FRAMEWORK MULTIPLATAFORMA PARA EL DESARROLLO DE SISTEMAS MULTIAGENTE

UNIVERSIDAD DE CASTILLA-LA MANCHA
ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA
EN INFORMÁTICA

PROYECTO FIN DE CARRERA

Framework Multiplataforma para el desarrollo de Sistemas Multiagente

Luis Miguel García-Muñoz Pérez

Septiembre, 2014

UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Tecnologías y Sistemas de Información

PROYECTO FIN DE CARRERA

Framework Multiplataforma para el desarrollo de Sistemas Multiagente

Autor: Luis Miguel García-Muñoz Pérez

Director: Dr. David Vallejo Fernández

Septiembre, 2014

Luis Miguel García-Muñoz Pérez

Ciudad Real – Spain

E-mail: luismgm@gmail.com
Web site:

TRIBUNAL:

Presidente:

Secretario:

Vocal:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	**SECRETARIO**	**VOCAL**
Fdo.:	Fdo.:	Fdo.:

Resumen

Los sistemas multiagente son una tecnología para afrontar problemas en los que se requiere de un enfoque de inteligencia artificial distribuida. Se utilizan cuando se requiere que diversas entidades (como pueden ser sensores, actuadores y monitores) trabajen juntas en un entorno sometido a cambios para afrontar un problema común pero para el que sus partes no están especializadas de manera individual. Para ello, hacen uso de los llamados agentes inteligentes, que son unidades básicas de inteligencia artificial con habilidades sociales para comunicarse entre sí y con automía para alcanzar sus propios objetivos; ya sea de manera competitiva, colaborativa o una combinación de ambas.

Existen ya varios *frameworks* para facilitar el desarrollo de estos sistemas, tanto libres como privativos. Sin embargo la mayoría están muy centrados en el lenguaje de programación Java y no todos mantienen un desarrollo actualizado. Surge la necesidad de ofrecer alternativas en cuanto al lenguaje de desarrollo y también de ofrecer tanto una arquitectura distinta como un nuevo modelo de agente para facilitar el desarrollo de sistemas multiagente.

En este documento se describe un *framework* para desarrollar sistemas multiagente multiplataforma e, incluso, multilenguaje; aunque con un mayor enfoque en el desarrollo en C++. Además, esta propuesta sigue un estándar para este tipo de sistemas de la organización FIPA. Este proyecto no sólo pretende ofrecer una solución al problema del desarrollo de sistemas multiagente, sino que su desarrollo pretende ser también una herramienta de estudio de este tipo de sistemas y de la problemática asociada a ellos. A lo largo de este documento se describirá cómo se inició y desarrolló dicho proyecto: cómo se planteó, qué problemas surgieron, cómo se solucionaron, resultado final, pruebas y conclusiones.

Índice general

Índice de cuadros

Índice de figuras

Índice de listados

Listado de acrónimos

GNU	GNU is Not Unix
RPC	Remote Procedure Call
SMA	Sistema Multiagente
AMS	Agent Management System
DF	Directory Facilitator
ICE	Internet Communications Engine
GPL	General Public License
JADE	Java Agent DEvelopment Framework
FIPA	Foundation for Intelligent Physical Agents
MTS	Message Transport System
CORBA	Common Object Request Broker Architecture
API	Application Programming Interface
STL	Standard Template Library
SSL	Secure Socket Layer
GCC	GNU Compiler Collection
GDB	GNU Debugger
ACL	Agent Content Language
OpenCV	Open Source Computer Vision Library
SDL	Simple DirectMedia Layer
AID	Agent Identifier
RMI	Remote Method Invocation
JNI	Java Native Interface
JDBC	Java Database Connectivity
REST	Representational State Transfer
OMG	Object Management Group
ORB	Object Request Broker

IDL	Interface Definition Languaje
IOR	Interoperable Object References

Agradecimientos

Gracias a mis padres por permitirme estudiar lo que me gusta y a mi director de proyecto por su dirección y trabajo previo para este proyecto.

Gracias también a los profesores que me dieron clase y que dieron lo mejor de sí mismos para ello.

Luis Miguel.

A mi abuelita, que ojalá siguiera entre nosotros.

Capítulo 1

Introducción

En este capítulo se introducirá el proyecto y se hablará de algunos conceptos iniciales, la motivación que ha llevado a hacerlo y la propuesta presentada. Se terminará indicando lo que puede encontrarse en cada capítulo de la presente documentación.

1.1 Contexto

Este proyecto trata sobre sistemas multiagente. Para introducir este concepto, es necesario hablar de todo lo que lleva hacia él, es decir, desde qué es la inteligencia artificial distribuida, qué son los agentes inteligentes y cómo estos dos conceptos se relacionan para dar lugar a los sistemas multiagente. En esta sección se explicarán brevemente estos conceptos, pero puede verse una explicación más detallada en el capítulo 3.

1.1.1 Inteligencia artificial distribuida

Desde su comienzo a mediados y finales de 1970, la Inteligencia Artificial Distribuida ha evolucionado y se ha diversificado rápidamente. Hoy es un área de investigación y campo de aplicación prometedor y bien establecido, que aúna conceptos, resultados e ideas de muy diversas disciplinas: Inteligencia Artificial, Ciencias de la Computación, Sociología, Economía, Ciencias de la Organización y la Gestión y Filosofía [RN04]. Su amplio alcance y naturaleza multidisciplinar hace de la Inteligencia Artificial Distribuida un concepto difícil de definir con precisión en pocas palabras.

Una posible definición es la siguiente: *La Inteligencia Artificial Distribuida es el estudio, construcción y aplicación de sistemas multiagente, que son sistemas en los cuales varios agentes inteligentes persiguen algún conjunto de objetivos o desempeñan algún conjunto de tareas* [Wei99].

1.1.2 Agente inteligente

Un **agente inteligente**, en este contexto, puede definirse como una entidad que percibe su entorno mediante sensores, lo modifica mediante actuadores y toma decisiones de manera autónoma.

La clave del problema al que se enfrenta un agente es decidir cuál es la mejor de todas las acciones posibles que debe llevarse a cabo para satisfacer sus objetivos de diseño. La complejidad de esta toma de decisiones se encuentra afectada por diferentes propiedades del entorno en el que se encuentra, como son accesibilidad, determinismo, dinamismo, etc.

Un agente inteligente debe tener la capacidad de ser flexible. Entendemos flexibilidad como una combinación de las siguientes características [Wei99]:

Reactividad:
Los agentes son capaces de percibir su entorno y responder a éste dentro de un tiempo apropiado, para así poder satisfacer sus objetivos de diseño.

Proactividad:
Los agentes tienen que ser capaces de exhibir comportamientos dirigidos por objetivos tomando la iniciativa para satisfacer sus objetivos de diseño.

Habilidad social:
Los agentes deben ser capaces de interactuar con otros agentes para satisfacer sus objetivos de diseño.

Los agentes son usados en las aplicaciones que requieren de inteligencia artificial, es decir, en problemas para los que no existe una solución analítica o bien no podemos calcularla por problemas de espacio y/o tiempo. Por tanto, intentamos obtener una buena solución aplicando otros criterios. Algunos ejemplos de sus aplicaciones son los siguientes: videojuegos, robótica, comercio electrónico, bolsa, agentes conversacionales (conversación con humanos simulando ser un humano real), etc.

1.1.3 Sistema Multiagente (SMA)

Un SMA consiste en un conjunto de agentes operando en el mismo entorno e interactuando entre ellos [Wei99]. Por eso, el entorno debe proporcionar una infraestructura para que dichas interacciones puedan tener lugar. Esta infraestructura incluirá protocolos para que los agentes puedan comunicarse e interactuar entre ellos.

Estos protocolos de comunicación permiten a los agentes intercambiar y entender mensajes. Así los agentes pueden tener conversaciones, que consisten en intercambios estructurados de mensajes.

Las características de un SMA son las siguientes:

- Proveen una infraestructura especificando protocolos de comunicación e interacción.
- Son típicamente abiertos y no tienen un diseñador centralizado.
- Contienen agentes que son autónomos y distribuidos, y pueden ser egoístas o cooperativos.

Algunos ejemplos de aplicaciones de SMA son los siguientes:

- **Sistemas multi-robot**: donde podemos representar cada robot como un agente y, debido a la necesidad de colaboración y/o competición, necesitamos dar soporte a la comunicación entre ellos.
- **Simulación**: hay múltiples fenómenos cuyo comportamiento puede ser modelado como agentes y, por tanto para modelar la interactuación entre ellos es necesario comunicarlos.
- **Comercio electrónico**: donde se pueden modelar compradores y vendedores como agentes, permitiendo así una negociación automática entre ellos siguiendo unos parámetros dados a los agentes (límites de gasto, estrategia de negociación, etc).

Dado que los SMA trabajan en entornos complejos en los cuales la información se encuentra distribuida en distintas localizaciones, se hacen necesarias herramientas software que den soporte a sus distintas necesidades, como son las siguientes: comunicaciones entre agentes, despliegue, movilidad, etc. Proporcionar todas estas herramientas mediante un *framework* permite al desarrollador de un SMA centrarse en el problema concreto a resolver.

1.2 Motivación

Desde el punto de vista del desarrollo y despliegue de un SMA, se nos plantean una serie de cuestiones:

- El **modelo de agente**. Hay varias opciones dependiendo de la flexibilidad que se quiera darle al desarrollador y el objetivo del sistema: basados en tareas, nada en concreto pero dando unas pocas funciones para comunicarse con el resto del SMA, definiendo distintos tipos de agente, etc.
- Cómo **comunicar los agentes** entre ellos. Lo más básico es usar sockets, pero este es un nivel de abstracción muy bajo. Por ello, son interesantes otras técnicas como Remote Procedure Call (RPC) (técnica para comunicar procesos distribuidos en distintas localizaciones que permite llamar funciones remotas como si fuesen locales) o paso de mensajes.
- El **lenguaje de los agentes**. Puede permitirse que los agentes se envíen cualquier mensaje y su interpretación quede a cargo del desarrollador de cada agente o también puede definirse un lenguaje común.
- El **lenguaje de programación**. Este puede condicionar aspectos como la facilidad en el desarrollo, la eficiencia y la portabilidad del sistema.

- La **percepción de cada agente sobre el resto del sistema**. Hay que estudiar cómo los agentes se localizan entre ellos, si pueden solicitar servicios antes de saber si hay algún otro agente que lo proporciona, si son informados correctamente de las altas y bajas de otros agentes, etc.
- **Problemas inherentes a la red**. Las comunicaciones en red tienen asociado un conjunto de problemas debido a su naturaleza, como pueden ser los siguientes: altas latencias, fallos en las comunicaciones, aislamiento de algunas partes, necesidad de sincronizar información entre distintos nodos, seguridad en el canal, etc.

Un desarrollador que tuviera que atender a todo esto podría verse empleando más tiempo y esfuerzo en desarrollar todo lo necesario para soportar el sistema que en el problema a resolver en sí. Por tanto, sería una gran ayuda si todo ello ya estuviese desarrollado de manera genérica, permitiendo así centrarse únicamente en la lógica de los agentes del sistema.

Ya existen en el mercado *frameworks* para desarrollar SMA. Un ejemplo es JADE (Java Agent DEvelopment Framework) [BPR99], que permite desarrollar y desplegar SMA desarrollados con el lenguaje Java. JADE sigue el estándar FIPA (Foundation for Intelligent Physical Agents) [1] y es software libre bajo licencia GPL (General Public License).

Usar JADE pondría dos limitaciones importantes:

- Obliga a usar el lenguaje Java, que por requisitos de diseño y/o eficiencia podría no ser el más adecuado para nuestros agentes.
- Al usar este lenguaje, puede haber problemas de dependencia de Oracle, que es la empresa dueña de él. Es lo que se conoce popularmente como *casarse con el software*.

1.3 Propuesta

Este proyecto propone como solución al problema anterior el desarrollo de un nuevo *framework* para SMA. Este nuevo *framework* proporciona infraestructura software y servicios asociados al desarrollo de sistemas multiagente, además de tener capacidad multiplataforma. Ha sido desarrollado, principalmente, en lenguaje C++, además de que todo lo que usa (bibliotecas, middleware, ...) es multiplataforma, por lo que puede funcionar en los principales sistemas operativos modernos.

Se parte del proyecto de fin de carrera de David Vallejo Fernández, que consistió en un sistema distribuido de rendering. Una parte de dicho sistema era un sistema multiagente de propósito general que respetase el estándar FIPA, utilizando el middleware ZeroC ICE (Internet Communications Engine) para dar soporte a las comunicaciones. Este middleware consiste en una capa intermedia de software que facilita la comunicación por red utilizando RPC, además de otros servicios útiles.

[1] http://www.fipa.org/

El proyecto de David estaba centrado en la aplicación de la tecnología de agentes, por lo que el SMA de base no era genérico y no estaba planteado como un *framework* para el desarrollo de SMA. Así, esta primera aproximación tenia ciertas limitaciones, como son el requerimiento de que el usuario reimplemente las interfaces de ICE para desarrollar los agentes, la falta de un modelo definido de agente y el soporte parcial al lenguaje de comunicación de agentes que define el estándar FIPA. Por ello se pretende ampliar dicho proyecto, añadiendo las características comentadas en la siguiente sección y reescribiendo todo aquello que sea necesario.

Partiendo de lo anterior se ha diseñado y desarrollado un nuevo *framework* multiplataforma para el desarrollo, configuración y despliegue de aplicaciones software basadas en agentes inteligentes. El nuevo *framework* tiene características como las siguientes:

- Un nuevo **modelo de agente**. Los agentes están compuestos de un tipo de tareas que se denominan comportamientos (**behaviours**), las cuales pueden ser de distintos tipos: simples, compuestas de otros subcomportamientos, cíclicas, etc.
- **Encapsulación de las comunicaciones**. Se continua usando el middleware de comunicaciones ZeroC ICE, añadiendo más características aprovechando los servicios que ofrece, como los canales de eventos. Este middleware permite conectar clientes/servidores desarrollados en distintos lenguajes de programación.
- Servicios de **búsqueda de agentes**:
  - Agent Management System (AMS): es el servicio de páginas blancas, que registra todos los agentes del SMA.
  - Directory Facilitator (DF): es el servicio de páginas amarillas, que permite buscar agentes por el servicio que dan.
- **Replicación** y **balanceo de carga**. Todos los servicios básicos (a saber: AMS, DF, MTS y las fábricas de agentes) poseen estas características, mejorando la robustez y tolerancia a fallos del sistema.
- Un sistema de **paso de mensajes** entre los agentes. Ahora los agentes pueden pasarse mensajes directamente entre ellos o utilizando el MTS, además de la posibilidad de suscribrise a canales de eventos.

En la figura 1.1 se muestra la arquitectura general del *framework* propuesto. Podemos identificar los siguientes componentes:

- Cada **host** (equipo físico) contiene uno o varios nodos.
- Cada **nodo** contiene **servicios** del *framework* y/o agentes.
- Los nodos que contienen agentes contienen también la **fábrica de agentes**, que es el servicio encargado de crear nuevos agentes en cada nodo.

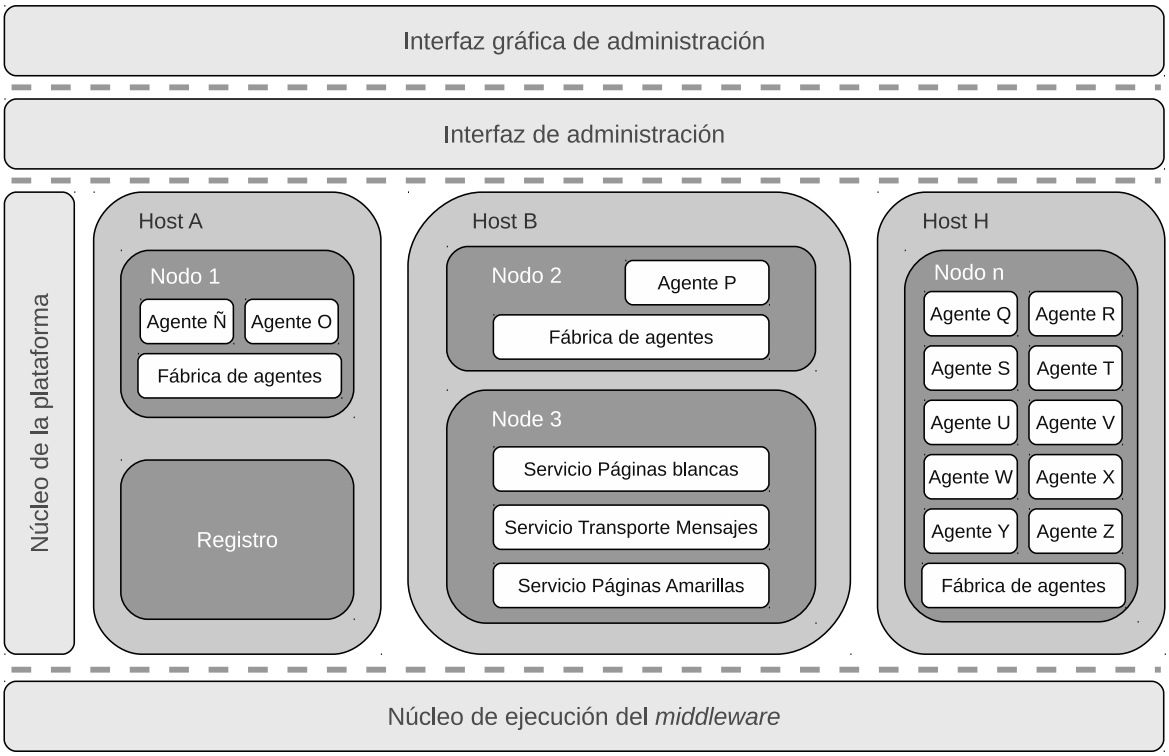

Figura 1.1: Esquema que representa un posible despliegue de un sistema multiagente desarrollado con este *framework*.

- Hay servicios de **páginas blancas** (AMS) y páginas amarillas (DF), que permiten que los agentes encuentren otros agentes en en sistema.
- Hay servicios de **transporte de mensajes**, que es una de las formas a través de la cual los agentes pueden comunicarse entre ellos.
- Existe un servicio llamado **registro**, el cual permite encontrar cualquier otro elemento del sistema sin necesidad de conocer su ubicación física.
- El núcleo de ejecución del ***middleware*** es la parte que encapsula las comunicaciones mediante RPC. .
- Hay una **interfaz de administración** que permite gestionar el despliegue de todo el sistema.
- También hay una **interfaz gráfica** de administración que permite que el administrador del sistema gestione el despliegue de manera amigable.

Para el RPC que encapsula las comunicaciones se ha hecho uso del middleware ZeroC ICE [Hen04], cuyos servicios facilitan también la gestión de los nodos, la administración y el registro.

De este modo, un desarrollador puede desarrollar y desplegar un sistema multiagente siguiendo los siguientes pasos:

- Definir cada uno de los agentes que participarán siguiendo el modelo de agente propuesto por el *framework*.
- Decidir qué servicios va a contener cada nodo.
- Gestionar el despliegue del sistema mediante la interfaz de administración.

Todo esto permite al desarrollador centrarse en la lógica de dominio de su problema a resolver, evitando emplear tiempo en tareas como el modelo de agente a usar, cómo paralelizar, cómo comunicar los agentes entre sí, etc. Además, este *framework* es flexible y permite que un desarrollador con conocimientos más avanzados de ICE utilice funciones adicionales o aproveche mejor lo que ya se ofrece.

En el presente proyecto, el *framework* se ha usado para implementar desde el principio una aplicación de SMA que consiste en un sistema de videovigilancia.

1.3.1 Caso de estudio: Sistema de videovigilancia

Para aportar una prueba que demuestre tanto la funcionalidad del *framework* como su viabilidad para el desarrollo de sistemas multiagente, este proyecto propone además un caso de estudio de una posible aplicación real del *framework*.

La propuesta de caso de estudio consiste en modelar un sistema de videovigilancia con agentes. Se trata de obtener un escenario por el que se mueven un conjunto de objetos móviles (coches y peatones), definir un conjunto de áreas y mostrar cuando estos objetos pasan sobre dichas áreas y en qué porcentaje las cubren.

Primero se obtiene el vídeo original y las áreas que se quieren vigilar desde un archivo XML. La lectura de este archivo es realizada por un primer agente, el cual crea desde él otros agentes los cuales monitorizan las áreas que serán vigiladas. A continuación, un nuevo agente lee la salida de un tracker de vídeo, el cual ha capturado los objetos que se mueven en la escena. Este agente crea nuevos agentes que monitorizan dichos objetos (o los actualiza si el agente ya existía) y, a su vez, estos agentes comunican su posición para que los agentes que representan las áreas la conozcan, calculen si están sobre ella y en qué porcentaje y, finalmente escriban los resultados a un archivo de salida. Posteriormente a la ejecución de este sistema, la salida de las áreas es leída con un script para ser representada sobre el vídeo original.

1.4 Estructura del documento

Capítulo 1: Introducción

El capítulo actual, donde se introduce el proyecto.

Capítulo 2: Objetivos

Se presentan los objetivos que pretenden cumplirse con este proyecto.

Capítulo 3: Estado del arte

Se habla del estado actual de la inteligencia artificial distribuida, los sistemas multiagente, tecnologías relacionadas y otros *frameworks* existentes.

Capítulo 4: Método de trabajo

Se describen la metodología de trabajo y las herramientas utilizadas durante el desarrollo del proyecto.

Capítulo 5: Arquitectura

Se presenta y desarrolla la arquitectura que da soporte al *framework* presentado en este proyecto y se detallan sus diferentes partes y funciones.

Capítulo 6: Evolución, resultados y costes

Se discute la evolución del proyecto, los resultados obtenidos y el coste del mismo.

Capítulo 7: Conclusiones y trabajo futuro

Se describen las conclusiones obtenidas de todo el desempeño del proyecto y las líneas futuras de trabajo a partir de lo presentado.

Apéndice A: Compilación y ejemplo de despliegue

Un manual para compilar el código presentado junto al proyecto y para desplegar sistemas multiagente con él.

Apéndice B: Interfaces Slice

Se citan las interfaces que permiten acceder a las operaciones remotas que ofrece este *framework*.

Capítulo 2

Objetivos

En este capítulo va a plantearse el objetivo general del proyecto y, a continuación, una descripción más detallada de los distintos objetivos funcionales y didácticos del proyecto.

2.1 Objetivo general

El objetivo general de este proyecto es el diseño e implementación de un *framework* que proporcione la base y las herramientas para crear y desplegar soluciones basadas en agentes software. Esto viene motivado por los siguientes aspectos:

- Necesidad de un *framework* que permita desarrollar y desplegar soluciones a problemas de inteligencia artificial distribuida.
- Necesidad de un framework que posibilite el desarrollo en distintos lenguajes de programación, al contrario que JADE (ver detalles en el capítulo 3).
- El uso e integración del *middleware* ZeroC ICE en un proyecto de estas características.
- Un enfoque didáctico orientado, no solo a realizar un proyecto real, sino también al estudio de problemas que surgen en un proyecto de esta magnitud y estas características.

2.2 Objetivos específicos

Los objetivos específicos están divididos en funcionales (funcionalidades que se esperan del proyecto) y didácticos (conocimientos que se esperan adquirir con el proyecto).

2.2.1 Objetivos funcionales

- **Modelo de agente**: este framework deberá establecer un modelo de agente que permita expresarlos de una manera adecuada.
- **Interoperatibilidad y seguimiento de estándares**: los servicios que proporciona este framework se basan en el estándar FIPA y se ajustarán a él en la medida de lo posible.
- **Facilitación de las comunicaciones en red**: deben proporcionarse facilidades para las comunicaciones en red entre los agentes y de estos con los servicios del framework, sin tener que recurrir a herramientas de más bajo nivel como los sockets.

- **Alta disponibilidad**: este framework deberá hacer lo posible por mantener la disponibilidad de sus servicios ante problemas de red y/o en los nodos que los contienen. Para ello se usarán técnicas de replicación y de tolerancia a fallos.
- **Robustez ante gran número de peticiones**: se deben proporcionar servicios para los cuales puedan distribuirse las peticiones que se les hacen siguiendo determinadas políticas de balanceo de carga.
- **Curva de aprendizaje reducida para desarrolladores**: se facilitará, en la medida de lo posible, el entendimiento de este framework y sus funcionalidades a los desarrolladores que vayan a usarlo. Para ello, se proporcionará un manual de uso con ejemplos. Esta documentación aparece en la presente memoria.
- **Aplicación real**: para probar que el framework es útil en problemas reales, se realizará un ejemplo de una solución a un problema representada con agentes utilizando este framework. Dicha solución se desplegará y se mostrarán los resultados.

2.2.2 Objetivos didácticos

- **Estudio y aprendizaje de C++**: se utilizará este lenguaje como lenguaje principal para el desarrollo del proyecto, permitiendo estudiarlo en profundidad en una aplicación real.
- ***Middleware***: se estudiará y utilizará un middleware de comunicaciones complejo, que será ZeroC ICE.
- **Sistemas distribuidos**: las aplicaciones que se desarrollen con este framework serán, a efectos prácticos, sistemas distribuidos. Por tanto, se estudiarán conceptos relacionados con ellos, como pueden ser la replicación de nodos, la tolerancia a fallos, el balanceo de carga y todos los problemas asociados al uso de una red.
- **Reutilización de código**: el proyecto partirá de un proyecto previamente existente, por lo que tendrá que trabajarse con código ajeno y/o modificarlo.
- **Interoperatibilidad entre lenguajes**: este proyecto utilizará varios lenguajes distintos y se estudiará cómo hacerlos trabajar juntos.
- **Proyecto complejo**: será un proyecto prolongado en el tiempo y con múltiples requisitos.
- **Organización del trabajo**: deberá encontrarse un método de trabajo que permita terminar el proyecto y compaginarlo con otras actividades, como pueden ser las asignaturas de la carrera.
- **Estimación de costes**: se estudiarán los costes que se generan en el desarrollo de un proyecto de estas características.

Capítulo 3

Estado del arte

Este capítulo introduce algunos antecedentes y tecnologías que ponen en contexto el proyecto. Se tratan conceptos teóricos para informar al lector de los fundamentos y también tecnologías que son interesantes por ser predecesoras, por ofrecer funcionalidad similar o que incluso se han usado dentro del proyecto.

3.1 Agentes inteligentes

Un agente es un sistema computacional que está situado en algún entorno y es capaz de relizar acciones autónomas en este entorno con el propósito de alcanzar sus objetivos delegados [WJ95]. En esta definición debe resaltarse lo siguiente:

1. No se refiere a *agente inteligente*, sino sólo a *agente*.
2. No dice nada sobre a qué tipo de *entorno* se refiere.
3. No define lo que es *autonomía*.

Las dos primeras se explicarán más adelante. Sobre la **autonomía**, no es un concepto fácil de definir. Se puede decir que autonomía, en este caso, significa que los agentes son capaces de actuar sin la intervención de humanos u otros sistemas, es decir, tienen control sobre su estado interno y su comportamiento [Wei13].

Un agente tiene un repertorio de acciones que es capaz de realizar. Esto se denomina su **capacidad efectiva**, es decir, la manera en la que modifica el entorno. No todas estas acciones pueden realizarse en cualquier situación. Por ello, el problema clave que deben resolver los agentes es decidir qué accion tomar en cada situación.

La complejidad del proceso de selección de acción puede estar afectada por una variedad de diferentes propiedades del entorno. Russel y Norvig sugieren la siguiente clasificación de **propiedades del entorno** [RN04]:

- **Accesible** o **inaccesible**.

  Un entorno accesible es aquel en el que el agente puede obtener información completa, precisa y actualizada sobre el estado del entorno. Muchos entornos moderadamente complejos son inaccesibles (como puede ser el mundo físico o internet). Cuanto más accesible sea un entorno, más sencilla es la creación de agentes que trabajen en él.

- **Determinista** o **no determinista**.

  Un entorno determinista es aquel en el que cada acción tenga un sólo efecto garantizado, es decir, que no haya incertidumbre acerca del estado que resultará al efectuar una acción. El mundo físico es, a todos los efectos, no determinista. Los entornos no deterministas presentan grandes problemas para el diseño de agentes.

- **Episódico** o **no episódico**.

  En un entorno episódico, el rendimiento de un agente es dependiente de un cierto número de epsisodios discretos, sin relación entre en rendimiento del agente en escenarios diferentes. Un ejemplo de este tipo de entorno puede ser un sistema de ordenamiento de correo. Los entornos episódicos son mucho más sencillos desde la perspectiva del desarrollador de agentes porque el eagente puede decidir qué acción tomar basándose sólo en el episodio actual, es decir, no necesita razonar sobre las interacciones entre el actual y los futuros episodios.

- **Estático** o **dinámico**.

  Un entorno estático es aquel en el que se puede asumir que no va a haber cambios excepto por la actuación de los agentes. Un entorno dinámico es aquel en el que hay otros procesos operando en el y, por tanto, ocurren cambios que estan fuera del control de los agentes. El mundo físico es un entorno altamente dinámico.

- **Discreto** o **contínuo**.

  Un entorno es discreto si hay un número fijo y finito de acciones y percepciones en él. Un ejemplo de entorno dsicreto puede ser un juego de ajedrez, mientras que la conducción de un taxi sería un entorno contínuo.

Por tanto, la clase más compleja de entornos serán aquellos que sean inaccesibles, no deterministas, no episódicos, dinámicos y contínuos.

Un agente que fuese, por ejemplo, un termostato o un servicio de UNIX no podría ser definido como **agente inteligente**. Por tanto, para considerar a un agente inteligente éste debe exhibir los siguientes tipos de comportamiento para obtener sus objetivos delegados [WJ95]:

- **Proactividad**: los agentes inteligentes son capaces de exhibir un comportamiento dirigido por sus objetivos tomando la iniciativa para poder satisfacerlos.
- **Reactividad**: los agentes inteligentes son capaces de percibir su entorno y de responder en un tiempo determinado a los cambios que ocurren en él para poder satisfacer sus objetivos delegados.
- **Habilidades sociales**: los agentes inteligentes son capaces de interactuar con otros agentes (e incluso con humanos) para poder satisfacer sus objetivos delegados.

3.2 Sistemas Multiagente

Cada agente es una entidad individual capaz de realizar acciones de manera independiente. Sin embargo, muchas aplicaciones requieren la interacción entre diversos individuos para poder alcanzar un determinado objetivo. Un ejemplo puede ser el de una red de sensores que monitoriza el tráfico en una intersección muy concurrida. Si planteamos cada sensor como un agente inteligente, entonces éstos deberían tener la capacidad de coordinar sus actividades. Estos sistemas, compuestos de varios agentes, son denominados Sistema Multiagente (SMA) [Wei13].

Un SMA a veces no puede ser descrito completamente como la suma de todas las descripciones de los agentes que hay en él. No sólo es que no todos los problemas puedan ser fácilmente descritos en términos de estados mentales individuales, sino que, en muchos casos, determinadas situaciones son descritas mejor basándose en actividades y restricciones que caracterizan el comportamiento exterior observable de toda la población. Por ejemplo, puede pensarse en recursos cuya propiedad sea colectiva para todos los miembros de una comunidad, como puede ser una conexión a internet compartida. Si cada agente sigue su propio objetivo de tener la máxima frecuencia de accesos a dicha conexión, pronto la calidad de la conexión disminuirá porque los agentes se estarán limitando unos a otros. Un ejemplo como éste muestra la necesidad de organizar de alguna manera los agentes para que todos se beneficien. A veces, esta organización surgirá de las interacciones entre los agentes, pero en muchos casos es otra entidad con sus propios objetivos la que regula los agentes en el entorno.

Un sistema mulitagente puede ser definido como una organización o sociedad de agentes, donde un conjunto de agentes interactuan y cooperan entre sí para alcanzar algún objetivo colectivo [FGM04]. Un sistema multiagente es un sistema compuesto de múltiples agentes inteligentes que interactuan para solucionar problemas que están más allá de las capacidades o conocimiento individual de cada uno de ellos.

Un sistema multiagente está caracterizado por lo siguiente [Syc98]:

- Cada agente tiene información incompleta o no tiene la suficiente capacidad para resolver el problema global. Es decir, tiene un punto de vista limitado.
- No existe un control global del sistema.
- Los datos están descentralizados.
- Los cálculos son asíncronos.

Por tanto, los sistems multiagente pueden ser vistos como solucionadores de problemas distribuidos [DR94]. Son usados para resolver problemas que serían imposibles o muy difíciles de resolver por un sistema monolítico o por un único agente.

3.3 FIPA

FIPA (Foundation for Intelligent Physical Agents) [1] es una organización internacional que está dedicada a promover la industria de los agentes inteligentes mediante el desarrollo de especificaciones abiertas que den soporte a agentes y a aplicaciones basadas en agentes. Esta labor se realiza mediante colaboración abierta con sus organizaciones miembro, las cuales son empresas y universidades que están activas en el campo de los agentes. FIPA hace que el resultado de sus actividades esté disponible a todas las partes interesadas e intenta contribuir con sus resultados a la formación de estándares formales.

Los miembros de FIPA compiten abiertamente, individual y colectivamente, en el desarrollo de aplicaciones basadas en agentes. La membresía en FIPA está abierta a cualquier grupo sin restricciones. En particular, sus miembros no están obligados a implementar o usar estándares específicos basados en agentes, recomendaciones o especificaciones de FIPA sólo por participar en en ella.

Las especificaciones de FIPA son desarrolladas a través de participación directa en su membresía. El estatus de una especificación puede ser preliminar, experimental, *deprecated* u obsoleto.

El estándar de FIPA al cual se refiere este proyecto se llama *FIPA Agent Management Specification*. Este estándar define un *framework* en el cual los agentes de FIPA existen y cooperan entre ellos. Establece el modelo lógico de referencia para creación, registro, localización, comunicación, migración y eliminación de agentes.

Las entidades contenidas en el modelo de referencia (ver figura 3.1) son conjuntos lógicos de capacidades (esto es, servicios) y no implican ninguna configuración física. Adicionalmente, los detalles de implementación de cada plataforma de agentes y sus agentes son decisiones de diseño de cada desarrollador individual de sistemas de agentes.

El modelo de referencia de gestión de agentes consiste en los siguientes componentes lógicos, cada uno representando un conjunto de capacidades, las cuales pueden ser combinadas en las implementaciones físicas de plataformas de agentes:

- Un **agente** es un proceso computacional que implementa la funcionalidad autónoma y comunicativa de una palicación. Los agentes se comunican utilizando el lenguage ACL (Agent Content Language). Un agente es un actor fundamental en una plataforma de agentes y combina una o más capacidades de servicio, tal y como son publicadas en las descripciones de servicio, en un modelo de ejecución unificado e integrado. Un agente debe tener al menos un propietario, por ejemplo, basado en afiliación a una organización o propiedad de un usuario humano y un agente debe soportar al menos una noción de identidad. Esta noción de identidad es el identificador de agente (Agent Identifier (AID)) que etiqueta un agente para que pueda ser distinguido unequívocamente

[1] http://www.fipa.org/

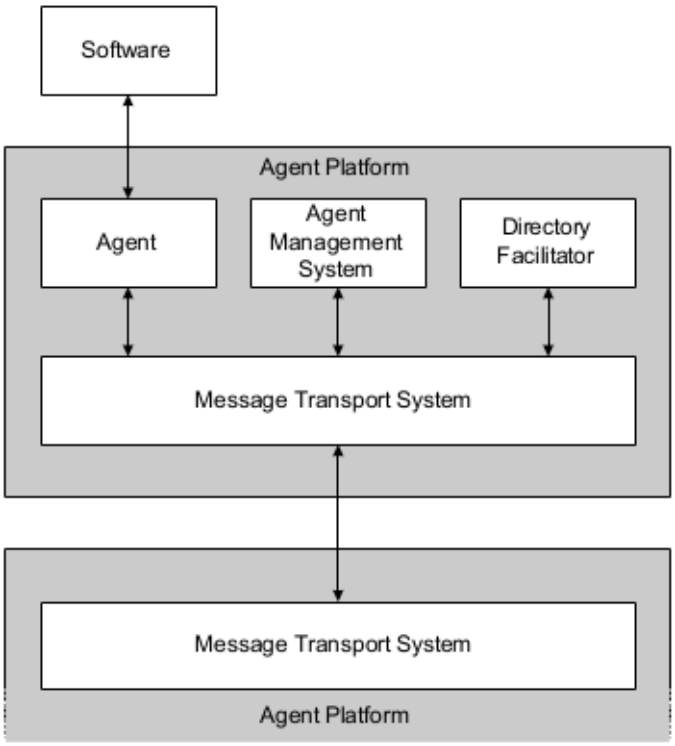

Figura 3.1: Modelo de referencia de gestión de agentes.

dentro del universo de agentes. Un agente podria ser registrado en varias direcciones de transporte por las cuales puede ser contactado.

- El **Directory Facilitator (DF)** es un componente opcional de la plataforma de agentes pero, si está presente, debe ser implementado como un servicio. El DF provee el servicio de páginas amarillas a otros agentes. Los agentes podrían registrar sus servicios en el DF o consultar al DF para buscar qué servicios son ofrecidos por otros agentes. Múltiples DF dentro de una plataforma pueden estar federados.
- El **Agent Management System (AMS)** es un componente obligatorio de la plataforma de agentes. El AMS ejerce control supervisorio sobre el acceso y uso de la plataforma. Sólo un AMS debe existir en cada plataforma. El AMS mantiene un directorio de AIDs el cual contiene direcciones de transporte, entre otras cosas, para los agentes registrados en la plataforma. El AMS ofrece servicios de páginas blancas a los agentes. Cada agente debe registrarse con un AMS para obtener una AID válida.
- El **Message Transport System (MTS)** es el método de comunicación por defecto entre agentes en diferentes plataformas.
- Una **plataforma de agentes** provee de la infraestructura física en la cual los agentes pueden ser desplegados. La plataforma consiste en la máquina, sistema operativo, sofware de soporte para los agentes, componentes de gestión de agentes de FIPA (DF, AMS y MTS) y agentes.

Nombre	Lenguaje	Estándar	Libre	Artículo relevante
JADE	Java	FIPA	Sí	[BCG07]
Jadex	Java	FIPA	Sí	[PBL05]
Cougaar	Java	No	Sí	[HW05]
Agent Factory	Java	FIPA	Sí	[RCO05]
JACK	Java	No	No	[Win05]

Cuadro 3.1: Plataformas de desarrollo de SMA relevantes en la actualidad.

El diseño interno de una plataforma de agentes es asunto de los desarrolladores de sistemas de agentes y no está sujeto a estandarización por FIPA. Las plataformas de agentes y los agentes nativos a esas plataformas, ya sea por creación directa o por migración a la plataforma, podrían usar cualquier método propietario de intercomunicación.

Debe hacerse notar que el concepto de plataforma de agentes no significa que todos los agentes residentes en una plataforma tengan que estar localizados en el mismo *host*. FIPA visualiza una variedad de diferentes plataformas, desde procesos únicos conteniendo hilos de agentes hasta plataformas completamente distribuidas construidas alrededor de estándares *middleware* propietarios o abiertos.

FIPA solo se preocupa de cómo se lleva la comunicación entre agentes nativos a una plataforma y agentes fuera de la plataforma. Los agentes son libres para intercambiar mensajes directamente por cualesquiera medios que soporten.

- El ***software*** describe todas las ejecuciones ejecutables de instrucciones (no agentes) accesibles a través de un agente. Los agentes podrían acceder a *software*, por ejemplo, para añadir nuevos servicios, adquirir nuevos protocolos de comunicación, adquirir nuevos protocolos y/o algoritmos de seguridad, adquirir nuevos protocolos de negociación y acceder a herramientas para soportar la migración.

Puede consultarse el estándar completo en el sitio web de FIPA.

3.4 *Frameworks* para sistemas multiagente

En este apartado se nombrarán algunos *frameworks* para desarrollar sistemas multiagente y, más en particular, se detallará el más importante de ellos: JADE.

3.4.1 *Frameworks* existentes para el desarrollo de SMA

En la tabla 3.1 se citan algunos *frameworks* ya existentes en la actualidad para el desarrollo de sistemas multiagente. En este documento, va a detallarse más en profundidad JADE; por considerarlo el más importante de ellos y por haber inspirado algunas características de este proyecto.

3.4.2 JADE

JADE (Java Agent DEvelopment Framework) es un middleware desarrollado por TILAB para el desarrollo de aplicaciones multiagente distribuidas basadas en la arquitectura de comunicación P2P.

JADE es un framework completamente desarrollado en Java. Simplifica la implementación de sistemas multiagente mediante un middleware que respeta el estándar FIPA y un conjunto de herramientas que soportan las fases de depuración y despliegue. La plataforma de agentes puede ser distribuida a través de distintas máquinas (que no tienen que tener necesariamente el mismo sistema operativo) y la configuración puede controlarse mediante una interfaz gráfica de usuario (GUI) remota. La configuración puede cambiarse en tiempo de ejecución añadiendo nuevos agentes o moviéndolos de una máquina a otra, cuándo y cómo se requiera. El único requisito es el entorno de ejecución Java versión 5 o superior (JRE $\geq$ 1.5).

La arquitectura de comunicaciones ofrece un sistema de mensajería eficiente y flexible, donde JADE crea y administra una cola de mensajes ACL entrantes, privados para cada agente. Los agentes pueden acceder a su cola mediante una combinación de varios modos: bloqueo, reconocimiento de patrones, etc. El modelo de comunicación FIPA completo ha sido implementado y sus componentes están claramente distinguidos y completamente integrados: protocolos de interacción, ACL, lenguajes de contenido, esquemas de codificación, ontologías y protocolos de transporte. El mecanismo de transporte, en particular, puede adaptarse a cada situación, eligiendo de manera transparente el mejor protocolo disponible. La mayoría de los protocolos de interacción definidos por FIPA están disponibles y pueden ser instanciados después de de definir el comportamiento dependiente de la aplicación en cada estado del protocolo. SL y la ontología de administración de agentes ha sido implementada también, así como el soporte para lenguajes definidos por el usuario y ontologías que pueden ser implementadas, registradas con los agentes y automáticamente usadas por el framework.

JADE está siendo usado por muchas compañías y grupos académicos, como BT, Telefónica, CNET, Universidad de Helsinky, INRIA, ATOS y muchos otros. JADE es software libre bajo la licencia LGPL.

Características técnicas/funcionales

- Aplicación multiplataforma distribuida con comunicación P2P.
- Respeta el estándar FIPA.
- Servicio de páginas blancas (AMS) y páginas amarillas (DF) con la posibilidad de crear grafos de federación en tiempo de ejecución.
- Herramientas gráficas para las fases de depuración, administración y monitorización.
- Soporte para migración de agentes y su estado de ejecución.

- Soporte de protocolos de interacción complejos.
- Soporte para creación y administración del contenido de los mensajes, incluyendo XML y RDF.
- Soporte de integración en páginas JSP.
- Soporte de seguridad a nivel de aplicación (sólo en J2SE).
- Protocolos de transporte elegibles en tiempo de ejecución. Actualmente: Java RMI, JICP (protocolo propiedad de JADE), HTTP y IIOP.

Ventajas y desventajas

Como ventajas de este *framework*, pueden nombrarse las siguientes:

- Es un proyecto de software libre (licencia LGPL), lo que permite a usuarios y desarrolladores colaborar mejor entre ellos.
- Simplifica el desarrollo de aplicaciones distribuidas compuestas de entidades autónomas que necesiten colaborar para hacer trabajar todo el sistema.
- Simplifica el desarrollo de aplicaciones que requieren de negociación y coordinación entre un conjunto de agentes, donde los recursos y la lógica de control están distribuidos en el entorno.
- Los agentes de JADE controlan su propio hilo de ejecución y, por tanto, pueden ser programados para iniciar la ejecución de acciones sin intervención humana. A esta característica se la llama proactividad.
- Permite el desarrollo más eficiente de aplicaciones multiplataforma, ya que el modelo P2P es más eficiente que el modelo cliente-servidor.
- JADE respeta el estándar FIPA, que permite la interoperatibilidad entre agentes de diferentes plataformas.
- JADE provee de un conjunto homogéneo de APIs que son independientes de la red y la versión de Java. En la práctica, provee las mismas APIs para los entornos J2EE, J2SE y J2ME. Esto permite a los desarrolladores reusar el mísmo código de aplicación para distintos dispositivos (como pueden ser PC y teléfono móvil).
- Las APIs de JADE son fáciles de aprender y de usar. JADE ha sido diseñado para simplificar las comunicaciones y el transporte de mensajes, haciendo transparente para el desarrollador la administración de las diferentes capas de comunicaciones usadas para enviar un mensaje desde un agente hasta otro, permitiéndole así concentrarse en la lógica de la aplicación.

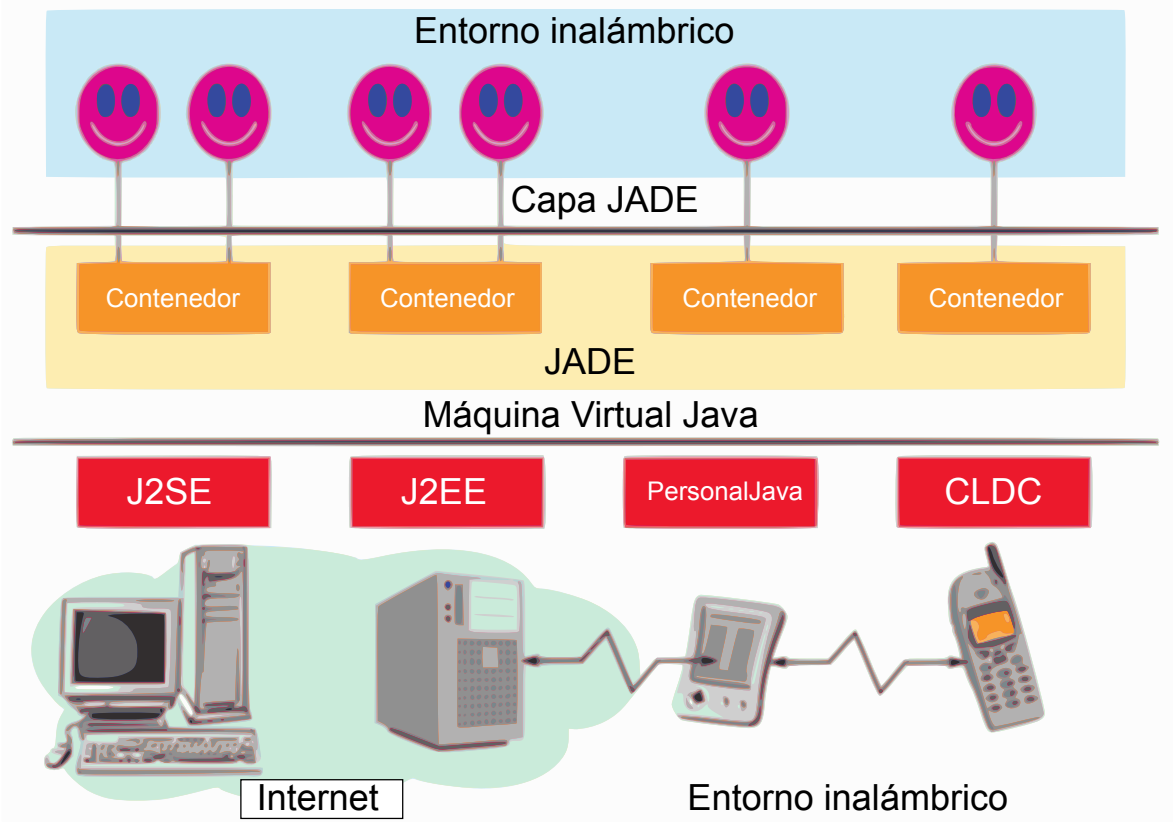

Figura 3.2: Arquitectura JADE.

También existen las siguientes desventajas:

- Sólo permite usar Java como lenguaje de programación.
- Java es un lenguaje interpretado, por lo que tiene peor rendimiento que otros lenguajes compilados.
- Aunque Java es libre en principio, está en una situación delicada después de la compra de Sun por parte de Oracle (patentes, etc.).

Arquitectura

JADE incluye tanto las bibliotecas requeridas para desarrollar aplicaciones de agentes como el entorno de ejecución que provee los servicios básicos y debe estar activo en el dispositivo antes de que los agentes sean ejecutados. A cada instancia del entorno de ejecución de JADE se le llama contenedor (ya que *contiene* agentes). Al conjunto de contenedores se le llama plataforma y provee de una capa homogénea que oculta a los agentes (y también a los desarrolladores) la complejidad y diversidad de las capas inferiores (hardware, sistema operativo, tipo de red, JVM).

Como puede verse en la figura 3.2, JADE es compatible con el entorno J2ME CLDC/-MIDP1.0. Ha sido probado con la red GPRS en diferentes terminales móviles. La memoria que ocupa el entorno de ejecución JADE, en un entorno MIDP1.0, es de alrededor de 100

KiB, pero puede ser reducida hasta los 50 KiB usando la técnica de ROMizing (compilando JADE junto con la JVM). JADE es extremadamente versátil y por eso, no sólo se ajusta a las limitaciones de entornos con recursos limitados, sino que también se integra bien en arquitecturas complejas como .NET o J2EE donde JADE se convierte en un servicio para ejecutar aplicaciones multiplataforma. El limitado uso de memoria que hace JADE permite instalarlo en todos los teléfonos móviles que puedan ejecutar aplicaciones Java.

Programación de agentes

Los agentes en JADE siguen un modelo basado en comportamientos. Un comportamiento representa una tarea que un agente puede hacer y es implementada como un objeto de una clase que extiende a jade.core.behaviours.Behaviour.

Un agente puede ejecutar varios comportamientos concurrentemente. Sin embargo, es importante informar de que la planificación de comportamientos en un agente no es preventiva (como en los hilos Java) sino cooperativa. Esto significa que cuando un comportamiento está planificado se llama a su método action() y se ejecuta hasta que termina. Así que es el programador el que define cuando un agente cambia desde la ejecución de un comportamiento al siguiente.

A pesar de que requiere un pequeño esfuerzo adicional por parte de los programadores, este enfoque tiene varias ventajas:

- Permite tener un solo hilo Java por cada agente. Esto es muy importante especialmente en entornos con recursos limitados, como los teléfonos móviles.
- Provee mejor rendimiento ya que el cambio entre comportamientos es mucho más rápido que el cambio entre hilos en Java.
- Elimina todos los elementos de sincronización entre los comportamientos concurrentes accediendo a los mismos recursos, ya que todos los comportamientos son ejecutados por el mismo hilo Java. Esto mejora el rendimiento.
- Cuando ocurre un cambio de comportamiento, el estado de un agente no incluye ninguna información de pila y es posible tomar una imagen (*snapshot*) de él. Esto hace posible implementar importantes características avanzadas, como guardar el estado de un agente en un almacenamiento persistente para una posterior reanudación (persistencia de agentes) o transferirlo a otro contenedor para ejecución remota (movilidad de agentes).

Tipos de comportamiento

En la figura 3.4 puede verse el diagrama de clases de todos los comportamientos ofrecidos por JADE. A continuación, se detallará la funcionalidad ofrecida por cada uno de ellos:

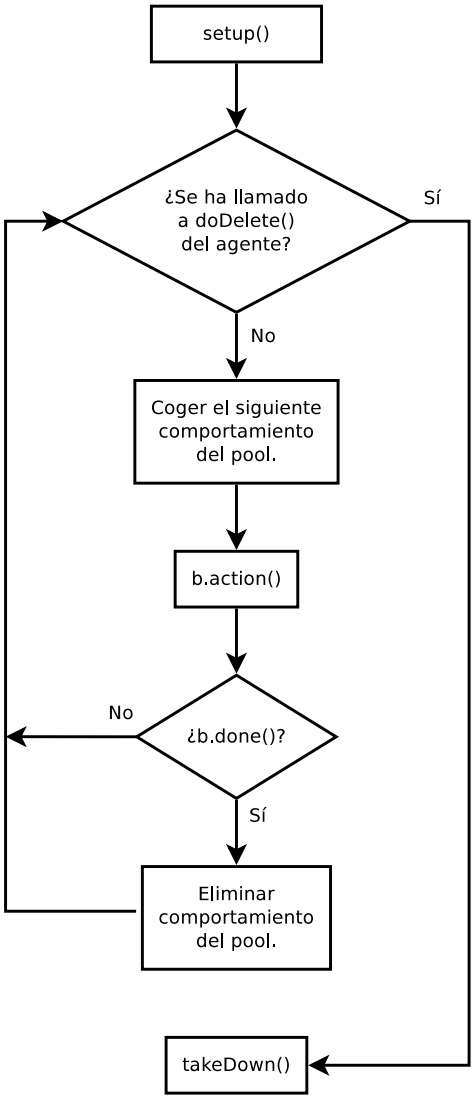

Figura 3.3: Camino de ejecución de un hilo de agente.

- **Comportamientos one-shot**: Este tipo de comportamientos se ejecutan una sola vez, es decir, modelan una tarea atómica. La clase jade.core.behaviours.OneShotBehaviour ya implementa el método done() devolviendo «verdadero», y puede ser extendida para implementar estos comportamientos.
- **Comportamientos cíclicos**: Son comportamientos que nunca se completan y cuyo método action() ejecuta las mismas operaciones cada vez que es llamado, es decir, modelan una tarea cíclica. La clase jade.core.behaviours.CyclicBehaviour implementa el método done() devolviendo siempre «falso», y puede ser extendida para implementar estos comportamientos.
- **Comportamientos complejos**: Estos comportamientos sirven para implementar tareas compuestas por otras subtareas (tareas complejas). Pueden ser de tres tipos:
  - FSMBehaviour: sus subtareas corresponden a estados en una máquina de estados finita.

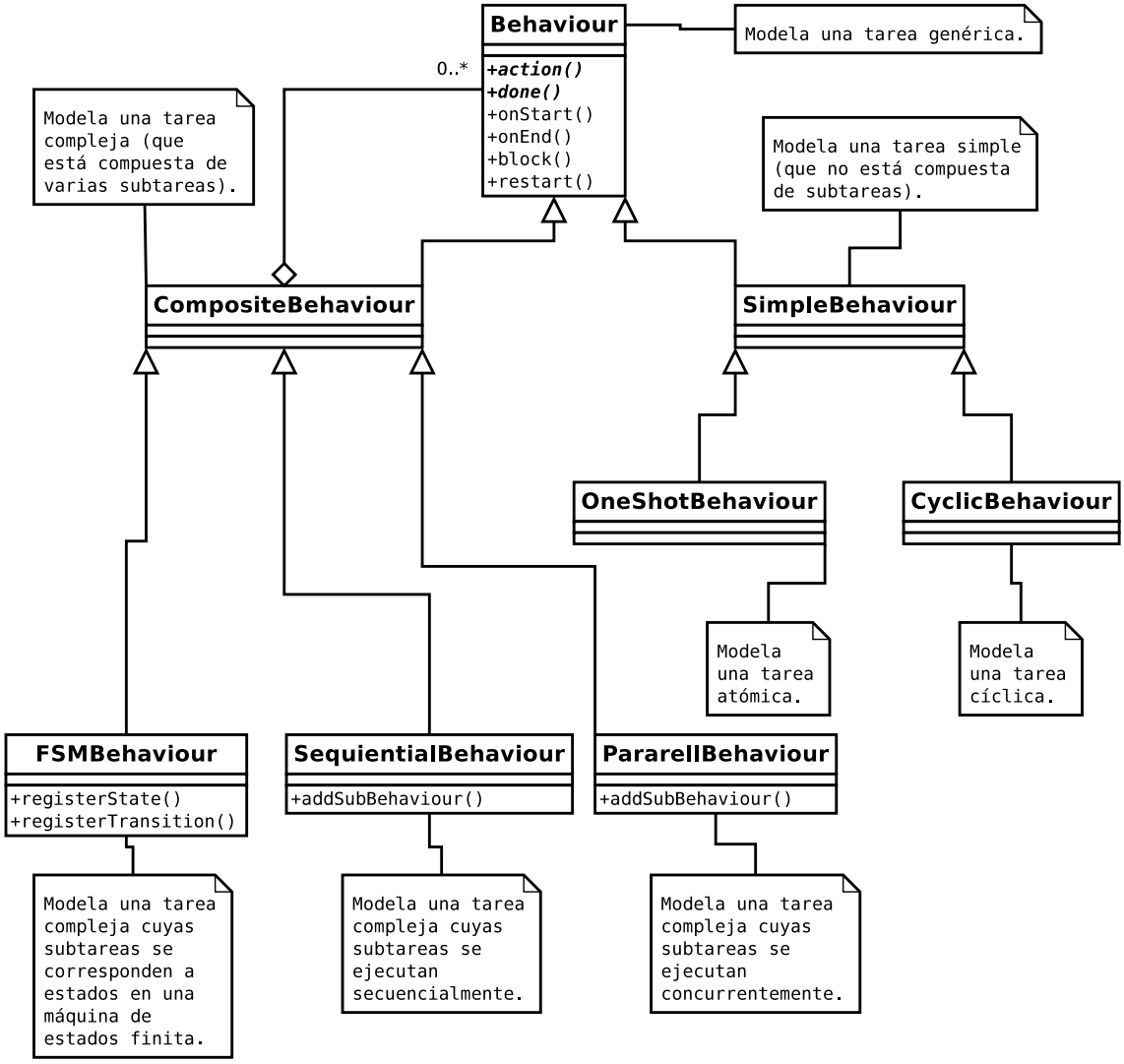

Figura 3.4: Diagrama de clases de comportamientos.

- SequentialBehaviour: sus subtareas se ejecutan secuencialmente.
- PararellBehaviour: sus subtareas de ejecutan concurrentemente.

- **Comportamientos genéricos**: Estos almacenan un estado y ejecutan una operación diferente dependiendo del estado. Terminan cuando una condición definida por el usuario se hace cierta.

3.5 Middleware de comunicaciones

El middleware es una parte esencial en este proyecto, ya que permite realizar las comunicaciones en un nivel de abstracción superior al de los sockets y además proporciona varios servícios útiles para el framework. En esta sección se estudiarán varias alternativas disponibles para las comunicaciones.

3.5.1 *Web services*

Web services (o, simplemente, servicios web) [2] provee un medio estándar de interoperatibilidad enttre diferentes aplicaciones software, ejecutándose en diversas plataformas o *frameworks*.

Un *web service* es un sistema software diseñado para soportar interoperatibilidad de máquina a máquina sobre una red. Tiene una interfaz descrita en un formato procesable por máquinas. Otros sitemas interactuan con el servicio web de una menera preescrita por su descripción usando mensajes SOAP, típicamente enviados a través de HTTP con una serialización XML en conjunto con otros estándares relacionados con la web.

La mayoría de servicios web no tienen por qué adoptar la compleja arquitectura propuesta por el W3C. Pueden reconocerse dos clases principales de *web services*:

- Servicios **REST (Representational State Transfer)**: su principal propósito es manipular representaciones de recursos en XML mediante un conjunto de operaciones sin estado. Estas operaciones son las presentes en el protocolo HTTP. No necesariamente la comunicación tiene que hacerse con XML. Puede usarse otros formatos como JSON, dependiendo de la operación y el recurso al que se esté accediendo.
- Servicios web **arbitrarios**: en ellos el servicio expone un conjunto arbitrario de operaciones sobre él.

3.5.2 Java RMI

Java Remote Method Invocation (Java RMI) [3] permite a un desarrollador crear aplicaciones distribuidas entre aplicaciones basadas en tecnología Java, de forma que los métodos de objetos remotos de Java pueden invocarse desde distintas máquinas virtuales e incluso desde distintos *hosts*. RMI utiliza las capacidades de serialización de objetos para el proceso de *marshalling* y *unmarshalling* (transformación de un objeto en un formato de datos apropiado para su transmisión por red y viceversa).

RMI proporciona un modelo simple y directo para la computación distribuida usando objetos de Java. Estos objetos pueden ser objetos nuevos o también pueden ser *wrappers* alrededor de una API ya existente. Java adopta el principio de *escribe una vez, ejecuta en cualquier sitio*. RMI extiende el modelo de Java para ser ejecutado en todas partes.

Debido a que RMI está centrado en Java, aporta la potencia de Java y su portabilidad a la computación distribuida. Se puede mover el comportamiento, como pueden ser agentes y lógica de negocio, a cualquier parte de la red en la que tenga más sentido.

RMI se conecta a sistemas ya existentes y antiguos utilizando la Java Native Interface (JNI). RMI también puede conectarse a bases de datos relacionales utilizando el paquete estándar

[2] http://www.w3.org/TR/2004/NOTE-ws-arch-20040211
[3] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html

JDBC (Java Database Connectivity). Las combinaciones de RMI/JNI y RMI/JDBC permiten usar RMI para comunicarse con servidores ya existentens en lenguajes diferentes a Java y expandir el uso de Java a aquellos servidores en los que tenga sentido.

Al nivel más básico, RMI es el mecanismo de RPC de Java. RMI tiene muchas ventajas sobre otros sistemas RPC tradicionales porque es parte del enfoque orientado a objetos de Java. Sistemas RPC tradiconales son neutrales respecto al lenguaje y, por tanto, son sistemas que reducen sus posibilidades al común denominador. Por tanto, no pueden aportar toda la funcionalidad que no esté disponible en todas las plataformas objetivo.

RMI está centrado en Java, con conectividad a sistemas existentes utilizando métodos nativos. Esto quiere decir que RMI puede tomar un enfoque natural, directo y completo para proveer a los desarrolladores de una tecnología de computación distribuida que permite añadir la funcionalidad de Java a través de un sistema de una manera incremental.

Las principales ventajas de RMI son:

- **Orientado a objetos**: RMI puede pasar objetos completos como argumentos y valores de retorno, no sólo tipos de datos predefinidos. Esto quiere decir que puedne pasarse tipos complejos, como una tabla *hash*, como un sólo argumento. En sistemas RPC existentes era necesario que el cliente descompusiese cada objeto en tipos primitivos, enviar esos tipos y luego recrearlos del lado del servidor. RMI permite enviar objetos sin necesidad de añadir código adicional en el cliente.
- **Comportamiento móvil**: RMI puede mover el comportamiento (es decir, las implementaciones de las clases) del cliente al servidor y viceversa.
- **Patrones de diseño**: el paso de objetos permite usar toda la potencia de la tecnología orientada a objetos en la computación distribuida. Cuando se puede pasar el comportamiento, se facilita el uso de patrones de diseño en las soluciones.
- **Seguridad**: RMI usa los mecanismos de seguridad de Java para permitir al sistema permanecer en un estado seguro cuando los usuarios descargan implementaciones.
- **Fácil de escribir y de usar**: RMI hace sencilla la escritura de servidores remotos y clientes que accedan a ellos en Java. Una interfaz remota es una interfaz normal de Java y se comporta como un objeto como cualquier otro de Java.
- **Conexión a sistemas ya existentes o antiguos**: RMI puede interactuar con estos sistemas a través de JNI. Usando RMI y JNI puede escribirse un cliente en Java y usar una implementación ya existente del servidor. Además, usando JNI pueden reescribirse partes del servidor en Java cuando se decida y así aprovechar las ventajas de Java en el nuevo código. De la misma forma, RMI puede intereactuar con bases de datos relacionales ya existentes usando JDBC sin modificar código no Java ya existente que usara estas bases de datos.

- **Escribe una vez, ejecuta en cualquier sitio**: RMI sigue este principio de Java. Cualquier sistema basado en RMI es 100 % portable a cualquier máquina virtual en Java.
- **Recolección de basura distribuida**: RMI usa esta característica para recolectar objetos remotos en el servidor cuando ya no son referenciados por ningún cliente en la red. De la misma forma que la recolección de basura en la máquina virtual de Java, esta recolección distribuida permite definir objetos en el servidor cuando son necesitados, sabiendo que serán eliminados cuando ya no sean accesibles por los clientes.
- **Computación en paralelo**: RMI es multihilo, lo que permite utilizar hilos de Java para un mejor procesamiento concurrente de las peticiones de los clientes.
- **Solución de computación distribuida de Java**: RMI es parte del núcleo de la plataforma Java desde la versión 1.1, por lo que existe en cualquier máquina virtual de Java a partir de esa versión. Todos los sistemas RMI utilizan el mismo protocolo, por lo que todos los sistemas Java pueden comunicarse entre sí directamente sin necesidad de ninguna traducción de protocolo.

3.5.3 CORBA

CORBA (Common Object Request Broker Architecture) [4] consiste en una arquitectura e infraestructura abiertas e independientes del vendedor, que puede ser usada para que las aplicaciones trabajen juntas sobre redes.

CORBA es un estándar del OMG (Object Management Group), uno de los mayores consorcios en la industria de la informática. Este estándar permite ejecutar objetos en diferentes plataformas, en diferentes máquinas y escritos en diferentes lenguajes de programación para interoperar. El OMG no crea software por sí mismo, sino que crea especificaciones que son implementadas por diferentes fabricantes. Existen un gran número de implementaciones de CORBA, tanto libres como privativas y comerciales, que pueden interactuar entre sí a menos que se usen extensiones propietarias.

El entorno de ejecución de CORBA consiste en un **Object Request Broker (ORB)** que enruta las llamadas de un objeto a otro y devuelve potenciales valores de retorno. Un ORB maneja los detalles de la invocación, como puede ser encontrar el objeto objetivo y el *marshalling* de los argumentos. Todas las invocaciones pasan por el ORB. Mediante el paso por este entorno de ejecución de todas las invocaciones de objetos, CORBA puede ocultar las diferencias entre los objetos comunicantes (como pueden ser el lenguaje de programación, el sistema operativo y la localización física) sin requerir que sus interfaces miren en la misma memoria. Por tanto, CORBA no es un estándar a nivel binario [5].

Todos los objetos son accedidos usando interfaces, que son especificadas en un dialecto de Interface Definition Languaje (IDL) propio de CORBA. Es este lenguaje, en conjunto con los

[4] http://www.corba.org/
[5] http://www.polberger.se/components/read/corba.html

bindings para cada lenguaje, el que sirve como el mecanismo de estandarización de CORBA. Aparte del ORB, todas las implementaciones de CORBA vienen con un compilador de **IDL**. Uno de los usos primarios del compilador de IDL es generar ***proxies*** específicos del lenguaje para el lado del cliente y el servidor (conocidos como ***stubs*** y ***skeletons***, respectivamente), que son usados para enviar y recibir las peticiones de objetos.

CORBA también define un gran número de *bindings* estandarizados, los cuales hacen posible acceder e implementar objetos de CORBA en muchos lenguajes. Estos *bindings* son implementados por un **adaptador de objetos**, dejando al ORB concentrarse en partes del entorno de ejecución agnósticas del lenguaje. Un objeto es una entidad abstracta a la que los clientes se refieren a través de **referencias a objetos** estandarizadas como **Interoperable Object References (IOR)**. Un IOR contiene toda la información necesaria para contactar el objeto, como la dirección IP y el número de puerto para un objeto remoto. A la implementación de un objeto se la llama **sirviente**. El adaptador de objeto es responsable de enlazar una referenca a objeto con un sirviente concreto, el cual puede servir la petición hecha al objeto.

Como se comprobará a continuación, muchos de los conceptos y procedimientos de CORBA son predecesores de los de ZeroC ICE.

3.5.4 ZeroC ICE

ZeroC ICE (Internet Communications Engine) es un conjunto de herramientas moderno y orientado a objetos [Hen04], el cual permite construir sistemas distribuidos con un mínimo esfuerzo.

Sus principales objetivos de diseño son:

- Proveer de una plataforma adecuada para ser usada en entornos heterogéneos.
- Proveer de un conjunto de características para permitir desarrollar aplicaciones distribuidas realistas en una amplia variedad de dominios.
- Evitar una complejidad innecesaria, haciendo la plataforma fácil de usar y entender.
- Proveer de una implementación eficiente en ancho de banda, uso de memoria y uso de CPU.
- Proveer de una implementación con seguridad incorporada, haciéndola adecuada para ser usada sobre redes públicas inseguras.

ICE es un middleware muy completo que ofrece muchas características. En el presente documento se estudiará lo más importante para este proyecto resumido desde su documentación oficial [6].

[6] http://doc.zeroc.com/display/Ice34/Home

Terminología

ICE es una tecnología con un vocabulario propio para definir lo que hace. A continuación definiremos algunos de sus términos para entender mejor su aplicación a este proyecto.

Clientes y servidores

Los términos *cliente* y *servidor* no son designaciones firmes de determinadas partes de la aplicación. En su lugar, determinan el rol que toma parte de la aplicación durante una petición.

- Los clientes son entidades activas. Hacen peticiones a los servidores.
- Los servidores son partes pasivas. Ofrecen servicios en respuesta a las peticiones de los clientes.

Frecuentemente, los clientes y servidores no son *clientes puros* realizando peticiones o *servidores puros* ofreciendo servicios, sino que ofrecen servicios y a la vez realizan peticiones de servicjios. Por eso, muchos sistemas no son descritos como de cliente-servidor, sino como peer-to-peer (de igual a igual).

Objetos de ICE

Un objeto de ICE es una abstracción. Está caracterizado por los siguientes puntos:

- Un objeto de ICE es una entidad local o en un espacio remoto de direcciones que puede responder a peticiones de clientes.
- Un solo objeto de ICE puede ser instanciado en un solo servidor o, redundantemente, en múltiples servidores. Si un objeto tiene múltiples instancias, sigue siendo un sólo objeto.
- Cada objeto de ICE tiene una o más interfaces. Una interfaz es una colección de operaciones con nombres que son ofrecidas por el objeto. Los clientes realizan peticiones invocando estas operaciones.
- Una operación tiene cero o más parámetros así como un valor de retorno. Los parámetros y valores de retorno tienen un tipo específico. Los parámetros tienen nombre y dirección: los parámetros de entrada son iniciados por el cliente y pasados al servidor; mientras que los parámetros de salida son iniciados pro el servidor y pasados al cliente. El valor de retorno es un tipo especial de parámetro de salida.
- Un objeto de ICE tiene una interfaz principal. Además de eso, puede tener cero o más interfaces alternativas, conocidas como facetas. Los clientes pueden elegir entre las facetas de un objeto la interfaz con la que quieren trabajar.
- Cada objeto tiene una identidad única. La identidad de un objeto es un valor de identificación que lo distingue de todos lso demás objetos. El modelo de objetos de ICE asume que cada identidad de un objeto es globalmente única, es decir,

no hay dos objetos en un dominio de comunicación de ICE que tengan la misma identidad. En la práctica, no es necesario utilizar identidades globalmente únicas, sólo identidades que no colisionen con otra identidad en un determinado dominio de interés.

Proxies

Para que un cliente sea capaz de contactar un objeto de ICE, el cliente debe mantener un *proxy* para ese objeto. Un proxy es un artefacto que es local al espacio de direcciones del cliente. Representa el objeto ICE (probablemente remoto) para el cliente. Un proxy actua como el embajador local para un objeto de ICE: cuando el cliente invoca una operación en el proxy, el entorno de ejecución de ICE realiza lo siguiente:

1. Localiza el objeto de ICE.
2. Activa el servidor del objeto si no se está ejecutando.
3. Acitva el objeto en el servidor.
4. Transmite los parámetros de entrada al objeto de ICE.
5. Espera a que la operación se complete.
6. Devuelve cualquier parámetro de salida y el valor de retorno al cliente (o en caso de error, lanza una excepción).

Un proxy encapsula toda la información necesaria para que tenga lugar esta secuencia de pasos. En particular, un proxy contiene:

- Información de direccionamiento que permite al entorno de ejecución en el lado del cliente contactar el servidor correcto.
- Una identidad de objeeto que identifica qué objeto en particular esta siendo objetivo de una petición en el servidor.
- Una identificador de faceta opcional que determina a qué faceta del objeto en particular se refiere el proxy.

Sirviente (*servant*)

Como se ha mencionado, un objeto de ICE es una entidad conceptual que tiene tipo, identidad e información de direccionamiento. Sin embargo, las peticiones del cliente deben acabar con un procesamiento concreto en el lado del servidor que pueda proveer del comportamiento para una operación. Es decir, una petición del cliente debe acabar finalmente ejecutando código dentro del servidor, un código que estará escrito en un lenguaje específico de programación y ejecutado con un procesador específico.

El artefacto en el lado del servidor que provee de comportamiento a una invocación de una operación es conocido como sirviente. Un sirviente encarna uno o más objetos de ICE. Los métodos en la clase corresponden a las operaciones del interfaz del objeto de ICE y proveen el comportamiento para las operaciones.

Un solo sirviente puede encarnar un solo objeto de ICE a la vez o varios objetos de ICE simultáneamente. En el primer caso, la identidad del objeto de ICE encarnada por el sirviente está implícita en el sirviente. En el segundo, el sirviente es proveido de la identidad del objeto de ICE en cada petición, para que pueda decidir qué objeto encarnar a lo largo de la duración de la petición.

A la inversa, un solo objeto de ICE puede tener múltiples sirvientes. Esto permite construir sistemas redundantes.

Slice

Cada objeto de ICE tiene una interfaz con un número de operaciones. Las interfaces, operaciones y los tipos de datos que son intercambiados entre clientes y servidores son definidos usando un lenguaje denominado Slice. Slice permite definir el contrato entre cliente y servidor en una manera que no es dependiente de lenguajes de programación específicos. Las definiciones en Slice son compiladas por un compilador a una API (Application Programming Interface) para un lenguaje de programación específico.

Mapeo de lenguajes

Las reglas que gobiernan cómo cada construcción en Slice es traducida en un lenguaje de programación específico son conocidas como mapeo de lenguajes. Por ejemplo, para el mapeo a C++, una secuencia en Slice aparece como un vector de la STL (Standard Template Library), mientras que para el mapeo a Java, una secuencia en Slice aparece como un array de Java. Para determinar cómo será la API para una construcción en Slice, el desarrollador sólo necesita la propia definición en Slice y el conocimiento sobre las reglas de mapeo a un determinado languaje. Las reglas son suficientemente simples y regulares para hacer innecesaria la lectura del código generado para averiguar cómo funciona la API generada.

En la versión 3.4.2 (la utilizada para el presente proyecto), ICE provee mapeo a los siguientes lenguajes: C++, Java, C#, Python, Objective-C y, para el lado del cliente, PHP y Ruby.

Estructura de cliente y servidor

En la figura 3.5 vemos un esquema que representa la estructura lógica interna de una aplicación desarrollada con ICE. Tanto el cliente como el servidor consisten en una mezcla de código de aplicación, código de bibliotecas y código generado de las definiciones Slice:

- El núcleo de ICE contiene el soporte del entorno de ejecución (tanto del lado del cliente como del servidor) para comunicaciones remotas. Una gran parte de este código se refiere a los detalles de transmisión por la red, hilos, ordenamiento de bytes y muchos otras cuestiones relacionadas con la red que quieren apartarse del código de la aplicación. El núcleo de ICE provee un número de bibliotecas que los clientes y servidores

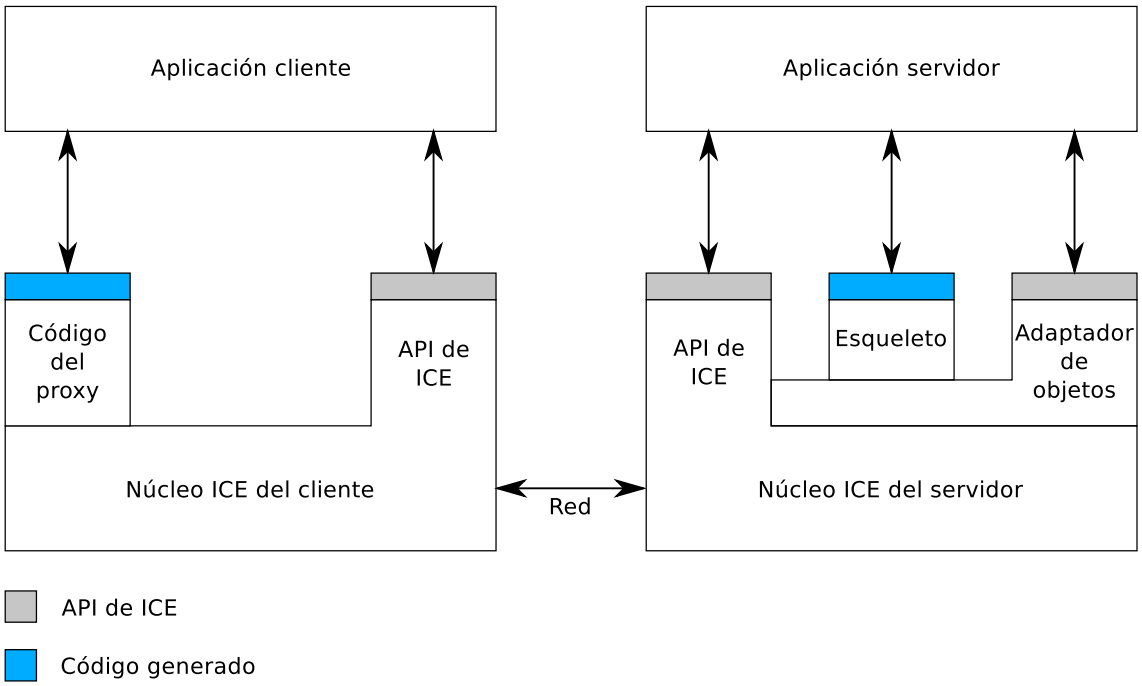

Figura 3.5: Estructura de clientes y servidores ICE.

usan.

- La parte genérica del núcleo de ICE (es decir, la parte que es independiente de los tipos específicos definidos con Slice) puede accederse a través de la API de ICE. Un desarrollador puede usar esta API para hacerse cargo de tareas administrativas, como iniciar y finalizar el entorno de ejecución. La API de ICE es idéntica para clientes y servidores (aunque los servidores usan una parte mayor de esta API que los clientes.
- El código de los proxis es generado desde las definiciones en Slice y, por tanto, específico para los tipos de objetos y datos que el desarrollador ha definido con Slice. El código de los proxis tiene dos funciones principales:
  - Provee una interfaz para que el cliente llame *hacia abajo*. Llamar a una función en la API del proxy generado termina enviando un mensaje RPC al servidor que invoca la función correspondiente en el objeto objetivo.
  - Provee código para hacer ***marshalling*** y ***unmarshalling***. *Marshalling* es el proceso de serializar estructuras de datos complejas, como secuencias o diccionarios, para transmitirlas a través de un canal. El código para el *marshalling* convierte datos en uan forma que es estandarizada pra transmitirla y es independiente de aspectos como el endian-ness (big endian o little endian, si el bit más significativo va primero o el bit menos significativo va primero) de la maquina local. *Unmarshalling* es el inverso de marshalling, es decir, deserializar datos que llegan a traves de la red y reconstruirlos como una representación local de los tipos de datos apropiados para el lenguaje de programación en uso.

- El código esqueleto es también generado desde las definiciones en Slice y, por tanto, es específico para los tipos de objetos y datos que el desarrollador ha definido con Slice. El código esqueleto es el equivalente en el lado del servidor del código del proxy en el lado del cliente: provee una interfaz para llamar *hacia arriba* que permite al entorno de ejecución de ICE transferir el el hilo de control al código de aplicación que ha escrito el desarrollador. El esqueleto también contiene código para el marshalling y unmarshalling, para que el servidor pueda recivir parámetros enviados por el cliente y devolver parámetros y excepciones al cliente.
- El adaptador de objetos es una parte de la API de ICE que es específica al lado del servidor: sólo los servidores utilizan adaptadores de objetos. Un adaptador de objetos tiene las siguientes funciones:
  - El adaptador de objetos mapea las peticiones entrantes desde los clientes a métodos específicos en los objetos del lenguaje de programación utilizado. En otras palabras, el adaptador de objetos rastrea qué sirvientes en qué identidades de objetos están en memoria.
  - El adaptador de objetos está asociado con uno o más endpoints de transporte. Si más de un endopoint de transporte está asociado con un adaptador, los sirvientes que encarnan objetos en el adaptador pueden ser alcanzados via múltiples transportes. Por ejemplo, puede asociarse un endpoint TCP y otro UDP en un mismo adaptador para proveer alternativas de calidad de servicio y rendimiento.
  - El adaptador de objetos es responsable de la creación de proxis que pueden ser pasados a los clientes. El adaptador de objetos conoce el tipo, identidad y detalles de transporte de cada uno de sus objetos y empotra los detalles correctos cuando el código de la aplicación del lado del servidor pide la creación de un proxy.

Nótese que, en lo que concierne a la vista de procesos, sólo hay dos procesos involucrados: el cliente y el servidor. Todo el soporte del entorno de ejecución para las comunicaciones distribuidas es proveído por las bibliotecas de ICE y el código que es generado desde las definiciones en Slice.

Vista rápida del protocolo de ICE

ICE provee un protocolo RPC que puede usar TCP o UDP como transporte en la capa inferior. Además, ICE permite usar SSL (Secure Socket Layer) como transporte, para que todas las comunicaciones entre cliente y servidor vayan cifradas.

El protocolo de ICE define lo siguiente:

- Una variedad de tipos de mensaje, como los de petición y respuesta.
- Un protocolo de máquina de estados que determina en qué secuencia son intercambiados los diferentes tipos de mensaje por el cliente y el servidor, junto con el establecimiento de conexión asociado y tear-down (?) semántica para TCP/IP.
- Reglas de codificación que determinan cómo es representado cada tipo de dato en el canal.
- Una cabecera para cada tipo de mensaje que contiene detalles como el tipo de mensaje, el tamaño y la versión del protocolo y codificación en uso.

ICE también soporta compresión en el canal: ajustando un parámetro de configuración, el desarrollador puede organizar todo el tráfico para que vaya comprimido, ahorrando ancho de banda. Esto es útil si la aplicación intercambia grandes cantidades de datos entre cliente y servidor.

El protocolo de ICE es adecuado para construir mecanismos de redirección de eventos altamente eficientes, porque permite redirigir un mensaje sin conocimiento de los detalles de la información que contiene. Esto significa que los conmutadores de mensajes no necesitan hacer el unmarshalling y otra vez el marshalling (el proceso para convertir una llamada a una operación en un mensaje a enviar y viceversa) de los mensajes, ya que pueden redirigir el mensaje simplemente tratándolo como un buffer opaco de bytes.

El protocolo de ICE tambien soporta operaciones bidireccionales: si el servidor quiere enviar un mensaje a un objeto de retrollamada proveido por el cliente, la retrollamada puede ser hecha sobre la conexión que fue originalmente creada por el cliente. Esta característica es especialmente importante cuando el cliente está detrás de un cortafuegos que permite conexiones salientes, pero no entrantes.

Servicios

El núcleo de ICE proporciona una sofisticada plataforma cliente-servidor para el desarrollo de aplicaciones distribuidas. Sin embargo, aplicaciones realistas normalmente requieren algo más que poder ejecutar objetos remotos: iniciar servidores bajo demanda, distribuir eventos asíncronos, configurar la aplicación, distribuir parches, etc.

ICE incluye un conjunto de servicios que permiten lo anterior, además de otras características. Estos servicios son implementados como servidores de ICE, de los cuales la aplicación que se está desarrollando es el cliente. Ninguno de estos servicios usan características internas de ICE ocultas a los desarrolladores, por lo que en teoría se pueden desarrollar servicios equivalentes manualmente. Sin embargo, tener estos servicios disponibles como parte de la plataforma permite centrarse en el desarrollo de la aplicación en lugar de tener que desarro-

llar la infraestructura primero. Además, estos servicios no son precisamente triviales, por lo que es mejor usar lo disponible que estar *reinventando la rueda*.

ZeroC ICE proporciona los siguientes servicios:

Freeze y Freezescript

ICE tiene incorporado un servicio de persistenca llamado Freeze. Este servicio facilita la tarea de almacenar el estado de un objeto en una base de datos: se define dicho estado en las interfaces Slice y el compilador de Freeze genera código que almacena y recupera el estado de la base de datos. Freeze usa por defecto Berkeley DB.

ICE también ofrece un conjunto de herramientas llamadas FreezeScript que facilitan el mantener bases de datos y migrar el contenido.

IceGrid

IceGrid es una implementación del servicio de localización que resuelve la información simbólica en un proxy indirecto a un par protocolo-dirección. El servicio de localización es sólo el principio de las capacidades de IceGrid:

- Permite registrar servidores para iniciarse bajo demanda.
- Provee de herramientas para configurar aplicaciones complejas con muchos servidores.
- Soporta replicación y balanceo de carga.
- Automatiza la distribución y parcheo de ejecutables y archivos dependientes.
- Provee una interfaz de consulta sencilla para obtener proxis a objetos.

IceBox

IceBox es una sencilla aplicación de servidor que puede organizar el inicio y parada de un número de componentes de la aplicación. Estos componentes pueden ser desplegados como bibliotecas dinámicas en lugar de como procesos. Esto reduce la carga total del sistema, permitiendo, por ejemplo, ejecutar varios componentes en una única máquina virtual Java en lugar de tener varios procesos cada uno con su propia máquina virtual.

IceStorm

IceStorm es un servicio de publicación y subscripción que desacopla clientes y servidores. Fundamentalmente, IceStorm actua como un switch de distribución de eventos. Los publicadores envian eventos al servicio, el cual pasa estos eventos a los subscriptores. De esta forma, un único evento publicado por un publicador puede ser enviado a múltiples subscriptores. Los eventos son categorizados por tema (topic), y los subscriptores especifican el tema en el cual están interesados. Sólo los eventos que igualan con el tema de un subscriptor son enviados a ese subscriptor. El servicio permite seleccionar una variedad de criterios de calidad de servicio para permitir a las aplicaciones elegir un buen equilibrio entre confiabilidad y rendimiento.

IceStorm es particularmente útil si se tiene la necesidad de distribuir información a un gran número de componentes de aplicación. IceStorm desacopla los publicadores de información de los subscriptores y se ocupa de la redistribución de los eventos publicados. Además, IceStorm puede ser ejecutado como un servicio federado, es decir, múltiples instancias del servicio pueden ser ejecutadas en diferentes máquinas para distribuir la carga de procesamiento sobre varias CPUs.

IcePatch2

IcePatch2 es un servicio de parcheo de software. Permite distribuir actualizaciones de software fácilmente a los clientes. Estos simplemente se conectan al servidor IcePatch2 y piden actualizaciones para una aplicación particular. El servicio automáticamente comprueba la versión del software de los clientes y descarga las actualizaciones en un formato comprimido para ahorrar ancho de banda. Los parcheos pueden hacerse más seguros utilizando el servicio Glacier2, para que sólo los clientes autorizados puedan descargar actualizaciones de software.

Glacier2

Glacier2 es el servicio de firewall transversal de ICE. Permite a los clientes y servidores comunicarse de manera segura a través de un firewall sin comprometer la seguridad. El tráfico cliente-servidor es cifrado usando certificados de clave pública y es bidireccional. Glacier2 ofrece soporte para autenticación mutua así como gestión de inicio de sesión segura.

Capítulo 4

Método de trabajo

Este capítulo habla de la metodología que se ha usado durante el desarrollo de este proyecto para mostrar al lector cómo ha sido su desarrollo a nivel organizativo. También se detallan las herramientas que se han usado durante todo el proceso.

4.1 Metodología

Dado que sólo hay un desarrollador en este proyecto, la metodología no tiene tanto interés a nivel de organización como cuando hay un grupo de desarrolladores. La metodología utilizada ha sido programación extrema [1], que es un tipo de metodología ágil.

4.1.1 Manifiesto ágil

Para ayudar a entender mejor la filosofía detrás de las metodologías y prácticas ágiles, a continuación se cita el Manifiesto Ágil [2]:

«Estamos descubriendo formas mejores de desarrollar software tanto por nuestra propia experiencia como ayudando a terceros. A través de este trabajo hemos aprendido a valorar:

- **Individuos e interacciones** sobre **procesos y herramientas**.
- **Software funcionando** sobre **documentación extensiva**.
- **Colaboración con el cliente** sobre **negociación contractual**.
- **Respuesta ante el cambio** sobre **seguir un plan**.

Esto es, aunque valoramos los elementos de la derecha, valoramos más los de la izquierda.»

4.1.2 Programación extrema

Esta metodología consiste en ir teniendo reuniones periódicas con el cliente, en cortos espacios de tiempo (alrededor de 2 semanas). En cada una de estas reuniones, el cliente comprueba el trabajo hecho hasta la fecha y, en función de él, se decide lo que se va a

[1] http://www.computerworld.com/s/article/66192/Extreme_Programming?taxonomyId=063
[2] http://agilemanifesto.org/iso/es/

hacer hasta la proxima reunión. El proyecto se considera acabado cuando el cliente está satisfecho.

En este caso, la labor del cliente era realizada por el director de proyecto y las reuniones no han sido en intervalos fijos, sino que han sido en función del trabajo encargado en cada periodo y el nivel de ocupación del director y el autor (por ejemplo, los exámenes y trabajos de la carrera tenían prioridad y, por tanto, retrasaban las reuniones). Nótese también que los roles han estado muy difuminados, ya que el director de proyecto además de cliente es también jefe de proyecto y el desarrollador también ha tomado decisiones sobre los requisitos.

4.2 Herramientas

4.2.1 Hardware

Este proyecto no tiene requisitos de hardware específicos más allá de los que puede tener cualquier ordenador moderno con conexión a una red de área local y/o a internet. Durante el desarrollo, se ha utilizado un equipo con las siguientes características:

- Procesador Intel Core i3 M 330, 2.13 GHz
- 3.8 GiB de memoria RAM
- Disco duro Toshiba, 465.76 GiB, 7200 rpm

4.2.2 Software

Sistema operativo

El sistema operativo utilizado para el desarrollo ha sido Debian GNU/Linux [3]. Debian tiene tres ramas:

- Estable: la rama más probada y aconsejada para entornos de producción.
- Pruebas: la rama que será la futura versión estable cuando esté suficientemente probada.
- Inestable: la rama a la que se le incorporan todos los nuevos cambios. A su vez, esta contiene otra rama experimental.

La versión del sistema operativo que pertenece a cada rama va cambiando con el tiempo, excepto la versión inestable cuya versión siempre se denomina *Sid*. Para este proyecto, se ha trabajado siempre con la rama de pruebas, por considerar que tiene el mejor equilibrio entre novedad del software que incluye y estabilidad de éste. Todas las versiones del software que se describe a continuación son las que han pertenecido a esta rama.

[3] https://www.debian.org

Documentación

Para los manuales y también para la escritura del presente documento se ha utilizado LaTeX. LaTeXes un lenguaje de marcas y sistema de preparación de documentos, que permite expresar su contenido y forma en forma de texto plano siguiendo una sintaxis que, posteriormente, se compila para generar el documento en un determinado formato. En este caso, ese formato ha sido PDF.

Para la documentación del código se ha utilizado Doxygen [4]. Se trata de una herramienta que, escribiendo los comentarios en el código fuente siguiendo una determinada sintaxis, permite extraer esa información de dichos comentarios y generar documentación. Esta documentación puede ser generada en diversos formatos, como pueden ser LaTeXy PDF.

Lo siguiente es un ejemplo de comentarios en formato doxygen:

```
/** Una clase de prueba */
class Test {

public:
  void member(); ///< Una función miembro de la clase Test.

private:
  int value; ///< Un atributo privado de la clase Test.
};
```

Gráficos

Se han utilizado, principalmente, dos herramientas para elaborar gráficos como los de este documento:

- Dia [5]: es una herramienta para generar diagramas estructurados que, además, permite exportarlos a muchos formatos.
- Inkscape [6]: es un editor de gráficos vectoriales.

Editor de texto

Vim [7] es un editor de texto que proporciona todas las herramientas del antiguo editor vi de entornos Unix junto con muchas más opciones avanzadas. Es áltamente configurable y está pensado para aumentar la productividad mediante combinaciones de teclas y comandos que permiten realizar multitud de tareas sin levantar las manos del teclado. Está considerado,

[4] http://www.stack.nl/~dimitri/doxygen/
[5] http://dia-installer.de/
[6] http://www.inkscape.org
[7] http://www.vim.org

principalmente, un editor para programadores, pero puede ser usado para cualquier tarea que consista en escribir texto. Tanto la programación como la escritura de documentación se ha realizado con esta herramienta.

Compiladores e intérpretes

La principal herramienta de compilación ha sido GNU Compiler Collection (GCC) [8]. Esta colección de compiladores contiene frontends para multitud de lenguajes, además de ser uno de los compiladores más famosos de C y C++.

Junto con GCC, también se ha usado la herramienta llamada GNU Debugger (GDB) [9], que consiste en un depurador en línea de órdenes que se complementa muy bien con GCC.

Otra herramienta más ha sido el intérprete oficial de Python [10], también llamado a veces CPython. Concretamente, se ha usado la rama 2.7 que es la correspondiente a la versión del lenguaje usada. También se ha usado IPython [11], que es una consola interactiva avanzada para el lenguaje Python.

También se ha utilizado la consola e intérprete de comandos GNU Bash [12]. Se trata de un intérprete de comandos compatible con sh que incorpora muchas otras características útiles.

Por último, una herramienta muy importante de este ámbito ha sido GNU Make [13]. Make no es un compilador, sino una herramienta para automatizar el proceso de compilación y las tareas auxiliares de dicho proceso. Esta herramienta no sólo facilita el trabajo durante el desarrollo, sino que también puede facilitar el trabajo a un futuro usuario en el momento en que tuviera que compilar e instalar el software recibido.

Control de versiones

El control de versiones consiste en utilizar herramientas software que faciliten la gestión de los cambios durante el desarrollo de proyectos software, aunque también sirve para gestionar otros artefactos como documentación y páginas web. Esto se hace manteniendo un repositorio que almacena los cambios realizados y permite también descargarlos, de forma que distintos desarrolladores pueden trabajar en el proyecto desde distintos equipos y mantenerse sincronizados, además de que añade seguridad al no estar el proyecto en un único sitio.

[8] http://gcc.gnu.org/
[9] http://www.gnu.org/software/gdb/
[10] https://www.python.org/
[11] http://www.ipython.org/
[12] http://www.gnu.org/software/bash/
[13] http://www.gnu.org/software/make/

Para el control de versiones se ha utilizado Mercurial [14]. Mercurial es una herramienta de control de versiones distribuida, multiplataforma y software libre. Dado que es distribuida, la información sobre las versiones no se encuentra en un único servidor, sino que tanto el servidor principal como los distintos equipos que participen en el desarrollo tienen una copia sincronizada de todo el repositorio, de forma que si alguno no está disponible esto no afecte al trabajo de los demás.

Gestión del proyecto

Para gestionar el proyecto se ha utilizado la herramienta web Redmine [15]. Es una herramienta software libre que proporciona utilidades para la gestión del proyecto como son seguimiento de *bugs*, asignación de tareas, vista del árbol de versiones del repositorio y calendarios y diagramas.

Comunicaciones

Para las comunicaciones a través de la red, se ha utilizado el middleware ZeroC ICE, del que ya se ha hablado más en detalle en 3.5.4.

Edición de vídeo

FFmpeg [16] es un framework multimedia que permite realizar multitud de operaciones con diversos formatos multimedia. En este caso, ha sido usada su versión en línea de órdenes para la descomposición en frames y creación del vídeo final utilizados en uno de los ejemplos realizados con este proyecto.

4.2.3 Lenguajes

C++

C++ es el lenguaje principal utilizado para desarrollar este proyecto. Se trata de un lenguaje de programación de propósito general pero con énfasis en la programación de sistemas. Tiene las siguientes características [17]:

- Es un mejor C.
- Soporta abstracción de datos.
- Soporta programación orientada a objetos.
- Soporta programación genérica.

[14] http://mercurial.selenic.com/
[15] http://www.redmine.org/
[16] https://www.ffmpeg.org/
[17] http://www.stroustrup.com/C++.html

C++ es un lenguaje multiparadigma que puede, incluso, subdividirse en tres lenguajes con estilos diferentes:

- Las características de C son un lenguaje estructurado.
- Con clases y sus características asociadas es un lenguaje orientado a objetos.
- Con la programación genérica se puede hablar de metaprogramación en tiempo de compilación.

Para el proyecto, se ha utilizado la versión estándar de este lenguaje, que data de 1998. Existe ya un estándar más moderno desde el 2011; que, aunque es compatible con el anterior, incorpora otras características como funciones lambda, *smart pointers* (clases que reservan un recurso al crearse y lo liberan al destruirse) y soporte de concurrencia en la biblioteca estándar. Además, durante 2014 se está desarrollando otra revisión menor. Nótese que muchas de estas características, como los smart pointers, son proporcionadas también por la biblioteca de utilidades de ZeroC ICE.

No se ha escogido este lenguaje sólo porque el proyecto en el que está basado ya tuviese código en este lenguaje, sino también porque C++ permite una gran flexibilidad; ya que combina las ventajas de un lenguaje de bajo/medio nivel con las de un lenguaje orientado a objetos moderno.

Python

Python ha sido utilizado como lenguaje de apoyo en el desarrollo de este proyecto. Se trata de un lenguaje de propósito general y de alto nivel. Su filosofía de diseño enfatiza la legibilidad del código y su sintaxis permite al programador expresar los conceptos en menos líneas que en otros lenguajes. Es un lenguaje multiparadigma, permitiendo los estilos estructurado, orientado a objetos e incluso funcional.

En este proyecto, se ha trabajado con la rama 2.7 de este lenguaje. La rama 3 es más moderna, pero también introduce cambios más drásticos en el lenguaje, por lo que se decidió seguir con la 2.7; que sigue siendo mantenida porque es usada aún en muchos otros proyectos.

Este lenguaje ha servido para automatizar distintas tareas, para facilitar tareas auxiliares al framework y también como lenguaje experimental para el desarrollo de agentes.

Bash

Bash no sólo es un intérprete de comandos, como se ha comentado anteriormente. También tiene su propio lenguaje de programación de scripts. Se ha usado para automatizar tareas, como las de arranque/parada de la plataforma.

Capítulo 5

Arquitectura

Este capítulo discute en detalle la arquitectura que da soporte al framework para sistemas multiagente planteado en este proyecto. Se discutirán temas relativos al diseño, desarrollo, configuración y despliegue del presente *framework* y de los sistemas multiagente desarrollados con él.

5.1 Descripción general

En la figura 5.1 puede verse un esquema representativo de la arquitectura del presente framework. Se trata de una arquitectura modular que facilita enormemente el diseño y desarrollo de sistemas multiagente.

También puede verse un diagrama de clases de todo el *framework* en la figura 5.2. En un primer vistazo, pueden reconocerse los siguientes elementos:

- Las clases que terminan en el sufijo Prx, que representan *proxies*, es decir, clases que dan acceso a objetos remotos. Tienen usos como representar distintas réplicas de los servicios básicos y representar otros agentes para el envío de mensajes.
- Los servicios básicos AMS y DF. Son clases abstractas que son extendidas dos veces: una para representar la replicación de los servicios y otra más para la implementación final del servicio. La clase MTS, el otro de los servicios básicos, no tiene replicación porque no contiene estado.
- Behaviour y todas sus subclases que representan los distintos tipos de comportamientos usados por los agentes.
- Los agentes, que están relacionados tanto con los servicios como con los comportamientos y están formados por una clase abstracta que representa su interfaz remota y su implementación interna.

Se irán detallando más las clases y sus funciones en las siguientes secciones de este capítulo.

La descripción de la arquitectura se hará en dos partes: una describiendo las capas del diseño y su función y otra comentando la capacidad de flexibilidad que aporta este diseño.

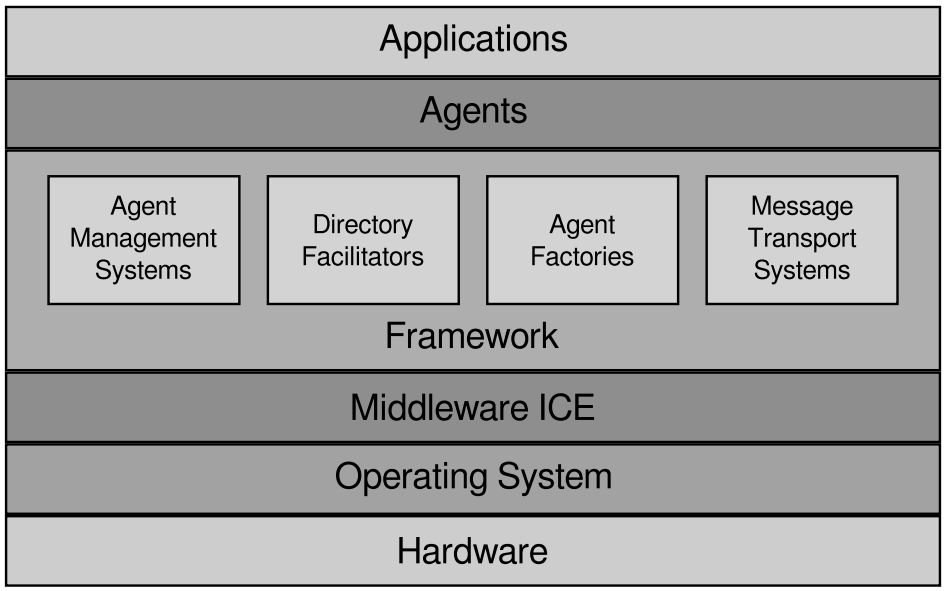

Figura 5.1: Arquitectura del framework.

5.1.1 Capas

Hardware

Son las piezas físicas con las que se trabaja en última instancia, como pueden ser discos duros, tarjeta de red y memoria RAM. Dado que este proyecto trata sobre sistemas distribuidos, la red es una parte especialmente importante.

Su función en este proyecto es la de hacer efectivas las instrucciones de los procesos que lo componen: operaciones matemáticas, transmisión de datos por la red, almacenamiento de datos, etc.

Sistema operativo

Podemos definir un sistema operativo como el conjunto de herramientas software que permiten al usuario interactuar con el ordenador. Un sistema operativo mínimo consta de los siguientes elementos:

- **Núcleo**: conjunto de funciones y herramientas que dan acceso a las funciones de bajo nivel para trabajar con el hardware. Proporciona funcionalidades como el planificador de tareas y la gestión de memoria.
- **Drivers**: son bibliotecas que permiten al sistema operativo comunicarse con hardware específico, como pueden ser tarjetas gráficas, tarjetas de red o impresoras.
- **Biblioteca de llamadas al sistema**: son funciones que permiten al software interactuar con el ordenador a través del sistema operativo.
- **Intérprete de comandos o *shell***: es un programa que permite a un usuario humano dar órdenes procesables por el ordenador. Puede ser por línea de comandos o incluso

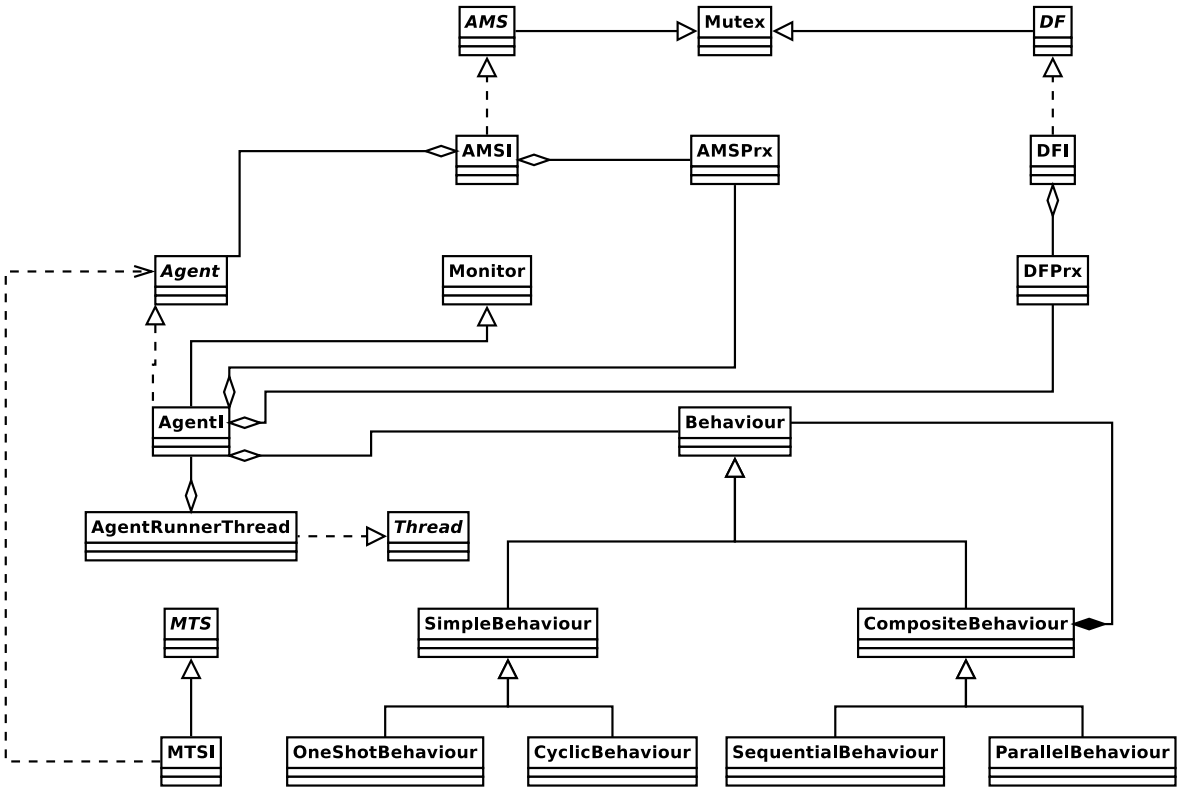

Figura 5.2: Diagrama de clases del *framework*.

también puede ser gráfico.

- **Editor de texto**: es un programa que permite crear, abrir, visualizar y modificar archivos de texto, que es una de las funciones más básicas que pueden realizarse con un ordenador.
- **Compilador**: una herramienta que permite transformar una secuencia de órdenes en un lenguaje de programación a un programa ejecutable. Normalmente es para un lenguaje de sistemas como pueden ser C o C++.

El listado anterior es un conjunto muy básico. Los sistemas operativos modernos vienen acompañados de múltiples herramientas adicionales, como pueden ser procesadores de texto, entornos de desarrollo (*IDE*) y navegadores web. Además de todo lo que acompañe al sistema operativo, el usuario también puede instalar software compatible con su hardware y sistema operativo.

Su función en este proyecto es la de proporcionar acceso al hardware y la de aportar funcionalidades de sistema, como pueden ser las comunicaciones entre procesos y el acceso a la red.

Middleware

El *middleware* es una capa de software intermedia entre el framework y el sistema operativo que debe proporcionar acceso a las funciones de las capas inferiores utilizando artefactos de mayor alto nivel que los que proporciona el sistema operativo sólo.

En este proyecto se utiliza ZeroC ICE. Se ha elegido este *middleware* por los siguientes motivos:

- Es un proyecto maduro y bien probado.
- Es software libre.
- Da soporte a múltiples lenguajes.
- Proporciona servicios muy apropiados para este proyecto, como pueden ser mecanismos de balanceo de carga o una implementación del patrón *observer*.
- Es utilizado también por mútliples empresas a nivel comercial.

Su función en este proyecto es la de proporcionar una capa de abstracción de más alto nivel para las comunicaciones por red. Tambien proporciona una capa de abstracción más alta para otros recursos como los hilos.

Framework

Es el proyecto propuesto, el cual consiste en un *framework* para crear sistemas multiagente distribuidos que siguen el estándar FIPA. Proporciona un conjunto de servicios, funciones y herramientas como pueden ser servicios de localización, tipos de comportamientos (tareas) para los agentes y funciones de comunicación entre agentes. Además, el *framework* permite que tanto los servicios básicos como los agentes se encuentren distribuidos en diferentes *hosts*. Es descrito a lo largo de este documento.

Agentes

Son las unidades básicas de inteligencia artificial que este framework es capaz de manejar y de las que depende la solución que se esté implementando con él. El tipo de agentes propuesto sigue un modelo orientado a comportamientos, que consisten en tareas que el agente ejecuta de una determinada manera dependiendo del tipo que sean, como pueden ser tareas paralelas o cíclicas. Puede consultarse una definición más detallada del concepto de **agente inteligente** en el capítulo 1.

Su función en este proyecto es la de hacer efectiva la solución que implemente un desarrollador que trabaje con el framework.

Aplicaciones

Cualquier otra pieza de software que trabaje de alguna forma con los agentes. Es una parte opcional, ya que los agentes pueden ser autónomos y trabajar sólo con otros agentes y la plataforma, pero pueden haber también otras aplicaciones que se comunican con ellos y añadan otras funciones como monitorización o acceso a recursos diversos. Normalmente hará falta, al menos, un lanzador para invocar a la fábrica de agentes para que cree el primer agente de un despliegue.

Su función en este proyecto es la de proporcionar apoyo a los agentes y otras tareas como la inicialización de un despliegue.

5.1.2 Flexibilidad

Existen casos en los que un desarrollador que use este framework decida no seguir estrictamente esta separación por capas y acceder a ellas de otra forma, como puede ocurrir en los siguientes casos:

- Los agentes pueden comunicarse directamente con el *middleware*, característica que podría ser aprovechada si un desarrollador que trabaje con este framework dispone de conocimientos específicos de ZeroC ICE.
- Otras aplicaciones pueden acceder directamente a funciones del framework sin pasar necesariamente por agentes. Por ejemplo, lo normal es que el primer agente que se crea en un despliegue sea porque un programa invoca la función crear agente de la fábrica de agentes.

La decisión de si seguir un acceso estricto a los recursos por capas o no se deja, por tanto, en manos de los desarrolladores que usen este framework.

5.2 Servicios de soporte FIPA

El estándar FIPA (comentado más en detalle previamente en 3.3) define un conjunto de servicios de los que debe disponer un sistema multiagente y que el framework que describe este documento implementa. A continuación se hará una descripción más detallada de la implementación que se ha realizado de ellos en este proyecto.

5.2.1 Agent Management System (AMS)

AMS es uno de los dos servicios de localización de agentes. Su labor podría definirse como de *páginas blancas*, ya que registra todos los agentes del sistema y permite buscarlos por el nombre del agente.

Para realizar esta labor, durante la creación de todos los agentes estos obtienen un proxy al AMS e inmediatamente se registran en él, de forma que desde su creación ya están presentes en el servicio para ser localizados por las búsquedas. De la misma forma, al destruir un

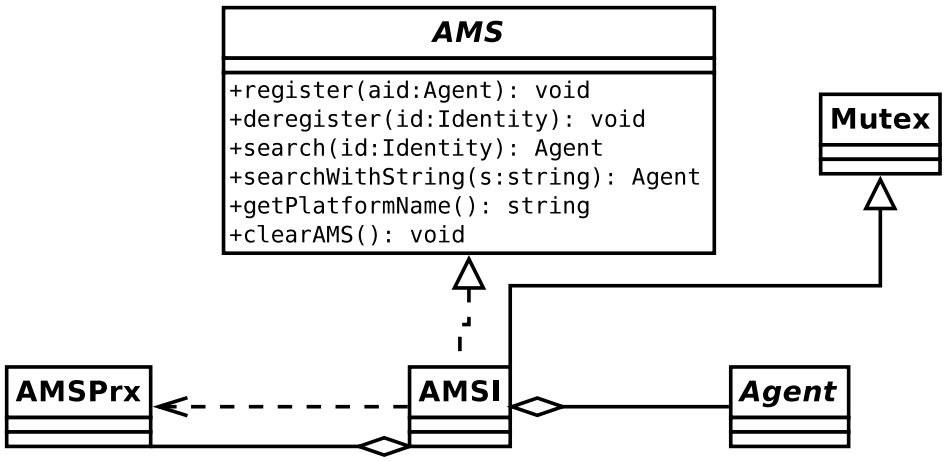

Figura 5.3: Diagrama de clases del servicio AMS.

agente éste solicita ser borrado del servicio.

En las pruebas, este servicio ha demostrado ser el método de búsqueda más utilizado para localizar agentes concretos.

En la figura 5.3 se muestra el diseño que se ha decidido usar para la implementación de este servicio en el *framework*. Podemos observar las siguientes características:

- La clase AMS consiste en la implementación generada automáticamente por ICE basándose en la interfaz definida. Todas sus funciones pueden llamarse remotamente. Además, esta clase contiene las estructuras de datos que deben guardarse para conservar la persistencia del servicio. sto es un requisito de la utilidad de ICE *Freeze*, que facilita la persistencia de la clase.
- La clase AMSPrx es la clase proxy generada por ICE para dar acceso remoto a los objetos de tipo AMS. Es necesaria para acceder al resto de réplicas del servicio.
- En la jerarquía de clases también se hereda de Mutex, que es la implementación de este tipo de artefacto que proporciona ICE. Un mútex es una estructura de datos que permite controlar el acceso con exclusión mutua a un recurso compartido.
- La clase AMSI es la que proporciona la implementación de las operaciones del servicio AMS.

Las funciones más importantes de este servicio son las siguientes:

- register: sirve para registrar un agente en el servicio. Es llamada automáticamente por los agentes durante su construcción.
- deregister: sirve para deshacer el registro de un agente en el servicio. Es llamada automáticamente por los agentes durante su destrucción.
- search: permite buscar un agente en el servicio utilizando su id. Esta id es del tipo que utiliza ICE para identificar objetos.

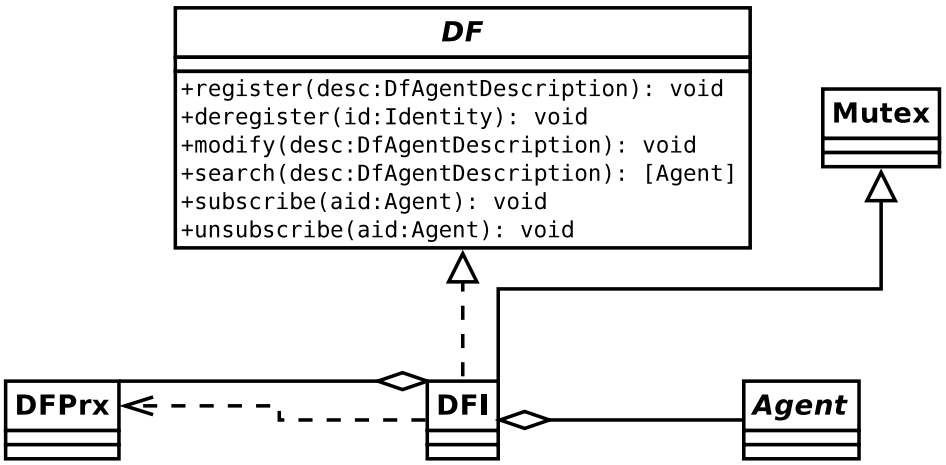

Figura 5.4: Diagrama de clases del servicio DF.

- searchWithString: es igual que la anterior, solo que en lugar de buscar utilizando un objeto Identity de ICE permite expresar esta id como una cadena y dejar que sea la propia función quien realice la conversión.
- getPlatformName: devuelve el nombre que se le ha dado a la plataforma actual (es decir, al despliegue de un sistema multiagente actual).
- clearAMS: borra todos los agentes registrados en el servicio.

5.2.2 Directory Facilitator (DF)

DF es el otro servicio de localización de agentes del framework. Su labor es la de un servicio de *páginas amarillas*, ya que, al igual que el AMS, registra todos los agentes del sistema y permite hacer búsquedas entre ellos. La diferencia con el anterior es que permite hacer búsquedas por la descripción del agente en lugar de por sólo su nombre.

Para ello, se proporciona una estructura DfAgentDescription (ver interfaces en el apéndice B) que permite a un agente preguntar por aspectos diversos como los servicios que se quieren buscar y los protocolos que soportan. Por ejemplo, un agente del sistema podría hacer una búsqueda preguntando por otros agentes que proporcionen un servicio de acceso a una base de datos; y para ello no es necesario que el agente que lo busca supiera previamente si existe algún agente en la plataforma desplegada que proporcione dicho servicio.

Este servicio dispone de una característica denominada **federación**, que consiste en que un DF puede registrarse en otro DF ya existente, formando un grafo de estos servicios en el que se puede buscar en todo él preguntando sólamente en uno de ellos.

La implementación del DF hecha en este proyecto hace uso de uno de los servicios del middleware ZeroC ICE denominado IceStorm (comentado más en detalle en 3.5.4). Éste consiste en una implementación del patrón *observer* [GHJV94] para objetos distribuidos.

En la figura 5.4 puede verse el diseño utilizado para la implementación de este servicio en el *framework*. El diseño es exactamente igual que en el caso anterior del AMS, excepto por que tiene otras funciones.

Las funciones más importantes de este servicio son las siguientes:

- register: sirve para registrar un agente en el servicio. Es llamada automáticamente por los agentes durante su construcción. A diferencia de su equivalente en el AMS, cada agente se registra utilizando una descripción en lugar de un objeto Agent.
- deregister: sirve para deshacer el registro de un agente en el servicio. Es llamada automáticamente por los agentes
- modify: permite modificar la descripción de un agente ya registrado.
- search: permite buscar en el servicio los agentes que coincidan con la descripción.
- subscribe: esta función suscribe un agente al servicio para tener la capacidad de recibir actualizaciones cuando se produzcan cambios en él.
- unsubscribe: deshace la suscripción de un agente previamente suscrito al servicio.

5.2.3 Message Transport System (MTS)

MTS es un servicio cuyo propósito es el paso de mensajes entre los distintos agentes de un mismo sistema. En el caso de este framework, este servicio pierde algo de protagonismo ya que los propios agentes tienen la capacidad de enviar y recibir mensajes sin necesidad de pasar por él; pero aún así, para respetar el estándar FIPA, este servicio tambien está implementado y funciona como se espera. De esta forma, este framework proporciona dos **métodos de comunicación** entre los agentes:

- Directo de agente a agente: para ello, todos los agentes disponen de una función llamada sendMessage.
- A través del MTS: para lo cual este servicio proporciona su propia función sendMessage.

En ambos casos, es necesario que el agente que va a mandar el mensaje conozca el proxy del agente receptor (o los proxies si fueran varios). Ese conocimiento puede obtenerse de varias formas:

- Mediante mensajes anteriores.
- Mediante algún mecanismo programado en los agentes por el desarrollador que utilice el framework.
- Preguntando a los servicios AMS y/o DF previamente comentados.

5.3 Servicios de replicación

La implementación realizada de todos los servicios básicos (AMS, DF y MTS) soporta **replicación**. Esto quiere decir que no tiene por que haber una única instancia de cada uno de estos servicios, sino que puede haber varias prestando todas el mismo servicio. Esto aporta ventajas como las siguientes:

- Aumenta la **disponibilidad del sistema**, al ser accesible desde distintas instancias.
- Mejora su **escalabilidad**, al permitir distribuir la carga entre distintas instancias.
- Proporciona **robustez**, ya que el fallo de alguna de sus partes no hace que falle todo el sistema.

Cada instancia de un mismo servicio es denominada **réplica**. La presencia de réplicas presenta un nuevo problema: ya que un cliente de un servicio puede usar cualquier réplica indistintamente, quizá deban compartir información para estar sincronizadas y que cualquiera de ellas pueda atender las peticiones igual. Por ello, para hablar de la replicación se deben estudiar los dos casos siguientes:

- Servicios sin estado.
- Servicios con estado.

Se habla de estado cuando un servicio tiene que mantener una cierta información que puede cambiar bajo ciertas condiciones (por petición de un agente, de otro servicio, con el paso del tiempo, etc). Un ejemplo de estado sería la lista de agentes que contiene el AMS, que cambia cada vez que se crea o destruye un agente.

En este escenario, hacen su aparicion diversos problemas típicos de los sistemas distribuidos, como pueden ser los siguientes:

- Necesidad de establecer una **política para actualizar todas las réplicas de un servicio**.
- Problemas en la **sincronización del estado**: debido a fallos temporales de la red, algunas réplicas no reciben la actualización del estado pero otras sí.
- Formación de **islas**: una parte de las réplicas queda incomunicada con el resto pero comunicada entre ellas, de forma que sigue funcionando de manera autónoma al resto pero no sincronizada con él.

A continuación, estos casos son desarrollados más en detalle.

5.3.1 Servicios sin estado

Es el caso más sencillo. Al no tener estado, no existe ningún problema de coordinación ni de consistencia de la información entre las réplicas, por lo que todo se reduce a la distribución de la carga entre ellas. Este es el caso del MTS.

Esta tarea de distribución ha sido facilitada por el middleware ICE, cuyo servicio IceGrid (ver 3.5.4) soporta la capacidad para crear grupos de réplicas. Un **grupo de réplicas** es un conjunto de éstas que comparten una cierta configuración común. En este caso, hablamos de la política que se sigue para decidir, una vez solicitado un servicio, cuál de las réplicas del grupo va a atender la petición. IceGrid proporciona las siguientes **políticas**:

- Aleatoria: se elige una réplica aleatoriamente (siguiendo una distribución uniforme) sin tener en cuenta más parámetros.
- Adaptativa: utilizando la información de carga del sistema (medida en función de las peticiones remotas que está atendiendo cada réplica en cada momento), se elige la réplica menos cargada.
- *Round Robin*: las réplicas están ordenadas y se va escogiendo una cada vez. Al llegar al final, se vuelve a empezar por la del principio.
- Ordenada: se escoge la réplica por unas prioridades previamente definidas.

Las políticas más utilizadas en las pruebas de este framework han sido la adaptativa y la aleatoria. Esta última es importante, ya que al distribuir las peticiones uniformemente ofrece un buen balanceo de carga con un coste computacional bajo.

5.3.2 Servicios con estado

En este caso la replicación es algo más compleja, ya que, al tener estado, aparecen problemas relacionados con la coordinación y la consistencia de información entre las réplicas. La parte de distribución de carga es exactamente igual que en el caso anterior. Éste es el caso de los servicios AMS y DF.

La replicación de estos servicios con estado se ha realizado siguiendo un **esquema maestro-esclavo**. En todo momento, una de las réplicas es el maestro y el resto son esclavos de ella. Eso da lugar a dos posibilidades para atender una petición:

- Cuando se recibe una petición de escritura: si es el maestro el que recibe una petición de escritura, la realiza y luego distribuye el cambio hacia todos los esclavos. Si es un esclavo el que recibe una petición de escritura, éste redirige dicha petición hacia el maestro. Puede verse una representación en la figura 5.5.
- Cuando se recibe una petición de lectura: tanto el maestro como cualquiera de los esclavos pueden atender peticiones de lectura en cualquier momento. Puede verse una representación en la figura 5.6.

Todo esto da lugar al problema de decidir quién es el maestro en cada momento y qué pasa si éste falla. Para garantizar la uniformidad del sistema, se ha usado una variante del algoritmo Bully [GM82]. El motivo de no haber usado el algoritmo original ha sido que las condiciones del framework no necesariamente eran las ideales enumeradas en el algoritmo

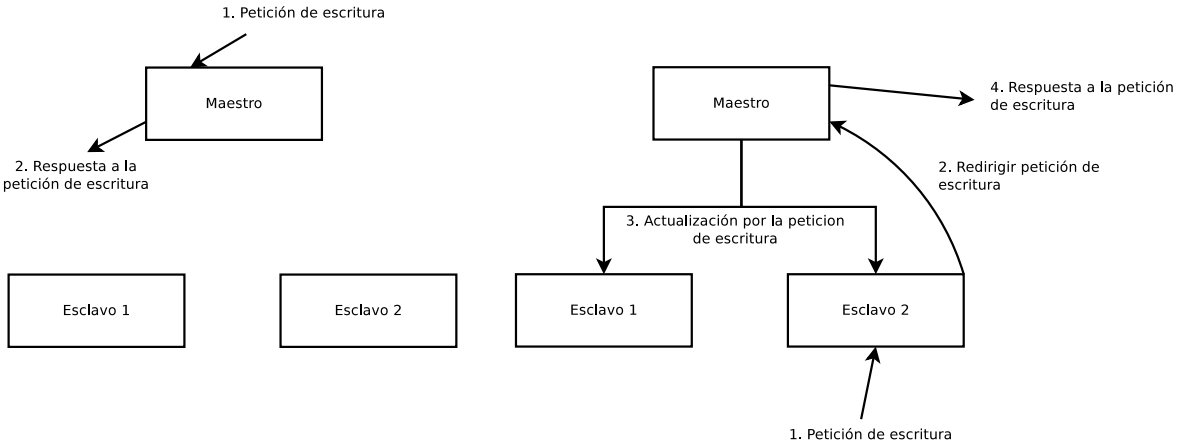

Figura 5.5: Ejemplo de peticiones de escritura.

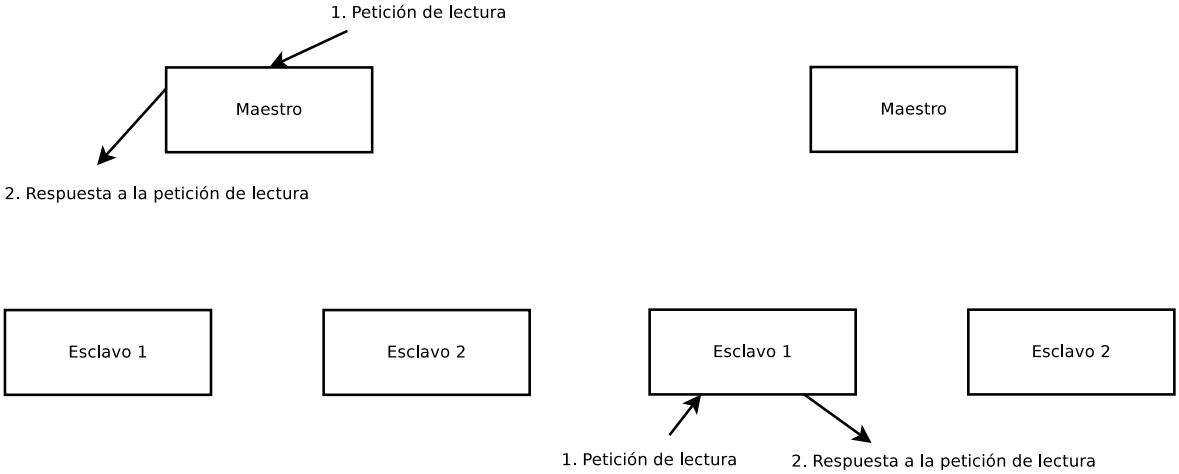

Figura 5.6: Ejemplo de peticiones de lectura.

(por ejemplo, el algoritmo habla de paso de mensajes y este framework utiliza un middleware de Remote Procedure Call (RPC)). Los siguientes **pasos** resumen el algoritmo utilizado a grandes rasgos:

1. Periódicamente o cuando pierde la conexión con el maestro, un proceso P inicia un procedimiento de elecciones enviando un mensaje a todos los procesos preguntando si existe alguno con mayor prioridad.
2. Si el proceso P no recibe respuesta de un proceso con mayor prioridad, entonces gana las elecciones y se autoproclama maestro, retransmitiéndolo a todos los procesos.
3. Si el proceso P recibe respuestas de uno o más procesos con mayor prioridad, espera un tiempo a que otro proceso se autoproclame maestro. Si esto no ocurre, vuelve a comenzar desde el paso 1.
4. Si el proceso P recibe un mensaje de un proceso con menor prioridad, envía un mensaje informando de ello y vuelve a empezar el proceso de elección.

Nótese también que muchos servicios del middleware ICE tambien soportan replicación, como son el registro y IceStorm (ver 3.5.4), que tambien son usados en este framework.

5.4 Servicios de persistencia

La persistencia consiste en la capacidad para mantener el estado de un servicio desde que éste se detiene (bien sea por un fallo, manualmente, etc) hasta que vuelve a ponerse en marcha. Para ello, el estado debe almacenarse en algún soporte secundario que no sea volátil como la memoria; bien sea un archivo en un disco duro, una base de datos, etc. Para que la persistencia sea efectiva, debe realizarse automáticamente, con el fin de que cuando se detenga el servicio se almacene su estado más reciente posible.

Dos de los servicios básicos (AMS y DF) tienen soporte de persistencia de datos gracias a la implementación planteada en este proyecto. El servicio MTS no requiere persistencia ya que no tiene estado.

La persistencia es proporcionada gracias al servicio que proporciona ICE llamado *Freeze Evictor* (ver 3.5.4). Su funcionamiento es el siguiente: se proporciona persistencia automática para los atributos que se indican en las interfaces Slice (que pueden verse en el apéndice B). Esta persistencia, en el caso de este framework, está configurada para que sea inmediata. Es decir, cada vez que se actualiza un valor de estos servicios, esta información se guarda inmediatamente con el propósito de que, en caso de fallo, la mayor parte de los datos del estado de los servicios queden almacenados.

La persistencia está implementada internamente por ICE con una base de datos Berkeley DB [1], aunque se puede configurar para otras bases de datos. En este caso, se ha utilizado la configuración por defecto.

5.5 Creación de agentes

Para la creación de agentes hay que recurrir al servicio AgentFactory implementado en este proyecto, el cual proporciona dos funciones, una para crear un sólo agente y otra para crear una secuencia de ellos. Los agentes que este servicio pueden crear depende de una **biblioteca dinámica** (detallada en esta sección más adelante) que debe indicarse en la configuración de la plataforma.

Para más detalles, consúltese el interfaz de éste servicio en el apéndice B y las instrucciones de despliegue en el apéndice A.

[1] http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html

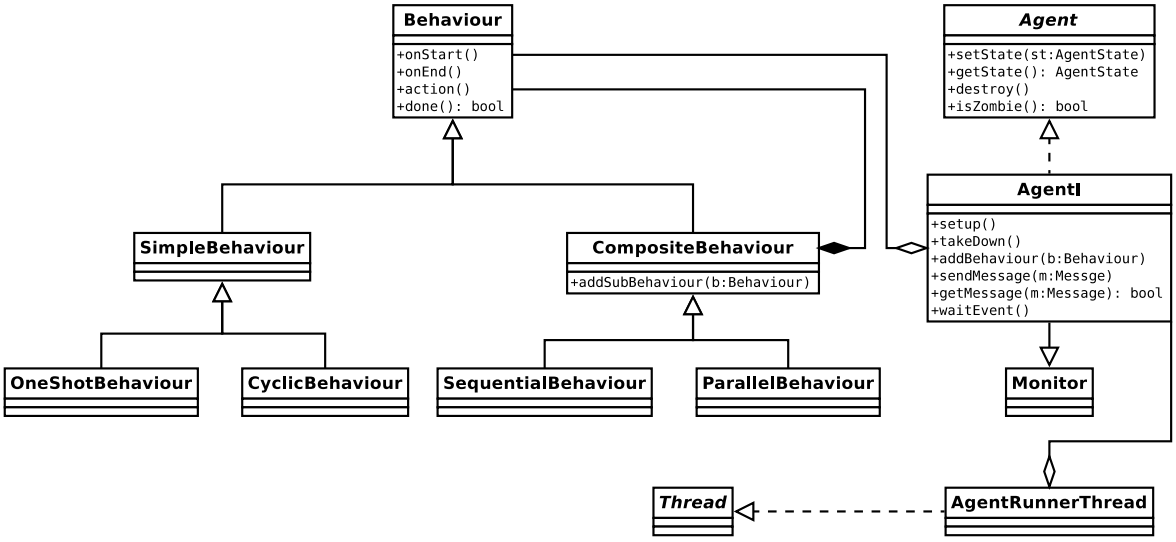

Figura 5.7: Diagrama de clases de agentes y comportamientos.

5.5.1 Agentes

El modelo de agente presente en este framework está fuertemente inspirado en el de JADE (comentado anteriormente en 3.4.2). De esta forma, un agente tiene asociados uno o varios **comportamientos**, los cuales consisten en tareas que el agente ejecuta de una determinada forma, dependiendo del tipo de comportamiento que sea. Este modelo permite diseñar los agentes de manera modular y teniendo un buen control sobre cómo se ejecutan estas tareas.

En la figura 5.7 puede verse el diagrama de clases que representa las relaciones entre los agentes y sus comportamientos. Se empezará describiendo los métodos de los comportamientos, que son los siguientes:

- onStart: se ejecuta la primera vez que se ejecuta el comportamiento. Su implementación es opcional.
- onEnd: se ejecuta al acabar la ejecución del comportamiento. Su implementación también es opcional.
- action: es el contenido del comportamiento. Su implementación es obligatoria.
- done: indica si el comportamiento ha terminado. Su implementación es obligatoria, pero dependiendo del tipo de comportamiento puede darse ya una implementación.
- addSubBehaviour: sirve para añadir *subcomportamientos* a los comportamientos compuestos, que son descritos más abajo en este documento.

Para entender mejor la función de cada uno de los métodos, puede verse el flujo de ejecución en la imagen 5.8. La secuencia es la siguiente:

1. Ejecutar setup del agente (descrito en la siguiente subsección).
2. Coger comportamiento de la lista, si lo hay. Si no, ir al paso 9.
3. Comprobar si el agente ha sido destruido.
4. Si el comportamiento se ejecuta por primera vez, ejecutar onStart.
5. Ejecutar action.
6. Si done devuelve falso, volver al paso 3. Si no, continuar.
7. Ejecutar onEnd.
8. Volver al paso 2.
9. Terminar ejecutando takeDown del agente (véase la siguiente subsección).

A la hora de diseñar agentes y sus comportamientos, es muy importante tener en cuenta el orden en que se van a ejecutar sus métodos y el orden en el que aparecen en la lista de comportamientos del agente. También hay que tener en cuenta que el agente sólo comprueba si ha sido destruido al pasar por el paso 3.

Tambien hay que resaltar, para evitar confusiones, que este flujo no es exactamente igual al que siguen los comportamientos en JADE.

En la figura 5.7 podemos observar las relaciones de herencia que se establecen desde la clase Behaviour. Estas clases describen los diferentes tipos de comportamiento. Un desarrollador debe crear comportamientos que hereden de estas clases, dependiendo del tipo de comportamiento que quiera. Se distinguen los siguientes **tipos de comportamiento**:

- Comportamientos genéricos: estos almacenan un estado y ejecutan una operación diferente dependiendo del estado. Terminan cuando una condición definida por el usuario se hace cierta. Sirven para crear comportamientos que no se ajusten a ninguna de las definiciones de los otros tipos. Se corresponden con la clase Behaviour.
- Comportamientos simples: esta categoría engloba aquellos comportamientos que proporcionan una implementación del método done pero no contienen otros comportamientos a su vez. Heredan de la clase SimpleBehaviour. Son los siguientes:
    - Comportamientos ***one-shot***: este tipo de comportamientos se ejecutan una sola vez, es decir, modelan una tarea atómica. La clase OneShotBehaviour ya implementa el método done() devolviendo «verdadero», y puede ser extendida para implementar estos comportamientos. Un ejemplo sería un comportamiento que se quisiera que fuese ejecutado una única vez.

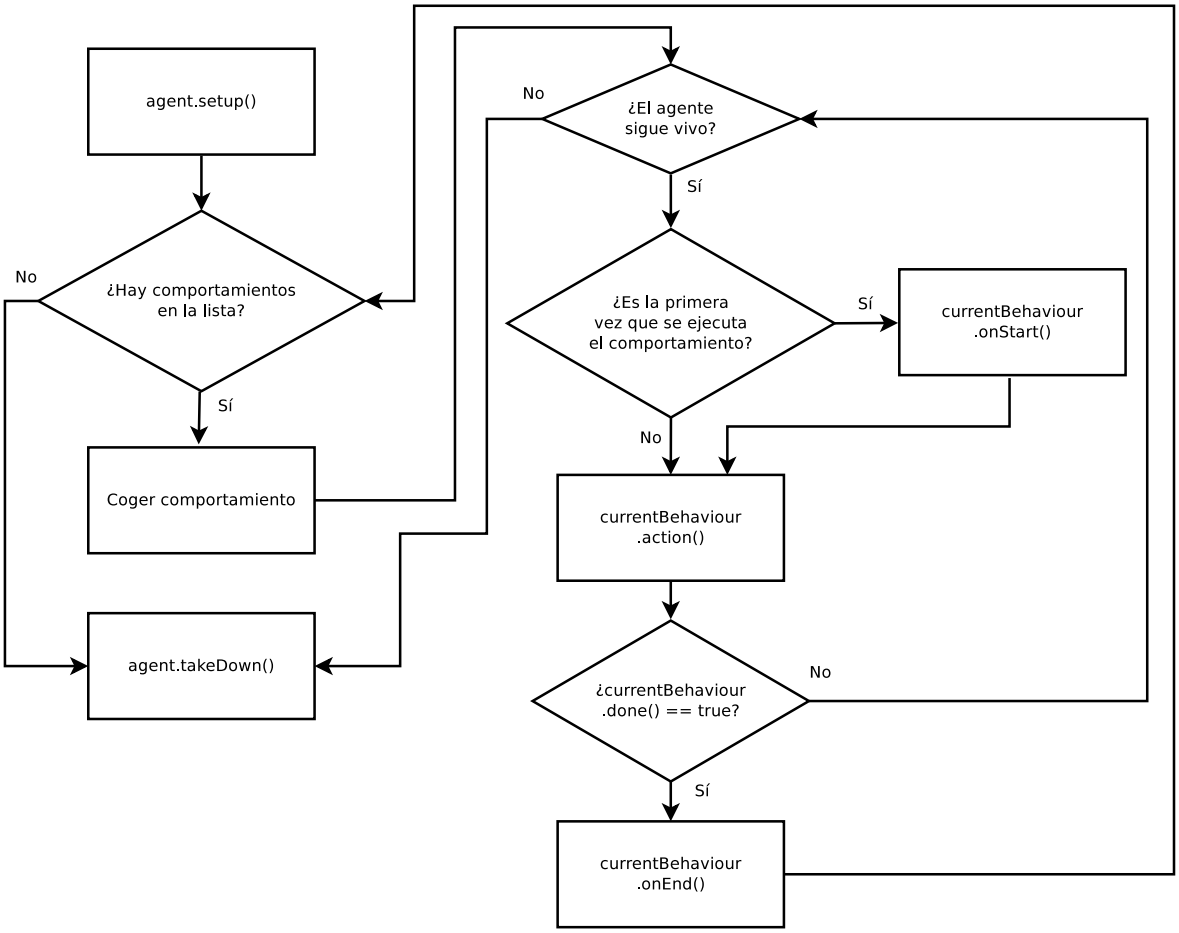

Figura 5.8: Flujo de ejecución de los comportamientos.

- Comportamientos **cíclicos**: son comportamientos que nunca se completan y cuyo método action() ejecuta las mismas operaciones cada vez que es llamado, es decir, modelan una tarea cíclica. La clase CyclicBehaviour implementa el método done() devolviendo siempre *falso*, y puede ser extendida para implementar estos comportamientos. Puede utilizarse para comportamientos que se quieran ejecutar cíclicamente hasta que se invoque el método destroy del agente.

- Comportamientos **complejos**: estos comportamientos sirven para implementar tareas compuestas por otras subtareas (tareas complejas o compuestas). Todos heredan de la clase CompositeBehaviour. Pueden ser de dos tipos:

  - SequentialBehaviour: sus subtareas se ejecutan secuencialmente. A efectos prácticos, no hay diferencia entre añadir *subcomportamientos* a un SequentialBehaviour y añadir comportamientos no complejos al agente, pero puede servir para agruparlos.
  - ParallelBehaviour: sus subtareas de ejecutan concurrentemente. Se ha implementado haciendo que cada uno de sus *subcomportamientos* se ejecute en un nuevo hilo.

5.5.2 Implementación de agentes.

Primero hay que implementar los comportamientos. Esto deberá hacerse dependiendo del tipo de comportamiento que queramos que tengan, los cuales se han dscrito en la subsección anterior. Luego, para implementar el agente propiamente dicho, hay que heredar de la clase AgentI. En la figura 5.7 podemos observar las relaciones que se establecen con la clase AgentI:

- Agregación hacia la clase Behaviour: esta relación representa el hecho de que un agente contiene un conjunto de comportamientos que debe ejecutar.
- Implementación de la clase Agent: esta clase es el interfaz definido para comunicarse remotamente con los agentes (ver más detalles en el anexo B).
- Herencia de la clase Monitor: un monitor es una estructura de datos que proporciona acceso con exclusión mutua a un recurso y además permite bloquearlo a la espera de una señal. Esta implementación de agente también es a su vez un monitor, lo que permite hacer espera pasiva para la recepción de mensajes y también puede ser aprovechado para poner al agente en espera de otros recursos.
- Agregación desde la clase AgentRunnerThread: esta es la clase que realiza la ejecución de un agente en su propio hilo.

Un agente dispone tanto de los métodos de la clase Agent como los de la clase AgentI. La diferencia entre los que aparecen en una y en la otra es que los métodos que define Agent son accesibles remotamente, es decir, podríamos ejecutarlos con un proxy a un objeto Agent que no tiene por qué estar en la misma máquina que el proceso que lo invoca. Los de la clase AgentI son para ser usados localmente.

Para implementar un agente, deben implementarse los siguientes métodos:

- setup: Se ejecuta al iniciar el agente. En él deben añadirse las tareas que ejecutará el agente mediante el método addBehaviour(Behaviour* b).
- takeDown: Se ejecuta al finalizar el agente. Debería contener instrucciones de limpieza.

Nótese que el método addBehaviour también podría llamarse desde los comportamientos, lo que podría servir para añadir nuevos comportamientos de manera dinámica respecto a condiciones que se cumplan durante la ejecución del agente y no sólamente antes de comenzar.

Aparte de los anteriores, un agente dispone de otros métodos ya implementados que pueden ser de utilidad:

- setState: establecer estado del agente (ver apéndice B). Estos estados son los definidos por el estándar FIPA.

- getState: obtener estado del agente.
- sendMessage: enviar un mensaje a otro agente, directamente de agente a agente.
- destroy: matar al agente (terminar su ejecución). Esto no ocurre inmediatamente, sino que en el ciclo de ejecución del agente se comprueba en determinados momentos si el agente ha sido destruido (ver gráfico 5.8).
- isZombie: saber si un agente ha sido destruido.
- getMessage: obtener un mensaje de la cola de mensajes, si lo hay.
- waitEvent: bloquear el agente hasta que ocurra un evento. Normalmente se refiere a la llegada de un mensaje.

También hay que implementar un método llamado getAgent, que es el que le devuelve a la fábrica de agentes el agente que debe crear. Toda la implementación de agentes realizada debe ser compilada como una biblioteca dinámica (archivos .so en GNU/Linux y .dll en Windows). Durante el despliegue, se le indica al *framework* la ruta hacia ésta biblioteca y luego pueden crearse los agentes que son proporcionados por la citada función getAgent. De esta forma, el código del *framework* se mantiene independiente de los sistemas mulitagente que se implementen e, incluso, permite cambiar esta implementación sin dejar de ejecutar el *framework* (aunque si hay agentes desplegados de la implementación anterior, estos deberán destruirse para no causar un error).

Puede verse un ejemplo de implementación de un agente en el anexo A.

Mirando el archivo de cabecera AgentI.h puede comprobarse que existen más métodos en la clase AgentI, sin embargo estos pertenecen a características experimentales o a métodos que, aunque sean públicos, son de uso interno del *framework*.

5.6 Interfaz gráfica de administración

El servicio IceGrid de ZeroC ICE se configura mediante un archivo XML en el cual se indican todos los elementos y parámetros del despliegue de un sistema distribuido. Este *framework* utiliza esa misma capacidad para definir los sistemas multiagente y sus despliegues.

Además, este servicio ofrece una herramienta de administración denominada IceGrid GUI. Esta herramienta proporciona todo lo necesario para configurar y administrar un entorno distribuido con IceGrid sin tener que editar manualmente el citado archivo XML. Permite hacer gráficamente operaciones como las siguientes:

- Definir plantillas para los servicios que se van a desplegar.
- Añadir nodos al entorno donde se despliegan servicios.
- Configurar cada servicio.
- Guardar todas las configuraciones en un archivo XML.

Esta herramienta se utiliza en el *framework* para los siguientes propósitos:

- Indicar qué servicios (ya sean los básicos o las fábricas de agentes) van a desplegarse en cada *host*.
- Añadir parámetros de la configuración, como pueden ser las identificaciones de las réplicas o la biblioteca dinámica que deben usar las fábricas de agentes para crear los agentes.
- Cambiar otros parámetros como la política de distribución de carga entre las réplicas.

Un ejemplo de su uso puede verse en el ejemplo de despliegue que hay en el apéndice A.

Capítulo 6

Evolución, resultados y costes

En este capítulo se van a discutir los distintos resultados que se han obtenido después del desarrollo de este proyecto, además de un caso de estudio que se ha planteado para comprobar un caso real de aplicación del *framework*. También se discutirá la evolución del proyecto a lo largo de su periodo de desarrollo y el coste que ha tenido.

6.1 Evolución

En Abril de 2010 el autor entró a trabajar en el grupo de investigación Oreto de esta universidad para tratar temas de inteligencia artificial distribuida. Se comenzó estudiando ontologías y el autor empezó a familiarizarse con el *middleware* de comunicaciones ZeroC ICE. También se recuperó la parte de sistema multiagente del proyecto de fin de carrera de David Vallejo, estudiando su código para usarlo para comenzar un nuevo proyecto y analizando la viabilidad de seguir con el estándar FIPA.

Un año más tarde, se comenzó con el desarrollo de este proyecto. En la figura 6.1 puede verse un diagrama orientativo sobre el tiempo que se le ha dedicado a las tareas. A continuación, se describe la cronología en semestres del desarrollo:

- Abril 2011 - Septiembre 2011: se retoma el desarrollo del *framework* de agentes. Se adapta a la versión más reciente de ICE y estudia un nuevo modelo de agente basado en comportamientos. Se decide seguir aplicando el estándar FIPA y se toma JADE como modelo.
- Octubre 2011 - Marzo 2012: se plantea la replicación y persistencia de los servicios básicos de la plataforma. Se hacen pruebas sobre la implementación de agentes realizada. Se desarrolla el paso de mensajes entre los agentes.
- Abril 2012 - Septiembre 2012: se prepara el primero de los artículos. Se trabaja en la solución de problemas varios con el *middleware* y resolución de bugs. Se prototipan los agentes y comportamientos para Python. Se empieza a crear documentación sobre uso y despliegue.
- Octubre 2012 - Marzo 2013: refactorizaciones varias, especialmente de los servicios básicos y sus capacidades de replicación.

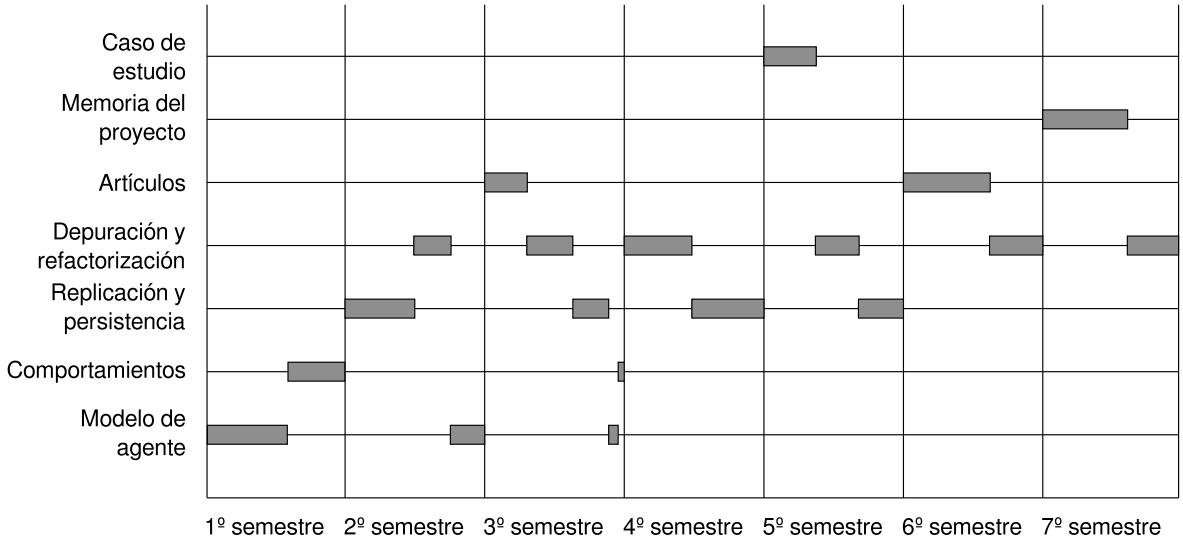

Figura 6.1: Diagrama de distribución de tiempo por tareas durante el proyecto.

- Abril 2013 - Septiembre 2013: se desarrolla el caso de estudio y se continua refactorizando y probando la replicación.
- Octubre 2013 - Marzo 2014: se prepara el segundo artículo. El autor comienza a trabajar en la empresa privada en Noviembre, lo que reduce el tiempo dedicado al proyecto.
- Abril 2014 - Septiembre 2014: se decide entregar el proyecto en Septiembre. Se recopila la documentación existente y se añade lo necesario a la memoria. Se repasan detalles menores del código fuente.

Con su entrega, el proyecto se considera cerrado en Septiembre de 2014.

6.2 Caso de estudio: sistema de videovigilancia

Para probar la funcionalidad ofrecida por el *framework*, se ha diseñado e implementado la solución a un problema relacionado con la videovigilancia. Esta decisión ha sido tomada teniendo en cuenta que la vigilancia inteligente es un campo muy apropiado para las soluciones basadas en sistemas multiagente y que puede aportar muchos problemas apropiados bien porque el *framework* ya aporta soluciones para ellos o bien para probarlo con nuevos problemas no previstos. A lo largo de esta sección se detallará el problema y la solución aportada.

6.2.1 Introducción

La vigilancia inteligente puede ser definida como la aplicación de técnicas, algoritmos y métodos de inteligencia artificial con el propósito de desarrollar sistemas de seguridad avanzados que desempeñen tareas de monitorización que, tradicionalmente, han sido llevadas a cabo por personal humano. En este contexto, la vigilancia inteligente representa un tema candente en investigación, abordando desafíos de investigación relevantes como pueden ser

la detección efectiva, *tracking* y, desde un punto de vista más general, la comprensión del comportamiento de objetos en movimiento [VV05].

El objetivo final de la vigilancia inteligente consiste en automatizar las tareas de monitorización de entornos complejos de una manera robusta y eficiente. En otras palabras, la vigilancia inteligente también apunta a avanzar el estado del arte de los sistemas de vigilancia tradicionales, donde el personal de seguridad es el responsable de supervisar actividades sospechosas en monitores. Sin embargo, la observación continua de este tipo de dispositivos ha demostrado ser inefectiva tras largos periodos de tiempo [Smi04]. En este contexto, la vigilancia inteligente puede reducir esta dependencia humana asistiendo correctamente al personal de seguridad y optimizando el proceso de monitorización.

Actualmente, existen sistemas artificiales capaces de detectar comportamientos anómalos, identificando objetos sospechosos o perdidos, analizando la trayectoria de objetos móviles o incluso detectando multitudes.

En los últimos años, dos de los principales factores que han determinado la evolución de los sistemas de vigilancia inteligente son los siguientes [RVFT07]:

- La distribución de sensores usada para recolectar datos del entorno (como, por jemplo, cámaras de seguridad).
- El hecho de que nuestra sociedad demanda una mayor seguridad.

Por una parte, la distribución física de sensores de vigilancia implica el uso de procesamiento distribuido y fusión de información. Por otra parte, una mayor seguridad implica el uso de técnicas de inteligencia artificial y métodos para diseñar soluciones más sofisticadas. Estas soluciones deben ser capaces de generar información de alto nivel cuando monitorizan entornos complejos.

Entonces, desde un punto de vista general, el diseño de sistemas de vigilancia avanzados puede ser entendido como el diseño de sistemas distribuidos basados en conocimiento. De esta manera, la información recolectada desde el entorno es usada para generar conocimiento y ese conocimiento es usado para proporcionar una vigilancia sofisticada. En este contexto, la tecnología de agentes representa una solución más que adecuada para afrontar este reto. De hecho, los sistemas multiagente han sido ampliamente usados para dirigir problemas relacionados con la vigilancia como pueden ser segmentación de imágenes [BDBR04], *tracking* multicámara [RSJ04], entendimiento del comportamiento [CCP09] o fusión de información [LLACSJ09], por nombrar unos pocos.

6.2.2 Planteamiento

El problema planteado es el siguiente: dada una secuencia de vídeo en la que se muestra la actividad de diversos objetos móviles (vehículos y peatones) en una determinada escena y un conjunto de áreas definidas sobre ésta, se quiere monitorizar estos objetos para detectar

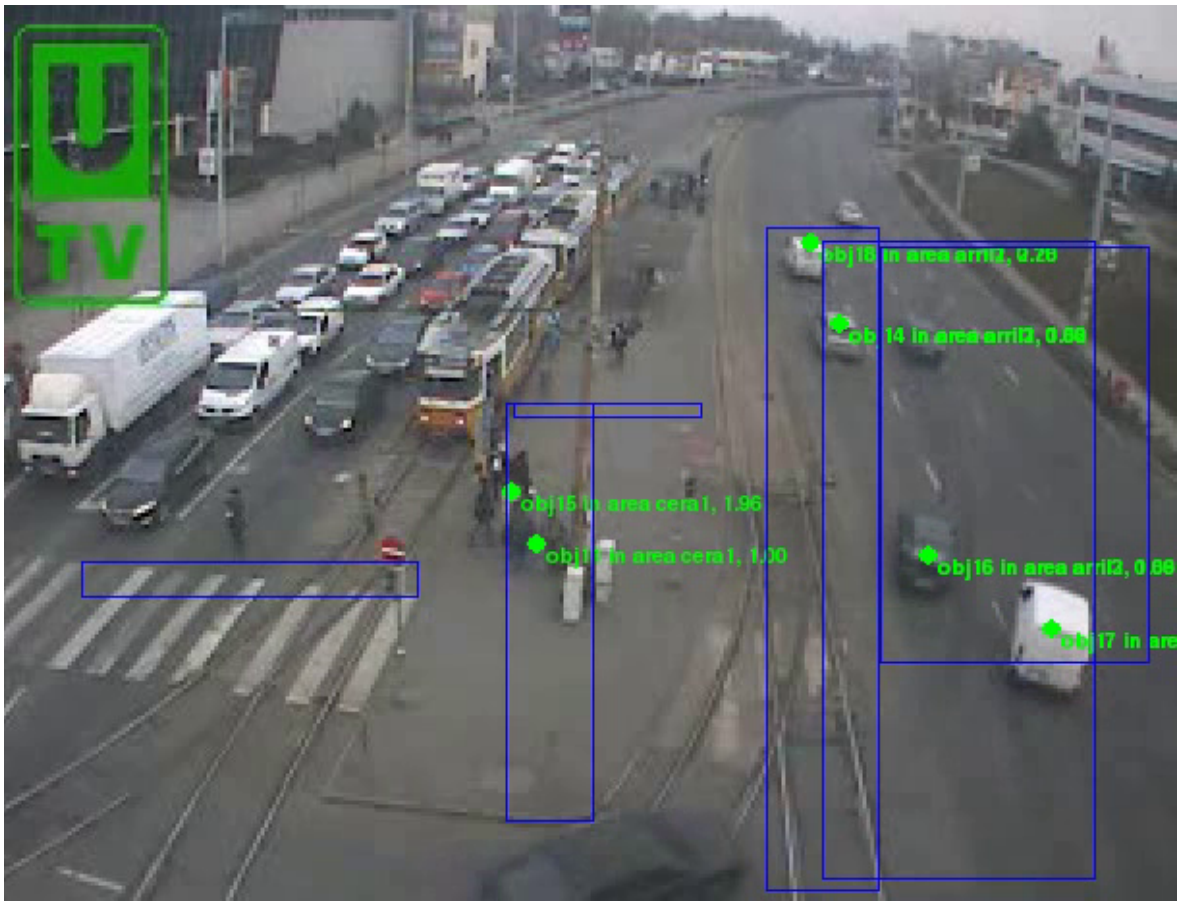

Figura 6.2: Ejemplo de vigilancia que muestra áreas y objetos móviles sobre ellas.

cuándo están en una zona prohibida para ellos. Para ello, se estudiará cuándo alguno de estos objetos móviles se superpone sobre alguna de las áreas definidas y qué porcentaje de dicho objeto está sobre el área.

En la imagen 6.2 se puede ver un ejemplo de un instante capturado de un vídeo de vigilancia, donde vemos un conjunto de áreas señaladas y de objetos móviles sobre ellas. La elección de las áreas a observar puede responder a varios motivos, como pueden ser los siguientes:

- Son áreas conflictivas de cruce de peatones.
- Son cercanas a la cámara y se está centrando la observación en ellas.
- Son intersecciones de caminos entre movimiento de peatones y de vehículos (pasos de peatones).

Se ha utilizado el framework para diseñar e implementar una solución a este problema basada en un sistema multiagente. Éste es un buen ejemplo de aplicación para probar el framework por los siguientes motivos:

- Requiere la creación y destrucción de gran cantidad de agentes, lo que permite poner a prueba las fábricas de agentes y su rendimiento cuando hay muchos agentes en la

misma máquina.

- Requiere un flujo continuo de comunicación entre los agentes, lo que permite probar el paso de mensajes implementado y la característica de espera pasiva cuando los agentes esperan mensajes.
- Requiere características que no proporciona el *framework*, permitiendo demostrar su flexibilidad. Por ejemplo el uso de IceStorm, que consiste en una implementación del patrón *observer* que proporciona ICE y que podría integrarse bien con los agentes.
- Es una posible aplicación real del *framework*.

6.2.3 Herramientas

Para este caso de estudio, se utilizarán las siguientes herramientas:

- Tracker: es un programa que, para una una secuencia de vídeo dada, es capaz de detectar los objetos que hay en movimiento y hacerles un seguimiento. En este caso, el tracker era una herramienta ya proporcionada basada en OPENCV (Open Source Computer Vision Library) [1]. OPENCV es una biblioteca libre de visión por computador.
- Ffmpeg [2]: es una herramienta que permite realizar operaciones con archivos de vídeo, entre otras cosas. En este caso se utilizará para convertir vídeos en *frames* y viceversa.
- Este *framework*.
- El lenguaje Python con la biblioteca Pygame [3], para dibujar resultados sobre las imágenes del vídeo. Pygame es un *wrapper* de la biblioteca SDL (Simple DirectMedia Layer) [4] para Python. SDL es una biblioteca multiplataforma que proporciona acceso a recursos multimedia como audio, gráficos y teclado.

6.2.4 Diseño de la solución

Se ha optado por un diseño el en cual habrá tres tipos de agente:

- Un tipo para introducir los datos de entrada.
- Otro tipo para representar las áreas que están siendo vigiladas.
- Y otro tipo más para representar los objetos móviles en la escena.

En la figura 6.3 se muestra el diagrama de clases del diseño planteado para solucionar el problema. En este diagrama pueden reconocerse los siguientes tipos de agente:

- TrackerAgent: es el agente encargado de poner todo en marcha. Tiene dos comportamientos:

[1] http://opencv.org
[2] https://www.ffmpeg.org/
[3] http://www.pygame.org
[4] https://www.libsdl.org

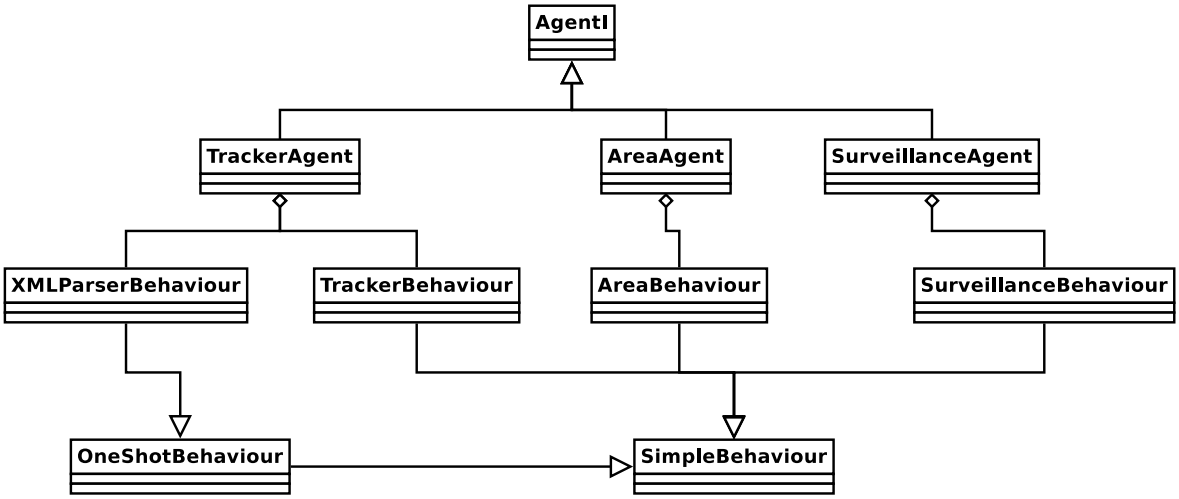

Figura 6.3: Diagrama de clases del diseño de agentes para la solución.

  - XMLParserBehaviour: lee el archivo XML donde están definidas las áreas y crea un agente para cada una.
  - TrackerBehaviour: lee el archivo de salida del *tracker* y envía un mensaje al agente que representa cada objeto con la información sobre su posición. Si previamente no existía un agente para ese objeto, lo crea.

- AreaAgent: representa una área que se esté observando. Tiene un único comportamiento, AreaBehaviour, que tiene como tarea esperar recibir la posición de otros objetos y calcular en qué porcentaje están dentro del área.
- SurveillanceAgent: representa un objeto en movimiento. Tiene un único comportamiento, SurveillanceBehaviour, cuya función es publicar su posición cada vez que ésta es acualizada. Esta publicación es escuchada por los agentes de tipo AreaAgent.

Como se puede apreciar, el modelo de agente elegido al desarrollar el *framework* hace muy sencilla la tarea de definir e implementar los agentes y las tareas que estos tienen que realizar.

6.2.5 Flujo de trabajo

Se plantean los siguientes pasos a seguir:

1. Se ejecutará el tracker con el vídeo que se va a estudiar, obteniendo un archivo de salida con los distintos objetos detectados en el vídeo.
2. Se ejecutará el sistema multiagente diseñado para la solución, obteniendo un archivo de salida con las áreas vigiladas y los objetos que han estado
3. Se usará la utilidad ffmpeg para dividir el vídeo en *frames*.

4. Se usará un script programado en Python que utiliza Pygame para dibujar los resultados obtenidos en el paso 2 sobre los *frames* generados en el paso 3 y así ofrecer realimentación al usuario sobre los datos obtenidos.
5. Se volverá a usar ffmpeg para unir los frames con la nueva información en un vídeo que muestre todo.

Puede verse un *frame* del vídeo que se obtiene al final del proceso en la imagen 6.2.

Debe notarse que, aunque en este caso se ha optado por un análisis de los datos *offline*, es decir, con los datos del vídeo previamente obtenidos, también sería posible plantear una solución de análisis de los datos en tiempo real. En tal caso, el diseño debería plantearse para que los agentes y las tareas que realizan sean lo suficientemente ágiles como para responder apropiadamente en los tiempos en que llegan los datos desde el vídeo.

6.2.6 Conclusiones del caso de estudio

Después de todo el proceso comentado en el flujo de trabajo, se ha obtenido un vídeo [5] que muestra los resultados del caso de estudio. Este vídeo se incluye en el CD que acompaña esta documentación.

Hay que señalar que el *tracker* utilizado no es muy preciso y eso da lugar a errores que pueden apreciarse en el vídeo final. Algunos de estos errores son los siguientes:

- Un mismo objeto móvil cuyo seguimiento es interrumpido y reanudado, dando lugar a que parezcan dos distintos.
- Mala detección del tamaño de los objetos móviles debido a problemas de perspectiva.
- Objetos móviles no detectados.

Tambien influye la calidad del vídeo utilizado. Sin embargo, no era propósito de este caso de estudio el probar el *tracker*, ya que lo importante era el procesamiento de sus resultados por un sistema multiagente.

En la imagen 6.4 pueden verse unas capturas del vídeo original enfrentadas a esos mismos instantes del vídeo obtenido. Los rectángulos azules representan las zonas que son observadas. El motivo de su forma es que son aproximaciones a zonas reales del vídeo que de no ser representadas por rectángulos harían más complicada la detección de objetos sobre ellas.

Los objetos móviles son representados por puntos verdes. Cuando estos objetos móviles se encuentran parcial o totalmente dentro de una zona observada, el punto verde va seguido de una cadena de información que aporta los siguientes datos:

- Una identificación única asignada al objeto.
- El área sobre la que se encuentra.

[5] http://www.esi.uclm.es/www/dvallejo/wi-iat/surveillance_output.mpg

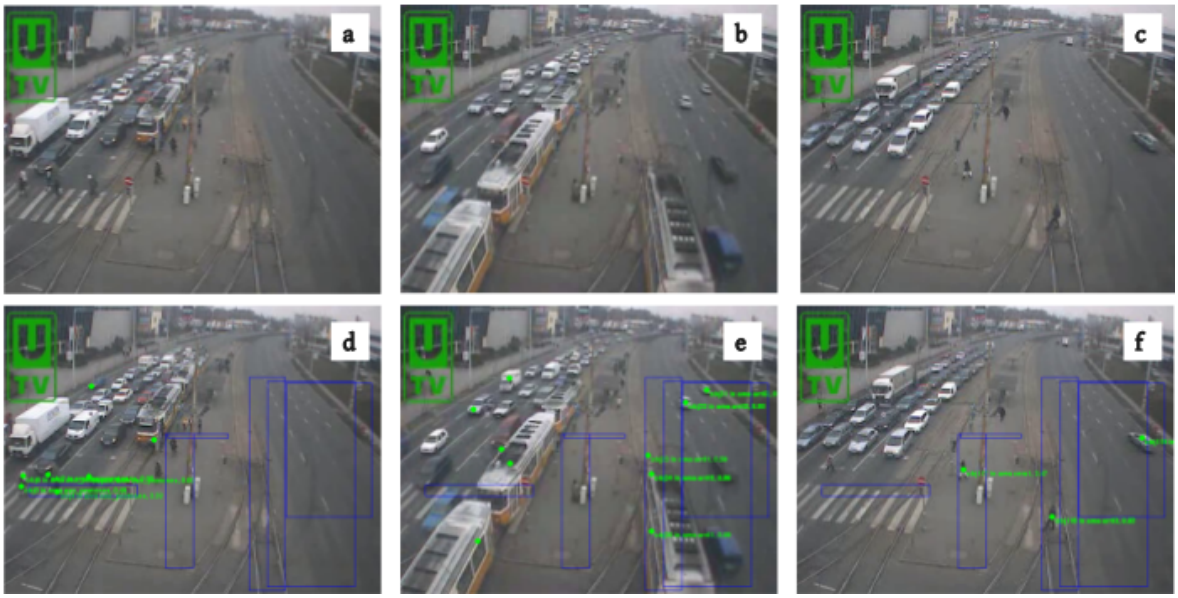

Figura 6.4: Imágenes capturadas del vídeo original (primera fila) y del resultado del caso de estudio (segunda fila).

- El porcentaje (sobre 100) del objeto que se encuentra sobre el área.

El caso de estudio permitió tambien mejorar la implementación del paso de mensajes entre los agentes, gracias a que sacó a relucir que la espera activa que los agentes realizaban al esperar cada mensaje reducía su rendimiento. Por ello, ahora la implementación del paso de mensajes utiliza espera pasiva basada en monitores.

Como conclusión final, se ha observado que la solución implementada con este *framework* responde con fluidez y se adapta correctamente a un problema real, demostrando así la capacidad del framework para ser usado en muchos otros problemas.

6.3 Publicaciones científicas

Este proyecto ha permitido generar dos publicaciones científicas, las cuales son descritas a continuación.

6.3.1 KES AMSTA 2012

En el artículo *Developing Intelligent Surveillance Systems with an Agent Platform* se discuten los avances en el desarrollo de este framework teniendo en mente el dominio de la vigilancia inteligente. Se presta especial atención a aspectos clave como son la escalabilidad, robustez y flexibilidad. Además, se centra en el esfuerzo por las siguientes características:

- Implementar un servicio de replicación para los principales componentes del *framework*.
- Proveer a los desarrolladores de un modelo de desarrollo de agentes flexible.
- Integrar negociaciones concurrentes.

Este artículo se publicó en la *6th International KES Conference on Agents and Multi-agent Systems, Technologies and Applications (KES AMSTA 2012), Dubrovnik (Croatia)* y puede consultarse en [VGMA+12].

6.3.2 SPUnd 2013

El artículo *An agent-based approach to understand events in surveillance environments* también habla del *framework* y de su aplicación en el dominio de la vigilancia inteligente, estando más centrado en cómo el modelo de agente orientado a comportamientos aporta la suficiente flexibilidad para afrontar los distintos problemas que surgen en las tareas de vigilancia. Se enseñan resultado experimentales para mostar cómo este enfoque de agentes puede contribuir a entender los eventos en entornos de tráfico urbano.

Este artículo se publicó en el *SPUnd 2013 Workshop of 2013 IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology, Atlanta* y puede consultarse en [VVG+13].

6.4 Costes

Es difícil dar una medida exacta del coste del proyecto debido a los cambios en rendimiento a lo largo de todo el desarrollo, habiendo diversos factores que han afectado a éste. Algunos de estos factores son los siguientes:

- Conocimientos del autor: al principio el avance era menor y de peor calidad. Conforme el autor iba aprendiendo sobre las tecnologías con las que trabajaba, se refactorizaba código anterior y se avanzaba más rápido.
- Tarea que se desarrollaba: hay tareas que, debido a su complejidad, han llevado más esfuerzo que otras. Un ejemplo es el sistema de replicación de servicios.
- Periodos en los que en los que había que entregar otros trabajos y prácticas en las asignaturas.
- Periodos de exámenes.

Por tanto, se hará una estimación del coste del proyecto siguiendo los siguientes criterios:

- Un sueldo de programador junior en C++, establecido en 20 euros por hora.
- 3 años de desarrollo, de los cuales de cada año se asumen 9 meses efectivos por descontar periodos de exámenes y de vacaciones.
- Horario laboral de media jornada (20 horas semanales).

Por tanto, el coste del proyecto se estima en 43200 euros. Debe señalarse que el elevado coste se debe a que es un proyecto de una complejidad importante y que requiere de conocimientos en múltiples áreas.

Capítulo 7

Conclusiones y trabajo futuro

En este capítulo se discutirán las conclusiones que se han obtenido de todo el trabajo realizado para este proyecto. También se enumerarán posibles líneas de trabajo para seguir en el futuro.

7.1 Conclusiones

Para comentar las conclusiones, éstas se enfrentarán con los objetivos comentados en el capítulo 2, que estaban divididos en objetivo general y objetivos específicos.

7.1.1 Objetivo general cumplido

Al final de todo el proceso de desarrollo, se ha diseñado y desarrollado un *framework* que permite crear y desplegar sistemas multiagente distribuidos. En concreto, se han respondido a los siguientes aspectos:

- El *framework* permite desarrollar y desplegar soluciones a problemas de inteligencia artificial distribuida.
- Permite utilizar dos lenguajes de programación: C++ y Python, aunque el soporte de Python es experimental y se ha puesto más énfasis en el soporte de C++. También presenta un diseño que facilita la incorporación de nuevos lenguajes de programación para los agentes.
- Se ha utilizado con éxito el *middleware* ZeroC ICE.
- El proyecto ha permitido al autor enfrentarse a multitud de problemas y a aprender sobre ellos.

Es importante resaltar la complejidad que ha alcanzado el proyecto, con problemas como sincronización de nodos, trabajo en múltiples lenguajes, problemas diversos relacionados con las comunicaciones, uso de tecnologías cuya formación se ha realizado *sobre la marcha*, reutilización de código no escrito por el autor, estudio de un *middleware* grande y complejo e incluso la generación de artículos científicos. El haber podido enfrentar esta complejidad y haber acabado el proyecto con todo ello también es considerado por el autor como un importante logro cumplido.

7.1.2 Objetivos específicos cumplidos

Los objetivos específicos estaban divididos en funcionales (funcionalidades que se esperaban obtener con el proyecto) y didácticos (conocimientos que se esperaban adquirir con el proyecto).

Objetivos funcionales cumplidos

- Se ha establecido un **modelo de agente orientado a comportamientos** (tareas).
- Se ha seguido el **estándar FIPA** durante el desarrollo de los servicios que ofrece este **framework**.
- Gracias al *middleware* ZeroC ICE, se han proporcionado **facilidades para las comunicaciones en red** haciendo uso de técnicas RPC.
- El framework ofrece un **sistema de replicación** para mejorar la disponibilidad de sus servicios.
- El framework utiliza **persistencia de datos** en sus servicios esenciales para mejorar la capacidad de recuperación ante fallos.
- Se ha pensado un diseño que permite a un desarrollador **aprender a usarlo de una manera relativamente fácil**, además de proporcionarse la presente documentación.
- Se ha solucionado e implementado un caso de uso para probar el *framework* en un posible **uso real**.

Objetivos didácticos cumplidos

El autor de este proyecto ha tenido la oportunidad de adquirir conocimientos y experiencia sobre las siguientes materias:

- Se ha mejorado el entendimiento del autor del **lenguaje de programación C++**.
- El autor ha aprendido a trabajar con el *middleware* **ZeroC ICE**.
- El autor ha obtenido conocimientos relacionados con los **sistemas distribuidos**.
- Se ha trabajado con éxito en un **proyecto complejo y prolongado en el tiempo**.
- Se ha trabajado con **varios lenguajes de programación** en el mismo proyecto.

7.2 Trabajo futuro

Para continuar con el desarrollo del proyecto a partir de lo presentado, se sugieren las siguientes líneas:

- **Agent Content Language (ACL)**: es el lenguaje definido por el estándar FIPA para comunicar los agentes entre sí. Actualmente se encuentran implementadas las estructuras

de datos que permiten tratarlo pero se deja ese tratamiento en manos del desarrollador que trabaje con el *framework*. Podrían añadírsele funciones y herramientas para facilitar ese tratamiento.

- **Soporte para más lenguajes**: se propone ampliar el soporte de lenguajes para implementar agentes a cualquiera que sea soportado por el *middleware* ZeroC ICE, que es el único límite que habría.
- **Interfaz gráfica propia**: actualmente se utiliza la proporcionada por ZeroC ICE, pero una interfaz gráfica especializada podría mejorar la facilidad para desplegar y administar un sistema multiagente desarrollado con este *framework*.
- **Comunicación con otros *frameworks***: siguiendo el estándar FIPA, podría hacerse que los agentes desarrollados para este *framework* pudieran comunicarse con los de otros *frameworks* y viceversa.
- **Negociaciones concurrentes**: consisten en poner de acuerdo a todas las partes de una negociación (vendedores y compradores) para efectuar una compra. Las técnicas de este área son muy utilizadas en comercio electrónico y se adaptan muy bien a un sistema multiagente. Podrían añadirse funciones y herramientas la *framework* para facilitar estas técnicas.
- **Promoción del uso de este *framework***: sería conveniente dar a conocer el *framework* para que éste fuese usado por desarrolladores de sistemas multiagente. Se propone, entre otros medios, la creación de un sitio web que muestre información, manual, tutoriales y descarga del framework y ejemplos.

ANEXOS

Anexo A

Compilación y ejemplo de despliegue

En este apéndice vamos se describe cómo compilar el framework y cómo desplegar una aplicación multiagente usándolo. Es recomendable tener algún conocimiento sobre el middleware ZeroC ICE, ya que el framework lo usa y hay muchos conceptos sobre él (registro, IceGrid, etc.) que deben conocerse.

A.1 Compilación

En primer lugar, debe existir una variable de entorno llamada $FIPA que contenga la ruta donde se encuentra el framework instalado. Puede indicarse en el archivo .bashrc añadiendo la siguiente línea:

export FIPA=/home/user/agentplatform/core

Para compilar, debe accederse al directorio:

$FIPA/cpp/

Luego, debe ejecutarse make. Deben estar instaladas las siguientes aplicaciones:

- GNU Make
- G++ 4.6
- ZeroC ICE 3.4.2

En la tabla A.1 se incluye la estructura de directorios del proyecto para facilitar la comprensión del contenido de cada uno de ellos.

A.2 Ejemplo de implementación de agente

MyAgent.h

```
#ifndef __MyAgent_h__
#define __MyAgent_h__

#include <Ice/Ice.h>
```

core/cpp/	Código fuente en C++.
core/python/	Código fuente en Python.
core/slice/	Interfaces en Slice.
core/doc/	Documentación.
core/doc/doxygen/	Documentación generada por Doxygen en html.
core/doc/articles/	Artículos publicados.
apps/surveillance/	Código y resultados del caso de estudio.

Cuadro A.1: Estructura de directorios.

```
#include <AgentI.h>
#include <MyBehaviour.h>

using namespace std;

class MyAgent: public virtual ::AgentI
{
 public:
  MyAgent(Ice::Identity, Ice::ObjectAdapterPtr);
  ~MyAgent();

  virtual void setup();
  virtual void takeDown();
};

#endif
```

MyAgent.cpp

```
#include <MyAgent.h>

// name y oa solo hay que pasarselos a la clase padre. Son necesarios
    para el
// buen funcionamiento del agente.
MyAgent::MyAgent(Ice::Identity name, Ice::ObjectAdapterPtr oa):
  AgentI(name, oa) {}

MyAgent::~MyAgent() {}

void MyAgent::setup()
{
  std::cout << "Metodo setup de 'MyAgent' ejecutado." << std::endl;
  addBehaviour(new MyBehaviour());
}

void MyAgent::takeDown()
{
```

```
  std::cout << "Metodo takeDown de 'MyAgent' ejecutado." << std::endl;
}
```

MyBehaviour.h

```
#ifndef __MyBehaviour_h__
#define __MyBehaviour_h__

#include <OneShotBehaviour.h>
#include <iostream>

using namespace std;

class MyBehaviour: virtual public OneShotBehaviour
{
 public:
  MyBehaviour();
  ~MyBehaviour();

  virtual void action();
};

#endif
```

MyBehaviour.cpp

```
#include <MyBehaviour.h>

MyBehaviour::MyBehaviour() {}

MyBehaviour::~MyBehaviour() {}

void MyBehaviour::action()
{
  cout << "Ejecutado metodo action() de la clase MyBehaviour." << endl;
}
```

MyGetAgent.cpp

```
#include <AgentFactoryI.h>
#include <MyAgent.h>

AgentI* AgentFactoryI::getAgent(const std::string& agentType, Ice::
    Identity id,
    Ice::ObjectAdapterPtr oa) {

  if (agentType == "myAgent")
    return new MyAgent(id, oa);
```

```
    assert(false);
}
```

A.3 Configuración inicial

Cada nodo debe tener su archivo de configuración. Estos irán guardados en $FIPA/config. Ejemplo de configuración:

```
## Los siguientes parametros solo deben ponerse si el registro se
    encuentra ##
## en este nodo. ##

# Nombre de la aplicacion. Es seguro dejarlo asi.
IceGrid.InstanceName = FIPA

# Endpoints para el registro
IceGrid.Registry.Client.Endpoints = tcp -p 4061
IceGrid.Registry.Server.Endpoints = tcp
IceGrid.Registry.Internal.Endpoints = tcp

# Permisos para conectarse al registro con icegridadmin o con icegrid-gui
IceGrid.Registry.AdminPermissionsVerifier = FIPA/NullPermissionsVerifier
IceGrid.Registry.PermissionsVerifier = FIPA/NullPermissionsVerifier

# Directorio donde almacena la base de datos del registro.
IceGrid.Registry.Data = ../db/registry

# Estos parametros deben ir asi.
IceGrid.Registry.DynamicRegistration = 1
IceGrid.Registry.DefaultTemplates = 1

# Este parametro indica que el registro se encuentra en este nodo.
IceGrid.Node.CollocateRegistry = 1

## Hasta aqui la configuracion para el registro. Lo siguiente debe
    ponerse en
## todos los nodos. ##

# Endpoints del nodo
IceGrid.Node.Endpoints = tcp

# Nombre del nodo
IceGrid.Node.Name = FIPAServices

# Directorio donde almacena la base de datos del nodo
IceGrid.Node.Data = ../db/FIPAServices
```

```
# Direccion del servicio Locator (que se encuentra en el mismo sitio que
    el
# registro)
Ice.Default.Locator = FIPA/Locator:tcp -h 127.0.0.1 -p 4061
```

A.4 Arranque y parada de la plataforma

Los scripts $FIPA/config/start.sh y $FIPA/config/stop.sh sirven para arrancar y parar los distintos nodos, respectivamente. El script $FIPA/config/clean.sh borra las bases de datos creadas por la plataforma, para empezar desde cero.

Ejemplo de start.sh

```
echo "Creating directories..."
mkdir -p ../db/FIPAServices ../db/registry ../db/Node1 ../logs

echo "Deploying FIPAServices..."
icegridnode --Ice.Config=config.fipa --daemon --nochdir
sleep 1
echo "Deploying Node 1..."
icegridnode --Ice.Config=config.node1 --daemon --nochdir
sleep 1
echo "Starting application..."
icegridadmin --Ice.Config=config.client -e "application add FIPA.xml"
echo "OK"

# Es importante respetar el orden: primero se despliegan los nodos y
    luego se
# arranca la aplicacion.
```

Ejemplo de stop.sh

```
#!/bin/bash
icegridadmin --Ice.Config=config.client -e "application remove FIPA"
sleep 1
echo "Shutting down the nodes..."
icegridadmin --Ice.Config=config.client -e "node shutdown Node1"
icegridadmin --Ice.Config=config.client -e "node shutdown FIPAServices"

# Debe respetarse el orden de parada: primero se para la aplicacion y
    luego
# cada nodo.
```

Ejemplo de clean.sh

```
#!/bin/bash
```

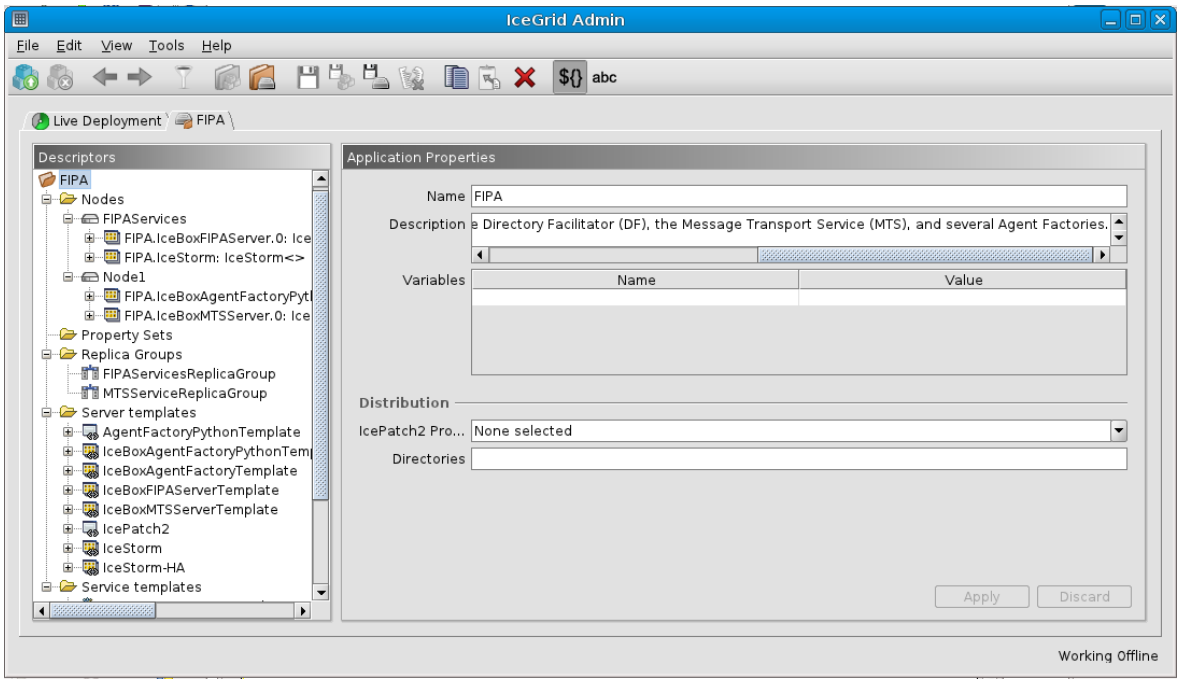

Figura A.1: IceGridGUI con FIPA.xml abierto.

```
echo "Cleaning..."
rm -rf *~ ../logs ../db
```

También existe el script $FIPA/config/reset.sh, que simplemente lanza consecutivamente stop.sh y clean.sh.

A.5 FIPA.xml

FIPA.xml es el archivo de configuración de IceGrid que contiene el despliegue propiamente dicho. La manera más cómoda de editarlo es usar la utilidad IceGridGUI.

Sólamente hay que editar la sección *Nodes* para añadir, quitar y modificar nodos. Las demás secciones no hay que editarlas.

Para añadir un nuevo nodo, hacemos click derecho en *Nodes*, y seleccionamos del menú emergente *New Node*. Después podemos añadir servidores para el nodo haciendo click derecho sobre éste y eligiendo *New Server from Template*, ya que todos los servicios deben ser añadidos desde las plantillas disponibles en *Server templates* (véanse las imágenes A.5 y A.5). Para que funcione al mínimo, en la plataforma deben estar desplegados al menos un servidor de cada tipo de los siguientes: *IceBoxFIPAServerTemplate*, *IceBoxMTSServerTemplate*, *IceBoxAgentFactoryTemplate* y *IceStorm*.

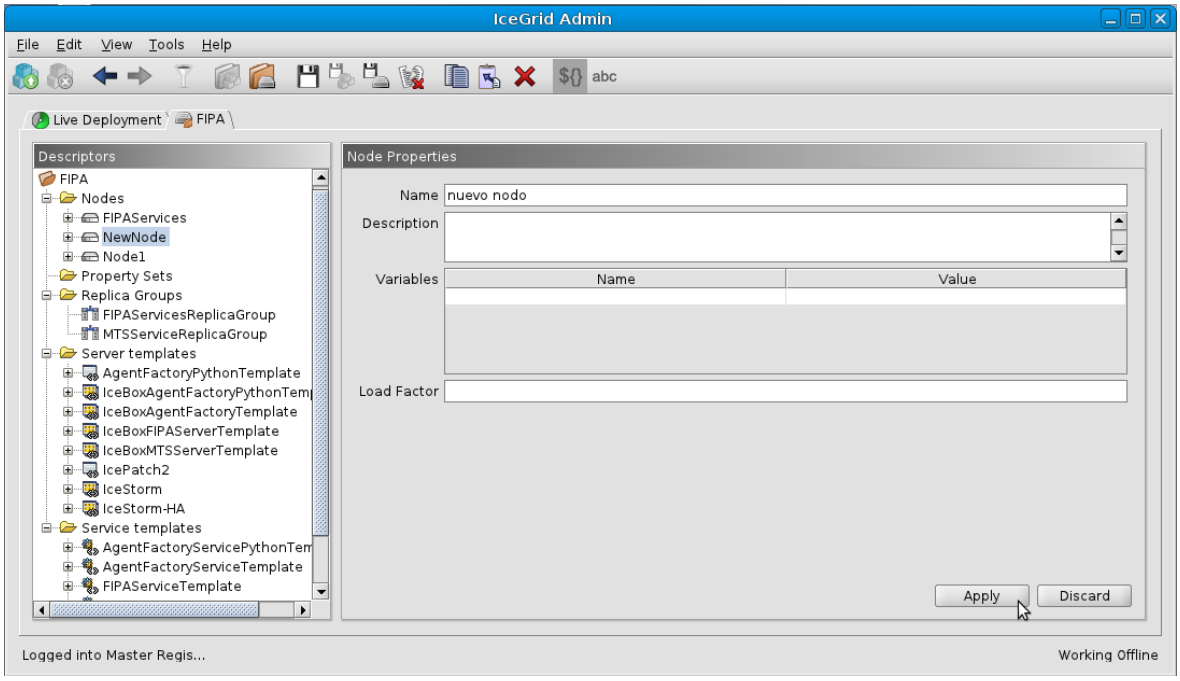

Figura A.2: Estableciendo el nombre de un nuevo nodo.

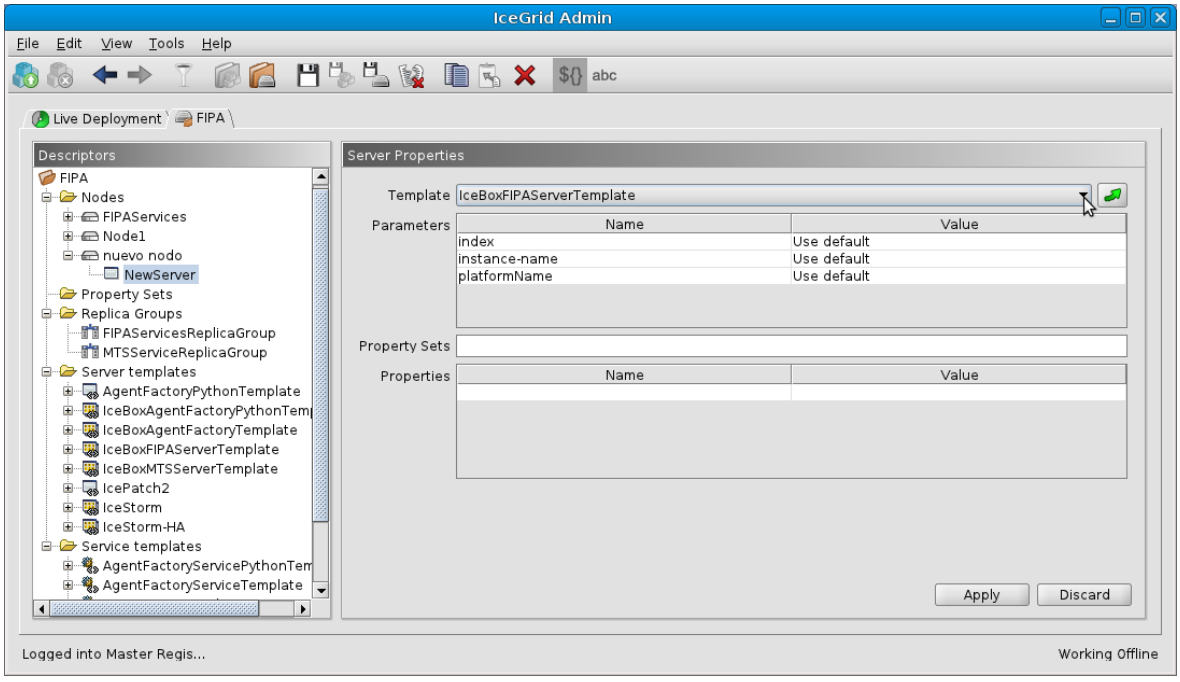

Figura A.3: Añadiendo el servidor FIPAServer al nuevo nodo desde su plantilla.

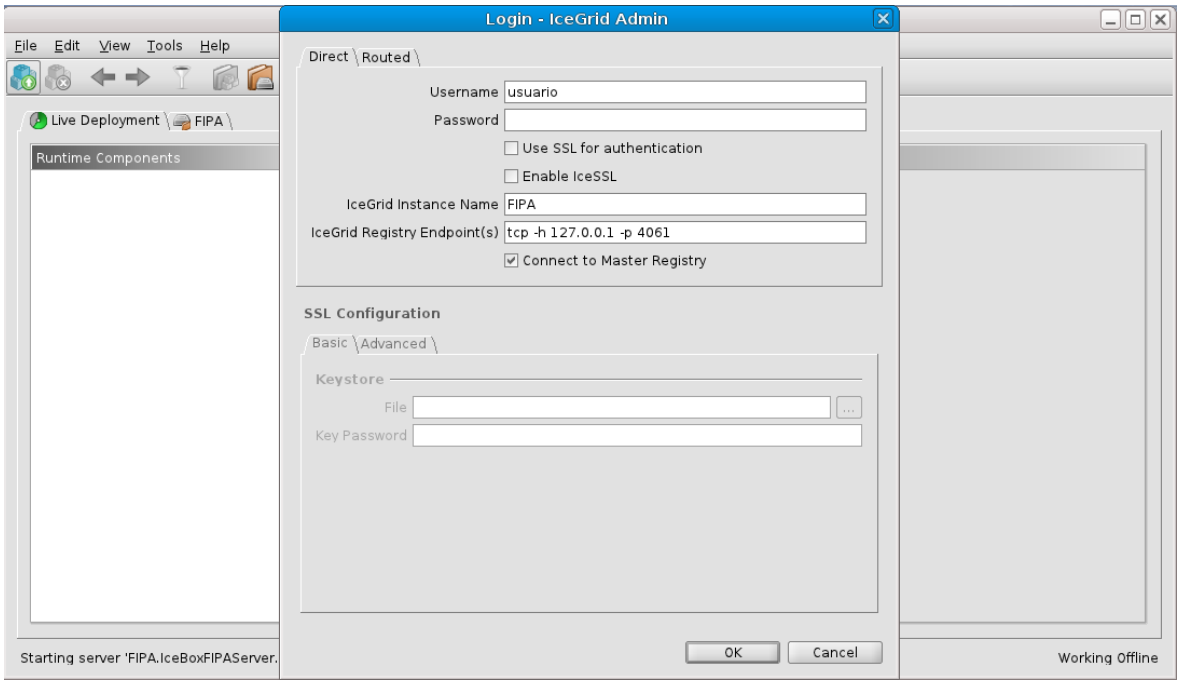

Figura A.4: Conectándose al registro.

A.6 Ejecución

Una vez ejecutado start.sh, ya podemos conectarnos al registro con IceGridGUI para ejecutar los nodos correspondientes a los AMS y a los DF (ambos servicios están contenidos en la plantilla *IceBoxFIPAServerTemplate*). Un ejemplo puede verse en las imágenes A.6 y A.6. El resto de servicios no es necesario lanzarlos manualmente, ya que se activan bajo demanda.

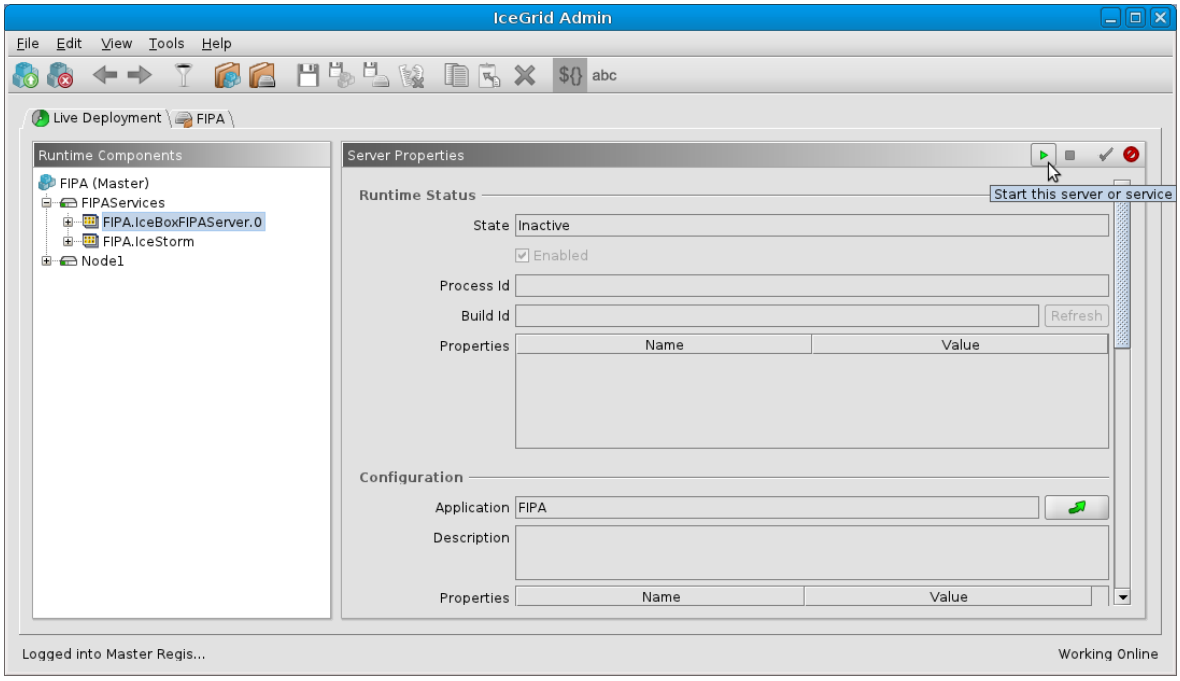

Figura A.5: Iniciando un nodo de tipo FIPAServer.

Anexo B

Interfaces Slice

En este anexo se citan las interfaces de los objetos accesibles remotamente del *framework*. Estas interfaces representan los servicios básicos del estándar FIPA, los tipos de datos utilizados en las llamadas remotas y los agentes. Están escritas en lenguaje Slice, que es el usado por el *middleware* ZeroC ICE para definir las interfaces de forma independiente del lenguaje en el que se implementan.

B.1 FIPA.ice

```
// -*- mode: c++; coding: utf-8 -*-

/**
 *
 * ICE implementation of the FIPA Agent Management Specification.
 * See http://fipa.org/specs/fipa00023/SC00023K.html for a detailed
    description.
 **/

#ifndef __FIPA_ice__
#define __FIPA_ice__

#include <FIPATypes.ice>
#include <Agent.ice>

module FIPA
{
  class AMS
  {
    // Parameters to save with the evictor.

    /// Name of the platform.
    string platformName;

    /// Agents in the platform.
    AMSAgentMap agents;

    /// Proxy to this servant.
```

```
FIPA::AMS* myProxy;

/// Proxy to the master.
FIPA::AMS* master;

/// True if this is the master.
bool iAmMaster;

/// Priority.
int priority;

/// Directory for the Freeze database.
string dbDir;

/// Name of the Freeze database.
string dbName;

/**
 * Register an agent with the agent platform.
 *
 * @param aid The proxy to the agent.
 **/
["freeze:write"] void register(Agent* aid) throws AgentExists;

/**
 * Register an agent with the agent platform (for replication).
 *
 * @param priority The priority of the emissor.
 * @param aid The proxy to the agent.
 **/
["freeze:write"] void registerR(int priority, Agent* aid);

/**
 * Deregister an agent with the agent platform.
 *
 * @param id The agent's identity.
 **/
["freeze:write"] idempotent void deregister(Ice::Identity id);

/**
 * Deregister an agent with the agent platform (for replication).
 *
 * @param priority The priority of the emissor.
 * @param id The agent's identity.
 **/
["freeze:write"] idempotent void deregisterR(int priority, Ice::
   Identity id);
```

```
/**
 * Search for an agent in the agent platform.
 *
 * @param id The agent's identity.
 *
 * @return The proxy to the agent.
 **/
idempotent Agent* search(Ice::Identity id);

/**
 * A version of search with a string argument instead of an Ice::
    Identity.
 *
 * @param s The agent's identity.
 *
 * @return The proxy to the agent.
 **/
idempotent Agent* searchWithString(string s);

/**
 * Get the name of the agent platform.
 *
 * @return The name of the agent platform.
 **/
idempotent string getPlatformName();

/**
 * Receives the announce of a new master
 *
 * @param priority Priority of the new master
 * @param master The new master
 **/
idempotent void setMaster(FIPA::AMS* master);

/**
 * Receives the designation of master.
 **/
idempotent void youAreMaster();

/**
 * Updates a slave replica the first time that it connect.
 *
 * @param slaveAMS The replica to update.
 **/
["freeze:write"] void updateSlave(FIPA::AMS* slaveAMS);
```

```
    /**
     * Clears all information of an AMS.
     **/
    ["freeze:write"] idempotent void clearAMS();

    /**
     * Returns the priority of the replica.
     **/
    idempotent int getPriority();
};

// TODO: DF federation.
class DF
{
    // Parameters to save with the evictor.

    /// Agents in the DF.
    DFAgentMap agents;

    /// Proxy to this servant.
    FIPA::DF* myProxy;

    /// Proxy to the master.
    FIPA::DF* master;

    /// True if this is the master.
    bool iAmMaster;

    /// Priority.
    int priority;

    /// Directory for the Freeze database.
    string dbDir;

    /// Name of the Freeze database.
    string dbName;

    /**
     * Register an agent's description with the DF.
     *
     * @param desc The agent's DF description.
     **/
    ["freeze:write"] void register(DfAgentDescription desc) throws
        AgentExists;

    /**
     * Register an agent's description with the DF (for replication).
```

```
     *
     * @param priority The priority of the emissor.
     * @param desc The agent's DF description.
     **/
    ["freeze:write"] void registerR(int priority, DfAgentDescription desc
        );

    /**
     * Deregister an agent with the DF.
     *
     * @param id The agent's identity.
     **/
    ["freeze:write"] idempotent void deregister(Ice::Identity id);

    /**
     * Deregister an agent with the DF (for replication).
     *
     * @param priority The priority of the emissor.
     * @param id The agent's identity.
     **/
    ["freeze:write"] idempotent void deregisterR(int priority, Ice::
        Identity id);

    /**
     * Modify an agent's description with the DF.
     *
     * @param desc The agent's DF description.
     **/
    ["freeze:write"] void modify(DfAgentDescription desc);

    /**
     * Modify an agent's description with the DF (for replication).
     *
     * @param priority The priority of the emissor.
     * @param desc The agent's DF description.
     **/
    ["freeze:write"] void modifyR(int priority, DfAgentDescription desc);

    /**
     * Search for an agent description in the DF.
     *
     * @param desc The agent's DF description.
     *
     * @return The proxies to the agents that match the description.
     **/
    idempotent AgentSeq search(DfAgentDescription desc);
```

```
/**
 * Subscribe an agent with the DF.
 *
 * @param aid The proxy to the agent.
 **/
//void subscribe (Agent* aid) throws AlreadySubscribed;
idempotent void subscribe(Agent* aid);

/**
 * Unsubscribe an agent with the DF.
 *
 * @param aid The Proxy to the agent.
 **/
idempotent void unsubscribe(Agent* aid);

/**
 * Notifies an action to a suscribed agent.
 *
 * @param action Action to notify
 * @param description Description of the action
 **/
void notify(FIPA::DFAction action, FIPA::DfAgentDescription
    description);

/**
 * Receives the designation of master.
 **/
idempotent void youAreMaster();

/**
 * Receives the announce of a new master
 *
 * @param priority Priority of the new master
 * @param master The new master
 **/
idempotent void setMaster(FIPA::DF* master);

/**
 * Updates a slave replica the first time that it connect.
 *
 * @param slaveDF The replica to update.
 **/
void updateSlave(FIPA::DF* slaveDF);

/**
 * Clears all information of an DF.
 **/
```

```
    ["freeze:write"] idempotent void clearDF();

    /**
     * Returns the priority of the replica.
     **/
    idempotent int getPriority();
  };

  interface MTS
  {
    /**
     * Receive an ACL message.
     *
     * @param aclMessage The ACL message.
     **/
    ["cpp:const"] void sendMessage(Message m);
  };

};

#endif
```

B.2 FIPATypes.ice

```
// -*- mode: c++; coding: utf-8 -*-

/**
 *
 * ICE implementation of the FIPA Agent Management Specification.
 * See http://fipa.org/specs/fipa00023/SC00023K.html for a detailed
    description.
 **/

#ifndef __FIPATypes_ice__
#define __FIPATypes_ice__

[["cpp:include:list"]]

#include <Ice/Identity.ice>
#include <Ice/BuiltinSequences.ice>

module FIPA
{
  // Exceptions.

  exception FIPAException {
```

```
    string reason = "Generic exception."; };

  exception AlreadySubscribed extends FIPAException {};
  exception AgentExists extends FIPAException {};
  exception FactoryFailed extends FIPAException {};
  exception AMSException extends FIPAException {};
  exception DFException extends FIPAException {};
  exception MTSException extends FIPAException {};
  exception AgentDeletedBeforeDestroy extends FIPAException {};
  exception LibraryException extends FIPAException {};
  exception FIPAServerFailed extends FIPAException {};

  exception AgentTypeNotExists extends FIPAException {
    string agentType; };

  /**
   * Possibles agent's states.
   **/
  enum AgentState {Initiated, Active, Suspended, Waiting, Transit,
      Deleted};

  /**
   * Possibles performatives of a ACL message.
   **/
  enum ACLPerformative {Accept, Agree, Cancel, Cfp, Confirm, Disconfirm,
      Failure, Inform, InformIf, InformRef, NotUnderstood, Propagate,
                        Propose, Proxy, QueryIf, QueryRef, Refuse,
                            RejectProposal, Request, RequestWhen,
                            RequestWhenever, Subscribe};

  /**
   * Possibles actions with the AMS and the DF
   **/
//enum DFAction {Registration, Deregistration, Modification};
  enum DFAction {RegisterDF, DeregisterDF, StartUpdateDF, EndUpdateDF,
      ModifyDF};
  enum AMSAction {RegisterAMS, DeregisterAMS, StartUpdateAMS,
      EndUpdateAMS};

  interface Agent;

  sequence<Agent*> AgentSeq;

  struct Envelope
  {
    AgentSeq to;
    Agent* from;
```

```
  string aclRepresentation;
  long date;
};

struct ACLPayload
{
  ACLPerformative performative;
  string content;
  // See FIPA00007.
  string language;
  // See FIPA00007.
  string encoding;
  string ontology;
  // See FIPA00025.
  string protocol;
  // To be unique.
  Ice::Identity conversationId;
};

struct Message
{
  Envelope env;
  ACLPayload payload;
};

struct ServiceDescription
{
  string name;
  string type;
  Ice::StringSeq protocols;
  Ice::StringSeq ontologies;
  Ice::StringSeq languages;
};

sequence<ServiceDescription> ServiceDescriptionSeq;

struct DfAgentDescription
{
  Agent* name;
  ServiceDescriptionSeq services;
  Ice::StringSeq protocols;
  Ice::StringSeq ontologies;
};

class AMS;
struct AMSMapEntry
{
```

```
    AMSAction action;
    Ice::Identity id;
    FIPA::Agent* proxy;
    FIPA::AMS* ams;
  };

  dictionary<Ice::Identity, FIPA::Agent*> AMSAgentMap;
  dictionary<Ice::Identity, FIPA::DfAgentDescription> DFAgentMap;

  class DF;
  struct DFMapEntry
  {
    DFAction action;
    Ice::Identity id;
    FIPA::Agent* proxy;
    FIPA::DF* df;
  };

  ["cpp:type:std::list< ::std::string>"] sequence<string> StringList;
};

#endif
```

B.3 Agent.ice

```
// -*- mode: c++; coding: utf-8 -*-

#ifndef __Agent_ice__
#define __Agent_ice__

#include <FIPATypes.ice>
// #include <Types.ice>

module FIPA
{
  interface Agent
  {
    /**
     * Set the agent's state.
     *
     * @param st The new agent's state.
     **/
    void setState(FIPA::AgentState st);

    /**
     * Get the agent's state.
```

```
     *
     * @return The new agent's state.
     **/
    ["cpp:const"] idempotent FIPA::AgentState getState();

    /**
     * Receive an ACL message.
     *
     * @param m The ACL message.
     **/
    void receiveMessage(FIPA::Message m);

    /**
     * Notification about a registration, deregistration or modification
        of an
     * agent with the DF.
     *
     * @param act The act of the notification.
     *
     * @param desc The agent's description.
     **/
    void dfAdv(FIPA::DFAction act, FIPA::DfAgentDescription desc);

    /**
     * Terminate the agent execution.
     **/
    idempotent void destroy();

    /**
     * Says if the agent was destroyed.
     *
     * @return True if was destroyed, false in other case.
     **/
    ["cpp:const"] bool isZombie();
};

interface AgentFactory
{
    /**
     * Create an agent.
     *
     * @param[in]  name      The agent's name.
     * @param[in]  agentType Agent type.
     *
     * @param[out] error Type of error ocurred.
     * 1 if the agent already exists, 2 if agentType invalid, 0 in
        another case.
```

```
     *
     * @return The agent's proxy. Null proxy if an error has ocurred.
     **/
    Agent* create(string name, string agentType, out int error);

    /**
     * Create a sequence of agents.
     *
     * @param[in]  names      The names of the agents.
     * @param[in]  agentTypes The types of the agents.
     *
     * @param[out] namesAlreadyRegistered The names of the agents already
         registered.
     * @param[out] agentTypesNotExists    The types of agents that not
        exists.
     *
     * @return The sequence of created agents proxies. Null proxies in
        error cases.
     **/
    AgentSeq createSeq(Ice::StringSeq names, Ice::StringSeq agentTypes,
                       out FIPA::StringList namesAlreadyRegistered,
                       out FIPA::StringList agentTypesNotExists);
  };
};

#endif
```

Bibliografía

[BCG07] Fabio Luigi Bellifemine, Giovanni Caire, y Dominic Greenwood. *Developing multi-agent systems with JADE*, volume 7. John Wiley & Sons, 2007.

[BDBR04] Ernst GP Bovenkamp, Jouke Dijkstra, Johan G Bosch, y Johan HC Reiber. Multi-agent segmentation of IVUS images. *Pattern Recognition*, 37(4):647–663, 2004.

[BPR99] Fabio Bellifemine, Agostino Poggi, y Giovanni Rimassa. JADE–A FIPA-compliant agent framework. En *Proceedings of PAAM*, volume 99, página 33. London, 1999. http://jade.tilab.com/.

[CCP09] Bo Chen, Harry H Cheng, y Joe Palen. Integrating mobile agent technology with multi-agent systems for distributed traffic detection and management systems. *Transportation Research Part C: Emerging Technologies*, 17(1):1–10, 2009.

[DR94] Edmund H Durfee y Jeffrey S Rosenschein. Distributed problem solving and multi-agent systems: Comparisons and examples. *Ann Arbor*, 1001(48109):29, 1994.

[FGM04] Jacques Ferber, Olivier Gutknecht, y Fabien Michel. From agents to organizations: An organizational view of multi-agent systems. En *Agent-Oriented Software Engineering IV*, páginas 214–230. Springer, 2004.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, y John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[GM82] Hector Garcia-Molina. Elections in a distributed computing system. *Computers, IEEE Transactions on*, 100(1):48–59, 1982.

[Hen04] Michi Henning. A new approach to object-oriented middleware. *Internet Computing, IEEE*, 8(1):66–75, 2004. http://www.zeroc.com.

[HW05] Aaron Helsinger y Todd Wright. Cougaar: A robust configurable multi agent platform. En *Aerospace Conference, 2005 IEEE*, páginas 1–10. IEEE, 2005.

[LLACSJ09] LM López-López, A Albusac, JJ Castro-Schez, y L Jiménez. Semantics-Provided Environment Views for Normality Analysis-Based Intelligent Surveillance. *International Conference on Agents and Artificial Intelligence*, páginas 1–10, 2009.

[PBL05] Alexander Pokahr, Lars Braubach, y Winfried Lamersdorf. Jadex: A BDI reasoning engine. En *Multi-agent programming*, páginas 149–174. Springer, 2005.

[RCO05] Robert Ross, Rem Collier, y Gregory MP O'Hare. AF-APL–bridging principles and practice in agent oriented languages. En *Programming Multi-Agent Systems*, páginas 66–88. Springer, 2005.

[RN04] Stuart J Russell y Peter Norvig. *Inteligencia Artificial: un enfoque moderno*. 2004.

[RSJ04] Paolo Remagnino, AI Shihab, y Graeme A Jones. Distributed intelligence for multi-camera visual surveillance. *Pattern recognition*, 37(4):675–689, 2004.

[RVFT07] Paolo Remagnino, Sergio A Velastin, Gian Luca Foresti, y Mohan Trivedi. Novel concepts and challenges for the next generation of video surveillance systems. *Machine Vision and Applications*, 18(3):135–137, 2007.

[Smi04] Gavin JD Smith. Behind the Screens: Examining Constructions of Deviance and Informal Practices among CCTV Control Room Operators in the UK. *Surveillance & Society*, 2(2/3):377, 2004.

[Syc98] Katia P Sycara. Multiagent systems. *AI magazine*, 19(2):79, 1998.

[VGMA+12] David Vallejo, Luis M García-Muñoz, Javier Albusac, Carlos Glez-Morcillo, Luis Jiménez, y José J Castro-Schez. Developing intelligent surveillance systems with an agent platform. En *Agent and Multi-Agent Systems. Technologies and Applications*, páginas 199–208. Springer, 2012.

[VV05] M Valera y SA Velastin. Intelligent distributed surveillance systems: a review. En *Vision, Image and Signal Processing, IEE Proceedings-*, volume 152, páginas 192–204. IET, 2005.

[VVG+13] David Vallejo, Felix Jesús Villanueva, LM Garcia, C Gonzalez, y Javier Albusac. An agent-based approach to understand events in surveillance environments. En *Web Intelligence (WI) and Intelligent Agent Technologies (IAT), 2013 IEEE/WIC/ACM International Joint Conferences on*, volume 3, páginas 100–103. IEEE, 2013.

[Wei99] Gerhard Weiss. *Multiagent systems: a modern approach to distributed artificial intelligence*. MIT press, 1999.

[Wei13] Gerhard Weiss. *Multiagent Systems*. MIT Press, 2013.

[Win05] Michael Winikoff. JACK^{TM} intelligent agents: An industrial strength platform. En *Multi-Agent Programming*, páginas 175–193. Springer, 2005.

[WJ95] Michael Wooldridge y Nicholas R Jennings. Intelligent agents: Theory and practice. *The knowledge engineering review*, 10(02):115–152, 1995.

[Woo02] Michael Wooldridge. *An introduction to multiagent systems*. Wiley, 2002.

Este documento fue editado y tipografiado con LaTeX
empleando la clase **arco-pfc** que se puede encontrar en:
https://bitbucket.org/arco_group/arco-pfc