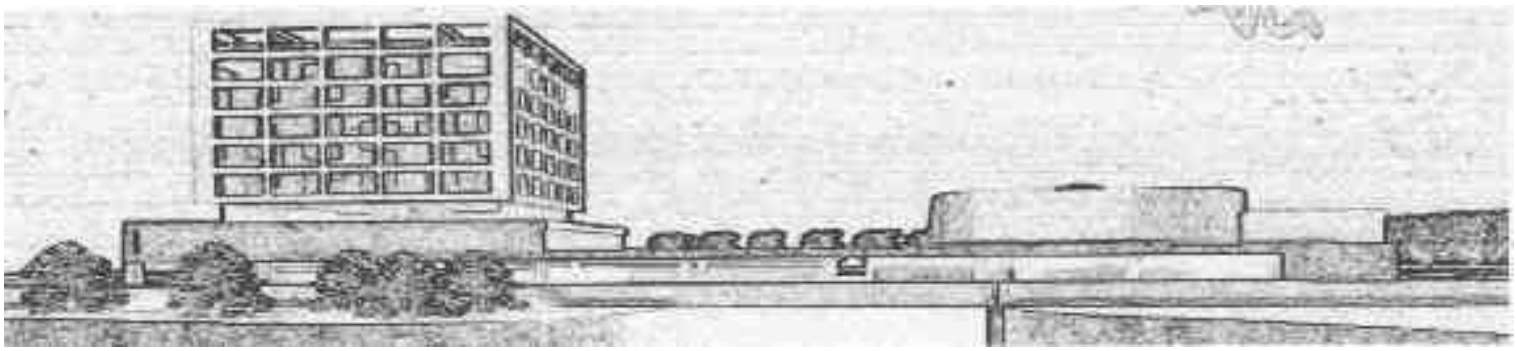
VOLUME VII, NUMBER 2



REPORTS ON SYSTEMS AND COMMUNICATIONS

Framework basado en AOP para la simulación distribuida según el estándar IEEE-1516

AGUSTÍN SANTOS-MÉNDEZ
LUIS RODERO-MERINO
ANDRÉS LEONARDO MARTÍNEZ-ORTÍZ
DANIEL IZQUIERDO-CORTÁZAR

Móstoles (Madrid), Spain, Abril 2007
Depósito Legal: 50653-2004
ISSN: 1698-7489

Framework basado en AOP para la simulación distribuida según el estándar IEEE-1516

Agustín Santos-Méndez, Luis Rodero-Merino
Andrés Leonardo Martínez-Ortíz, Daniel Izquierdo-Cortázar
Laboratorio de Algoritmia Distribuida y Redes, Universidad Rey Juan Carlos
Escuela Superior de Ciencias Experimentales y Tecnología
Campus de Móstoles (Madrid), C/ Tulipán S/N, 28933
Teléfono: 91 488 81 07 Fax: 91 664 74 90
E-mail: {asantos,lrodero,aleonar,dizquierdo}@gsyc.es

Abstract *Here we introduce a C# framework for the development of distributed simulations that follows the IEEE-1516 standard. It provides the concepts defined by that standard such as* Federation, Federated, *etc, and the* Run Time Infrastructure *that allows the communication among members of the simulation. The main characteristic of this framework is its intensive use of the AOP paradigm both for its development and to ease the creation of simulation software on top of it. By this approach, developers of distributed simulations do not need to take heed of the 1516 standard requirements, communications tasks, etc. Another interesting property of our middleware is how participants of the simulation organize in a* peer-to-peer *fashion, instead of applying the classic client-server architecture often found in distributed simulation environments.*

1. Introducción

Es habitual aplicar técnicas de simulación distribuida para la creación de mundos virtuales distribuidos, juegos en red, sistemas de entrenamiento donde los partipantes están repartidos geográficamente, o en simulaciones complejas que requieren utilizar los recursos de más de un sistema.

La simulación distribuida suele ser un reto ya que plantea problemas propios de la computación distribuida (comunicaciones, coordinación de procesos, etc.). Uno de estos problemas es el incremento de la complejidad que sufre el software. Esto es debido a que es necesario añadir, a la propia lógica de la simulación, la funcionalidad relacionada con las tareas de comunicación y coordinación. Esto a su vez disminuye el mantenimiento y la portabilidad del sistema.

Otro problema es que las soluciones para la simulación distribuida suelen basarse en el paradigma *cliente-servidor*, lo que impone limitaciones importantes como falta de escalabilidad y menor fiabilidad (el servidor es un punto único de fallo). Las limitaciones de la aproximación centralizada han motivado que la comunidad investigadora empiece a estudiar middleware basado en arquitecturas *entre iguales* (*Peer-to-Peer*, P2P). Por ejemplo en [13] se presenta un sistema P2P para dar soporte a sistemas multijugador.

En nuestra opinión, es potencialmente interesante investigar nuevos frameworks de simulación distribuida que cumplan con los siguientes objetivos:

- El número de participantes en la simulación debe de ser escalable.
- Los mecanismos de distribución han de ser transparentes al programador.
- Las simulaciones deben de poder ejecutarse en diversas plataformas software/hardware.

El estándar IEEE-1516 [3] es una propuesta para la construcción de arquitecturas sobre las que ejecutar simulaciones distribuidas. Dichas arquitecturas reciben el nombre de arquitecturas de alto nivel o *High Level Architecture,* **HLA** (en este trabajo usaremos los términos HLA y estándar IEEE de forma indistinta). Este estándar describe una arquitectura middleware sobre la que los desarrolladores pueden construir simulaciones distribuidas. Gracias a HLA, simulaciones individuales en máquinas distintas pueden colaborar para formar una *simulación global.*

En este artículo presentamos nuestro framework basado en HLA para la construcción de simulaciones distribuidas. Este framework implementa el estándar IEEE-1516, y tiene además varias características novedosas que lo diferencian de otras implementaciones y creemos importante resaltar:

- Incorpora la *programación orientada a aspectos* [12] (*Aspect Oriented Programming*, **AOP**) a la construcción de simulaciones distribuidas. Esto nos ayuda a cumplir uno de los objetivos de nuestro framework: hacer

transparente al programador la tarea de integrar código con la simulación global. El programador no necesita introducir cambios en su código para que su funcionalidad sea incorporada a la simulación (ver sección 4). Esto mejora además la modularidad del sistema al hacer independiente al código de simulación del middleware sobre el que será ejecutado.

- Usa el paradigma *Peer-to-Peer* (P2P) [15] para la comunicación y coordinación entre los participantes de la simulación, con el objetivo de superar las limitaciones del enfoque centralizado. Así, se aumenta la escalabilidad del sistema, ya que el añadir nuevos participantes en la simulación no sólo aumenta la demanda de recursos, sino también su oferta.

- Programado en C# [4]. Esto hace a la implementación multiplataforma, y permite integrar software de simulación en distintos lenguajes de programación gracias a las capacidades de C# para incorporar y ejecutar código escrito en otros lenguajes. Que nosotros sepamos, esta es la primera implementación en C# de este estándar.

El artículo está dividido en los siguientes apartados. La sección 2 presenta el estado del arte. La sección 3 introduce los conceptos básicos del estándar IEEE-1516. La sección 4 comenta los mecanismos en C# para la construcción de soluciones AOP. La sección 5 explica el diseño de nuestro framework de simulación, comentando las características principales de las tres capas que lo constituyen. Después, la sección 6 da un pequeño ejemplo de cómo usar nuestro middleware para hacer que un código de simulación para a ser distribuido. Finalmente, en la sección 7 damos algunas conclusiones y comentamos posible trabajo futuro.

2. Estado del arte

La propuesta presentada en este artículo combina conceptos diversos, como el uso de AOP aplicado a middleware, P2P y HLA. En esta sección damos una breve introducción al estado del arte en cada unos de estos campos, comentando otras propuestas relacionadas con nuestro sistema.

2.1. Programación orientada a aspectos

La *Programación Orientada a Aspectos* (*Aspect Oriented Programming* o AOP) [12] es un paradigma de programación que se ha propuesto como herramienta para reducir la complejidad del software.

AOP ayuda a especificar y aislar requisitos o características de un sistema software que no pertenecen a ningún módulo en particular, sino que pueden afectar a diversas partes del sistema (*cross-cutting concerns*). Estos requisitos son denominados *aspectos*. Típicamente, un aspecto está definido por *puntos de corte* (*pointcuts*), lugares dentro del flujo del programa donde se debe disparar el uso del aspecto (*e.g.* al entrar o salir de una función, etc), y por la funcionalidad a ejecutar al atravesar cualquiera de esos punto de corte (*e.g.* mostrar un mensaje de log, chequeos de seguridad, etc).

Aunque existen varias soluciones que añaden capacidades avanzadas de AOP a C#, tales como *AspectSharp* [1] o *PostSharp* [2] hemos decidido aprovechar las posibilidades que la misma plataforma .Net proporciona para realizar tareas de AOP mediante el uso de *atributos*. La razón principal es no introducir dependencias externas que creemos no son necesarias, ya que las capacidades que C# proporciona son suficientes para los requisitos de nuestro sistema. Además, la mayoría de frameworks de AOP requieren de ficheros extra en los que se incluye la descripción de los aspectos, lo que impacta negativamente en la mantenibilidad, y pueden en ocasiones ralentizar sensiblemente la ejecución del sistema, empeorando la eficiencia.

2.2. Utilización de AOP en sistemas middleware

La creación y utilización de middleware implica una gran complejidad tanto interna (en su construcción) como externa (interfaces, contratos ofrecidos a las aplicaciones). El uso de AOP, por otro lado, se está revelando como una herramienta muy útil para el desarrollo de sistemas distribuidos al limitar el impacto de estos problemas [5] [9].

Un claro ejemplo de aplicación de AOP a la construcción de middleware es *Remoting* [14], el framework para el desarrollo de aplicaciones distribuidas de la plataforma .Net. Sin embargo, Remoting impone un modelo que no se ajusta a las necesidades de replicación de estados de objetos; este es un requisito esencial en la simulación distribuida. Por otra parte, además, Remoting no tiene la capacidad para transferir la propiedad de objetos remotos. Finalmente, Remoting tiene limitaciones a la hora de especificar los mecanismos de comunicación. Por ejemplo, el estándar HLA permite definir cómo transmitir cada uno de los atributos de un objeto por un canal diferente, mientras que esto no es posible en Remoting. Estas limitaciones también se encuentran en otros middleware orientados a objeto tales como CORBA, RMI, etc.

2.3. Implementaciones del Estándar HLA

Existen diversas implementaciones propietarias de HLA[1] de las que no nos es posible recabar información por ser cerradas. Existen otras implementaciones abiertas, como *xrti* [10] . Ninguna de ellas utiliza AOP para su programación y todas usan un enfoque centralizado.

El estándar define un conjunto de interfaces que todo framework basado en HLA debe implementar. El estándar da la especificación de dichas interfaces para tres lenguajes: Java, C++ y Ada95. Esta es la principal razón por la que todas las implementaciones están en alguno de estos lenguajes.

Una de las novedades que aporta nuestra solución es que usa el lenguaje C# debido a las ventajas que ofrece (como portabilida, etc, ver sección 1). Además, como se explica en la sección 4, C# posee características que facilitan el uso de AOP, que es uno de los fundamentos de nuestro framework. Para ello, tras un estudio detallado, se adaptó a C# las interfaces especificadas por el estándar. Esta tarea no fue una mera traducción del lenguaje Java a C#, ya que se tuvieron en cuenta las caracteríscas propias de C#, como la posibilidad de describir eventos, propiedades, etc.

2.4. Middlewares de simulación no centralizados

Como se dijo anteriormente, existe un creciente interés en el desarrollo de sistemas de simulación distribuidos que no tengan dependencia de un servidor central. Por ello, se están estudiando soluciones más cercanas al paradigma P2P, donde cada participante del sistema es responsable de una parte de la simulación [13] [11].

En [6] encontramos una propuesta para el uso de P2P dentro del ámbito del estándar HLA. En su propuesta, Eklof utiliza la plataforma JXTA [8] para implementar un sistema de simulación distribuida P2P. Esta propuesta permite que nuevos participantes ofrezcan sus recursos a la simulación, y hace posible que los federados puedan migrar de una máquina a otra. Sin embargo, a diferencia de nuestro sistema, no permite la visibilidad automática de elementos replicados de la simulación entre participantes.

En este artículo presentamos una solución que combina el uso de AOP para la construcción de un middleware de simulación distribuida siguiento el estándar IEEE-1516 en un arquitectura descentralizada tipo P2P. A nuestro entender, esta propuesta es la primera que integra todas estas características.

3. Estándar de simulación distribuida IEEE-1516

El estándar IEEE-1516 define una arquitectura de alto nivel (*High Level Architecture*, **HLA**) para simulaciones distribuidas. La arquitectura HLA dicta las características a cumplir por el *middleware* sobre el que se construirán las simulaciones. El objetivo de HLA es permitir la interacción entre simulaciones de una forma sencilla y cómoda para el programador. Proporciona, en forma de servicios, funcionalidades propias de los entornos de simulación distribuidos, y que no se encuentran implementados en otro tipo de sistemas distribuidos de carácter más general, como por ejemplo servicios específicos de subscripción a eventos o control del tiempo global de simulación.

Una de las características propias de HLA es que cada *federado* (nombre técnico usado dentro de HLA para referirse a un participante de una simulación distribuida) puede interactuar de manera sencilla con otros federados, preguntando directamente a éstos o a través, por ejemplo, de subscripciones a *atributos* de esos otros federados. De esta forma, cuando un federado modifica sus atributos (por ejemplo, el federado que controla la simulación de un avión cambia el atributo *altura de vuelo*), otros federados automáticamente recibirán esas modificaciones y podrán actuar en consecuencia.

3.1. Conceptos básicos de HLA

HLA define los siguientes conceptos asociados a una simulación distribuida:

- *Federate*. Un federado forma con otros una simulación global. Cada federado ejecuta una parte de dicha simulación. Un federado no puede existir nunca por sí solo, siempre debe de estar asociado a una *federación*.

- *Federation*. Una federación es un conjunto de federados que conforman un entorno o una simulación global. Cada federado simula una parte (*e.g.* simulación de un avión) dentro de la simulación global (*e.g.* simulación del tráfico aéreo de un país). Un federado sólo puede interactuar con otros federados de su misma federación.

- *RunTime Infraestructure* (**RTI**). Parte clave del HLA, proporciona una capa de abstracción a los federados que encapsula la problemática adscrita al entorno de simulaciones distribuidas. Cualquier comunicación que se realice dentro del sistema entre simulaciones ha de pasar a través del RTI.

[1] En `https://www.dmso.mil/public/transition/hla/vendorlist` hay una lista de implementaciones propietarias de HLA.

Un federado se construye sobre un *Soporte RTI*, término con el que se define el soporte software que le permite comunicarse con otras federaciones, y a su vez permite a los federados comunicarse entre sí. Todas las comunicaciones entre federados deben ser mediante el RTI.

En la Fig. 6 podemos ver un diagrama en el que se muestran los conceptos anteriores.

Por otra parte, el estándar define un conjunto de documentos para la especificación de los modelos de la simulación. Estos son los siguientes:

- *Federation Object Model* (**FOM**). Para permitir la interoperabilidad entre las simulaciones de una federación, es necesario especificar qué información se intercambian entre ellas y la semántica de dicha información. Dicha información está descrita en el FOM. El FOM es un documento que contiene el modelo de la simulación: las *entidades* que forman parte la misma (*e.g.* avión), sus *atributos* (*e.g.* altura), las *interacciones* o eventos entre simulaciones (*e.g.* la pista está libre), el *formato de los mensajes* (*e.g.* la altura es un entero de 32 bits), el *tipo de transporte* (TCP, UDP u otro), etc. En definitiva, es un 'contrato' de qué información se intercambian los federados, y de cómo se modela dicha información.
  Por ejemplo, si el FOM de una federación incluye la entidad *avión* todos los federados sabrán que existe esa entidad y sus atributos, y cómo se codifican y envían los datos de dichos atributos. Todos los FOM deben seguir las reglas y restricciones dictadas por el estándar *Object Model Template* (**OMT**).
- *Simulation Object Model* (**SOM**). Cada federado debe indicar qué entidades, atributos, interacciones, etc, aporta con su simulación a la simulación global. Esto lo hace mediante el documento SOM. Cada SOM debe de ser un subconjunto del FOM de la federación, nunca contendrá elementos no contemplados en el FOM. Al igual que el FOM, sigue las normas del OMT.

4. Técnicas de intercepción de llamadas en C#

AOP se basa en la asociación de aspectos con *puntos de corte*. La capacidad de un lenguaje de programación para interceptar el flujo de control en dichos puntos de corte es clave en un sistema AOP. En C# esto es posible mediante la combinación de *atributos* y herencia de la clase `ContextBoundObject`.

En C# es posible asociar metadatos a elementos tales como clases, métodos, etc., mediante el uso de *atributos*[2]. Por ejemplo, un atributo muy común es `[Serializable]`, que indica que los objetos de una clase pueden ser serializados para ser enviados a traves de la red o guardados en disco. También es posible usar atributos para indicar que una clase está relacionada con un aspecto. Por ejemplo, un desarrollador podría definir un atributo `[Log]` indicando que cada vez que se llama a las funciones de las instancias de una clase se debe de ejecutar cierta funcionalidad de log:

```
[Log]
class C : ContextBoundObject { ... }
```

Figura 1: Ejemplo para intercepción

Es tarea del programador definir la funcionalidad asociada a cada atributo. Dicha funcionalidad se encargará a su vez de crear y registrar los manejadores que procesaran los eventos de llamada a los métodos de la clase.

Como ya se comentó en la sección 1, uno de nuestros objetivos a la hora de construir este middleware es que fuera transparente al programador. Gracias al uso de las técnicas descritas es posible interceptar las llamadas a la lógica de la simulación e incorporar de forma transparente el código propio de un sistema distribuido (replicación de estados, comunicaciones, reparto de tareas, etc.).

Para este framework hemos creado el atributo `[Observable]`, que usamos para indicar que las instancias de una clase han de ser vigiladas por nuestro software. Sirve, por lo tanto, para definir puntos de corte que indican que el código debe incorporarse a la simulación distribuida. Es el framework el que se encarga de integrar, de forma transparente, la funcionalidad en la simulación. Por un lado, esto mejora la modularidad, ya que no se introducen dependencias nuevas en el código de simulación. Por otro lado, la reutilización de código ya existente se simplifica: basta con introducir atributos en el código para incorporarlo a la simulación. Un ejemplo de esto se da en la sección 6.

5. Diseño del framework

Nuestro framework de simulación tiene tres partes o capas bien diferenciadas: la capa inferior se encarga de la construcción de mensajes y su envío a través de la red, la capa intermedia funciona como un gestor de replicación basado en AOP, y la capa superior se encarga de implementar los conceptos propios de HLA y proporciona la interfaz a los federados para que puedan acceder al RTI, usando para ello los servicios de las capas inferiores. En este apartado explicaremos estas tres capas.

[2] Los atributos de C# son similares, por ejemplo, a las *anotaciones* (*e.g.* `@Override`, `@Deprecated`, etc) usadas en Java, con las que quizás el lector esté más familiarizado

5.1. Capa de protocolo genérico

Esta parte implementa la funcionalidad de comunicación de mensajes entre elementos de un sistema. Es responsable de la serialización y envío en origen, y de la recepción y deserialización en destino, de la información a mandar.

Una característica importante de esta capa es su flexibilidad: no está pensada para el manejo de mensajes de un protocolo fijo y determinado, sino que puede serializar y deserializar mensajes según formato variable definido por configuración. Esto la hace utilizable en muchos otros escenarios, no sólo nuestro framework de simulación. Por ello la denominamos de *protocolo genérico*.

Para describir tanto los posibles mensajes a enviar como la forma de construirlos e interpretarlos se utiliza una información de configuración que puede proceder de ficheros o de metadatos (atributos). Así, posteriores cambios en el protocolo ya no requerirán modificaciones en el código. Si se hace mediante fichero, este documento ha de seguir la normativa OMT. Es cierto que podría haberse optado por otro formato, por ejemplo IDL, o WSDL. Sin embargo, al hacerlo así se pudo reutilizar gran parte del software, y en nuestra opinión se dió mayor consistencia al producto final.

5.2. Capa de gestión de la replicación

Quizás lo más relevante de nuestro middleware es que se encarga de replicar el estado de la simulación entre los participantes de la misma. Esto posibilita la construcción de un sistema distribuido sin servidor central, siguiendo la filosofía P2P.

Esta capa se encarga de detectar los cambios de estado en el sistema y de propagarlos, manteniendo así el estado del sistema replicado. Para ello, se intercepta la creación y destrucción de objetos (entidades) y los cambios en sus atributos.

Esta capa se basa en el atributo `[Observable]`, definido por nosotros. Cada vez que se crea una instancia de una clase marcada con ese atributo esta capa crea de forma automática un *proxy* que se encarga de vigilar cuando se llama al objeto correspondiente. Una vez se produce la llamada, se reenvía la información en forma de eventos a los componentes interesados.

Uno de esos componentes es, precisamente, la capa de protocolo. Cuando se produce cualquier modificación, se avisa mediante un evento a dicho componente, que siguiendo las especificaciones dadas construye el mensaje correspondiente y lo envía a los otros miembros del sistema. Para indicar a la capa de protocolo el formato de los mensajes se usa un documento OMT denominado *Bootstrap Object Model* (**BOM**). EL BOM fue inicialmente propuesto por el proyecto XRTI [10], que intenta proporcionar algunas extensiones a la arquitectura HLA. Sobre esa propuesta inicial hemos realizado algunos cambios para adaptarlo más a nuestras necesidades. En definitiva, el BOM contiene cómo construir los mensajes con los que esta capa notifica eventos tales como la creación de una instancia o cambios en un atributo.

Tanto esta capa de gestión de la replicación como la anterior de protocolo genérico son independientes entre sí, y podrían usarse por separado en otros sistemas. Por otro lado, la unión de ambas capas nos proporciona un middleware *de propósito general* para la replicación de objetos. Este middleware podría usarse en cualquier aplicación distribuida que necesite soporte para la replicación automática de instancias y sus propiedades. De forma transparente, comunica al resto de miembros del sistema eventos tales como la creación o destrucción de una instancia de una clase, o la modificación de sus atributos (ver Figura 5). Este middleware puede tener diversas aplicaciones, como por ejemplo la construcción de un sistema tolerante a fallos.

Es importante subrayar que este middleware de replicación es P2P, es decir, no depende de servidores centralizados sino que son los propios miembros del sistema distribuido los que hablan entre sí para indicar cuando se incorporan al sistema, cuando lo dejan, cuando se producen eventos, etc.

En el caso del simulador, hemos utilizado este middleware como base de la capa superior, que se encarga de implementar la funcionalidad de RTI (ver siguiente sección 5.3). La suma de las tres capas es lo que nos proporciona la implementación de HLA.

5.3. Implementación del RTI

Esta capa contiene la implementación propiamente dicha de los conceptos de RTI, tales como federación, federado, gestión de tiempo, etc. El código con la lógica de la simulación se construiría sobre este módulo.

Como se ha comentado anteriormente, el estándar especifica una serie de interfaces a mostrar a las aplicaciones para acceder a los servicios del RTI. Nuestra implementación cumple este requisito, permitiendo llamar al RTI mediante esas interfaces.

La mayoría de las implementaciones de RTI se basan en la idea de tener un servidor central, llamado típicamente *Executive*, que gestiona y mantiene el estado del sistema. Por ejemplo, qué federados existen, qué federaciones están activas en cada momento, etc. También se encarga del reenvío de eventos y coordinación entre simulaciones.

Como ya se indicó, las limitaciones de las arquitecturas centralizadas hacen que crezca el interés por las redes P2P en la comunidad investigadora. Incluso dentro del grupo que desarrolla el estándar HLA, se está trabajando en una especificación del mismo para redes P2P.

La principal característica a destacar de nuestra implementación del RTI es que, gracias a que usa los servicios de la capa de gestión de replicación, no necesita una arquitectura centralizada. En nuestra implementación, el estado es mantenido y propagado por la capa de replicación de forma automática. De este modo el estado del sistema está distribuido de forma natural entre los participantes de la simulación.

Así, por ejemplo, en una implementación tradicional, cuando el usuario solicita crear una federación, el RTI envía un mensaje al Ejecutivo y este crea una estructura de datos centralizada para mantener la información de esa federación. En nuestro caso, el RTI se limita a crear la misma estructura de forma local. Gracias al mecanismo de replicación esa estructura de datos será visible para todos los procesos miembros de la simulación de forma automática.

Otra consecuencia de usar esta arquitectura es la versatilidad a la hora de elegir los formatos de los mensajes entre RTIs.

5.4. Nivel de aplicación, federados

Finalmente, también en el nivel de aplicación, donde se sitúa el código del usuario (los federados), se hace uso de la arquitectura de replicación propuesta con el objetivo de facilitar las tareas del programador de simulaciones.

En una implementación tradicional, el usuario, tras la creación de cada objeto a simular (*e.g.* avión), o tras el cambio de estado del mismo (*e.g.* altura del avión), es responsable de avisar al RTI para que este propague la información al sistema. Como consecuencia, el código del objeto a simular está 'contaminado' con llamadas al middleware que añade complejidad innecesaria.

En nuestro sistema, el código de los objetos a simular se encuentra libre de esas llamadas. Es el motor de replicación el que de forma transparente se encarga de capturar los cambios de estado de la simulación y propagarlos. Así se consigue una mayor limpieza de código, ya que se separa la lógica de simulación del middleware de simulación. A su vez, simulaciones pensadas para un contexto monoproceso pueden ser fácilmente adaptadas o migradas a un entorno distribuido. El programador sólo debe los atributos requeridos al código ya existente.

6. Ejemplo de aplicación

Finalmente, en esta sección damos un ejemplo sencillo de uso del framework. Los objetos a simular serán países, implementados en la clase `Country`. Cada país tiene una propiedad que representa su población (`Population`). El código se muestra en la Fig. 2. Puede verse en el ejemplo como se encapsula la variable `population` dentro de la propiedad `Population`.

```
class Country {
    private double population;
    public double Population {
        get { return population; }
        set { population = value; }
    }
    ...
}
```

Figura 2: Código monoproceso típico

El objetivo ahora es hacer que las instancias de la clase puedan participar en una simulación HLA. Para ello, el programador sólo necesita hacer los ajustes para que la clase sea visible por la capa de replicación. Más específicamente el programador debe hacer que la clase herede de `HLAobjectRoot` (que a su vez hereda de `ContextBoundObject`) y marcarla con el atributo `HLAObjectClassAttribute` (definido a partir del atributo `Observable`). Así, el código en Fig. 2 pasaría a ser el representado el la Fig. 3.

```
[HLAObjectClassAttribute(
  Name = "Country",
  Sharing = SharingType.PublishSubscribe,
  Semantics = "A country.")]
class Country : HLAobjectRoot {
    private double population;
    [HLAAttribute(
      Name = "Population",
      Semantics = "The country population.")]
    public double Population {
        get { return population; }
        set { population = value; }
    }
    ...
}
```

Figura 3: Código distribuido

Los objetos de la clase `Country` serán a partir de aquí visibles para los miembros de la simulación global. Sea por ejemplo una simulación del mundo, en la que cada continente es simulado en un programa distinto. Cada programa no debe hacer nada para que las instancias creadas por él sean visibles para el resto. Por ejemplo, el código del programa para simular Europa podría ser:

```
class EuropeSim {
    static void Main(string[] args) {
        InitializeSimulator();
        Country spain = new Country();
        spain.Population = 45000000;
        ...
        double growRate = 1.02;
        for(int year = 1; year <= 10; year++)
            spain.Population *= growRate;
    }
}
```

Figura 4: Bucle de simulación principal

El código anterior no hace ninguna referencia a que las instancias de `Country` son parte de la simulación, eso ya fue determinado con los atributos añadidos a la misma clase. Además, los cambios en la población del país son propagados de forma transparente sin necesidad de incluir en los métodos de la propiedad (`get` y `set`) código extra.

Obsérvese también como en los atributos añadidos (anotaciones) se añade la información necesaria para el HLA/RTI.

7. Conclusiones y trabajo futuro

En este artículo hemos presentado la arquitectura de un middleware de simulación que sigue la norma IEEE-1516. Las características más importantes, y a nuestro entender, más novedosas de este middleware son:

- El estudio y aplicación, por primera vez, de técnicas de AOP para un middleware de simulación. Estas técnicas se han utilizado tanto a nivel interno para la construcción del sistema, como para facilitar la creación de simulaciones al usuario final.
- Es la primera implementación (al menos, hasta donde conocen los autores) de un middleware de simulación que sigue una filosofía no centralizada (basada en P2P) para la propagación de la información.
- La posibilidad de utilizar el middleware de replicación que sirve como base del sistema para la construcción de otros sistemas distribuidos (por ejemplo Computación P2P).
- Es la primera propuesta de una implementación en C# del estándar IEEE-1516.

Como posibles líneas de trabajo futuro, los autores se proponen usar este sistema para crear una versión distribuida de un simulador de redes similar a J-Sim [16] o a ns-2 [7]. Esto nos permitirá estudiar el comportamiento del sistema, sobre todo en ciertos aspectos que aún no han sido suficientemente probados, como por ejemplo su rendimiento en situaciones de carga alta.

Una extensión natural de nuestro trabajo es la adaptación del mismo para trabajar con protocolos JXTA [8]. Algunos servicios de JXTA son fácilmente adaptables al estándar IEEE-1516. Por ejemplo, una federación se podría implementar directamente con el concepto de grupo de JXTA. Sin embargo, otros requeriran un estudio más detallado (replicación, etc.).

Referencias

[1] The aspectsharp project. http://wiki.castleproject.org/index.php/AspectSharp.

[2] The postsharp project. http://www.postsharp.org/.

[3] Ieee std 1516-2000: Ieee standard for modeling and simulation high level architecture (hla), 2000.

[4] Standard ecma-334, C# Language Specification, 4th edition, 2006. ISO/IEC 23270:2006, http://www.ecma-international.org/publications/standards/ecma-334.htm.

[5] Adrian Colyer and Andrew Clement. Large-scale aosd for middleware. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development*, pages 56–65. ACM Press, 2004.

[6] Martin Eklof, Magnus Sparf, Farshad Moradi, and Rassul Ayani. Peer-to-peer-based resource management in support of hla-based distributed simulations. *SIMULATION*, 80:181–190, 2004.

[7] K. Fall and K. Varadhan. The ns manual. http://www.isi.edu/nsnam/ns/doc. UC Berkeley and Xerox PARC.

[8] Li Gong. Jxta: A network programming environment. *IEEE Internet Computing*, 5:88–95, 2001.

[9] Frank Hunleth, Ron Cytron, and Christopher Gill. Building customizable middleware using aspect oriented programming. In *Proceedings of the OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[10] Andrzej Kapolka. The extensible run-time infraestructure (xrti): An experimental implementation of proposed improvememtns to the high level architecture. Master's thesis, Naval Postgraduate School, Monterey, California, EEUU, 2003.

[11] Shanika Karunasekera, Scott Douglas, Egemen Tanin, and Aaron Harwood. P2p middleware for massively multi-player online games (demo). In *Proceedings of the 6th International Middleware Conference (Demonstrations Extended Abstracts)*, Grenoble, Francia, 2005.

[12] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Lecture Notes in Computer Science (Proceedings of the 11th European Conference on Object-Oriented Programming, ECOOP'97)*, volume 1241, pages 220–242. Springer-Verlag, 1997.

[13] Bjorn Knutsson, Honghui Lu, Wei Xu, and Bryan Hopkins. Peer-to-peer support for massively multiplayer games. In *Proceedings of the Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2004*, volume 1, pages 96–107. IEEE, 2004.

[14] Ingo Rammer. *Advanced .NET Remoting (C# Edition)*. Apress, 2002.

[15] Ralf Steinmetz and Klaus Wehrle. *Peer-to-Peer Systems And Applications*. Springer, 2005.

[16] Hung ying Tyan. *Design, Realization, and Evaluation of a Component-based Compositional Software Architecture for Network Simulation*. PhD thesis, The Ohio State University, 2002.

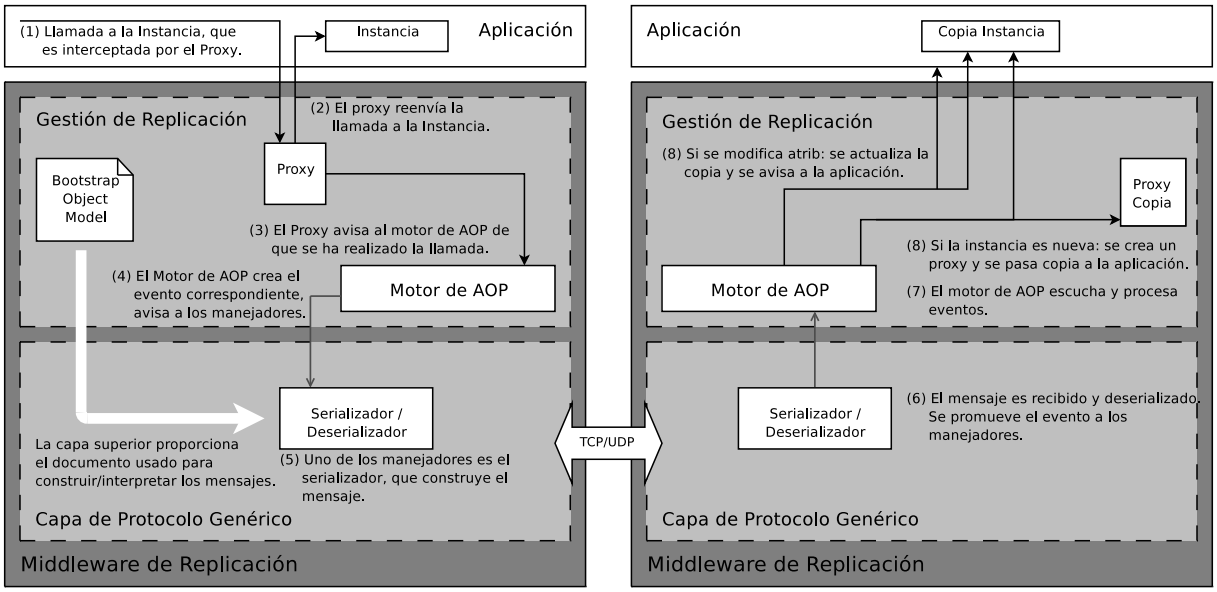


Figura 5: Middleware de replicación

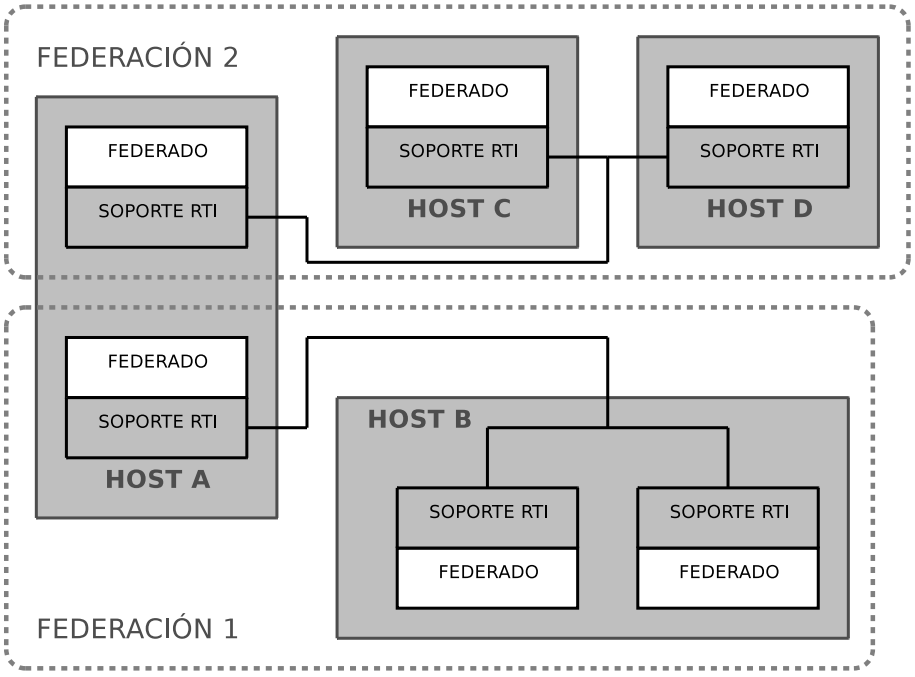


Figura 6: Conceptos básicos HLA