, pp 422-427.

[Ali05b] Ali, N., Pérez, J., Ramos, I., Carsí, J.A. *Introducing Ambient Calculus in Mobile Aspect-Oriented Software Architectures* In Proc. of the Fifth Working IEEE/IFIP Conference on Software Architecture (WICSA), © IEEE Computer Society Press, Pittsburgh, Pennsylvania, USA, 6-9 November, 2005 (position paper)

[Ali06] Ali, N., Millán, C., Ramos, I. *Developing Mobile Ambientes using an Aspect-Oriented-Software Architectural Model,* In Proc of the Distributed Objects and Applications (DOA 2006), LNCS, October 30-November , 2006, Montpellier, France.

[Ali06b] Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J.A. *Mobile Ambients in Aspect-Oriented Software Architectures*. In Proc. of the IFIP Working Conference on Software Engineering Techniques, LNCS ISSN: **1571-5736,** October 18-20, 2006,Warsaw, Poland

[Ali06c] Ali, N., Pérez, J., Costa, C., Ramos, I., Carsí, J.A. *Replicación Distribuida en Arquitecturas Software Orientadas A Aspectos Utilizando Ambientes*, XI Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Octubre 2006, Sitges, Barcelona, España.

[Aosd] *Aspect-Oriented Software Development*, http://aosd.net

[AsJ03] The AspectJ Team. *AspectJ Programming Guide*, 2003.

[AsJ04] AspectJ Project, en http://eclipse.org/aspectj/

[Bal85] Balzer R., *A 15 Year Perspective on Automatic Programming*. IEEE Transactions on Software Engineering, vol.11, num.11, págs. 1257-1268, Noviembre 1985.

[Bau00] Baude F. Caromel D. Huet F. Vayssière J. *Communicating Mobile Active Objects in Java* In Proc. of the 8th International Conference - High Performance Computing Networking'2000 (HPCN Europe 2000), LNCS vol. 1823, pp 633--643, 2000.

[Bau00b] Baumer C. Breugst M. Choy S. Magedanz T. *Grasshopper: a universal agent platform based on OMG MASIF and FIPA standards*. Technical report, IKV++ GmbH, 2000

[Be01] L. Bettini L., De Nicola R. *Translating Strong Mobility into Weak Mobility*. In Proc. of the Fifth IEEE Int. Conf. on Mobile Agents. 2001

[Bern96] Bernstein, P.A. *Middleware: a model for distributed system services*. Communications of the ACM, Volume 39, Issue 2, February 1996 ISSN:0001-0782, p 86-98

[Bur03] Burbidge R., Macleod R., Sutton M., Wheelright S. y Wigley A. *Microsoft® .NET Compact Framework (Core Reference)* Microsoft Press. 2003

[Ca05] Cabedo R. *Aplicación Del Lenguaje De Descripción De Arquitecturas Prisma A Un Sistema Robótico De Ámbito Industrial*. PFC UPV, 2005

[Car97] Cardelli, L. *Mobile Ambient Synchronization* SRC Technical Note Report 1997-013, Digital SRC, 1997.

[Car98] Cardelli, L., Gordon, A. D. *Mobile Ambients*, Foundations of Software Science and Computational Structures: First International Conference, FOSSACS '98, LNCS 1378, Springer, 1998, pp. 140-155.

[Car98b] Cardelli, L. *Abstractions for Mobile Computation* In Vitek, J. and (Eds.), C. J., editors, Secure Internet Programming: Security Issues for Distributed and Mobile Objects, volume 1603 of LNCS, Springer Verlag, pp. 51-94. 1998

[Cha03] Chaumette, S. and Vignéras, P. *A Framework for Seamlesly Making Object Oriented Applications Distributed*. In International conf. on Parallel Computing (PARCO 2003): 305-312, 2003.

[Chess95] Chess, D., Harrison, C., Kershenbaum, A. *Mobile Agents: Are They a Good Idea?* IBM Research Report RC. 1995

[Cle97] Clements P. Papaioannou T. Edwards J *Aglets : Enabling the Virtual Enterprise* In Proc. of the Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement Intl. Conference (MESELA '97), Loughborough University, UK, 1997

[Con04] Contreras M. Germán E. Chi M. Sheremetov L *Design and implementation of a FIPA compliant Agent Platform in .NET* In Journal of Object Technology, vol. 3, no. 9, October 2004, Special issue: .NET Technologies 2004 workshop, pp. 5-28, 2004

[CORBA] Página Web Oficial de OMG CORBA: http://www.corba.org/

[Cor01] Cormen T., Leiserson C., Rivest R., Stein C. *Introduction to Algorithms, Second Edition*. MIT Press and McGraw-Hill, 2001. ISBN 0262032937. Section 22.3: Depth-first search, pp.540–549.

[Cos05] Costa C., *Estudio E Implementación De Un Modelo De Arquitecturas Orientado A Aspectos Y Basado En Componentes Sobre Tecnología .Net*. PFC UPV, 2005

[Cu97] Cugola G., Ghezzi C., Picco G., Vigna G., *Analyzing Mobile Code Languages*, Mobile Object Systems, Lecture Notes in Computer Science, No. 1222, Springer-Verlag (D), pp. 94-109, February 1997.

[dotNET] *Microsoft .NET Developer Center*
http://msdn.microsoft.com/netframework/

[Eli00] Eliëns, A. *Principles of Object-Oriented Software Development*. Addison-Wesley (2000), ISBN 0-201-39856-7.

[FIPA] FIPA Web Oficial: http://www.fipa.org

[Free04] Freeman, E., Freeman E., Sierra K., and Bates B. *Head First Design Patterns*, O'Reilly. ISBN 0-596-00712-4. 2004.

[Fug98] Fuggetta, A., Picco, G.P., and Vigna, G. *Understanding Code Mobility*. In IEEE Transactions on Software Engineering, 24(5): 342-361, 1998.

[Gamma95] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[Gray93] Gray, J., Reuter, A. *Transaction Processing -Concepts and Techniques*. Morgan Kaufmann, 1993.

[Har84] Harel,D. *Dynamic Logic*. Handbook of Philosophical Logic II, editors D.M.Gabbay, F.Guenthner; pags. 497-694. Reidel 1984.

[He03] Herrero, J.L. *Proposal of a Platform, Language and Design, for the Development of Aspect Oriented Applications*. P.H.D. Thesis, Extremadura, Spain,2003.

[Kal99] Kalbarcyk Z. T., Iyer R. K., Bagchi S., Whisnant K., *Chameleon: A software infrastructure for adaptive fault tolerance*, IEEE Transactions on Parallel and Distributed Systems, vol. 10, pp. 560-579, June 1999.

[Kic97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J. *Aspect-Oriented Programming*. In proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

[Let98] Letelier P., Sánchez P., Ramos I., Pastor O., *OASIS 3.0: "Un enfoque formal para el modelado conceptual orientado a objeto"*. Universidad Politécnica de Valencia, SPUPV - 98.4011, ISBN 84-7721- 663-0, 1998.

[Lo02] Lopes, A. Fiadeiro, J.L., Wermelinger, M. *Architectural Primitives for Distribution and Mobility*, 10th Symposium on Foundations of Software Engineering, SIGSOFT FSE 2002, pp. 41-50.

[Lo97] Lopes, C. *D. A Language Framework for Distributed Computing* Ph.D. Dissertation, College of Computer Science, Northeastern University, Boston, 1997.

[MASIF] Especificación MASIF, OMG (1995), http://www.omg.org/cgi-bin/doc?orbos/97-10-05

[Mil91] Milner, R., *π-Cálculo Poliádico: A tutorial*, 1991.

[Moss87] Moss, J. E. B. *Log-based recovery for nested transactions*. In Proceedings of the International Conference on Very Large Data Bases VLDB (1987), pp. 427-432.

[MSRem] *Microsoft .Net Remoting: A Technical Overview*, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/hawkremoting.asp

[Na00] Natarajan B., Gokhale A., Yajnik S., Schmidt D., *DOORS: Towards High-Performance Fault Tolerant CORBA*, doa, p. 39, International Symposium on Distributed Objects and Applications, 2000.

[Ozsu98] Öszu T., Valduriez, P. *Principles of Distributed Database Systems 2nd Ed.* Prentice Hall, 1998.

[Pa02] Pahl S. *Descripción de los Enterprise Services (COM+) en .NET*, http://www.microsoft.com/spanish/msdn/articulos/archivo/19( 702/voices/entserv.asp, 2002

[Pe05] Pérez, J., Ali, N., Carsí, J.A., Ramos, I. *Dynamic Evolution in Aspect-Oriented Architectural Models*, Proc. of the European Workshop on Software Architecture, Pisa, June 2005 © Springer LNCS vol n.3527.

[Pe05b] Pérez, J., Ali, N., Costa, C., Carsí, J.A., Ramos, I. *Executing Aspect-Oriented Component-Based Software Architectures on .NET Technology*,3rd International Conference on .NET Technologies, Pilsen, Czech Republic, May-June 2005 , 2005

[Pe06] Pérez J., Ali N., Carsí J.A., Salavert I. *Designing Software Architectures with an Aspect-Oriented Architecture Description Language* The 9th International Symposium on Component-Based Software Engineering (CBSE 2006), Mälardalen University, Västerås near Stockholm, Sweden, June 29th -1st July 2006

[Ramm02] Rammer, I., *Advanced .NET Remoting (C# Edition)*. Apress, 2002.

[Ro04] S. Robinson, B. Harvey, C. McQueen, C. Nagel, M. Skinner, J. Glynn, K. Watson, O. Cornes, and J. Moemeka. *Professional C# (3rd Edition)*. Wrox Press Inc, 2004.

[Rya04] Ryan, C. and Westhorpe, C. *Application Adaptation through Transparent and Portable Object Mobility in Java*. In proc. of 2004 International Symposium on Distributed Objects and Applications (DOA 2004), Agia Napa, Cyprus, 2004, Springer-Verlag LNCS3291

[Scott03] McLean S., Naftel J., Williams K., *Microsoft .NET Remoting*. Microsoft Press, 2003.

[Soa02] Soares, S. and Borba, P. *PaDA: A Pattern for Distribution Aspects*: In Second Latin American Conference on Pattern Languages Programming *SugarLoafPLoP*, Itaipava, Rio de Janeiro, Brazil, 2002, 87-99.

[Soa02b] Soares, S., Laureano, E. and Borba, P. *Implementing Distribution and Persistence Aspects with AspectJ*: Proceedings of the 17th ACM Conference on Object-Oriented programming systems, languages, and applications, OOPSLA'02, Seattle, WA, USA,2002, 174-190

[Sta04] Staneva D. Dobreva D. *MAPNET: a .NET-Based mobile-agent platform*, In Proc. of the 5th international conference on Computer systems and technologies, 2004

[Sta04b] Staneva D. Gacheva P. *The communication infrastructure of the MAPNET Mobile-Agent Platform*, CompSysTech, 2004

[Szy02] Szyperski, C., *Component Software: Beyond Object Oriented programming*, ACM Press and Addison Wesley, New York, USA, 2002.

[Tre04] Treaster M., *A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems* , http://www.citebase.org/cgi-bin/citations?id=oai:arXiv.org:cs/0501002 , 2004

[Tre05] Treaster M. *A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems*. ACM Computing Research Repository (CoRR), (cs.DC/ 0501002), January 2005

[Tro03] Troger, P. and Polze, A. *Object and Process Migration in .NET*. The 8th IEEE Intern. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003), Mexico, January 2003.

[Va04] Vasudeva, E., Joshi, Rushikesh K. *A Scheme for Implementing Ambient Calculus over an ARC Framework*, Software Design and Architecture Workshop, Dec 2004, IIIT Bangalore (position paper).

# ANEXO 1
# PATRONES DE SOFTWARE

Los Patrones de Diseño (*Design Patterns*) son la base para la búsqueda de soluciones a problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces. Un patrón de diseño es una solución a un problema de diseño no trivial, efectiva (ya se resolvió el problema satisfactoriamente en ocasiones anteriores) y reutilizable (se puede aplicar a diferentes problemas de diseño en distintas circunstancias).

Uno de los objetivos que se han marcado para la refactorización y ampliación de las características distribuidas del *middleware* PRISMANET ha sido aplicar patrones de diseño, para de esta forma, conseguir un grado más alto de eficiencia, reutilización del diseño y del código. Además es posible que con la incorporación de los patrones de diseño a las estructuras existentes se facilite la comprensión del sistema, incluso para quien no esté familiarizado con los patrones. Por la misma razón, se ha intentado aplicar dichos patrones para la incorporación de nuevos conceptos.

Siguiendo este planteamiento se han aplicado dos patrones de diseño. Durante la modificación del modelo de ejecución de *attachments* y *bindings* se ha aplicado el patrón Observer, para conseguir un canal de comunicación más eficiente En la implementación de las transacciones ha sido necesario establecer un mecanismo que guarde la información de un aspecto que va a ser modificada por si hay que recuperarla en el fututo, por lo que se ha decidido usar el patrón Memento.

A continuación se describe el problema que se buscaba solucionar en cada caso y como se ha resuelto haciendo uso de los patrrones de diselño propuestos por Gamma.

## A1.1 Patrón Observer

En una primera revisión del código se ha detectado que el funcionamiento de los *attachments* y los *bindings* cuando no recibían ninguna petición desde los puertos de salida de la componente o sistema al que estaban conectados, era altamente ineficiente, pues entraban en un bucle de espera-reactivación costoso, que en muchos casos era innecesario; aunque en la cola de peticiones de los puertos de salida del componente (sistema) no se encolara ninguna petición, el *attachment* (*binding*) comprobaba periódicamente dicha cola para verificar si su estado había cambiado.

La manera de resolver esta ineficiencia pasa por hacer que el propio puerto de salida avise a los *attachments* (*bindings*) que están conectados a él en el momento que ocurra un evento que les pueda interesar, como, en este caso, encolar una petición en la cola del puerto. De esta forma, el hilo de escucha de los *attachments* (*bindings*) se puede suspender en el momento que no hayan más peticiones en la cola y volverlo a reactivar cuando aparezca una nueva petición, liberando el coste de procesamiento innecesario que suponía dormirlo y despertarlo cada cierto tiempo.

El comportamiento descrito anteriormente está recogido por el patrón *Observer* [Gamm95], cuya aplicabilidad se describe para los siguientes casos:

- Cuando una abstracción tiene dos aspectos y uno depende del otro. Encapsular estos aspectos en objetos separados permite modificarlos y reutilizarlos de forma independiente.

- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuantos objetos deben cambiarse

- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quienes son dichos objetos. En otras palabras, cuando no queremos que estos objetos estén fuertemente acoplados.

El último punto se ajusta al caso descrito: El puerto de salida ha de notificar a los *attachments* (*bindings*) conectados a él que se ha encolado una petición en su cola, para que los *attachments* (*bindings*) reanuden, en caso de estar suspendidos, su hilo de escucha para atenderla.

La estructura que se propone en la bibliografía [Gamm95] para el patrón *Observer* es la siguiente se muestra en la Figura 91.

El Sujeto (*Subject*) conoce a sus observadores y proporciona una interfaz para añadir y quitar objetos Observador. El SujetoConcreto *(ConcreteSubject)* almacena el estado de interés para los ObservadoresConcretos y les envía una notificación cuando cambia de estado. El objeto Observador *(Observer)* define una interfaz para actualizar los objetos que deben ser modificados ante un cambio en el Sujeto. El ObservadorConcreto (*ConcreteObserver*) mantiene una

referencia al SujetoConcreto, guarda un estado que ha de ser consistente con el del SujetoConcreto e implementa una interfaz de actualización.

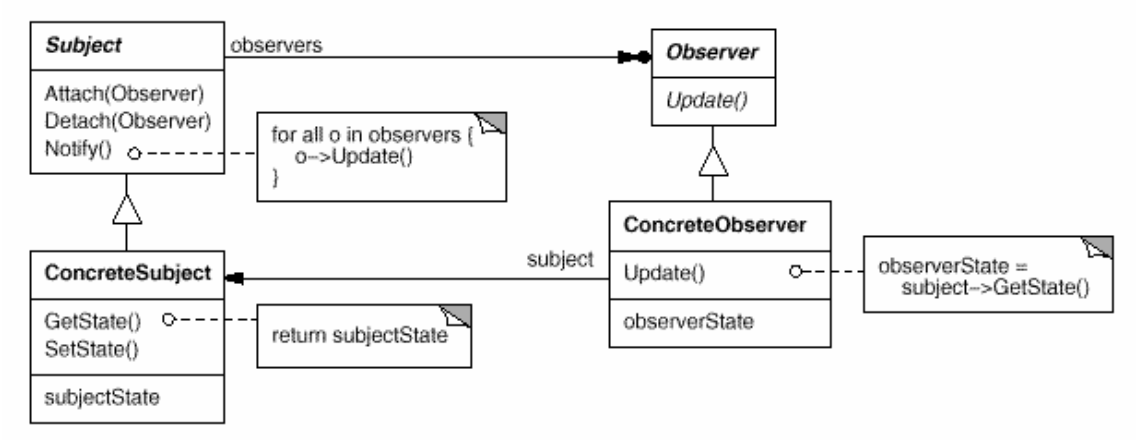



**Figura 91: Estructura del patrón Observer**

En el caso que nos atañe, el Sujeto está representado por el `OutPort` que contiene una lista con la información de los Observadores. El Observador está representado por la interfaz `ISubscriber`, que define el método `getRequest()` con el que el Observador será avisado de un cambio en el Sujeto. El objeto SujetoConcreto en realidad está representado por la cola, pues es su estado el que necesita obtener el observador. Pero hay que tener en cuenta que cada observador tiene asignada una cola independiente, aunque a la hora de encolar un elemento se haga de manera global a todas las colas de todos los Observadores del puerto de salida. Esto se hace mediante la clase `ListenerPort`, cuyos objetos poblarán la lista `subscribersList` del `OutPort`. Un objeto `ListenerPort` está formado por una referencia al `ISubscriber` correspondiente y por la cola en la que escuchará. La clase `OutPort` contiene los métodos `RegisterListener` y `UnregisterListener` para añadir y eliminar `ListenerPorts` a la lista `subscribersList` respectivamente. El objeto ObservadorConcreto estará representado por un `AttachmentClient` (`ComponentBindingClient`), que hereda de `Attachment` (`ComponentBinding`) que a su vez implementa la interfaz `ISubscriber` que, como se ha comentado representa al Observador. El *attachment* cliente (*binding* cliente) contiene una referencia a la cola de su `ListenerPort`, esto es, al SujetoConcreto, tal y como indica el patrón.

El funcionamiento del patrón *Observer* estipula que el sujeto concreto notifica a sus observadores cada vez que se produce un cambio que pudiera hacer que el estado de éstos fuese inconsistente con el suyo. Tras ser informado, un objeto ObservadorComcreto puede pedirle al sujeto más información, que utilizará para sincronizar su información con la del sujeto. A continuación se muestra un diagrama de interacción donde se muestra el funcionamiento:

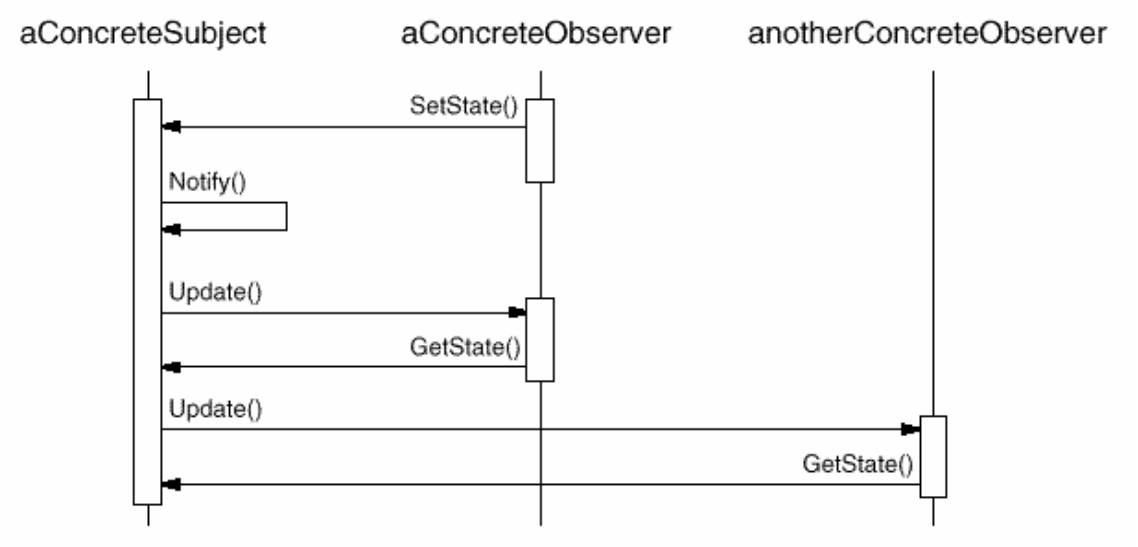



**Figura 92: Diagrama de secuencia del patrón Observer**

Nótese como en la figura, es un objeto concreto quien inicia el cambio que después será notificado al resto de observadores. Esto no tiene porqué ser así, siendo el propio sujeto o incluso una clase externa quien comience la interacción.

En el caso de los puertos de salida, es elemento arquitectónico quien comenzará la interacción encolando un elemento en el puerto de salida. Esto provocará una llamada al `Notify()` del `OutPort` que recorrerá la lista de suscriptores (`subscribersList`) encolando la petición en su cola particular e invocando a su método `getRequest()` para notificarle que ha ocurrido un evento de encolado.

De esta forma, el patrón *Observer* permite modificar los sujetos y los observadores de forma independiente, así como añadir nuevos observadores sin modificar el sujeto u otros observadores. Es decir, podemos añadir y eliminar *attachments* (*bindings*) al puerto de salida sin que esto afecte al puerto de salida ni a los otros *attachments* (*bindings*) que están conectados.

Otras ventajas que proporciona la aplicación de este patrón son las siguientes:

- **Acoplamiento abstracto entre sujeto y observador:** Todo lo que un sujeto sabe es que tiene una lista de observadores, cada uno de los cuales se ajusta a la interfaz simple de la clase abstracta Observador (en este caso, la interfaz *ISubscriber*). El sujeto no conoce la clase concreta de ningún observador, por tanto el acoplamiento es mínimo.

- **Capacidad de realizar comunicación mediante difusión:** A diferencia de una petición ordinaria, la notificación enviada por un sujeto no necesita especificar su receptor, ya que se envía automáticamente a todos los objetos interesados que se hayan

suscrito a ella, dejando en manos del observador el manejar u obviar la notificación. Por ejemplo, cada vez que se añada un elemento a la cola de un `OutPort` de una componente se notificará a todos los *attachments* (*bindings*) conectados. Cuando un *attachment* concreto reciba la notificación mediante su `getRequest`, si su hilo de escucha está suspendido lo reanudará, en caso contrario, no hará nada.

- **Actualizaciones inesperadas:** Dado que los observadores no saben de la presencia de los otros, no pueden saber el coste último de cambiar el sujeto. Es decir, puede que al aplicar un cambio mínimo en éste inicien una actualización en cascada de los demás observadores y de sus objetos dependientes. Esto se puede corregir aplicando protocolos adicionales al patrón que permitan a los observadores descubrir que ha cambiado en el sujeto. Esto, no obstante, no ha sido necesario en el caso de los puertos de salida y sus suscriptores, pues el único cambio que se notifica es el de añadir elementos a una cola.

A continuación se van a analizar algunas cuestiones relativas a la implementación del patrón *Observer* en la escucha de los puertos de salida por parte de los *attachments* y *bindings*.

- **Correspondencia entre los sujetos y sus observadores.** El modo en que el sujeto conoce a sus observadores es guardando una referencia a éstos. De esta manera, el puerto de salida guarda una lista `ListenersList` en donde se registran los observadores, en este caso los suscriptores[9]. De esta manera, cada vez que un suscriptor se asocia a un puerto de salida para escuchar las peticiones que se encolan en éste, se invocará al método `AddListener()` del puerto para añadir el suscriptor al listado de *Listeners*. De manera análoga, cuando un *attachment* se desvincule de un puerto se llamará al método `RemoveListener()`. Como se puede ver, estos dos métodos realizan la funcionalidad de los métodos *Attach* y *Detach* descritos en el patrón.

- **Observar más de un sujeto.** En el patrón se describe la posibilidad de que en determinadas ocasiones un observador pueda depender de más de un sujeto, en cuyo caso es necesario implementar un mecanismo para saber que sujeto ha enviado la notificación para actuar en consecuencia. Esta no es una situación aplicable a los puertos de salida y los suscriptores, pues cada suscriptor (en concreto, su parte cliente) sólo está asociada con un único puerto de salida (no tiene sentido pensar en un *attachment*

[9] De ahora en adelante se utilizará el término suscriptor para hacer referencia tanto a los *attachments* como a los *bindings*.

conectado a dos `OutPorts`, en todo caso se hablaría de dos *attachments*)

- **¿Quién dispara la actualización?** El patrón *Observer* describe dos posibilidades a la hora de disparar la notificación. Una opción es que las operaciones que provocan el cambio de estado en el sujeto llamen a `Notify()`, de esta manera, la notificación a los observadores es transparente al cliente. Otra opción es que los clientes sean los encargados de llamar a `Notify()` en el momento adecuado. De esta forma, solo lanzaran la actualización en los observadores en el momento que crean que el cambio de estado en el sujeto es suficiente, evitando así innecesarias actualizaciones intermedias. En el caso que se analiza, debido a su simplicidad, es el propio sujeto, el puerto de salida, quien tras encolar una petición lo notifica a sus suscriptores, es decir, la primera opción descrita. Este comportamiento, si bien simplifica mucho la implementación, puede resultar algo ineficiente, pues lanza una difusión cada vez que es encolado un elemento en el puerto de salida, y, como se comenta anteriormente, la finalidad de la notificación a los suscriptores es despertarlos en el caso que sus hilos estén suspendidos al no recibir peticiones. Si se piensa en el caso de un puerto de salida que está continuamente encolando peticiones, y cuyos suscriptores no llegan nunca a suspenderse, estas difusiones son altamente ineficientes, ya que no harán nada. Una solución a este problema es que la orden `Notify()` solo se lance en el caso en que se pasa en la cola del puerto de salida de ningún elemento a uno.

- **Referencias perdidas a los sujetos borrados.** Borrar un sujeto no debería producir referencias perdidas en sus observadores. Esto no se va a dar, ya que en PRISMANET, cuando se borra un puerto de salida, es porque se va a borrar el elemento arquitectónico que lo posee, y por tanto también se borran todos los suscriptores relacionados. Es decir, eliminar un sujeto supone eliminar a todos sus observadores.

- **Asegurarse de que el estado del Sujeto es consistente consigo mismo antes de la notificación.** Es importante que la llamada a notificar sea la última operación que realice el sujeto para evitar que los observadores se actualicen cuando el sujeto aún no ha alcanzado un estado consistente. Esto se puede dar cuando las operaciones de las subclases del Sujeto llaman a operaciones heredadas. En el caso en que aplicamos el patrón hemos de garantizar que la llamada a `Notify()` se realiza tras encolar la petición en la cola interna del suscriptor y no antes.

- **Evitar protocolos específicos del observador: los modelos *push* y *pull*.** En ocasiones las implementaciones del patrón

*Observer* suelen hacer que el sujeto envíe información adicional sobre el cambio. Esta información se pasa como parámetros de Actualizar. Hay dos modelos para este envío de información: El modelo *push* en el que el sujeto le envía toda la información parametrizada al observador, la requiera éste o no y el modelo *pull*, en el que el sujeto le envía la información mínima al observador y éste ya se encarga de pedir los detalles explícitamente. Los puertos de salida no han de enviar ninguna información adicional a los suscriptores, así que este punto se ha ignorado.

- **Especificar las modificaciones de interés explícitamente.** Se puede mejorar la eficiencia extendiendo la interfaz de registro del sujeto, haciendo que los observadores se suscriban solo a aquellos eventos concretos que les interesen. Cuando ocurra un evento, el sujeto avisará únicamente a los observadores que estén registrados con este evento. De nuevo, este punto no tiene sentido en la aplicación comentada, pues tan solo hay un tipo de evento a notificar: El encolado de una petición.

- **Encapsular la semántica de las actualizaciones complejas.** Cuando la relación de dependencia entre sujetos y observadores es muy compleja, puede ser necesario un objeto intermedio que mantenga estas relaciones. Esto no es necesario en el caso de estudio.

- **Unir las clases del sujeto y el observador.** Las bibliotecas de clases escritas en lenguajes que carecen de herencia múltiple, generalmente no definen clases separadas Sujeto y Observador, sino que juntan sus interfaces en una clase. Eso permite definir un objeto que haga tanto de sujeto como de observador sin usar herencia múltiple. En el caso expuesto no es necesario contemplar esta situación, ya que un sujeto no ha de ser al mismo tiempo observador ni viceversa. No obstante, la implementación realizada en C# (lenguaje que no soporta la herencia múltiple), ha requerido la implementación del patrón haciendo uso de interfaces en lugar de con clases abstractas, como recomienda la bibliografía.

## A1.2 Patrón Memento

Durante el diseño del modelo transaccional de PRISMANET ha surgido la necesidad de incorporar un mecanismo capaz de guardar el estado de un aspecto previo a la ejecución de una operación transaccional sobre él. La operación transaccional puede modificar el valor de los atributos del aspecto, esto es, su estado. Si la transacción finalmente falla, el estado previo ha de ser recuperado, por lo que se ha de guardar en algún lugar.

Lo que se pretende es que, si durante una transacción, formada por un conjunto de llamadas a servicios de un aspecto, algún método falla, se deshagan todos los cambios producidos en el estado del aspecto desde que se inició la transacción. Por ello, se debe encapsular dicho estado previo al inicio de la transacción para recuperarlo en caso de que no se complete correctamente la transacción. Para ello, una buena solución es la aplicación del patrón de software *Memento* descrito por Gamma [Gamma95].

El patrón de software *Memento* se usa para extraer el estado interno de un objeto sin violar su encapsulamiento, para poder recuperarlo en un momento dado. La aplicación de este patrón está definida en [Gamma95] para los casos en los que se desee obtener una instantánea del estado del objeto (o de una porción de éste) de manera que pueda ser restaurado posteriormente.

La estructura del patrón se muestra a continuación se muestra en la Figura 93.

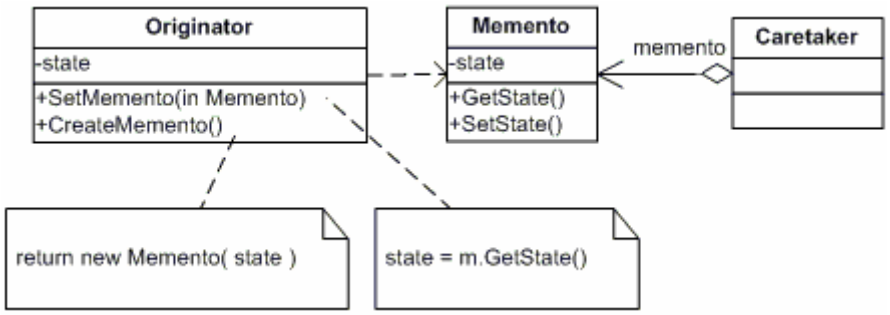



**Figura 93: Estructura del patrón Memento**

Las clases y/o objetos participantes en el patrón se describen a continuación:

- **Memento:** Es quien guarda la información del estado interno del objeto *Originator*. Protege contra el acceso de objetos que no sean el *Originator*. Tiene dos interfaces: una que tan solo pasa el memento al resto de objetos (*CareTaker*) y otra que da acceso a toda la interfaz (*Originator*) de forma que permite obtener la información necesaria para restaurarse al estado anterior.

- **Originator.** Es quien crea el Memento que contiene su estado interno, y lo usa, posteriormente, para recuperar dicho estado. No guarda el objeto memento.

- **Caretaker:** Es el responsable de guardar el Memento. Nunca examina o interopera con él.

El funcionamiento se muestra en la bibliografía a través del diagrama de la Figura 94.

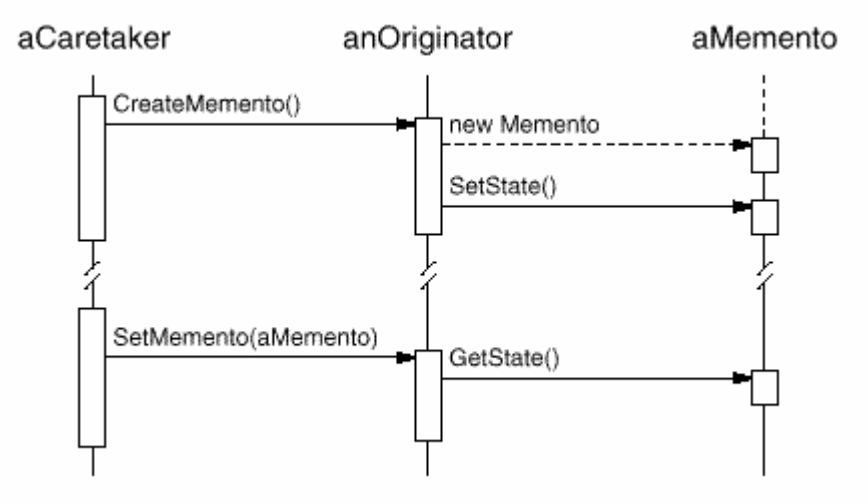



**Figura 94: Diagrama de secuencia del patrón Memento**

El *Caretaker* invoca al servicio `CreateMemento()` del *Originator*, que le devuelve un nuevo objeto *Memento* que encapsula el estado actual del *Originator*. Cuando el *CareTaker* estime necesario, invocará al método `SetMemento()` del *Originator*, pasándole como parámetro el *Memento* que está guardando. El método obtendrá el estado del objeto *Memento* pasado y lo restablecerá.

En la aplicación del patrón, el Originador será cada uno de los aspectos que quieran almacenar una instantánea de su estado. El *Caretaker* es un objeto agregado al aspecto (`aspectStatecareTaker`) que se encargará de realizar las acciones de gestión del Memento, que será un objeto genérico capaz de guardar una instantánea del estado de cualquier aspecto, es decir, del estado de sus atributos.

La aplicación del patrón de diseño Memento conlleva las siguientes consecuencias:

- **Preserva los límites de la encapsulación**: *Memento* evita que la información que solo el originado puede manipular sea consultada por otros elementos, pese a que sea almacenada fuera del Originador. Esta es la tarea del *CareTaker*, en concreto del `stateCareTaker` del aspecto, que guardará el *Memento* con el estado del aspecto sin dar la opción que nadie más que el mismo aspecto consulte su estado.

- **Simplifica el Originador:** El Originador no es el encargado de albergar la lógica para guardar el estado anterior. De esto se encarga el `stateCareTaker` asociado a cada aspecto, abstrayendo así al Originador de dicha funcionalidad, simplificando así su diseño.

- **El uso de Mementos puede sobrecargar el sistema:** La creación y gestión de *Mementos* puede incurrir en una sobrecarga considerable si el Originador ha de almacenar grandes cantidades

de información para guardar en el *Memento*, o si se crean y recuperan *Mementos* con mucha frecuencia. Esto en gran medida depende del dominio de aplicación, pero la separación aspectual de la funcionalidad de un sistema reduce el tipo de información que guarda cada aspecto (solo la relativa al *concern* que representa) por lo que no deberían ser grandes cantidades de información las que guardase un Memento. Por otro lado, se ha de tener en cuenta que el Memento solo se usará bajo el contexto de una transacción, y sólo se restaurará el estado que almacena en el caso de que esta falle, por lo que el sobrecoste de su creación y recuperación está asociado a casos muy concretos.

# ANEXO 2
# TRANSACCIONES EN .NET

En esta sección se va a dar un breve resumen de los mecanismos que ofrece el *framework* .NET para el uso de transacciones. En concreto se explicarán las transacciones asociadas a servicios COM proporcionadas por el especio de nombres `EntrepriseServices` y el espacio de nombres `System.Transaction`, que recientemente se ha añadido a la versión 2.0 del *framework*.

## A2.1 Enterprise Services: Transacciones orientadas a componentes

El espacio de nombres `EnterpriseServices` proporciona funcionalidad para el uso de transacciones asociadas a componentes en .NET. Para ello es necesario que las clases que vayan a hacer uso de dichas transacciones hereden de `ServiceComponent` y que el ensamblado en que están definidas reciba un *strong name* (mediante la inclusión de una llave generada con la utilidad `sn.exe`)

Las transacciones están ligadas a contextos, que son los que definen los límites de la transacción. *EnterpriseServices* proporciona mecanismos para el paso de contextos entre componentes. Un dominio de aplicación puede tener diferentes contextos. Un contexto es usado para agrupar objetos con requisitos de ejecución similares. Los contextos están compuestos por un grupo de propiedades que son usadas para intercepción: cuando un objeto interno al contexto es accedido desde otro contexto, el interceptor realiza determinadas acciones previas a que la llamada llegue al objeto. Este preproceso, es usado, por ejemplo para realizar las acciones pertinentes para mantener el estado inicial en el caso que tras una serie de accesos al objeto se decida deshacer todos los cambios, como en una transacción.

Las clases son marcadas con el atributo `[Transaction]`, para indicar que van a trabajar en el contexto de una transacción, configurado con alguna de estas opciones:

- `Required`: El componente se ejecuta dentro de una transacción. Si una transacción ha sido creada previamente (por ejemplo, en otro componente que llama a éste), el componente se ejecutará en la misma transacción. Si no ha sido creada la transacción previamente, el componente creará una.

- `RequiredNew`: El componente siempre se ejecutará en una nueva transacción, independientemente de si el que le llame se ejecuta sobre una transacción o no.

- `Supported`: El componente no necesita ejecutarse sobre una transacción propiamente, no obstante, si quien le llame se ejecuta sobre una transacción, éste se unirá a dicha transacción.

- `NotSupported`: El componente nunca se ejecutará sobre una transacción, independientemente de lo que haga el que le llame.

- `Disabled`: La posibilidad de transacción del contexto actual es ignorada.

Las transacciones son sensibles a la modificación de sus bits `consistence` y `done` dentro del contexto. La transacción es aceptada si todos los componentes participantes dentro de la transacción terminan correctamente. Si el *bit* de consistencia (`consistence`) es puesto a falso por alguno de los componentes, la transacción se abortará cuando el objeto que comenzó la transacción finalice. Si el *bit* `done` es puesto a cierto, el objeto podrá ser destruido una vez termine la llamada, creando una nueva instancia en la siguiente llamada. La configuración de estos bits se modifica con los métodos de la clase `ContextUtil` de la siguiente forma:

| ContextUtilMethod | Consistent Bit | Done Bit |
|---|---|---|
| `SetComplete()` | True | True |
| `SetAbort()` | False | True |
| `EnableCommit()` | True | False |
| `DisableCommit()` | False | False |

También es posible la utilización del atributo `[AutoComplete]` dentro de los métodos del componente, para que sea el *framework* quien decida la configuración de los *bits* en función de lo que ocurra.

## A2.2 La clase TransactionManager

El *framework* .NET 2.0 incorpora un nuevo espacio de nombres llamado `System.Transaction` que ofrece una infraestructura de clases para la gestión de procesos transaccionales iniciados en SQL Server, ADO.NET, *Message Queuing* (MSMQ), y el *Microsoft Distributed Transaction Coordinator* (MSDTC).

El espacio de nombres `System.Transactions`, proporciona un modelo implícito y un modelo explícito de programación con transacciones, basados en las clases `Transaction` y `TransactionScope`, respectivamente. La diferencia entre ambos modelos radica en la gestión de las transacciones, realizada de manera manual por el programador en el modelo explícito, y es gestionada por la infraestructura en el modelo implícito.

Para la implementación de las transacciones implícitas, tan solo hay que marcar el código que se quiere que forme parte de la transacción. El código marcado pasará a ser el ámbito de la transacción. El ámbito de la transacción es iniciado desde el momento en que se crea una instancia de `TransactionScope`. Una vez hecho esto, el Transaction Manager (TM) determina en que transacción va a participar el código marcado. Esto dependerá de dos factores: La existencia de un ambiente de transacción previo y el valor que tenga el parámetro `TransactionScopeOption`. Los diferentes valores se muestran a continuación:

- **Required:** Si el código marcado está incluido dentro del ámbito de otra transacción (zonas `TransactionScope` anidadas), se unirá a dicha transacción. Cuando una zona de código marcada como transaccional se une a una transacción creada anteriormente, la finalización del nuevo ámbito no termina la transacción más genérica, a no ser que en el nuevo ámbito se anule la transacción. En caso de no existir una transacción más genérica, el TM creará una nueva transacción.

- **RequiredNew:** Tanto si el código marcado pertenece a una transacción previamente creada, como si no, el TM creará una nueva transacción para el código marcado.

- **Supress:** El código marcado no se ejecutará transaccionalmente.

Cuando se complete la ejecución de todo el código incluido en el ámbito de la transacción, se debe llamar al método `Complete` para indicar al TM que puede realizar el `commit`. Si durante la ejecución del código marcado se hubiese producido alguna excepción, el TM hubiera abortado la transacción. En caso de haber transacciones anidadas que pertenezcan al mismo ambiente transaccional, solo se dará por buena la transacción si todas las transacciones internas han concluido correctamente.

La implementación explícita de transacciones es útil para definir transacciones que engloben llamadas a múltiples funciones o hilos de ejecución. A diferencia del caso implícito, el programador ha de indicar cuando se debe hacer un `commit` o un `rollback`, para confirmar o abortar la transacción, respectivamente. En este caso se usa la clase `CommitableTransaction` que incorpora toda la funcionalidad de la clase `Transaction` gracias a la herencia, y además, incorpora un método `commit`. Este método nos permite pasar el objeto que realiza la transacción a otros métodos, de forma que sean capaces de votar sobre la transacción. No obstante, tan solo el creador del objeto `CommitableTransaction` tiene la habilidad de hacer un `commit`. Es importante señalar que hasta que la transacción no es confirmada o abortada, los recursos involucrados en ella permanecerán bloqueados.