AMAAVEM: Arquitectura Multi-Agente Adaptativa para la Vigilancia de Entornos Monitorizados



UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

INGENIERÍA

EN INFORMÁTICA

PROYECTO FIN DE CARRERA

AMAAVEM: Arquitectura Multi-Agente Adaptativa para la Vigilancia de Entornos Monitorizados

Francisco Javier Torrecillas Pérez

Junio, 2012



UNIVERSIDAD DE CASTILLA-LA MANCHA

ESCUELA SUPERIOR DE INFORMÁTICA

Departamento de Tecnologías y Sistemas de Información

PROYECTO FIN DE CARRERA

AMAAVEM: Arquitectura Multi-Agente Adaptativa para la Vigilancia de Entornos Monitorizados

Autor:

Francisco Javier Torrecillas Pérez

Director:

David Vallejo Fernández

Junio, 2012

Francisco Javier Torrecillas Pérez

Ciudad Real – Spain

E-mail: FcoJavier.Torrecillas@alu.uclm.es
Web site: `http://code.google.com/p/amaavem/`
© 2012 Francisco Javier Torrecillas Pérez

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

Se permite la copia, distribución y/o modificación de este documento bajo los términos de la Licencia de Documentación Libre GNU, versión 1.3 o cualquier versión posterior publicada por la *Free Software Foundation*; sin secciones invariantes. Una copia de esta licencia esta incluida en el apéndice titulado «GNU Free Documentation License».

Muchos de los nombres usados por las compañías para diferenciar sus productos y servicios son reclamados como marcas registradas. Allí donde estos nombres aparezcan en este documento, y cuando el autor haya sido informado de esas marcas registradas, los nombres estarán escritos en mayúsculas o como nombres propios.

TRIBUNAL:

Presidente:

Vocal 1:

Vocal 2:

Secretario:

FECHA DE DEFENSA:

CALIFICACIÓN:

PRESIDENTE	VOCAL 1	VOCAL 2	SECRETARIO
Fdo.:	Fdo.:	Fdo.:	Fdo.:

Agradecimientos

A mi padre, a mi madre y a Isa por estar día a día ayudándome y apoyándome. Especial mención a toda mi familia, a mi hermana, a mis tíos, a mis tías, a mis primos y en especial a mis abuelos. A mis amigos fuera de la carrera, a los amigos que he hecho dentro (en especial a César, Javi, Jorge y Paco), que me soportan a diario y me ayudan demasiado. En resumen, este proyecto va dedicado a todos mis seres queridos.

Por supuesto, agradecer a mi director David Vallejo Fernández, la oportunidad de realizar esta idea, haciendo tan sencillo el día a día gracias a su comprensión y paciencia.

Un placer haber realizado este proyecto.

Fco Javier Torrecillas Pérez

Resumen

Un sistema de vigilancia inteligente es un sistema que aplica técnicas, algoritmos y métodos de Inteligencia Artificial, cuyo objetivo es la monitorización y análisis inteligente de un entorno. El objetivo principal de la vigilancia inteligente no es captar imágenes de una zona para que un operador las revise después, sino automatizar este proceso tanto como sea posible, ofreciendo un soporte eficaz al operador humano.

En la actualidad, existen múltiples arquitecturas que implementan sistemas de vigilancia inteligente. Sin embargo, existen limitaciones en algunas de ellas que conducen a problemas arquitectónicos tales como la escalabilidad, la expansibilidad, la adaptabilidad, el rendimiento, la dificultad de uso, etc. Por ello, con la creación de Amaavem se intenta ofrecer una arquitectura potente, que resuelve este tipo de problemas y ofrece al desarrollador la posibilidad de instalar un sistema de vigilancia inteligente en entornos monitorizados.

Amaavem es una arquitectura multi-agente adaptativa para la vigilancia de entornos monitorizados. Amaavem es un sistema distribuido, gracias a la aplicación de técnicas distribuidas y al soporte del *middleware* de comunicaciones, llamado *ZeroC Ice*. Amaavem ofrece una serie de funcionalidades básicas para analizar las situaciones que suceden en cualquier entorno. Amaavem está creado empleando herramientas y estándares libres, consiguiendo una fácil portabilidad.

Para probar la arquitectura, se realizó un despliegue de la arquitectura en un dos casos de estudio: uno en una universidad y otra en una calle peatonal de una ciudad. Se estudiaron comportamientos básicos de los tipos de objetos específicos de estos entornos y posteriormente, se analizaron los resultados obtenidos.

En conclusión, el objetivo principal de este proyecto es facilitar al desarrollador final una arquitectura, que posibilita el despliegue de un sistema de vigilancia inteligente en un entorno monitorizado de manera sencilla.

Abstract

An intelligent surveillance system is a system that applies techniques, algorithms and artificial intelligence methods aimed at monitoring and analyzing environments in an intelligent way. The main objective of intelligent surveillance is not capturing images of an area for an operator to review them later, but to automate this process as much as possible, providing effective support to the human operator.

Currently, there are many architectures that implement intelligent surveillance systems. The problem is that many of them suffer from architectural issues such as scalability, expandability, adaptability, performance, difficulty of use, etc. Therefore, the creation of Amaavem attempts to provide a powerful architecture capable of solving these problems and offer customers the option of desploying a surveillance system in any environment.

Amaavem is an adaptive multi-agent architecture for monitoring environments. Amaavem is a distributed system, through the application of techniques and is supported by a distributed communications middleware, called ZeroC Ice. Amaavem offers a number of basic functions to analyze situations that happen in any environment. Amaavem is created using free tools and open standards, getting an easy portability.

The main objective of this project is to provide the customer an architecture that facilitates the deployment of an intelligent surveillance system in a simple way.

Índice general

Resumen XIII

Abstract XV

Índice general XVII

Índice de figuras XXI

1. Introducción 1

1.1. ¿Qué es un Sistema de Vigilancia Inteligente? 2

1.1.1. Evolución e impacto socio-económico 3

1.1.2. Problemática 4

1.2. Estructura del Documento 6

2. Objetivos 9

2.1. Objetivo general 9

2.2. Objetivos específicos 9

3. Antecedentes 11

3.1. Inteligencia Artificial 12

3.1.1. Introducción 12

3.1.2. Agentes Inteligentes 13

3.1.3. Sistemas Multi-Agente 20

3.2. Sistemas de Vigilancia Inteligente 26

3.2.1. Introducción 26

3.2.2. Evolución 28

3.2.3. Arquitecturas Multi-Agente 33

3.3. Middlewares de Comunicaciones 39

3.3.1. Introducción 39

3.3.2. Alternativas más relevantes 40

4. Método de Trabajo **47**
4.1. Metodología de trabajo 47
4.2. Herramientas 48
4.2.1. Lenguajes de Programación 48
4.2.2. Hardware 49
4.2.3. Software 49

5. Arquitectura **53**
5.1. Descripción general 54
5.2. Módulo Generador de Eventos 60
5.2.1. Submódulo de preprocesamiento. 62
5.3. Módulo Monitor de Vigilancia 66
5.4. Módulo Creación de Agentes 72
5.4.1. Factoría de agentes 72
5.4.2. Submódulo sistema de comunicaciones 78
5.4.3. Submódulo de comprobaciones 81
5.5. Módulo Análisis de Comportamientos 83
5.5.1. Submódulo clasificación de objetos 84
5.5.2. Submódulo análisis de eventos 86
5.6. Módulo Análisis de Resultados 88

6. Evolución, Resultados y Costes **93**
6.1. Evolución 93
6.1.1. Iteraciones 94
6.2. Caso de estudio de la arquitectura 100
6.3. Recursos y Costes 107
6.3.1. Coste 107
6.3.2. Estadísticas 107
6.3.3. Rendimiento 108

7. Conclusiones y Propuestas **113**
7.1. Objetivos alcanzados 113
7.2. Propuestas de trabajo futuro 116
7.3. Conclusión personal 118

A. Manual de usuario de Amaavem **121**
A.1. Consideraciones iniciales 121

A.2. Ajustar los archivos de configuración 122
A.3. Configurar la base de datos 125
A.4. Configuración del archivo Makefile 129
A.5. Compilación de Amaavem 130
A.6. Ejecución del programa 130
A.7. Manual de uso del módulo análisis de vídeo 135
A.8. Manual de uso del parseador de ficheros XML 136

B. Guía de desarrollo con Amaavem **137**

C. Archivos de configuración de Amaavem **147**
C.1. Archivo XML para desplegar Amaavem 149

D. Esquema de la base de datos empleada por Amaavem **153**

E. Interfaces de comunicación de Amaavem **155**

F. Código fuente **159**
F.1. Código del parseador de ficheros XML 159
F.2. Código del analizador de vídeos 161

G. GNU Free Documentation License **167**

Bibliografía **175**

Índice de figuras

1.1. Esquema general de un sistema de vigilancia inteligente. 2
1.2. Fases que abarca Amaavem dentro de un sistema de vigilancia inteligente . 5

3.1. Mapa conceptual de conocimiento relacionado con la vigilancia inteligente. 11
3.2. Interacción Agente-Entorno 13
3.3. Diagrama de un agente reactivo 16
3.4. Diagrama de un agente con estado 17
3.5. Diagrama de un agente basado en objetivos 18
3.6. Diagrama de un agente basado en utilidad 18
3.7. Diagrama de un agente que aprende 19
3.8. Sistema Multi-Agente 20
3.9. Métodos de Comunicación entre agentes: pizarra 24
3.10. Métodos de Comunicación entre agentes: paso de mensajes 25
3.11. Centro de Vigilancia Tradicional (metro de Londrés) 26
3.12. Etapas en un sistema de Vigilancia Inteligente 30
3.13. Arquitectura PRISMATICA (Esquema adaptado de [VBL05]) 34
3.14. Monitor MIPSA : Proyecto Prismática ([VBL05]) 35
3.15. Diagrama de Secuencia Sistema PRISMATICA 36
3.16. Arquitectura del Sistema de Vigilancia IBM ([TBH04]) 37
3.17. Arquitectura del Sistema de Vigilancia propuesto por [DVD08] 38
3.18. Middlewares como mecanismo de abstracción. 40
3.19. Arquitectura abstracta de ZeroC ICE. 41
3.20. Arquitectura de RMI (Remote Method Invocation). 45

4.1. Esquema de una metodología iterativa e incremental. 48

5.1. Esquema de la arquitectura de Amaavem. 54
5.2. Diagrama de componentes de Amaavem 56
5.3. Diagrama del archivo de configuración de Amaavem 57

5.4. Diagrama de las interfaces de Ice para Amaavem. 57
5.5. Diagrama de clases de Amaavem. 59
5.6. Diagrama de secuencia general de Amaavem. 60
5.7. Esquema de la base de datos. 63
5.8. Esquema de la selección de objetos en la base de datos. 64
5.9. Diagrama de interacción del monitor de vigilancia. 66
5.10. Diagrama de secuencia del módulo monitor de vigilancia. 70
5.11. Diagrama de los adaptadores de objetos para el monitor de vigilancia. . . . 71
5.12. Diagrama conceptual de las factorías de agentes. 74
5.13. Diagrama del sistema de comunicaciones de Amaavem. 78
5.14. Diagrama de la especialización de comportamientos 84
5.15. Diagrama de secuencia: análisis de los eventos 85
5.16. Ejemplo de un análisis de resultados 90
5.17. Ejemplo de otro análisis de resultados 91

6.1. Ejemplo de caso de estudio. Calle de una universidad de USA. 99
6.2. Diagrama de los agentes para un dominio concreto. 101
6.3. Gráfico de actividad del repositorio de Amaavem. 107
6.4. Despliegue para 3 nodos de la arquitectura Amaavem. 108
6.5. Diagrama de componentes Ice para 3 nodos. 109
6.6. Tiempos de análisis de la arquitectura Amaavem. 110
6.7. Tiempos de comunicación de la arquitectura Amaavem. 111

A.1. Manual usuario: configuración BD 1 126
A.2. Manual usuario: configuración BD 2 126
A.3. Manual usuario: configuración BD 3 127
A.4. Manual usuario: acceso remoto BD 1 127
A.5. Manual usuario: acceso remoto BD 2 128
A.6. Manual usuario: acceso remoto BD 3 128
A.7. Manual usuario: acceso remoto BD 4 129
A.8. Manual usuario: Despliegue Amaavem 1 131
A.9. Manual usuario: Despliegue Amaavem 2 132
A.10. Manual usuario: Despliegue Amaavem 3 132
A.11. Manual usuario: Despliegue Amaavem 4 133
A.12. Manual usuario: Despliegue Amaavem 5 133
A.13. Manual usuario: Despliegue Amaavem 6 134

C.1. Archivo de configuración de IceGrid para las aplicaciones cliente. 147
C.2. Archivo de configuración de IceGrid para el registry (nodo principal). . . . 148

Capítulo 1

Introducción

Actualmente, el terrorismo, los constantes robos, los secuestros, el vandalismo y demás fechorías han sufrido un incremento importante en las grandes ciudades. Por tanto, se hace necesario reforzar las medidas de seguridad para tratar de prevenir dichos actos. Por ello, gobiernos, instituciones y empresas privadas han tomado como medida de seguridad la instalación de cámaras de videovigilancia para intentar reducir el número de actos delictivos.

En las grandes urbes de todo el planeta, cada vez es más frecuente la instalación de cámaras de videovigilancia en zonas muy concurridas. En este punto es donde la vigilancia inteligente irrumpe en la sociedad para tratar de ayudar a conseguir entornos más seguros. Surge la necesidad de crear sistemas de vigilancia inteligente automatizados debido a que un operador humano no puede desempeñar esta tarea de manera óptima.

En los sistemas tradicionales de vigilancia, el ser humano es el responsable de procesar todas las imágenes captadas por el sistema. Dicha tarea agota rápidamente al operador porque una persona no puede fijar su atención en una serie de monitores durante un periodo de tiempo prolongado. La fatiga y el cansancio son estados naturales que sufren los seres humanos y repercuten negativamente en el rendimiento del operador.

Por estos puntos explicados anteriormente, surge la necesidad de crear sistemas de vigilancia inteligente para intentar reducir el índice de delitos perpetrados en entornos públicos así como dar un soporte efectivo a los operadores humanos.

La **Vigilancia Inteligente** [VV05] se puede definir como la aplicación de técnicas, algoritmos y métodos de Inteligencia Artificial con el fin de desarrollar sistemas de seguridad inteligentes, que lleven a cabo las tareas de vigilancia y monitorización realizadas tradicionalmente por operadores humanos. No se trata de sustituir radicalmente al operador humano, sino ofrecerle un soporte para que pueda aumentar su rendimiento en la vigilancia de entornos.

Este proyecto de fin de carrera propone una arquitectura distribuida adaptativa para el diseño, desarrollo, configuración y despliegue de sistemas de vigilancia inteligente en entornos de monitorización.

1.1. ¿Qué es un Sistema de Vigilancia Inteligente?

Un sistema de vigilancia inteligente es un sistema que aplica técnicas, algoritmos y métodos de Inteligencia Artificial cuyo objetivo es la monitorización y análisis inteligente de un entorno. El objetivo principal de la vigilancia inteligente no es captar imágenes de una zona para que un operador las revise después, sino automatizar este proceso tanto como sea posible, ofreciendo un soporte eficaz al operador humano.

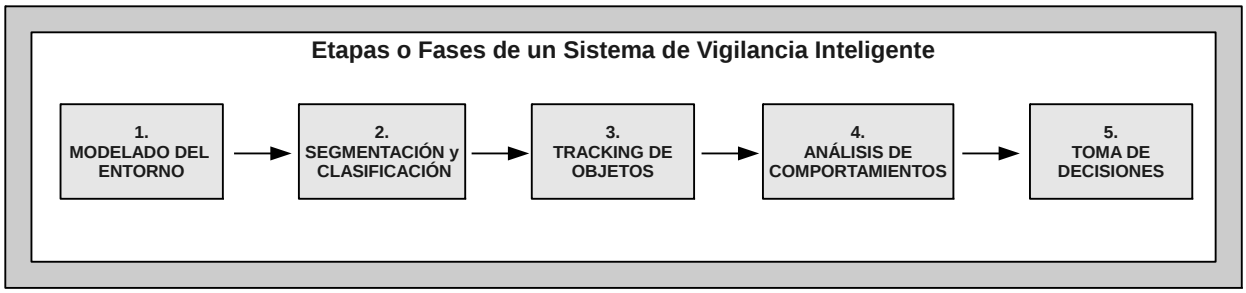


Figura 1.1: Esquema general de un sistema de vigilancia inteligente.

Un sistema de vigilancia inteligente se puede dividir en 5 etapas claramente diferenciadas. La salida de una etapa es la entrada de la siguiente (véase figura 1.1)

1. **Modelado del entorno:** En esta etapa se determina el entorno a monitorizar así como los objetos que van a intervenir en la escena y sus relaciones.

2. **Segmentación y clasificación:** En esta etapa se identifican y clasifican los elementos y las acciones que están realizando en cada momento de la escena.

3. **Tracking:** En esta etapa se realiza el seguimiento del objeto hasta que deja de aparecer en las imágenes. El proceso por el cual se determina la posición del objeto en cada instante de tiempo se denomina *tracking*.

4. **Análisis de Comportamientos:** En esta etapa, una vez recogida la información de las etapas anteriores, se procesa dicha información para poder determinar si el comportamiento del objeto se asemeja con sus patrones de comportamiento que tiene asociado dicho objeto por naturaleza.

5. **Toma de decisiones:** En esta etapa el sistema de vigilancia inteligente debe ser capaz de tomar una serie de decisiones, o dar soporte a la toma de las mismas, a partir de la información obtenida en las etapas anteriores y elaborar un plan de emergencia en caso de que sea necesario. En esta etapa también se debe ofrecer soporte al usuario que esté empleando el sistema.

1.1.1. Evolución e impacto socio-económico

Los sistemas de vigilancia inteligente se pueden encuadrar, desde un punto de vista general, en tres generaciones [VV05] :

- **Primera Generación**.

  En esta primera generación se incluyen los tradiciones sistemas de videovigilancia analógicos de circuito cerrado CCTV(*Close-Circuit Television*). El sistema consigue una baja tasa de fallos y su funcionamiento resultaba adecuado. A pesar de sus limitaciones, son sistemas muy usados debido al bajo que coste que tienen y a la madurez de su tecnología. Entre sus problemas destacan principalmente el uso de técnicas analógicas para el tratamiento, la distribución y el almacenamiento de las imágenes y su alta dependencia de la actividad humana para detectar las irregularidades que suceden en el entorno monitorizado.

- **Segunda Generación**.

  En esta segunda generación aparecen las primeras técnicas de vigilancia automatizada, donde se emplean los algoritmos de visión por computador y los sistemas de la primera generación. Como ventajas se pueden citar el incremento de la eficiencia de los sistemas y la mayor capacidad de observación y análisis de situaciones de manera simultánea. También se reduce la dependencia del ser humano. Sin embargo, esta generación todavía no satisface las necesidades de obtener algoritmos para la visión por computador en tiempo real, conseguir patrones de comportamientos y aprender de las variaciones del entorno.

- **Tercera Generación**.

  En esta tercera generación es donde surgen los sistemas más avanzados de vigilancia, apoyados en sensores heterogéneos distribuidos por todo el entorno a monitorizar. Surgen problemas para distribuir la información y gestionar el espacio de almacenamiento. Para resolver estos problemas, se pueden hacer uso de *middlewares* de comunicaciones para lograr que los sistemas sean altamente distribuidos.

Actualmente, es común el uso de sistema de vigilancia inteligente en entornos públicos de las grandes ciudades. La vigilancia inteligente ayuda a las autoridades del orden a vigilar entornos públicos, como por ejemplo aeropuertos, hoteles, hospitales, estaciones de autobuses, estaciones de tren, carreteras, puertos, etc. Debido a la actual demanda, se genera la necesidad de crear aplicaciones potencialmente adecuadas para cumplir el objetivo principal de la vigilancia inteligente: conseguir una **monitorización inteligente** de un espacio con la mínima intervención del ser humano.

A continuación se van a citar algunos ejemplos de uso de sistemas vigilancia inteligente que impactan directamente en la vida diaria.

La Dirección General de Tráfico en España (**DGT**) ha implantando sistemas de vigilancia en todo el territorio nacional. Como se puede apreciar en este estudio [DGT10], la DGT asegura que en ciudades como Madrid o Barcelona la tasa de accidentes y atropellos se redujo en un 60 % en el año 2010.

En una de las estaciones de Metro más importante de Londres (Liverpool Street) está instalando uno de los proyectos más innovadores en el campo de la Vigilancia Inteligente. En concreto, el sistema PRISMATICA [VBL05], el cual se analiza detalladamente en la sección 3.2.3. Es un sistema que cuenta con millares de cámaras instaladas en todo el recinto capaz de detectar comportamientos anómalos en toda la zona de la estación. Ofreciendo la posibilidad de reproducir cualquier imagen captada por el sistema en cualquier instante y siendo capaz de reconocer equipajes abandonados, entre otras muchas funcionalidades.

Según [NMW04], para apreciar la expansión de la vigilancia inteligente es interesante observar el número de cámaras CCTV desplegadas internacionalmente. En este contexto, el Reino Unido es el gran líder. En 2009, el número aproximado de habitantes del Reino Unido rondó los 62 millones, mientras que en número de cámaras desplegadas en dicho país se estimó en torno a 4 millones, es decir, una cámara de vigilancia por cada 15 personas.

1.1.2. Problemática

El desarrollo de Sistemas de Vigilancia Inteligente es una tarea ardua y compleja debido a la cantidad de problemas que surgen al analizar un entorno en concreto.

Por ejemplo, los sistemas de vigilancia de última generación se enfrentan a multitud de adversidades en cada una de las etapas. Actualmente, muchas empresas optan por sistemas de vigilancia más sencillos debido a la falta de madurez de estos sistemas de tercera generación.

Como se puede apreciar en la figura 1.1, cada salida de una etapa de un sistema de vigilancia inteligente es la entrada de la siguiente etapa. Es decir, los fallos en las primeras etapas se arrastran a las sucesivas etapas, siendo necesario tratar de resolver los problemas tan pronto como sea posible.

Si bien el ámbito de este proyecto se centra en las fases de Análisis de Comportamientos y Toma de Decisiones (véase figura 1.2), **en este trabajo se propone una arquitectura que de soporte al desarrollador final en la creación de un sistema de vigilancia inteligente**. Es decir, con la creación de este proyecto, se pretende crear una arquitectura para que resulte más sencillo crear y desplegar un sistema de vigilancia inteligente en un entorno monitorizado.

En la etapas de segmentación, clasificación y *tracking* surgen problemas provenientes de las infinitas situaciones que pueden darse debido a la heterogeneidad y complejidad de los entornos.

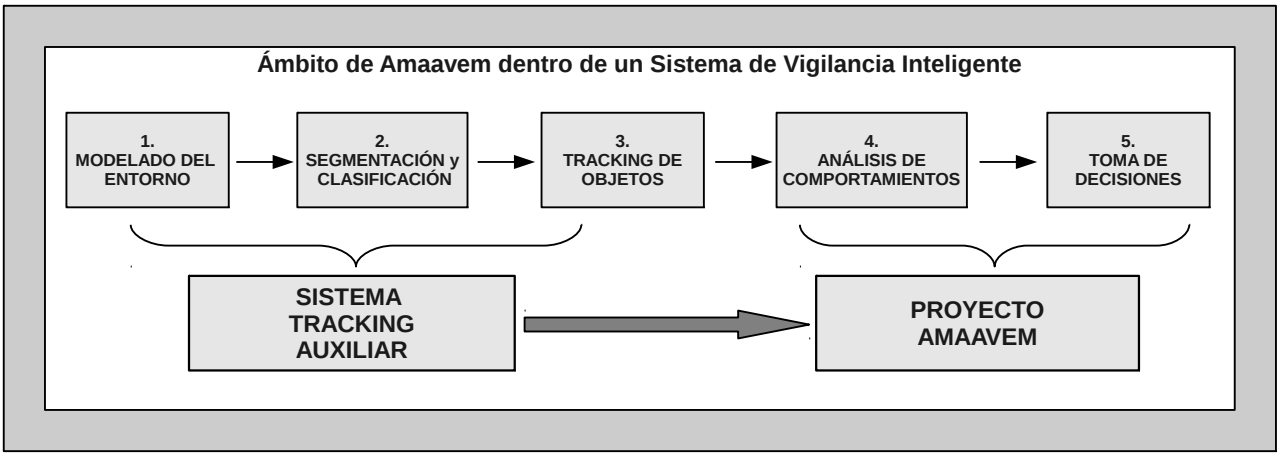


Figura 1.2: Fases que abarca Amaavem dentro de un sistema de vigilancia inteligente

Es interesante tratar y resolver estos errores de la mejor manera posible, ya que repercute directamente en los resultados finales.

En general, los principales problemas que han de ser tratados en el presente proyecto fin de carrera se muestran a continuación:

- **Problemas de adaptabilidad**. Algunas arquitecturas se centran en analizar un concepto concreto (velocidad, tamaño, trayectoria,etc) y no dan la posibilidad para que dichas arquitecturas puedan ser ampliadas de una manera sencilla. Es interesante proponer una arquitectura que facilite la inclusión de nuevos conceptos y funcionalidades al sistema.
- **Problemas de escalabilidad**. Algunas arquitecturas estudiadas no ofrecen la posibilidad de dar soporte a muchos usuarios o su rendimiento empeora considerablemente cuando se tiene que analizar multitud de objetos simultáneamente. Se pretende que el sistema sea escalable en tanto como sea posible ofreciendo una arquitectura distribuida para lograr que la escalabilidad no sea un problema.
- **Soporte para fuentes de información heterogéneas**. Los sistemas de vigilancia inteligente deben de dar soporte a multitud de fuentes de información tales como: vídeos captados por cámaras, información auditiva captada por micrófonos o información proveniente de diversos sensores (de temperatura, de presión, de potencia, etc)
- **Problemas con la diversidad de entornos**. En la actualidad, existen una gran variedad de entornos para monitorizar (un aeropuerto, una estación de tren, un colegio, una tienda de muebles,etc). Muchos sistemas similares centran su diseño en un entorno en concreto, reduciendo así la flexibilidad. Sería útil que el sistema pueda utilizarse en multitud de entornos de manera sencilla.

- **Problemas de Robustez**. Algunas arquitecturas estudiadas no ofrecen la posibilidad de recuperación de fallos en caso de que, por ejemplo, algún artefacto hardware falle. Se pretende crear una arquitectura tolerante a fallos capaz de seguir trabajando en caso de fallos no previstos inicialmente.
- **Problemas de Modularidad**. Algunas arquitecturas estudiadas no poseen una división de su arquitectura en componentes, las cuales sufren de un alto acoplamiento y una baja cohesión. Para tratar que el sistema no sufra de acoplamiento y los componentes tengan alta cohesión, se pretende que la arquitectura resuelva este problema ofreciendo una serie de servicios, cada uno de los cuales serán implementados por los diferentes módulos del sistema.
- **Despliegue sobre múltiples plataformas**. Algunas arquitecturas no resuelven el problema de la multiplaforma y su diseño está ligado a un sistema operativo. Por tanto, se pretende crear una arquitectura que pueda ejecutarse en al menos *Microsoft Windows* y *GNU/Linux*.

El **objetivo principal** de Amaavem consistirá en la creación de una **Arquitectura Multi-Agente Adaptativa** para vigilar entornos monitorizados ofreciendo el soporte al diseño, desarrollo, configuración y despliegue de sistemas de vigilancia inteligente basado en la aplicación de una serie de restricciones y análisis de normalidad de las condiciones del entorno, determinando si se producen hechos fuera de lo normal.

Se usa una arquitectura multi-agente debido a que la tecnología de agentes se adapta bien a problemas complejos, abiertos y con fuentes de información de naturaleza heterogénea, como los planteados en entornos de monitorización.

Para conseguir el análisis de normalidad de las condiciones del entorno, se determinarán una serie de comportamientos asociados a la naturaleza de cada objeto. Entonces, para poder decidir si lo que está ocurriendo se encuentra en un grado de normalidad aceptable, los agentes analizarán dichos comportamientos y decidirán en cuánto difieren de lo acontecido.

En la sección 3 de este documento se explican los objetivos del proyecto más detalladamente.

1.2. Estructura del Documento

Este documento ha sido redactado siguiendo la normativa de Proyecto de Fin de Carrera de la Escuela Superior de Informática de la Universidad de Castilla-La Mancha.

Dicho documento está compuesto por los siguientes capítulos:

Capítulo 1: Introducción

En este capítulo se ha realizado una breve introducción del área en la que se enmarca el presente Proyecto Fin de Carrera.

Capítulo 2: Objetivos

En este capítulo se desglosa y se describen la lista de objetivos planteados para este Proyecto Final de Carrera.

Capítulo 3: Antecedentes

En este capítulo se realiza un estudio de las áreas de conocimiento necesarias para desarrollar este proyecto. Se explican las bases y el funcionamiento de un Sistema de Vigilancia Inteligente, un estudio detallado de algunas de las arquitecturas propuestas por sistemas similares existentes en el ámbito de la vigilancia inteligente y un análisis de los *middlewares* de comunicaciones más usados actualmente.

Capítulo 4: Metodología de Desarrollo y Herramientas

En este capítulo se recoge la metodología utilizada para desarrollar este proyecto, así como las herramientas necesarias para poder implementarlo.

Capítulo 5: Arquitectura de Amaavem

En este capítulo se describe el diseño e implementación del sistema, detallando los inconvenientes surgidos y las soluciones aportadas. En este capítulo se realiza una clasificación de los módulos y submódulos que componen Amaavem, describiendo cada uno de ellos desde un enfoque funcional hasta llegar a un nivel de detalle técnico.

Capítulo 6: Evolución, Resultados y Costes

En este capítulo se describe una evolución del sistema durante su periodo de desarrollo. Se detallan las iteraciones, los problemas y los resultados que se han producido durante el desarrollo de este proyecto.

Capítulo 7: Conclusiones y Propuestas futuras

En este capítulo se realiza un resumen del desarrollo y las metas alcanzadas. También se enumeran una serie posibles líneas de trabajo futuras para poder continuar con el desarrollo realizado en este proyecto.

Anexos

En esta sección se recogen anexos importantes para que el lector pueda entender cómo ha sido desarrollado este proyecto. El desarrollador final debe entender y aplicar estos anexos, ya que Amaavem es una arquitectura para crear sistemas de vigilancia inteligente y es necesario ofrecer un manual de usuario detallado (Anexo A) y una guía de desarrollo (Anexo B).

Capítulo 2

Objetivos

2.1. Objetivo general

El **objetivo principal** de Amaavem es el desarrollo de una **Arquitectura Multi-Agente Adaptativa** para vigilar entornos monitorizados ofreciendo el soporte al diseño, desarrollo, configuración y despliegue de sistemas de vigilancia inteligente basado en la aplicación de una serie de restricciones de una serie de restricciones y análisis de normalidad de las condiciones del entorno.

Se ha optado por usar una arquitectura basada en la tecnología de agentes ya que es una solución adecuada para resolver problemas complejos, como son los planteados en el ámbito de la vigilancia inteligente. Los entornos para monitorizar son entornos complejos con fuentes de información heterogéneas. Por ello, el uso de agentes inteligentes se convierte en una buena solución para poder monitorizar dichos entornos procurando que se necesite el mínimo soporte del operador humano.

Para conseguir el análisis de normalidad de las condiciones del entorno, se determinarán una serie de comportamientos asociados a la naturaleza de cada objeto. Entonces, para poder decidir si lo que está ocurriendo se encuentra en un grado de normalidad aceptable, los agentes analizarán dichos comportamientos y decidirán en cuánto difieren de lo acontecido.

2.2. Objetivos específicos

A partir de este objetivo principal se definen una serie de subobjetivos funcionales específicos que detallan el alcance de Amaavem:

- **Independencia del dominio**. Diseño e implementación de una arquitectura multiagente para el despliegue de sistemas independientes del dominio a monitorizar y de los aspectos o eventos de interés a monitorizar.
- **Escalabilidad**. Integración de soluciones escalables a nivel de comunicación y despliegue de agentes. Se pretende conseguir que el sistema sea cómodamente escalable para tener la posibilidad de añadir nuevas funcionalidades sin que suponga un cambio en la arquitectura.

- **Adaptabilidad**. Diseño de una arquitectura adaptativa basada en el uso de patrones de diseño y que pueda ser fácilmente extensible.
- **Modularidad**. Implementación de una serie de servicios, cada uno de los cuales serán implementados por los diferentes módulos que componen la arquitectura. Cada módulo será lo más independiente posible y tendrá unas responsabilidades bien acotadas para que el sistema gane en cohesión y no sufra de acoplamiento. Se pretende también que sea una tarea sencilla añadir nuevos módulos para que la arquitectura sea flexible y adaptable.
- **Robustez**. Diseño de una arquitectura tolerante a fallos capaz de seguir trabajando en caso de fallos en el hardware o software de soporte.
- **Balanceo de carga**. Estudio de técnicas y métodos, e implementación de un sistema que permita distribuir el trabajo en varios equipos en caso de que el tiempo de cómputo sea muy elevado.
- **Portabilidad**. El desarrollo de Amaavem se ha realizado siguiendo estándares y tecnologías libres, multiplataforma y consolidados, con el objetivo de que pueda ser portado al mayor número de plataformas posibles. Amaavem podrá ser ejecutado al menos en dos plataformas: *GNU/Linux* y *Microsoft Windows*.
- **Facilidad de uso**. Una premisa importante en el desarrollo de Amaavem fue la creación de una arquitectura orientada a desarrolladores con nivel medio en Informática. La creación de un manual de usuario que resuelva dudas de uso del sistema, así como ofrecer la mayor información en este documento son objetivos primordiales.

Para evaluar la arquitectura planteada, se pretende realizar un despliegue de un sistema de monitorización en un entorno concreto. El caso de estudio se realizará en un entorno urbano común. Se implementarán una serie de comportamientos tanto para personas como vehículos y extraer la información relevante del análisis de dichos comportamientos.

Capítulo 3

Antecedentes

El campo de la Vigilancia Inteligente es un área que necesita un apoyo sólido en otros conocimientos de la Informática. En concreto, para este proyecto, se ha necesitado un conocimiento previo de la Inteligencia Artificial, concretamente de los sistemas multi-agentes, las características de los sistemas de vigilancia más relevantes existentes y los *middlewares* de comunicaciones.

El objetivo primordial de este capítulo es poner de manifiesto los conceptos previos que se adquirieron para poder desarrollar este proyecto, así como la exposición y explicación de algunas arquitecturas y trabajos similares al proyecto Amaavem.

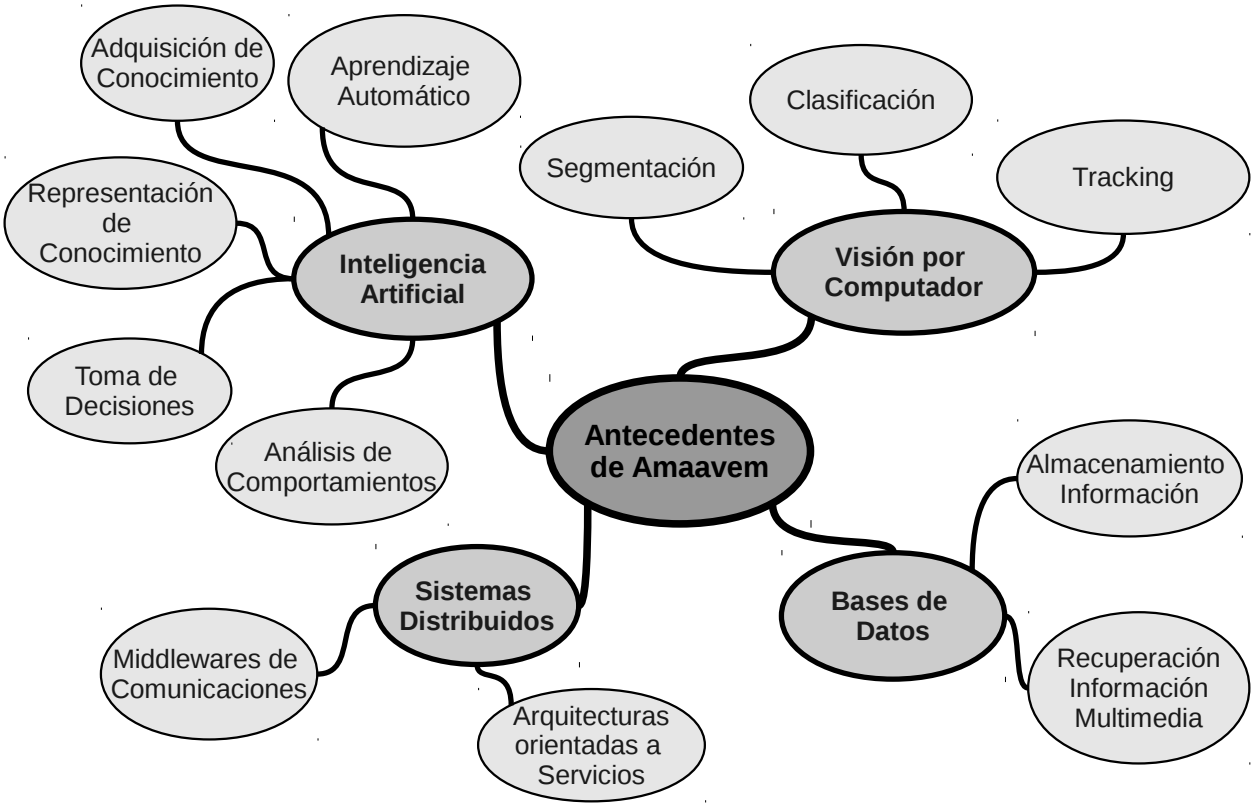


Figura 3.1: Mapa conceptual de conocimiento relacionado con la vigilancia inteligente.

En la figura 3.1, se puede apreciar un mapa conceptual de conocimiento, donde se observa los campos y conceptos necesarios para realizar una aplicación de vigilancia inteligente.

3.1. Inteligencia Artificial

3.1.1. Introducción

Durante miles de años, el ser humano ha tratado de entender cómo piensa; es decir, comprender cómo una persona puede entender, percibir y manipular un mundo más grande y complicado que ella misma. En cambio, la **Inteligencia Artificial (IA)** va más allá, no sólo pretende comprender, sino que también se esfuerza en construir entes inteligentes. La IA es una de las ciencias más recientes. El trabajo comenzó poco después de la Segunda Guerra Mundial.

LA IA abarca en la actualidad una gran variedad de sub-áreas, que van desde áreas de propósito general, como el aprendizaje y la percepción, a otras mucho más específicas, como puede ser la definición de un agente inteligente encargado de recomendarte libros en función de los gustos de una persona. La IA sintetiza y automatiza tareas intelectuales y es, por lo tanto, potencialmente relevante para cualquier ámbito de la actividad intelectual humana. Se puede decir que la IA es un campo universal.

Es complicado definir la IA en una sóla frase; por ello, se dispone una lista de definiciones aceptadas de la IA a lo largo del tiempo:

- «El nuevo y excitantes esfuerzo de hacer que los computadores piensen...máquinas con mentes, en el más amplio sentido literal.»(Haugeland,1985)
- «El estudio de las facultades mentales mediante el uso de moldes computacionales. »(Charniak y McDermott, 1985)
- «El estudio de los cálculos que hacen posible percibir, razonar y actuar. »(Winston, 1992)
- «El estudio de cómo lograr que los computadores realicen tareas que, por el momento, los humanos hacen mejor.»(Rich y Knight, 1991)
- «IA...está relacionada con conductas inteligentes en artefactos. »(Nilsson, 1998)

3.1.2. Agentes Inteligentes

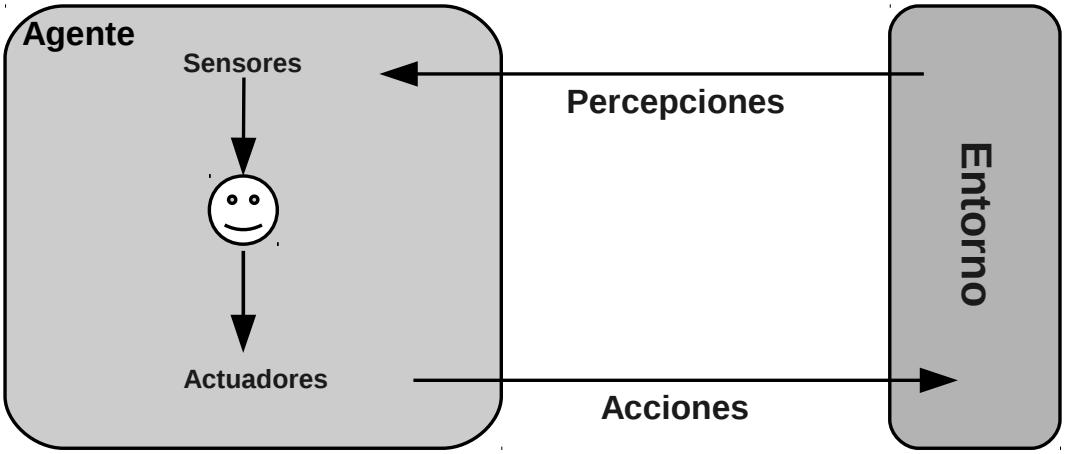


Figura 3.2: Interacción Agente-Entorno

Un **agente** es cualquier cosa capaz de percibir su entorno con la ayuda de sensores y actuar en ese medio utilizando actuadores [RN04]. Esta idea se ilustra en la figura 3.2

Un agente humano tiene ojos, oídos y otros órganos que actúan de sensores así como piernas, boca, brazos y demás partes para actuar. Un agente robot recibe pulsaciones del teclado, archivos o paquetes a modo de sensores y actúa sobre el medio escribiendo en ficheros, en el monitor, enviando paquetes por la red,etc.

El término percepción se usa en este contexto para indicar que el agente puede recibir información en cualquier instante. Un agente toma una decisión en función a una serie de percepciones a lo largo de un tiempo. Si se definen una serie de comportamientos en función las percepciones, se puede decir que el comportamiento de un agente viene dado por la función del agente, que actúa en consecuencia de una percepción.

A modo de ejemplo sencillo, cuando un termostato recibe como percepción que una habitación está a una temperatura menor de 19 grados, éste enciende la calefacción. Para percibir este hecho del entorno donde esté desplegado el termostato, es necesario un sensor de temperatura. El actuador será el encargado de activar la calefacción gracias a un circuito externo que pueda realizar dicha acción.

Estas funciones pueden ser escritas en una pequeña tabla para ser implementadas por el agente. Siguiendo con el ejemplo de los termostatos, veamos una tabla que simplifica el programa de un agente.

Temperatura <19º	Encender Calefacción
Temperatura >24º	Apagar Calefacción

El **agente racional** es aquel que hace lo correcto. Lo correcto es aquello que permite al agente obtener un resultado mejor que un comportamiento incorrecto.

Para poder medir el rendimiento de un agente se necesitan diseñar una medidas de rendimientos objetivas con el medio que se esté tratando. Normalmente estas medidas las determinará el diseñador encargado de la construcción del agente.

La racionalidad en un instante determinado va a depender de cuatro factores:

- La medida de rendimiento que define el criterio de éxito.
- El conocimiento del medio en el que habita el agente.
- Las acciones que el agente puede realizar.
- Las percepciones que el agente ha captado hasta ese momento.

  Por tanto, se puede definir a un **agente racional** de la siguiente manera [RN04]: «*En cada posible secuencia de percepciones, un agente racional deberá emprender aquella acción que supuestamente maximice su medida de rendimiento, basándose en las evidencias aportadas por la secuencia de percepciones y en el conocimiento que el agente mantiene almacenado.* »

Una vez definido el concepto de racionalidad, otro punto importante a tratar cuando se quiere desarrollar un agente inteligente es el **entorno de trabajo** de dicho agente. Para poder definir un agente se necesita una media de rendimiento, un entorno, actuadores y sensores. Este termino también es conocido como REAS (Rendimiento,Entorno,Actuadores,Sensores).

A modo de ejemplo, supongamos el caso de un tutor interactivo que se dedique a dar clases de apoyo de Matemáticas a alumnos de Bachillerato.

- Tipo de Agente : Profesor de Matemáticas interactivo.

- Medida de Rendimiento : Maximizar la puntuación de los alumnos.

- Entorno : Estudiantes, agencia examinadora.

- Sensores : Teclado de entrada.

- Actuadores : Visualizar los ejercicios,sugerencias y correcciones.

Los entornos de trabajo pueden clasificarse para que el agente pueda adaptarse en función de las características del propio entorno. Russell y Norvig proponen la siguiente clasificación en función de las propiedades del entorno [RN04]:

- **Totalmente observable** o **parcialmente observable**. En función de si el agente es capaz de acceder a toda la información del medio o solamente a una parte.
- **Deterministas** o **estocástico**. Un entorno determinista es aquél en el cual cada acción tiene ligada un efecto. No existe una incertidumbre sobre el estado que resultará después de aplicar la acción. En caso contrario, el entorno es estocástico.
- **Episódico** o **secuencial**. En un entorno episódico, la experiencia del agente se divide en episodios automáticos. Cada episodio consiste en la percepción del agente y la

realización de una única acción posterior. El siguiente episodio no depende de las acciones que se realizaron en episodios anteriores. En entornos secuenciales, la decisión presente puede afectar a decisiones futuras.

- **Estáticos** o **dinámicos**. Un entorno estático no tiene cambios hasta que el agente desarrolla una acción. Por otro lado, uno dinámico puede sufrir cambios como consecuencia de las acciones desarrolladas por otros elementos del entorno.
- **Discretos** o **continuos**. La distinción entre discreto y continuo puede aplicarse al estado del medio, a la forma en la que se maneja el tiempo y a las percepciones y acciones del agente. Por tanto, un entorno es discreto si existe un número finito de acciones y percepciones en el mismo.

Un agente puede describirse como:

$Agente = Arquitectura + Programa$

Siendo la arquitectura el conjunto de sensores y actuadores, y el programa es el programa del agente que implementa la función de dicho agente. Obviamente, el programa tiene que ser apropiado respecto a la arquitectura. Es decir, si el programa tiene que implementar acciones como *Golpear*, la arquitectura debe proveer de un brazo articulado.

Los programas de los agentes tienen la misma estructura: reciben las percepciones actuales como entradas de los sensores y devuelven una acción a los actuadores. Los programas de los agentes sólo reciben la percepción actual como entrada porque no hay nada más disponible en el entorno; si las acciones del agente dependen de la secuencia completa de percepciones, el agente tendría que recordar las percepciones.

La arquitectura abstracta de un agente se puede formalizar teniendo en cuenta las características del entorno y la interacción del agente con el mismo [Woo01].

Un entorno puede definirse como un conjunto de estados

$$S = \{s_1, s_2 ... s_n\} \tag{3.1}$$

Las interacciones de un agente con un entorno pueden definirse como un conjunto de acciones

$$A = \{a_1, a_2 ... a_n\} \tag{3.2}$$

Entonces, un agente puede definirse como la siguiente función:

$$Agente : S* \longrightarrow A \tag{3.3}$$

Esto significa que un agente realiza una acción en función del estado del entorno. Esto puede simplificar matemáticamente qué significa un agente pero esto es una aproximación de un agente estándar y existen agentes más complejos.

A continuación, se van a estudiar los distintos tipos de agentes en función de su programa:

* Agentes reactivos simples.
* Agentes con estado.
* Agentes basados en objetivos.
* Agentes basados en utilidad.
* Agentes que aprenden.

Agentes Reactivos Simples.

Es el tipo de agente más sencillo ya que seleccionan las acciones sobre la base de las percepciones actuales, ignorando el resto de percepciones anteriores.

Siguiendo con el ejemplo anterior del termostato, se puede afirmar que un termostato en un agente reactivo simple ya que el agente simplemente comprueba la temperatura actual, y en caso de que sobrepase o no llegue a una temperatura T, realiza una acción.

Estos agentes pueden definirse con un conjunto de reglas condición - acción. En la figura 3.3 se puede apreciar un esquema de un agente reactivo simple.

si *CONDICIÓN* **entonces** *ACCIÓN*.

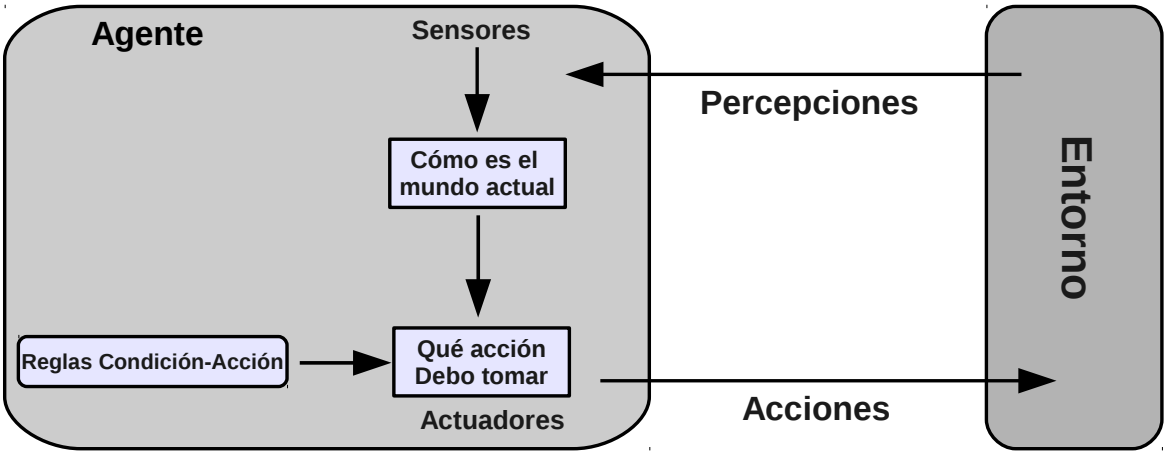


Figura 3.3: Diagrama de un agente reactivo

Agentes Con Estado.

Hasta ahora la decisión de agente sólo se realiza en función a una única percepción. Los agentes con estado son capaces de tomar una decisión en función a un histórico de percepciones que el agente ha ido almacenando.

Este tipo de agente mantiene una estructura de datos interna, empleada para registrar información del entorno. Cuando el agente decida la acción que debe realizar, tendrá que valorar la secuencia de estados almacenados, ver cómo evoluciona el mundo en función de la secuencia de estados y conocer también qué efectos pueden ocasionar sus acciones.

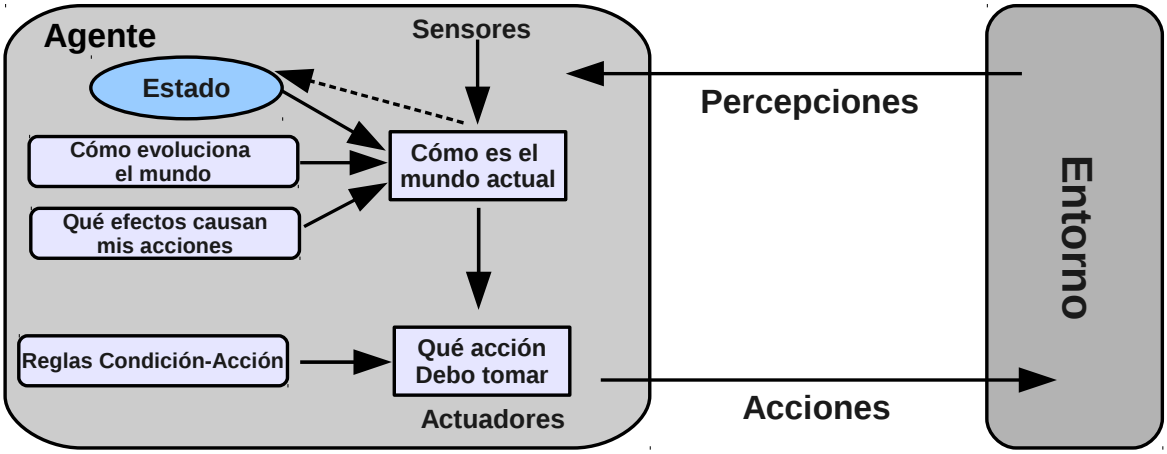


Figura 3.4: Diagrama de un agente con estado

En la figura 3.4 se muestra un diagrama de los agentes con estados.

Agentes basado en Objetivos.

El conocimiento sobre el estado actual del mundo no es siempre suficiente para decidir qué hacer. Además de la descripción del estado actual, el agente necesita algún tipo de información sobre su objetivo o meta que describa las situaciones que se deben alcanzar. El programa del agente puede combinarse con información sobre los resultados de las acciones posibles para elegir las acciones que permitan alcanzar el objetivo.

En conclusión, un agente basado en objetivos almacena información del estado del mundo así como un conjunto de objetivos que intenta alcanzar, y que es capaz de seleccionar la acción que eventualmente lo guiará hacia la consecución de sus objetivos.

En la figura 3.5 se muestra un diagrama de los agentes basados en objetivos.

Agentes basado en Utilidad.

Los objetivos por sí sólos a veces no son suficientes para generar comportamientos óptimos en la mayoría de entornos. El agente debe ser capaz de decidir si se encuentra más feliz en un estado A o en un estado B. Para poder realizar esta decisión se hace necesaria una

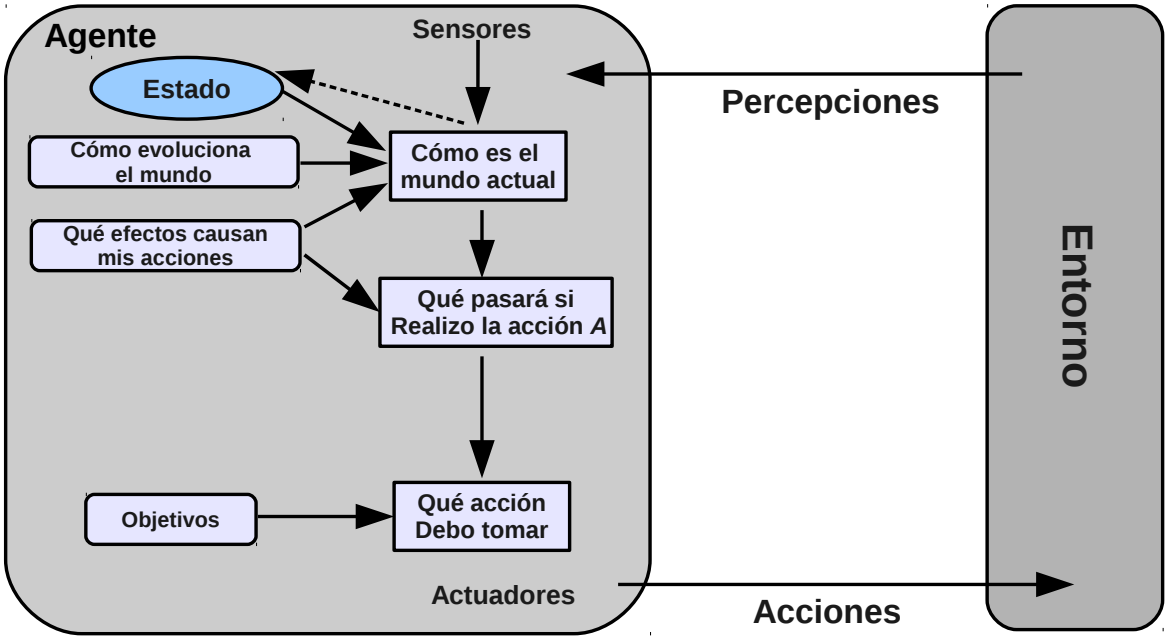


Figura 3.5: Diagrama de un agente basado en objetivos

función de utilidad, la cual proyecta un estado en un número real, que representa un nivel de felicidad.

Entonces, un agente de este tipo utiliza un modelo del mundo, junto con una función de utilidad que calcula sus preferencias entre los distintos estados del mundo. Después, selecciona la acción que le lleva a alcanzar la mayor utilidad esperada, que se calcula haciendo la media de todos los estados resultantes posibles, ponderado con la probabilidad del resultado. En la figura 3.6 se muestra un diagrama de los agentes basados en utilidad.

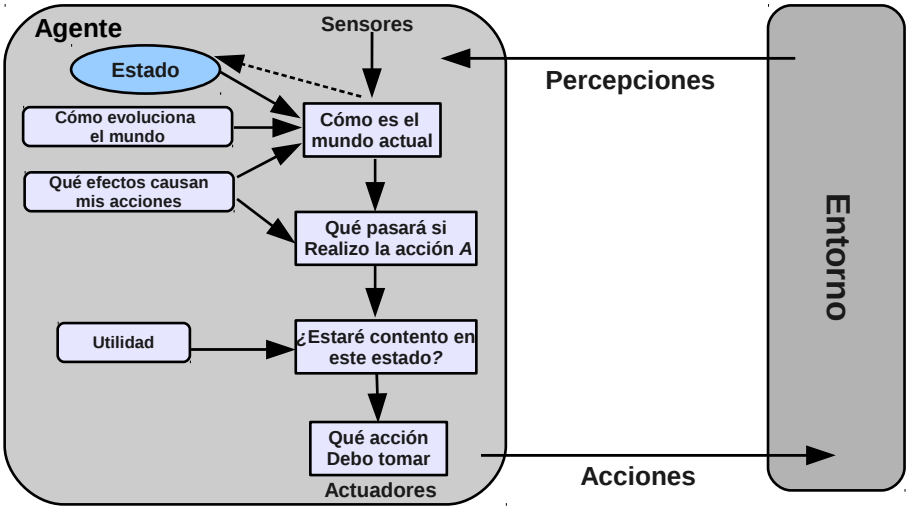


Figura 3.6: Diagrama de un agente basado en utilidad

Agentes que aprenden.

Todos los agentes descritos anteriormente pueden mejorar su eficiencia con la ayuda de algún mecanismo de aprendizaje.

Como se aprecia en la figura 3.7, un agente que aprende tiene cuatro módulos diferenciables y que se listan a continuación:

- Críticas.
- Elemento de Aprendizaje.
- Generador de Problemas.
- Elemento de Actuación.

El elemento de aprendizaje es el encargado de hacer mejoras en el agente. Se realimenta con críticas sobre la actuación del agente y determina cómo se debe modificar el elemento de actuación para proporcionar resultados más óptimos. El elemento de actuación es el encargado de la selección de acciones externas. Se podría decir que el elemento de actuación es lo que anteriormente se trató como el agente completo.

La crítica indica al elemento de aprendizaje qué tal lo está haciendo el agente con respecto a un nivel de actuación fijo. Es necesaria porque las percepciones por sí mismas no prevén un acierto del agente.

El generador de problemas es el encargado de sugerir acciones que lo guiarán hacia experiencias nuevas e informativas. Si el agente está dispuesto a explorar un poco y llevar a cabo algunas acciones que a corto plazo seguramente no sean óptimas, puede realizar acciones con mejoras a largo plazo.

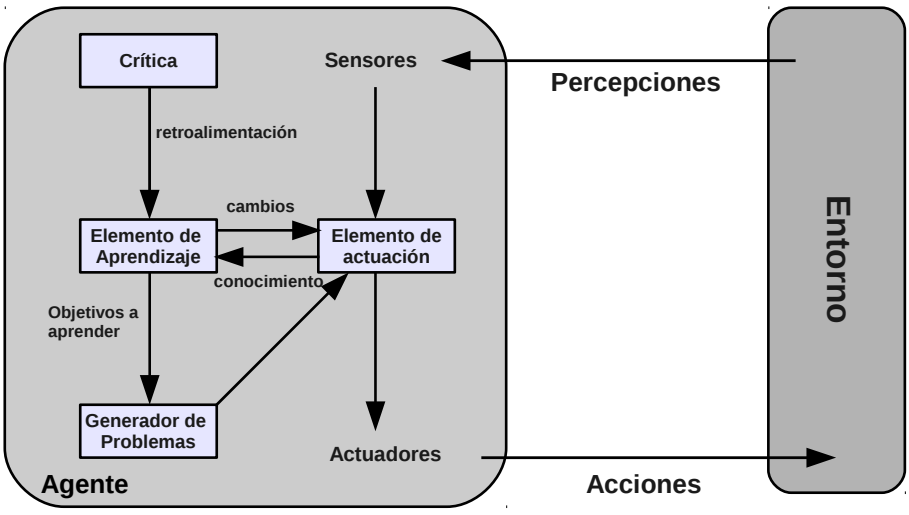


Figura 3.7: Diagrama de un agente que aprende

3.1.3. Sistemas Multi-Agente

La Inteligencia Artificial Distribuida es un campo de la Inteligencia Artificial dedicada al estudio de las técnicas y el conocimiento necesario para la coordinación y distribución del conocimiento y las acciones en un entorno con múltiples agentes.

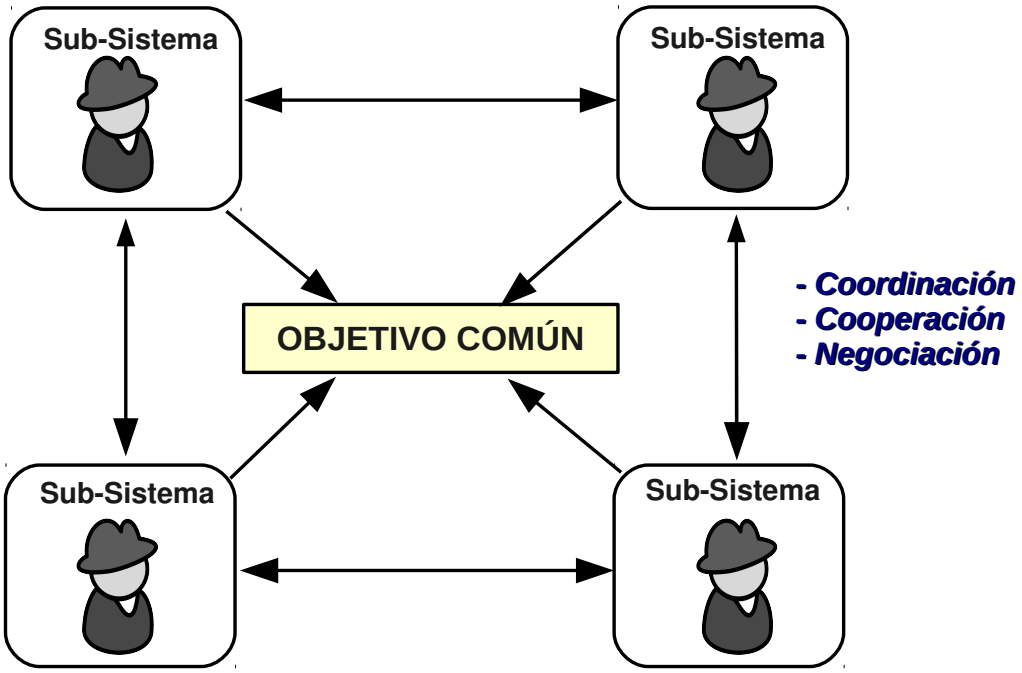


Figura 3.8: Sistema Multi-Agente

El hecho de que cada vez las aplicaciones sean más complejas y la necesidad de que los distintos sub-sistemas que componen un sistema complejo deban comunicarse entre ellos, es recomendable el uso de **Sistemas Multi-Agentes**.

La tecnología de agentes se adapta bien a problemas complejos, abiertos y con fuentes de información de naturaleza heterogénea, como pueden ser los expuestos en entornos de monitorización.

Un sistema multi-agente es un sistema compuesto por múltiples agentes inteligentes (véase la figura 3.8 que interactúan entre ellos. Los sistemas multi-agente pueden ser utilizados para resolver problemas que son difíciles o imposibles de resolver para un agente individual o un sistema monolítico [Woo01].

Cada sub-sistema tiene capacidad para decidir, por tanto, la dificultad surge en la coordinación entre los sub-sistemas. Para conseguir una gestión inteligente de las aplicaciones se necesita definir una serie de **políticas de cooperación, coordinación y negociación** entre los múltiples agentes del sistema.

Todos los sub-sistemas deben poseer las siguientes características:

- El sub-sistema debe estar por un conjunto de agentes, cada uno de los cuales mantiene sus propias habilidades.
- Los sub-sistemas tienen un mismo objetivo común.
- Los agentes del sub-sistema tiene un conocimiento limitado pero posee una cierta especialización para realizar alguna tarea específica. Dicha especialización está determinada por lo que el agente conoce y sus habilidades.

Algunas de las características que debe cumplir cada agente que forme un sub-sistema son las siguientes [Woo01]:

- **Autonomía**. Un agente puede operar sin la intervención directa de otros agentes, teniendo un control sobre sus acciones y su estado interno.
- **Sociabilidad**. Un agente debe ser capaz de interactuar con otros agentes, a través de un lenguaje de comunicación entre agentes, para resolver sus problemas y ayudar a la resolución de los problemas de otros agentes.
- **Reactividad**. Un agente debe ser capaz de percibir estímulos de su entorno y de actuar ante ellos.
- **Proactividad**. Un agente, además de reaccionar ante los estímulos de su entorno, debe mostrar iniciativa para cumplir sus propios objetivos.
- **Adaptabilidad**. Un agente puede cambiar su comportamiento en respuesta a cambios en las condiciones del entorno, aumentando su conocimiento sobre el problema.
- **Movilidad**. Un agente puede cambiar su situación física para mejorar en la resolución del problema.
- **Veracidad**. Un agente no comunicará información falsa a propósito.
- **Racionalidad**. Un agente tratará de cumplir sus objetivos siempre que pueda y no exista una causa importante por la que dejar de hacerlo.

Por otro lado, la distribución de las decisiones entre los distintos sub-sistemas permite lo siguiente:

- La autonomía de los agentes se mantiene.
- La información del sistema no se encuentra centralizada en un único nodo. Con estos sistemas, la información queda distribuida entre los distintos nodos.
- La toma de decisiones ya no es centralizada. Ahora cada agente toma decisiones locales y se coordina con los demás agentes para llegar al objetivo común.
- Adaptación al entorno en tiempo real.
- Mayor capacidad de procesamiento de eventos en un instante de tiempo.

La **comunicación** entre agentes es un punto fundamental a tratar, debido a que los agentes cooperan entre ellos para alcanzar un objetivo común. Por ejemplo, un caso de comunicación entre agentes podría ser que un agente solicite a otro agente la realización de alguna acción que el primer agente no puede realizar. El segundo agente tendrá la autonomía de aceptar o rechazar dicha acción.

La **coordinación** es una propiedad de los sistemas multi-agentes que posibilita la realización de una acción en un entorno compartido. El grado de coordinación entre los distintos sub-sistemas determinará hasta qué punto un sistema es capaz de evitar realizar acciones no necesarias así como no gastar recursos sin sentido y mantener unas condiciones de seguridad. Una mala coordinación entre los agentes puede ocasionar resultados indeseados.

La **cooperación** surge como la interacción entre agentes que busquen un objetivo común. La negociación se refiere a la coordinación entre agentes competitivos o simplemente interesados en su propio beneficio. Para llevar a cabo una cooperación exitosa, es necesario que los agentes mantenga un modelo interno del resto de agentes.

Todo ser inteligente para poder comunicarse con otro de su mismo ser necesita utilizar un mismo lenguaje y un canal de comunicación. Según [Wei99], existen tres aspectos básicos para estudiar de una manera formal la comunicación entre agentes:

1. **Sintaxis**, que especifica cómo se estructuran los símbolos de la comunicación.
2. **Semántica**, que define qué significa cada símbolo.
3. **Pragmática**, que se refiere a cómo se interpretan los símbolos.

Un acto comunicativo debe transmitirse desde el hablante (*activo*) al oyente (*pasivo*) en forma de sentencia (*afirmaciones o preguntas*).

Métodos de interacción entre agentes.

Los métodos o protocolos de interacción especifican cómo se realiza el intercambios de mensajes, es decir, definen las reglas de una conversación entre agentes.

En sistemas multi-agentes se puedan dar dos situaciones; situaciones de *cooperación*, donde los agentes deben ser capaces de combinar sus habilidades para poder lograr un objetivo común. El objetivo de los protocolos de interacción para estas situaciones es mantener un rendimiento coherente y no perjudicar la autonomía de los agentes.

Por otro lado, pueden darse situaciones de *competición*, en el caso de que los agentes sólo estén interesados en su propio objetivo o que sus objetivos estén en conflicto con otros objetivos de otros agentes. Para este caso, el objetivo de los protocolos de interacción es maximizar la utilidad de los agentes. En este tipo de sistemas juegan un papel considerable las estrategias de negociación que permiten resolver conflictos, asignar tareas y tomar decisiones.

Es muy común que los agentes compitan por recursos cuando éstos son limitados. Los agentes han de coordinarse para alcanzar sus propios objetivos u objetivos comunes porque existe una limitación de recursos, lo que implica que es necesario satisfacer una serie de restricciones globales en el sistema.

La estrategia básica de muchos protocolos de interacción actuales se basan en descomponer y distribuir tareas, siguiendo el modelo basado en *divide y vencerás*. Esta estrategia puede reducir la complejidad de una tarea dividiéndola en sub-tareas que requieran agentes de menor capacidad y que consuman menos recursos. Será el propio sistema el encargado de realizar la descomposición teniendo en cuenta las capacidades de cada agentes así como los recursos del sistema.

Según las tareas deben distribuirse en función a una serie de criterios generales [DLC87]:

- Las tareas deben ser asignadas a agentes que sean capaces de completarlas.
- La sobrecarga de recursos cruciales debe ser evitada tanto como sea posible.
- Un agente que posee una perspectiva global del sistema puede asignar tareas a otros agentes.
- Las responsabilidades solapadas pueden ser asignadas a varios agentes para lograr cierto grado de coherencia.
- Las tareas que sean dependientes deben ser asignadas a agentes próximos (en espacio o en semántica) para intentar minimizar los costes de comunicación y sincronización.
- La reasignación de tareas urgentes debe ser permitida para completar dichas tareas lo más rápido posible.

Métodos de comunicación entre Agentes.

Los métodos de comunicación se pueden diferenciar en dos grupos.

1. **Sistemas de pizarra.**

   Las pizarras son comúnmente usadas en sistemas inteligentes [Cor03]. En sistemas con multiplicidad de fuentes de conocimiento se hace necesario algún mecanismo de comunicación. Este tipo de sistemas se basa en la colaboración de un grupo de agentes expertos en distintas tareas apoyándose en un repositorio común de conocimiento. Véase la figura 3.9.

   La pizarra es una estructura de datos usada como mecanismo general de comunicación entre las diferentes fuentes de conocimiento, comúnmente gestionada por un controlador. Al igual que las pizarras de un aula de clase, son usadas para compartir conocimiento entre un grupo de seres inteligentes.

   La pizarra sirve para que cada agente vaya anotando sus soluciones parciales y utilizar el conocimiento que ya haya escrito en dicha pizarra para ampliar su conocimiento

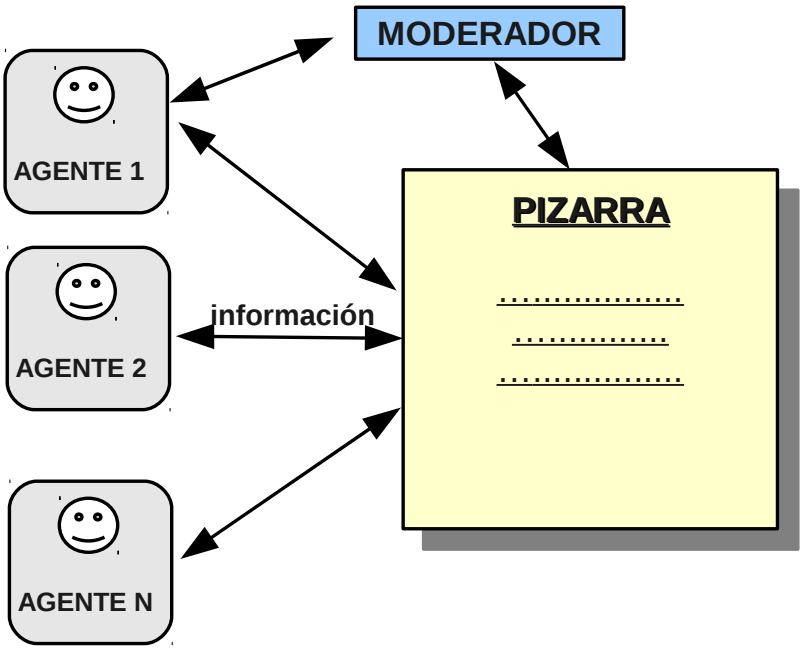


Figura 3.9: Métodos de Comunicación entre agentes: pizarra

local. Los especialistas empiezan a observar la información de la pizarra con el fin de llegar a una solución final.

$$SolucionFinal = \sum SolucionesParciales$$

Según [Wei99], un sistema de pizarra debe poseer las siguientes características:

- **Cada agente trabaja con su parte del problema**. No hay comunicación directa entre agentes y por ello el agente resuelve de forma autónoma su subproblema y bajo su responsabilidad.
- **Lenguaje común para interactuar**. El resto de agentes deben de ser capaces de leer y entender los datos incluidos en el sistema de pizarra.
- **Multiplicidad en las técnicas de solución**. La representación interna y el mecanismo de extraer el conocimiento por cada agente están ocultos.
- **Representación flexible de la información de la pizarra**. No hay restricciones a la hora de anotar información en la pizarra.
- **Idea de un Moderador**. Ya que la información puede ser muy densa y dispar se hace necesaria una cierta coordinación para el mantenimiento de la pizarra. Surge la idea de un Moderador encargado de dirigir el curso del problema y organizar el trabajo entre los agentes.
- **Envío de Eventos**. En vez de que el agente tenga que explorar la pizarra cada cierto tiempo, es la pizarra la que puede enviar algún tipo de evento cuando se

anota información nueva. Se establece un mecanismo por el cual la pizarra notifica a las fuentes de información los eventos en los cuales están interesadas.

2. **Sistemas de Paso de Mensajes.**



Figura 3.10: Métodos de Comunicación entre agentes: paso de mensajes

La comunicación utilizando mensajes forma una base sencilla y flexible. Este tipo de comunicación se suele emplear cuando las necesidades de comunicación son simples o cuando el alcance del sistema es limitado. Por tanto, para sistema poco complejos es preferible este tipo de comunicación.

El formato libre de los contenidos de los mensajes proporcionan capacidades de comunicación muy versátiles que no están restringidas a comandos simples.

El funcionamiento de esto de comunicación es bastante sencillo: un agente emisor envía un mensaje concreto a un agente emisor por una vía de comunicación predefinida(véase la figura 3.10). Al contrario que en los sistemas de pizarra, los mensajes son intercambiados directamente entre dos agentes. No se utiliza memoria, ni otros agentes son capaces de leer el mensaje si no va dirigido a ellos.

3.2. Sistemas de Vigilancia Inteligente

3.2.1. Introducción

La sociedad en la que vivimos actualmente exige un entorno seguro para vivir y por ello la Vigilancia Inteligente puede ayudar en este punto. Atrás quedan esos sistemas de vigilancia donde sólo existe un flujo de vídeo que se obtiene de una simple cámara de vigilancia y que son administrados por un número reducido de personas (véase figura 3.11).



Figura 3.11: Centro de Vigilancia Tradicional (metro de Londrés)

La Vigilancia Inteligente plantea una serie de **retos** mucho más ambiciosos que la vigilancia tradicional, como por ejemplo la monitorización de espacios complejos, la vigilancia en tiempo real o la identificación de comportamientos de los objetos en una escena de un entorno monitorizado.

Actualmente, la vigilancia inteligente aboca a la necesidad de monitorizar espacios debido a dos factores :

- Reducción del el número de actividad delictivas.
- Aumento de la calidad de vida de la sociedad creando entornos más seguros.

En la mayoría de instalaciones con cámaras CCTV resulta imposible comprobar en todo momento lo que están captando las cámaras. Normalmente, estas imágenes solo se revisan cuando pasa algún hecho fuera de lo normal (análisis forense). Además, el ser humano no es

una máquina perfecta para poder prestar atención en todo momento. Por ejemplo, en el caso de un agente de seguridad que trabaje ocho horas diarias vigilando un conjunto de cámaras de un centro comercial, es imposible que dicho agente sea capaz de estar concentrado durante un largo periodo de tiempo [SMI04].

Existe un estudio práctico [WD88] donde se indica que un operador puede monitorizar realmente entre 1 y 4 cámaras simultáneamente, de manera que en un sistema con 100 cámaras y 3 operarios, solo se podrían estudiar un 3 % de las imágenes activas captadas por el sistema. Cabe citar que para este estudio no se tuvieron en cuenta otros efectos como la fatiga y el cansancio, estados naturales que sufren los seres humanos.

Es por ello, que en los sistemas de vigilancia tradicionales sólo se revisa una parte pequeña de las imágenes captadas. Por consiguiente, la vigilancia inteligente puede ayudar a evolucionar este área.

La **complejidad** de los entornos monitorizables es el principal factor a tener en cuenta para poder monitorizar un entorno. Las diferencias físicas entre un entorno y otro pueden ser tan significativas que hace que los espacios de vigilancia tanto privados como públicos sean complejos. Una posible solución sería añadir una topología de cámaras extensas para poder cubrir cualquier entorno, pero esto no es viable debido al elevado coste que supondría una implantación y despliegue de tanto hardware especifico por cada zona a monitorizar. Además, se necesitaría un número elevado de personas para poder controlar la red de cámaras de vigilancia.

Por todos los motivos expuestos anteriormente, se hace necesario el desarrollo de sistemas de Vigilancia Inteligente. Para ello es vital la implementación de algoritmos inteligentes capaces de controlar un sistema de vigilancia sin la ayuda directa del operador humano, es decir, deben ser sistemas totalmente autónomos con capacidad para decidir en todo momento, gracias a el apoyo de las imágenes captadas por el sistema de cámaras.

A continuación se detallan una serie de **características** que resultan imprescindibles para poder desarrollar un sistema de Vigilancia Inteligente:

- **Portabilidad**. El sistema debe poder ser usado en distintas condiciones o entornos.
- **Escalabilidad.** El sistema debe ser fácilmente escalable para poder dar soporte a la mayor cantidad de usuarios posibles o en este caso, poder analizar multitud de situaciones simultáneamente .
- **Adaptabilidad.** El sistema debe poder adaptarse a distintos entornos monitorizados así como facilitar la extensión de nuevas funcionalidades al sistema en caso de que sea preciso.

- **Modularidad.** El sistema debe proveer una serie de servicios para poder monitorizar un entorno y para ello deben existir una serie de módulos que resuelvan cada una de las características del sistema.
- **Robustez** El sistema debe ser tolerante a fallos y poder seguir trabajando en caso de que el hardware de soporte falle.

3.2.2. Evolución

La evolución tecnológica de los Sistemas de Vigilancia ha sufrido un proceso evolutivo a lo largo de estos últimos años. En [VV05] se clasifican estos sistema en tres generaciones que van a ser discutidas brevemente a continuación:

1. **Primera Generación.**

   También conocida como CCTV (Close-Circuit Television). Es la etapa más analógica de todas. Se le conoce como circuito cerrado ya que todos los sistemas están unidos unos con otros. En esta generación el sistema consta de un conjunto de cámaras distribuidas en un entorno monitorizable, y una sala con diversos monitores que controlan las imágenes captadas por el sistema.

   Los problemas que presentan estos tipos de sistema son dos principalmente; el almacenamiento se hace muy pesado (normalmente, en cintas de vídeo antiguas) y se necesitan técnicas analógicas para su distribución, lo que hace complicado poder mantener adecuadamente el sistema. El otro problema es que para poder monitorizar el entorno se necesita el uso de una o varias personas que estén continuamente vigilando las imágenes captadas o bien se revisen posteriormente dichas imágenes. Pero de una forma u otra es necesaria la actuación humana para poder dar uso al sistema.

   Pese a sus problemas, los sistemas de videovigilancia CCTV son los sistemas más utilizados, sobre todo en entornos industriales o comerciales.

   Un avance en los sistemas de primera generación fue integrar la tecnología IP para poder supervisar los sistemas. Ya no es necesaria una sala centralizada para poder revisar las imágenes. Con este sistema el encargado de la empresa puede observar las grabaciones desde cualquier lugar del mundo con acceso a internet. Gracias a este cambio, las imágenes se envían a través de la red por lo que no es necesario el uso de cintas de vídeo para grabar la información. Estos sistemas mejoran sustancialmente a los sistemas CCTV tradicionales por lo siguientes motivos:

   - Facilidad de mantenimiento.
   - Ahorro en los costes debido a que no es necesaria tanta cinta de vídeo. Ahora la imagen se puede almacenar directamente en uno o varios computadores.
   - Facilidad de uso.

- Accesibilidad remota.
- Posibilidad de un tratamiento digital de las imágenes.
- Fácil integración con otros sistemas similares.

2. **Segunda Generación**.

   En esta segunda etapa de la Vigilancia Inteligente los sistemas intentan buscar una autonomía mayor, es decir, estos sistemas intentar reducir la interacción de los humanos con el mismo. Para ello es necesario que los sistemas implementen algoritmo de Visión por Computador e Inteligencia Artificial para poder tratar las imágenes.

   Estos sistemas deben poder interpretar comportamientos y eventos en un entorno monitorizable. Actualmente, la interpretación de comportamientos y eventos en un entorno real no es problema trivial debido a las múltiples dificultades que ofrecen los entornos. Estos sistemas cuentan con muchos handicaps para poder alcanzar la optimalidad. A continuación se explican los retos que se plantean en esta generación:

   - **Representación de cualquier dominio.**

     Normalmente los entornos son entornos muy complejos y lograr una representación del conocimiento adecuada es difícil. El sistema inteligente debe conocer los objetos que pueden participar en las escenas así como las relaciones entre ambos.

   - **Tratamiento apropiado de la incertidumbre y de la vaguedad**.

     En cualquier escenario que se trate en un entorno real, el sistema nunca podrá asegurar con total veracidad que está pasando pero es necesario que el sistema sea capaz de decidir que está pasando en cada instante. Normalmente, asignando a cada objeto de la imagen un nivel de normalidad en función de como dicho objeto se esté comportando. Por poner un ejemplo, imaginemos un sistema de Vigilancia Inteligente implantando en unos grandes almacenes. Un comportamiento fuera de lo normal es que una persona esté corriendo por los pasillos. El sistema debería ser capaz de asumir que ese objeto no está dentro de la normalidad.

   - **Implementación de algoritmos eficientes**.

     Es necesario que los algoritmos sean eficientes para poder dar resultados en tiempo real. Si seguimos con el ejemplo anterior, el sistema no es capaz de avisar a nadie cuando detecta comportamientos anómalos, la mejora con respecto a los sistemas de primera generación es nula e incluso la empresa estaría perdiendo más dinero. Por tanto, este punto es de vital importancia a la hora de desarrollar un sistema de Vigilancia Inteligente.

3. **Tercera Generación**.

   Estos sistemas se caracterizan por ser distribuidos. Los sistemas de Vigilancia Inteligente en esta etapa tienen como meta monitorizar entornos más amplios gracias a que

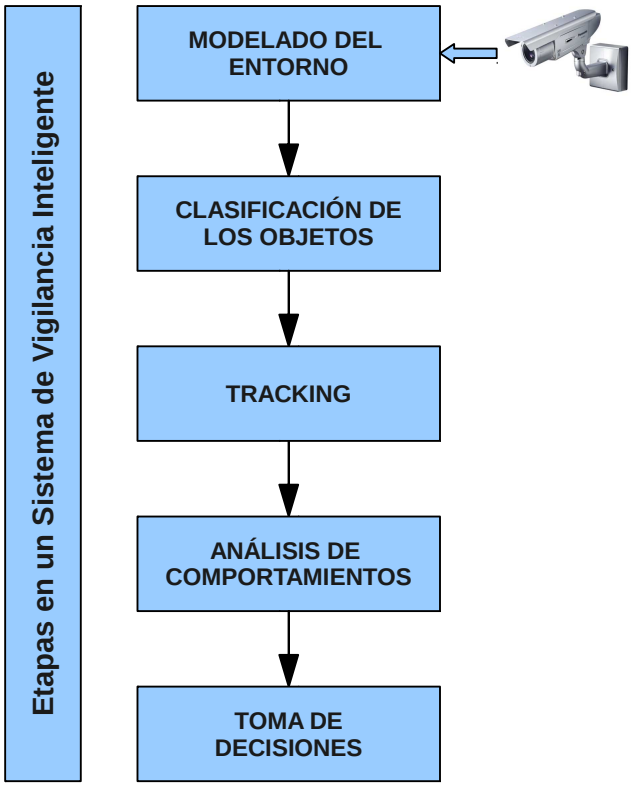


Figura 3.12: Etapas en un sistema de Vigilancia Inteligente

la información obtenida está distribuida. Por ello, es necesario el uso de múltiples sensores distribuidos inteligentemente por todo el entorno a monitorizar, los cuales envían información en tiempo real.

En este tipo de sistemas, la carga de información ya no está centralizada por lo que el sistema tiene mejores garantías para poder responder en tiempo real. También, el sistema gana en robustez ya que el sistema puede seguir trabajando en caso de que algunos componentes queden dañados. Pero la distribución de la información también tiene problemas ya que se complica la vigilancia del entorno debido a la necesidad de combinar múltiples dispositivos. Y por otro lado, también es complicado relacionar la información de los sensores para que el sistema pueda razonar correctamente.

Etapas en un sistema de Vigilancia Inteligente

Un sistema de Vigilancia Inteligente cuenta con las siguientes etapas funcionales (véase figura 3.12):

- **Modelado**. Esta etapa consiste en definir los elementos que participan en el entorno, así como sus propiedades y relaciones entre los elementos existentes en el entorno
- **Segmentación**. En en etapa se identifican los elementos y las acciones que están realizando en cada momento.

- **Clasificación**. En esta etapa se clasifican de manera adecuada de los objetos móviles de la escena. Este punto es vital ya que el análisis de comportamientos para un tipo suele ser totalmente distinto para otro tipo. A modo de ejemplo, el comportamiento que debe tener una persona normalmente en una calle pública no puede parecerse al comportamiento de un vehículo. Por ello, el sistema inteligente debe clasificar correctamente los objetos para dar soluciones reales.

  Típicamente, la clasificación de objetos en movimiento se realiza a partir de la silueta que tenga el objeto en la imagen o en función de los movimientos que realicen.

  Para el primer caso, el sistema tendrá un mecanismo para asignar un valor numérico que indicará el grado de pertenencia de un objeto a una clase. En el segundo caso, muchos autores argumentan que la silueta de la persona varía porque son objetos no rígidos y realizan un movimiento periódico en sus desplazamientos. En cambio, los vehículos son objetos rígidos y no tienen movimientos periódicos.

- **Tracking**. Esta etapa consiste en el seguimiento del objeto hasta que deja de aparecer en las imágenes. El proceso en el cual se determina la posición del objeto en cada instante de tiempo se denomina *tracking*. La salida de esta etapa es la entrada de la siguiente, ya que sin un una evolución temporal de los objetos, no se podrían sacar conclusiones de su comportamiento en la escena.

  Los algoritmos de *tracking* normalmente sufren de un problema que se denomina oclusión. Este hecho aparece cuando un objeto de la escena es ocultado total o parcialmente. La mayoría de sistemas que realizan *tracking* de objetos sufren ciertas dificultades para poder seguir con el *tracking* cuando ocurre la oclusión de un objeto. Tienden a considerar el objeto como un nuevo objeto cuando ya no existe la oclusión. Actualmente, la mayoría de sistemas de Vigilancia Inteligente tratan las oclusiones utilizando múltiples cámaras.

  Algunos de los problemas más comunes en estas etapas se listan a continuación:

  - El problema principal de la etapa de segmentación es detectar que objetos están en movimiento sin equivocaciones.
  - Puede darse el caso que se den oclusiones totales o parciales de los objetos que aparecen en la escena.
  - Reducción de la visibilidad de las cámaras en ciertas zonas puede ocasionar incertidumbre en las posiciones de los objetos , en clasificar los objetos y detectar los movimientos.
  - El sistema puede seguir y clasificar objetos que no tienen mucha importancia como por ejemplo podría ser el movimiento de un árbol por el aire, un cartel luminoso de publicidad, reflejos en algún cristal y un largo etcétera de situaciones.

  - Cuando un objeto que está siendo seguido por el sistema se separa en varios objetos.
  - Cuando dos objetos cruzan sus trayectorias es posible que el sistema los trate en ese justo instante como un único objeto.

- **Interpretación de Comportamientos**. En esta etapa se identifican los comportamientos de un objeto en la escena. El comportamiento de un objeto se identifica por una acción o evento que se produce en un instante de tiempo concreto, o bien por una secuencia de eventos en distintos instantes de tiempo. El problema de esta etapa es su dura dependencia con etapas anteriores. Una clasificación fallida o una trayectoria equivocada normalmente desemboca en una interpretación incorrecta en la conducta de un objeto.
- **Toma de Decisiones**. En esta etapa, un Sistema Inteligente debería ser capaz de tomar una serie de decisiones y elaborar un plan de emergencia en caso de que suceda una alarma de nivel medio-alto. Esta etapa, igual que la anterior, sufre de una fuerte dependencia con las etapas anteriores y ya que un mala decisión en esta etapa puede afectar a la integridad de las personas en el entorno, suele ser supervisada por una persona humana antes de que se realicen acciones.

Dentro de las aplicaciones de Vigilancia Inteligente, predominan las aplicaciones que analizan personas y vehículos, debido a que son los objetos más frecuentes de los entornos. A continuación se expone una lista de las aplicaciones de Vigilancia Inteligente más frecuentes:

- **Control de acceso en áreas reservadas**.
- **Identificación de personas concretas en escenarios complejos**.
- **Análisis de aglomeraciones de personas**.
- **Detección de anomalías y alarmas**.
- **Vigilancia interactiva mediante múltiples cámaras**.

Los lugares donde se implementan estas aplicaciones suelen ser variados pero principalmente se despliegan sistemas de Vigilancia Inteligente en lugares tales como:

- **Aeropuertos**. Son los entornos en los que se demanda alta seguridad ya que suele ser objetivo importante de grupos terroristas.
- **Estaciones de trenes**. La mayoría de sistemas para la vigilancia en estaciones, se suele desplegar en las estaciones de metro, donde hay un alto grado de altercados. En Londres, hay un proyecto en funcionamiento, que se detallará en secciones posteriores, donde se analiza el comportamiento de las personas en una estación de ferrocarril.
- **Vigilancia de tráfico**. La principal función de los sistemas implantados para la vigilancia de trafico consisten en la detección de congestiones y posibles accidentes.

- **Entornos marítimos**. Cuyo objetivo principalmente se centra en puertos de carga y descarga de mercancías, donde existe un alto nivel de robos.
- **Entornos Industriales**. Aparte de la vigilancia de la fábrica, también se usan para el control de calidad en el proceso de producción.
- **Lugares públicos** como bancos, casinos, parkings o grandes almacenes.

3.2.3. Arquitecturas Multi-Agente

En la actualidad existe una gran variedad de Arquitecturas Multi-Agentes que implementan sistemas de Vigilancia Inteligente. En esta memoria se van a presentar y estudiar una serie de Arquitecturas Multi-Agente ampliamente probadas en situaciones cotidianas. El uso de múltiples agentes inteligentes resulta adecuado para resolver problemas complejos debido a sus propiedades (estudiadas en la sección 3.1.2), sobretodo en entornos complejos con fuentes de información heterogéneas. Por ello, las arquitecturas multi-agentes resultan una solución adecuada para la implementación de Sistemas de Vigilancia Inteligente.

Proyecto PRISMATICA

El proyecto PRISMATICA[VBL05] fue un proyecto financiado por la Unión Europea cuyo objetivo era conseguir sistemas de transportes más seguros y más atractivos para las personas. El aspecto más innovador de este proyecto fue la integración de aspectos variados tales como la ingeniera social, la legalidad y la tecnología.

Como ya se explicó en capítulos anteriores la principal limitación de los sistemas de vigilancia de circuito cerrado(CCTV) es que tiende a ser utilizado como una herramienta reactiva y se hace necesaria la intervención de un operador para procesar las imágenes. El proyecto PRISMATICA tenía como objetivo crear una herramienta proactiva que fuera capaz de reconocer comportamientos en vías publicas de transporte, resultando ser una herramienta potente de apoyo a los trabajadores del lugar.

El sistema PRISMATICA es un sistema de vigilancia distribuido perteneciente a la **tercera generación** de sistemas de vigilancia que consta de un conjunto de diversos dispositivos, cada uno de los cuales realiza una tarea local para contribuir al objetivo común. Es decir, cada dispositivo (por ejemplo, una cámara, un micrófono, un sistema de acceso a la vía) trata con una área física relativamente pequeña sin que sea necesaria la información global del sistema.

La función de coordinar las acciones entre los dispositivos y la recogida de información generada por los dispositivos es llevada a cabo por un componente conocido como MIPSA (*Modular Integrated Passenger Surveillance Architecture*). Esta parte del sistema proporciona un único punto de contacto entre un operador y un medio de control así como la comunicación con los dispositivos inteligentes. Es el encargado de tomar las decisiones cuando tenga una información consistente para realizar una acción.

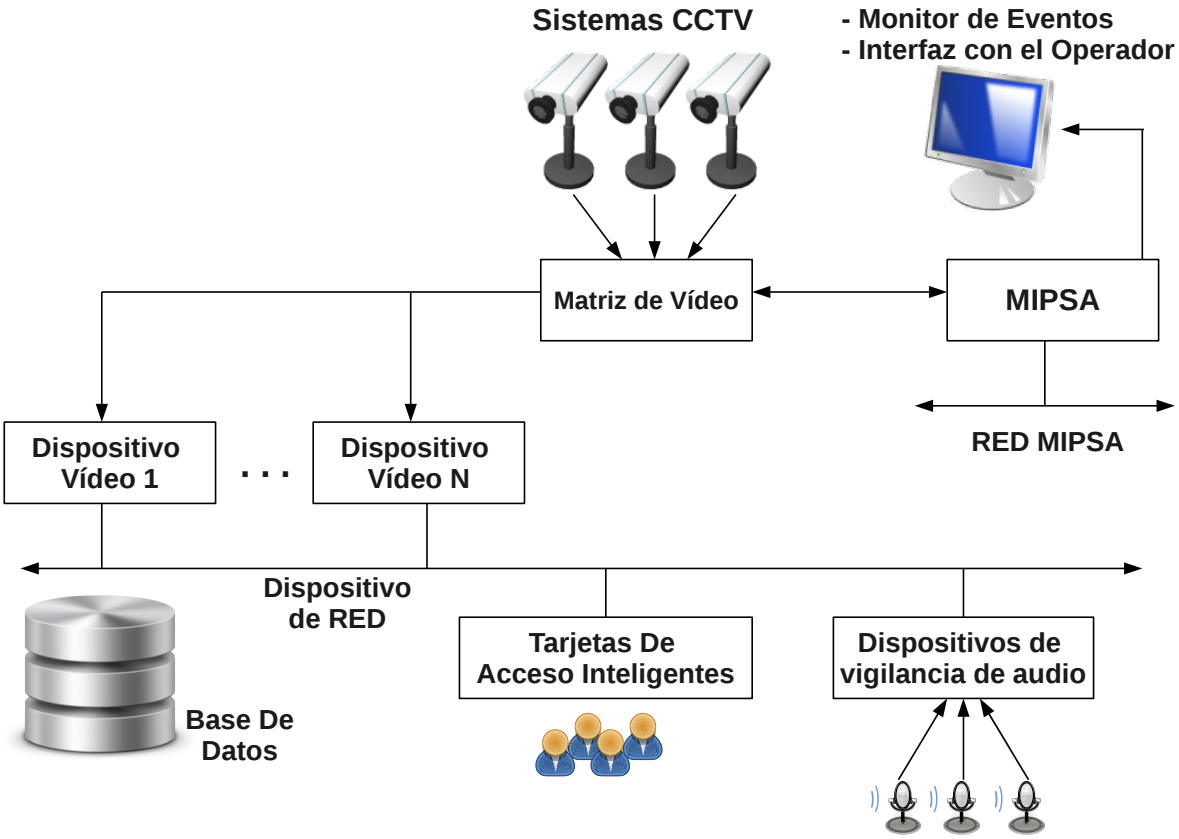


Figura 3.13: Arquitectura PRISMATICA (Esquema adaptado de [VBL05])

Esta comunicación se realiza a través de una red de área local utilizando una arquitectura basada en el *middleware* CORBA[1] usando un protocolo encapsulado en XML para el envío y recepción de mensajes y usando sockets para las transferencias de datos.

En la figura 3.13 se muestra un diagrama de la arquitectura general del sistema PRISMATICA y en la figura 3.14 se puede apreciar una visualización de un monitor MIPSA.

Las señales procedentes de los sistemas CCTV se almacenan en una matriz de vídeo, controlada por un MIPSA. Esto sirve para dos propósitos: en primer lugar, las imágenes pueden ser reproducidas inmediatamente por un operador y en segundo lugar, en caso de que sea preciso las imágenes pueden ser analizadas por los dispositivos de vídeo inteligente.

Además, el sistema incluye un dispositivo para capturar las señales de las tarjetas de los pasajeros. Cuando un pasajero introduce su tarjeta en los puntos de control para poder acceder a la vía, la señal es recogida por el punto de control y se transmite la información al MIPSA. De forma que el MIPSA puede reconocer la posición del punto de control, localizar al pasajero , mostrar imágenes de las cámaras que cubren esa zona e incluso generar un

[1](Common Object Request Broker Architecture). CORBA es un estándar definido por el OMG (Object Management Group) que permite escribir componentes de software en múltiples lenguajes y ejecutarlos en múltiples computadoras para trabajar juntos.

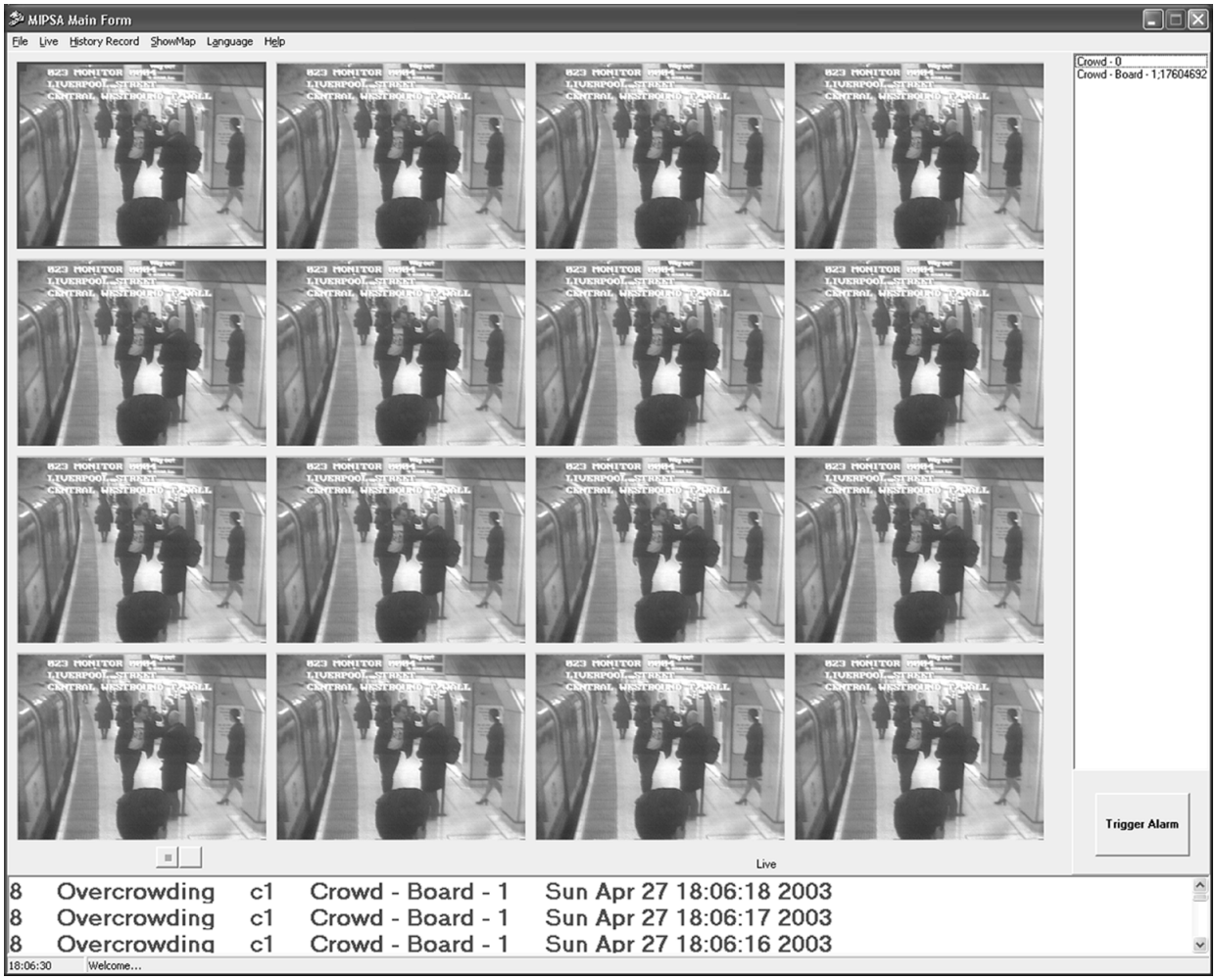


Figura 3.14: Monitor MIPSA : Proyecto Prismática ([VBL05])

mensaje de alarma en caso de irregularidades.

El sistema integró también un dispositivo de análisis de audio, desarrollado por *Thales Underwater Systems* (Francia) capaz de detectar sonidos inusuales como por ejemplo llamadas de socorro o peleas.

El sistema contiene una base de datos para almacenar la configuración del sistema, las anotaciones de los eventos y las secciones de los vídeos para poder recuperar en cualquier momento.

En la figura 3.15 se puede apreciar un diagrama de secuencia que resumen las interacciones propias del sistema.

La **evaluación** del sistema consistió en dos etapas:

- La arquitectura se puso a prueba en el metro de París y se demostró con éxito que los protocolos de comunicación y la integración de los dispositivos funcionaban correctamente.

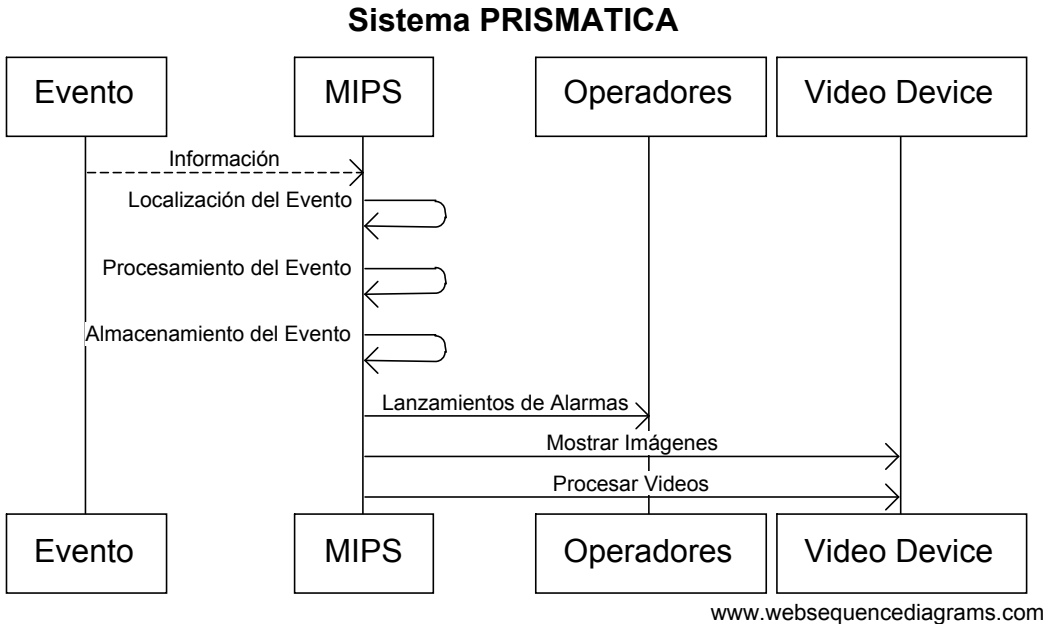


Figura 3.15: Diagrama de Secuencia Sistema PRISMATICA

- Un importante despliegue del sistema se llevó a cabo en una de las estaciones de metro más concurridas del mundo. En concreto, la estación de Liverpool Street de Londrés.

El trabajo actual de este proyecto está centrado en seguir mejorando el rendimiento en la parte de visión por computador e incorporar conocimiento experto en la tarea de análisis automático de comportamientos. Dotar al sistema de un conocimiento extra en función del contexto de cada hora concreta de los días. Por ejemplo, cuando los niños vuelven de la escuela, se está trabajando para que el sistema sea capaz de conocer esa situación y no alterar sus resultados. Otro caso podría ser cuando un día señalado se produzca algún evento deportivo en la ciudad, lo que puede originar una masificación de personas en la estación.

Sistema de Vigilancia Inteligente propuesto por IBM

En 2004 [TBH04] la empresa IBM propuso una arquitectura para un sistema de vigilancia inteligente. Por estos tiempo, IBM afirmaba que la mayoría de los sistemas de vigilancia inteligente se centraban en la obtención de información y pocos investigaban en usar dicha información para realizar búsquedas en tiempo real. Por tanto, la mayor prioridad del sistema se centró en la realización de búsquedas en tiempo real sobre las base de datos donde se almacenaban los datos de los vídeos.

Desde un punto de vista arquitectónico, el sistema de vigilancia de IBM se basa en dos módulos principales (véase figura 3.16):

1. **IBM: SSE o Motor de Vigilancia Inteligente.**

   Es el módulo encargado de llevar a cabo los análisis de los vídeos y comportamientos de los objetos. Es un software de vídeo basado en la tecnología de detección de even-

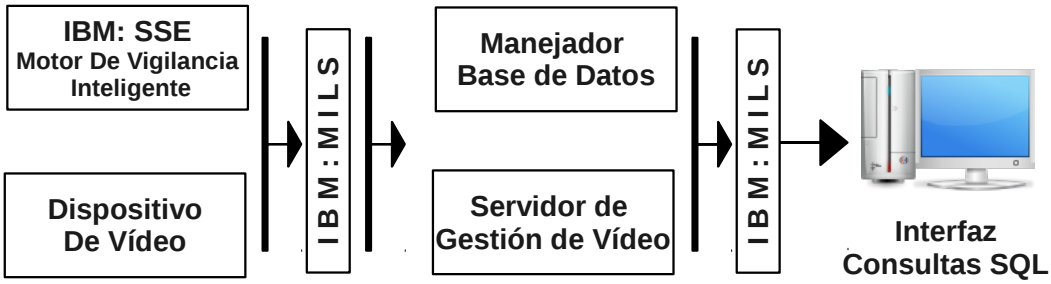


Figura 3.16: Arquitectura del Sistema de Vigilancia IBM ([TBH04])

tos. La SSE esta diseñada para poder realizar un despliegue adecuado de un sistema de vigilancia inteligente. El sistema implementa tecnologías para la detección de objetos, seguimiento de objetos y clasificación de objetos. Ofrece funcionalidades diversas como por ejemplo detectar movimientos, identificar objetos abandonados en la escena, tolerancia en los cambios de iluminación o de tiempo (por ejemplo, el soplo de los árboles) y generar alertas en tiempo real.

El módulo SSE detecta y rastrea todos los objetos que se mueven en las escena. La SSE crea un conjunto de documentos XML llamado VVI donde se codifican todas las actividades interesantes que ocurren en los vídeos, como por ejemplo; número de objetos en una escena, clase de cada objeto, propiedades de los objetos, posición, trayectoria, velocidad de los objetos en movimiento y otros aspectos del evento.

2. **IBM: Middleware MILS.**

   MILS ofrece los servicios de gestión de datos necesarios para construir una aplicación de vigilancia inteligente. Ofrece las siguientes funcionalidades:

   - Convertir el conjunto de documentos XML en tablas relaciones. MILS es capaz de tomar como entrada el VVI generado por el módulo SSE y almacenar dichos datos en tablas relaciones de una base de datos. Con esto se consigue poder realizar consultas SQL sobre la base de datos.
   - Servicios de consulta de datos. MILS proporciona a las aplicaciones una API de consulta de datos sobre las base de datos relacionales y soporta una amplia gama de consultas.

En 2004 se probó el sistema en un aparcamiento de un edificio de IBM (*IBM Watson Research Center*) con éxito. Se realizó un análisis de vídeo y se realizaron consultas tan diversas como por ejemplo; las últimas 100 actividades almacenadas en la base de datos, todos los coches que pasaron en un tiempo específico, todas las personas que pasaron en un intervalo de tiempo, automóviles que pasaron por una zona determinada y con un tamaño especificado por el propio usuario, todos los coches estacionados de color rojo,etc.

Middleware para la vigilancia de vídeo distribuido propuesto por H.Detmold

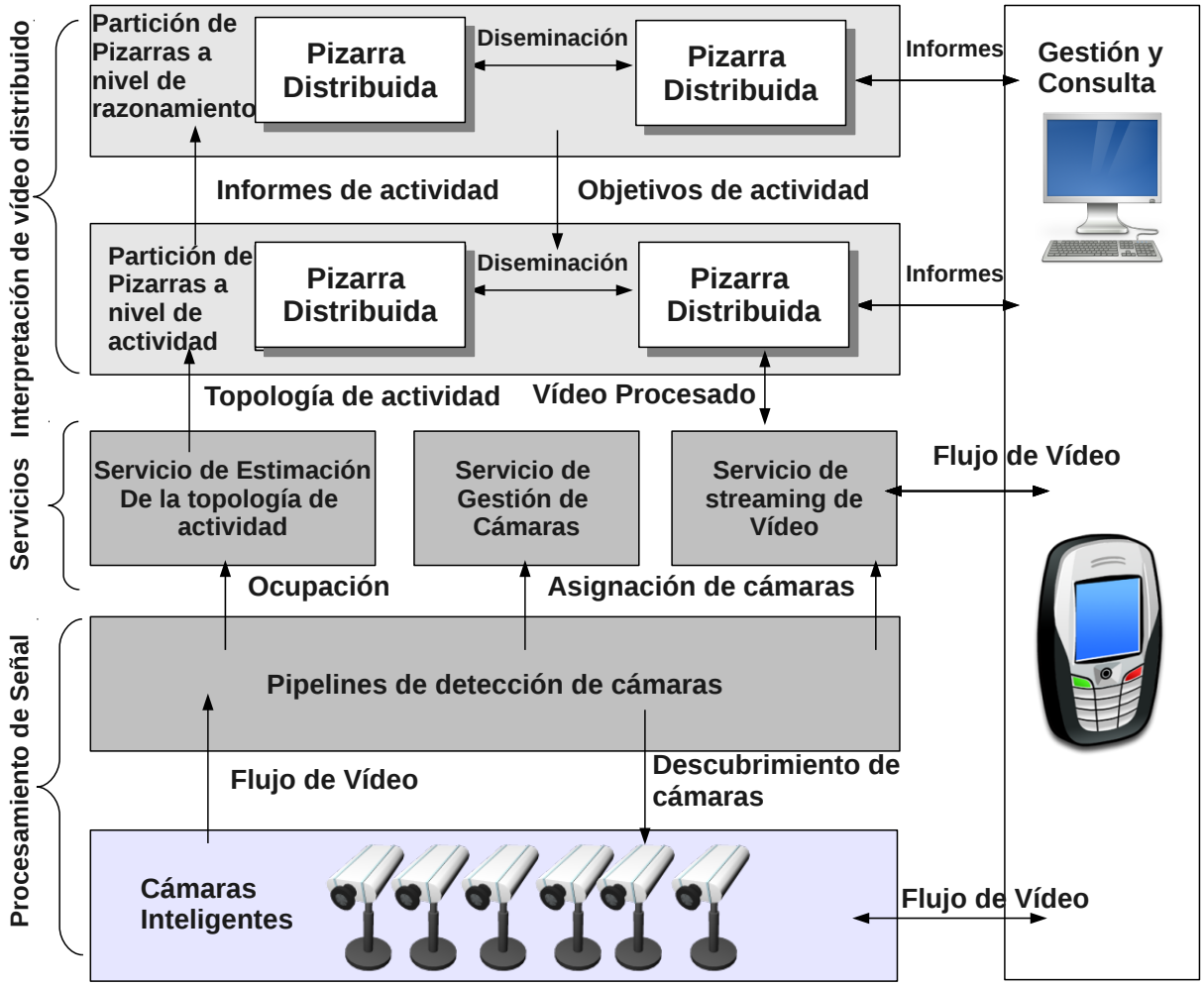


Figura 3.17: Arquitectura del Sistema de Vigilancia propuesto por [DVD08]

En 2008, en [DVD08], surgió un proyecto más innovador en el campo de la Vigilancia Inteligente. Se trata de un *middleware* como mecanismo de despliegue de sistemas de vigilancia inteligente basado en una topología que describe la actividad de los objetos de la escena. Esta novedosa arquitectura se basa en una arquitectura jerárquica que va desde los sensores (hojas) hasta el operador humano (raíz) y se apoya en el uso de pizarras distribuidas para lograr un sistema escalable y flexible.

La arquitectura está compuesta por diferentes módulos, los cuales tienen funciones específicas. En concreto, el sistema esta dividido en tres módulos principales (véase figura 3.17):

- Servicios. La capa de servicios ofrece interfaces intermedias entre los demás módulos de la arquitectura. También ofrece un servicio de gestión de las cámaras en el que se incluyen facilidades para registrar las cámaras, detectar anormalidades o asignarle tuberías para conseguir un balanceo de carga entre cámaras y tolerancia a fallos.

- Video-Streaming. Este módulo es el encargado de mantener los archivos de vídeo y los pone a disposición siempre que sea requerido.
- Análisis inteligente. Este modulo es el responsable de procesar los vídeos almacenados usando la tecnología de pizarras distribuidas.

Aunque los autores de [DVD08] mantienen que el rendimiento de la aplicación en tiempo real es un requisito fundamental para este tipo de aplicaciones, el objetivo principal del *middleware* fue construir un sistema robusto y eficaz, considerando este punto más importante que el rendimiento. Para conseguir la robustez del sistema desplegaron cada pizarra distribuida en nodos distintos, y así lograr que siempre se pudiera brindar el servicio.

3.3. Middlewares de Comunicaciones

3.3.1. Introducción

La organización IETF (Internet Engineering Engineering Task Force) en Mayo de 1997 define el concepto de *middleware* de la siguiente manera:

> *Un middleware puede ser visto como un conjunto de servicios y funciones reutilizables, expansibles, que son comúnmente utilizadas por muchas aplicaciones para funcionar bien dentro de un ambiente interconectado.*

Un **middleware** [Ber96][Hen04] es un software que asiste a una aplicación para poder comunicarse con otras aplicaciones, redes y/o sistemas operativos. Funciona como una capa de abstracción de software, que se sitúa entre las capas de aplicaciones y las capas inferiores (sistema operativo y red), cuya función principal consiste en proporcionar los mecanismos de comunicación necesarios entre los distintos elementos de un sistema distribuido.

Gracias al uso de *middlewares* el trabajo de los programadores para generar las conexiones que son necesarias en los sistemas distribuidos se hace de forma más liviana. De esta forma se provee una solución que mejora la calidad de servicio, seguridad, envío de mensajes, directorio de servicio, etc.

El *middleware* abstrae de la complejidad y heterogeneidad de las redes de comunicaciones subyacentes (véase figura 3.18), así como de los sistemas operativos y lenguajes de programación, brindando una *API* para la fácil programación y manejo de aplicaciones distribuidas. Dependiendo del problema a resolver y de las funciones necesarias, serán útiles diferentes tipos de servicios del *middleware*. Por lo general, el objetivo principal de un *middleware* es acercar el modelo de programación a un modelo local.

Sin embargo, esta definición ha ido evolucionando a distintos ámbitos, y se puede realizar una clasificación de *middlewares* en función de su objetivo e implementación:

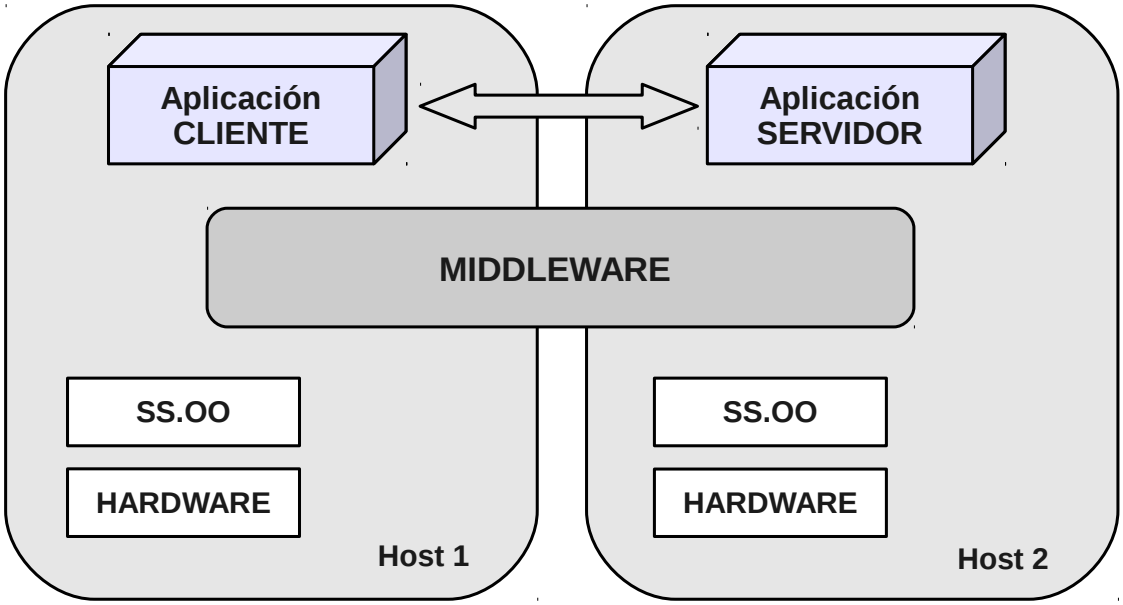


Figura 3.18: Middlewares como mecanismo de abstracción.

- **Middlewares de aplicaciones**. Su objetivo consiste en facilitar la ejecución e interacción de distintas aplicaciones en un mismo servidor.
- **Middlewares de comunicaciones**. Su objetivo es abstraer a una aplicación distribuida de las redes de comunicaciones así como sus protocolos. Dentro de esta categoría se pueden incluir los sistemas de invocación a procedimientos remotos (RPC/RMI), los sistemas de colas de mensajes (MQ) y los sistemas orientados al procesado paralelo de datos (DRI, PAS).
- **Middlewares para bases de datos**. Su objetivo es facilitar la interacción entre una aplicación y una base de datos (JDBC, ODBC, mySQLClient,etc).

3.3.2. Alternativas más relevantes

ZeroC ICE

ICE (Internet Communication Engine) [Hen09] es un *middleware* orientado a objetos. ICE otorga una serie de APIs y soporte de bibliotecas para construir aplicaciones distribuidas. Una aplicación ICE puede ser usada en diversos entornos, los clientes y servidores pueden ser escritos en distintos lenguajes de programación, pueden ejecutarse en distintos sistemas operativos y pueden comunicarse empleando diferentes tecnologías de red.

ICE fue diseñado por gente muy experta en el sector y el principal objetivo fue ofrecer una solución sencilla a las comunicaciones en una aplicación distribuida (véase figura 3.19). ICE es fácil de entender y programar pero a la vez es un sistema muy flexible que puede dar cabida a aplicaciones más exigentes.

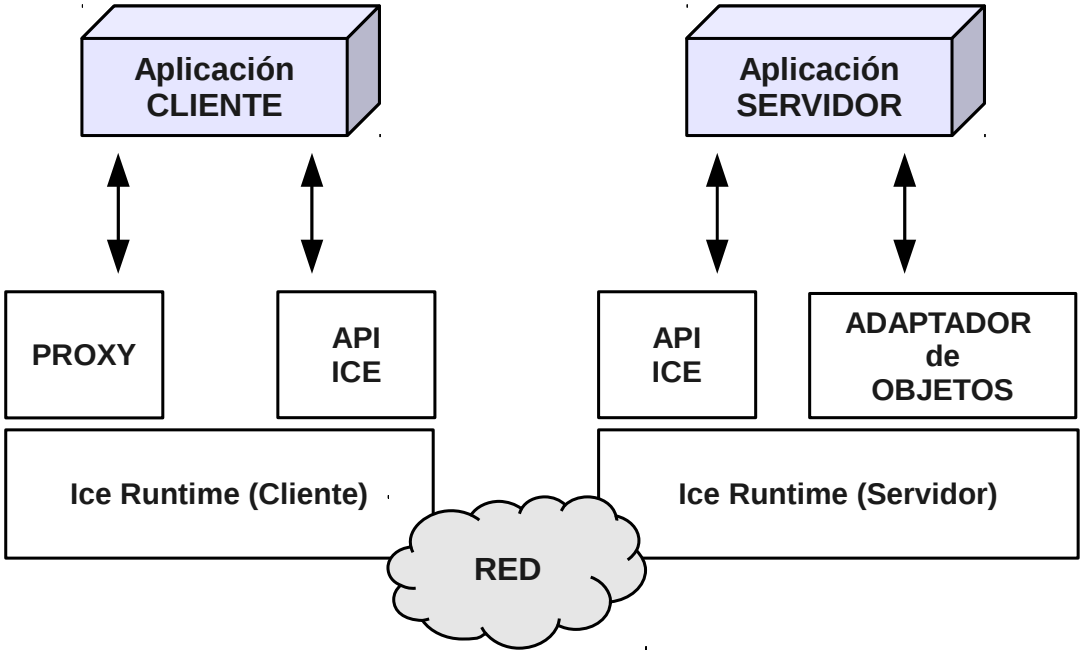


Figura 3.19: Arquitectura abstracta de ZeroC ICE.

A continuación se listan algunas de las características de ICE [Ice11]:

- Permite desarrollar las aplicaciones distribuidas en diversos lenguajes de programación: C++, Java, C# (y de otro tipo. NET, como Visual Basic), Python, Ruby, PHP y ActionScript.
- Ofrece un conjunto de APIs que destacan por su simplicidad y facilidad de uso. Esto acorta el tiempo de desarrollo, disminuye el esfuerzo de prueba y reduce el tiempo de comercialización.
- Proporciona una implementación eficiente en ancho de banda, en uso de memoria y en ciclos de CPU.
- Proporciona una serie de servicios que dan soporte para el desarrollo de aplicaciones distribuidas complejas. Véase el cuadro 3.1.
- ICE crea sistemas tolerantes a fallos gracias al uso de la replicación. Varias instancias de un servidor pueden ser desplegadas en diferentes equipos para lograr disponibilidad en al menos uno de los servidores. Esto no sólo hace que las aplicaciones sean resistentes ante los fallos, sino que también aumenta el rendimiento porque ICE te permite equilibrar la carga de un sistema distribuido en varios servidores.
- Es mucho más que un mecanismo de llamada a procedimientos remotos debido a que permite llamadas asíncronas como llamadas síncronas.

IceGrid	Servicio de localización, despliegue y activación de las aplicaciones.
IceBox	Servidor de aplicaciones ICE.
IceStorm	Servicio de publicación-subscripción.
IcePatch2	Servicio de distribución de archivos.
Glacier2	Servicio de cortafuegos de ICE.

Cuadro 3.1: Servicios avanzados de Zeroc ICE.

Típicamente, las aplicaciones reales necesitan algo más que las capacidades remotas, como por ejemplo la activación de servidores bajo demando, la distribución de los proxies a los clientes, la distribución de eventos asíncronos, la configuración para el despliegue de las aplicaciones, etc.

ICE posee una serie de servicios que gestionan estas características. Dichos servicios se implementan como servidores ICE de forma que la aplicación desarrollada actúe como un cliente.

A continuación se detallan una serie de características de los servicios ofrecidos por ICE:

- **IceGrid:** Es la implementación de un **servicio de localización ICE** que convierte la información simbólica en un proxy indirecto asociado a un par protocolo-dirección. IceGrid permite el registro de servidores para un arranque automático. Proporciona las herramientas necesarias para configurar aplicaciones complejas. Soporta la replicación y el balanceado de carga y otras muchas funcionalidades.
- **IceBox:** Es un **servidor de aplicaciones** que posibilita una fácil gestión para arrancar y parar componentes de una aplicación. Dichos componentes se despliegan como una biblioteca dinámica en vez de como un proceso, para lograr una reducción de la sobrecarga del sistema.
- **IceStorm:** Es un **servicio de publicación-subscripción** que actúa como un distribuidor de eventos entre servidores y clientes. Los publicadores envían eventos al servicio IceStorm, el cual los notifica a los subscriptores. Permite que un único evento sea enviado de forma transparente a múltiples subscriptores. Los eventos están categorizados en función de un tema (*topic*), para que los subscriptores puedan suscribirse al tema que les interese.
- **IcePatch2:** Es un servicio que permite la **fácil distribución de actualizaciones** de software a los clientes. Los clientes únicamente deben conectarse al servidor y solicitar las actualizaciones. El servidor chequea la versión de los clientes y envía las actualizaciones comprimidas.

- **Glacier2:** Es el **servicio de cortafuegos de ICE**, el cual posibilita la seguridad en la comunicación entre servidores y clientes. El tráfico entre el servidor y el cliente queda encriptado utilizando certificados de clave pública.

Por todo lo descrito anteriormente, el sistema ICE resulta flexible, fácil de usar, robusto, escalable y ofrece un rendimiento superior a otras alternativas.

CORBA

CORBA [Obj08] (*Common Object Request Broker Architecture*) es una plataforma software usada para dar soporte en la creación de sistemas distribuidos cuyo objetivo se basa en facilitar la invocación de procedimientos remotos y emplear un paradigma orientado a objetos.

Fue desarrollado por un consorcio llamado **OMG** (*Object Management Group*), una organización sin fines lucrativos a la cual pertenecen más de 800 miembros. Dentro de esta asociación CORBA representa la arquitectura de comunicaciones.

CORBA empaqueta el código escrito en otro lenguaje, en un paquete que contiene información adicional sobre las capacidades del código que contiene y sobre cómo llamar a sus métodos. Los objetos resultantes, pueden ser llamados desde otro programa (u objeto CORBA) desde la red. Es decir, CORBA permite que varias aplicaciones "hablen.^entre si incluso si están en diferentes maquinas, si usan distintos sistemas operativos o si están implementadas con diferentes lenguajes.

CORBA cuenta con los siguientes componentes:

- **Objetos**. Se utiliza un lenguaje de definición de interfaces (IDL) para especificar la funcionalidad de los objetos. Implementaciones estándar existen para Ada, C, C++, Smalltalk, Java, Python, Perl y Tcl.
- **Clientes**. Invocan métodos sobre los objetos de manera transparente. El código del cliente generado tras compilar el archivo *IDL* sirve para poder realizar las llamadas a métodos remotos. Es el conocido como *stub*, el cual incluye un *proxy* (representante) del objeto remoto en el lado del cliente. El código generado para el servidor consiste en unos *skeletons* (esqueletos) que el desarrollador tiene que rellenar para implementar los métodos del objeto.
- **ORB** (*Object Request Broker*). La principal tarea del *ORB* es resolver las solicitudes de referencias a objetos, permitiendo a los componentes de la aplicación establecer la comunicación con otros componentes. Mediante la utilización de un de *ORB*, un cliente puede invocar de manera transparente un método de un objeto sirviente que se encuentre en un proceso de la misma máquina o en otra distinta. No es necesario que el cliente sepa dónde se localiza el objeto que ejecutará el método, el lenguaje en el que está programado, el sistema operativo, ni cualquier otro aspecto que no sea su interfaz.

En resumen, CORBA proporciona un mecanismo estándar para definir interfaces entre componentes así como algunas herramientas que facilitan la implementación de dichas interfaces utilizando el lenguaje elegido por el desarrollador.

Además, la OMG especifica una gran abundancia de servicios estándar, tales como servicios de directorio, de nombrado, servicios de objetos persistentes, servicios de publicación-subscripción y servicios de transacciones. Cada uno de estos servicios está definido cumpliendo con la especificación de CORBA, y por tanto están disponibles para todas las aplicaciones CORBA. Finalmente, CORBA proporciona los medios que permiten que varios componentes de una aplicación, o de otras aplicaciones, puedan comunicarse entre si de una manera sencilla.

WEB SERVICES

El consorcio W3[2] definió una arquitectura de servicios distribuidos alrededor de estándares comunmente usados en Internet, como por ejemplo HTTP o XML. La arquitectura resultante es conocida como **Web Services Architecture** [WS004].

Es una arquitectura que utiliza un conjunto de protocolos y estándares que sirven para intercambiar datos entre aplicaciones. Distintas aplicaciones de software desarrolladas en lenguajes de programación diferentes, y ejecutadas sobre cualquier plataforma, pueden utilizar los servicios web para intercambiar datos en redes de ordenadores como Internet. La interoperabilidad se consigue mediante la adopción de estándares abiertos. Definen un entorno de desarrollo de aplicaciones dinámico y flexible.

Los estándares abiertos empleados se listan a continuación:

- **Web Services Protocol Stack**. Conjunto de servicios y protocolos de los servicios Web.
- **XML**. Es el formato estándar para los datos que se vayan a intercambiar. Los datos en XML también pueden enviarse de una aplicación a otra mediante protocolos normales como HTTP, FTP o SMTP.
- **SOAP** (Simple Object Access Protocol). Protocolo sobre el que se establece el intercambio de datos.
- **WSDL** (Web Services Description Language). Es el lenguaje de la interfaz pública para los servicios Web. Es una descripción basada en XML de los requisitos funcionales necesarios para establecer una comunicación con los servicios Web.
- **UDDI** (Universal Description, Discovery and Integration). Protocolo para publicar la información de los servicios Web. Permite comprobar qué servicios web están disponibles.

[2] http://www.w3c.es/

- **WS-Security** (Web Service Security). Protocolo de seguridad aceptado como estándar por OASIS (Organization for the Advancement of Structured Information Standards). Garantiza la autenticación de los actores y la confidencialidad de los mensajes enviados.

Como ventajas de los servicios web podemos descartar; la interoperabilidad que aporta entre aplicaciones independientemente de las plataformas sobre las que se instalen, la fomentación de protocolos basados en texto, que hacen más fácil acceder a su contenido y entender su funcionamiento así como soportar el concepto de servicio.

La principal desventaja de este *middleware* es su bajo rendimiento comparado con otras alternativas similares como pueden ser CORBA e ICE. Otras desventajas pueden ser su falta de madurez, la falta de soporte para orientación a objetos, falta de estandarización de las APIs y al apoyarse en HTTP, pueden esquivarse las medidas de seguridad basadas en *firewall*.

Java RMI

La arquitectura **RMI (Remote Method Invocation)** [Boo96] permite que una aplicación se comunique con objetos que residen en programas ejecutados en máquinas remotas. Por consiguiente, en lugar de crear un objeto, el programa delega el objeto remoto con un representante local, más conocido como *stub*. Los mensajes dirigidos al objeto remoto se envían al stub local, como si se tratase de otro objeto local. El *stub* gestiona los mensajes que se le envíen, y a su vez, los delega al objeto remoto, el cual ya puede acceder a sus métodos. El resultado de la invocación de los métodos en el objeto remoto se envía de regreso al *stub* local, que los remite al emisor original de la llamada (véase figura 3.20).

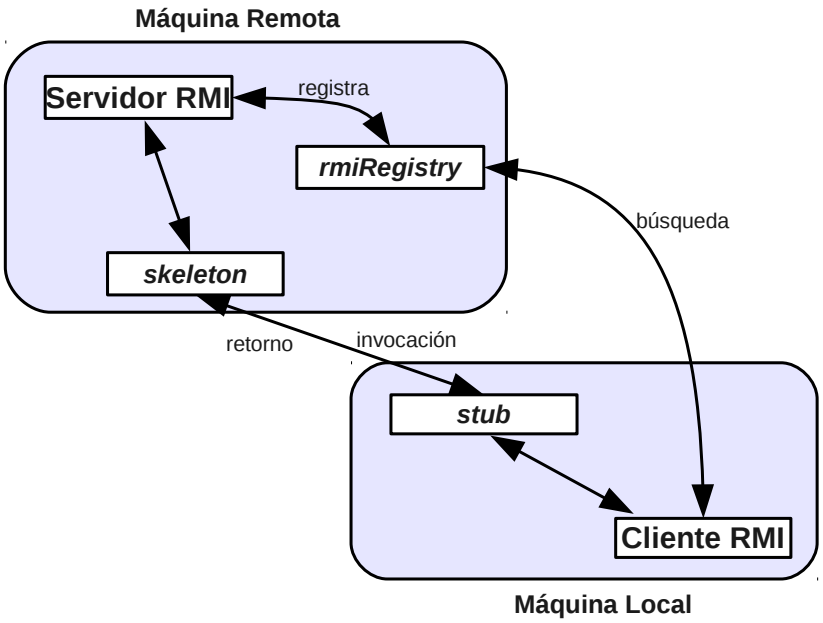


Figura 3.20: Arquitectura de RMI (Remote Method Invocation).

Las aplicaciones RMI normalmente comprenden dos programas separados: un servidor y un cliente. Una aplicación servidor típica crea los objetos remotos, hace accesibles unas referencias a dichos objetos remotos, y espera a que los clientes llamen a estos métodos u objetos remotos. Típicamente, una aplicación cliente obtiene una referencia remota de uno o más objetos remotos en el servidor y llama a sus métodos. RMI proporciona el mecanismo por el que se comunican y se pasan información del cliente al servidor y viceversa.

Para comprender mejor esta arquitectura a continuación se listan los pasos de comunicación que surgen en la una llamada RMI:

- El servidor debe registrar su nombre en la utilidad de registro, conocida como *rmiregistry*.
- El cliente debe buscar el nombre del servidor en la utilidad de registro con el fin de obtener una referencia del objeto remoto.
- Un cliente invoca a un método remoto, la invocación es redirigida primero al *stub*.
- El *stub* es el responsable de transmitir la invocación remota hacia el *skeleton*, que está en el lado del servidor. Para ello el *stub* abre una conexión (mediante un *socket*) con el servidor remoto, empaqueta los parámetros de la invocación y la redirige a través del flujo de datos hacia el skeleton.
- El *skeleton* posee un método que recibe las llamadas remotas, desempaqueta los parámetros e invoca a la implementación real del objeto remoto.

Capítulo 4

Método de Trabajo

En este capítulo se detalla el ciclo de vida software utilizado durante el desarrollo de Amaavem. En el Capítulo 6 se describen más extensamente las iteraciones del desarrollo, junto a la complejidad y resultados de cada una de ellas.

4.1. Metodología de trabajo

La metodología de desarrollo elegida fue la que sigue un *prototipado evolutivo*. Fue elegida esta metodología por los siguientes motivos:

- Permite desarrollar un sistema de manera incremental, permitiendo al desarrollador sacar ventaja de lo que se ha aprendido a lo largo del desarrollo anterior.
- No es necesario que los requisitos sean fijados antes de empezar el desarrollo y pueden ir variando a la largo del mismo.
- Ofrece una planificación sencilla.
- Involucración total del cliente para la refinar los prototipos funcionales de cada iteración (en este caso, el director del proyecto David Vallejo Fernández).
- Se generan prototipos funcionales con más frecuencia que otras alternativas.

Este modelo se ajusta bien al ámbito del proyecto Amaavem ya que el desarrollador y el director del proyecto están en continuo contacto, mejorando y refinando los prototipos funcionales que se desarrollan en cada iteración. A medida que se van refinando los prototipos, pueden surgir nuevos requisitos y tras un breve consenso de aprobación entre los miembros que trabajan en el proyecto, dichos requisitos pueden ser implementados en otra nueva iteración.

En la figura 4.1 se puede apreciar un esquema de un modelo de esta metodología. En el Capítulo 6 se detallan las iteraciones y planificaciones que se siguieron para construir Amaavem.

4.2. Herramientas

A continuación se enumeran y detallan los recursos software y hardware empleados durante el desarrollo de Amaavem, indicando las versiones que se emplearon.

4.2.1. Lenguajes de Programación

Para la implementación de Amaavem se han empleado los siguientes lenguajes:

- **C++:** Es el lenguaje principal en el desarrollo del proyecto. Se ha optado por *C++* porque es un lenguaje de programación eficiente para poder desarrollar aplicaciones distribuidas, está orientado a objetos permitiendo una capa de abstracción, encapsulación y modularización en la implementación. Previamente se adquirieron los conocimientos necesarios del lenguaje para poder desarrollar el proyecto, con ayuda del libro [Eck04].
- **Slice:** Es el lenguaje de especificación para ICE empleado para definir las interfaces de los objetos distribuidos. Es el mecanismo de abstracción fundamental para separar las interfaces de los objetos de su implementación. *Slice* establece un contrato entre el cliente y el servidor que describe los tipos y las interfaces de los objetos utilizados por la aplicación.
- **Python:** Es el lenguaje utilizado para realizar pruebas sobre los resultados obtenidos por el sistema. Se eligió *Python* para llevar a cabo esta tarea porque es un lenguaje que ofrece muchas facilidades en la modificación y creación de imágenes (usando *PyGame*). Debido a que seguir la traza del sistema de manera textual resultaba complicado,

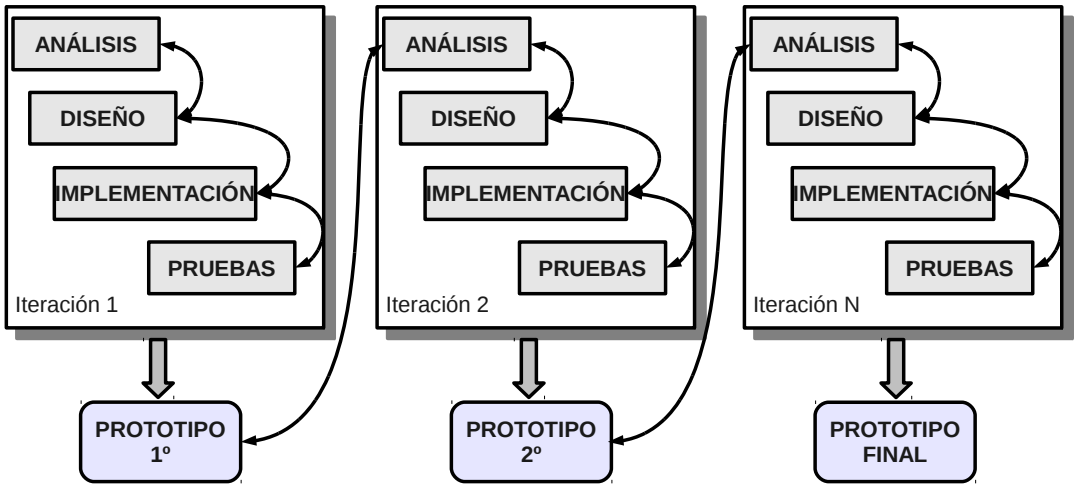


Figura 4.1: Esquema de una metodología iterativa e incremental.

se optó por crear un módulo usando *Python* que creaba vídeos con información adicional a los existentes y así poder determinar si el programa está arrojando resultados idóneos.

4.2.2. Hardware

El desarrollo de Amaavem se realizó con dos equipos, que se detallan a continuación:

- Computador personal portátil *HP NC6400* con un procesador Intel Core 2 T5500 y una memoria RAM de 2GB.
- Puesto instalado en el laboratorio de Oreto, de la Escuela Superior de Informática de Ciudad Real, en concreto, de un equipo que cuenta con un *Zotac*.
- Otros puestos del laboratorio Oreto, para la realización de las pruebas en varios nodos. También, son equipos *Zotac*.

Para el control de versiones se disponía del servicio que ofrece la empresa *Google Inc*, *Google Code*[1] , empleando *Mercurial*[2] para la administración del mismo.

4.2.3. Software

A continuación se listan todas las herramientas y bibliotecas software utilizadas para el desarrollo de este proyecto.

Sistema operativo

- **Ubuntu:** Fue el sistema operativo principal y se ha utilizado *Ubuntu* 11.04, al ser una de las distribuciones de *GNU/Linux* más estables.
- **Debian:** Es el sistema instalado en el puesto del laboratorio Oreto, concretamente una versión *Debian testing* .
- **Windows XP:** para probar la aplicación en plataformas *Microsoft Windows* se utilizó *Windows XP SP2*.

Software de desarrollo

- **Emacs:** Se usó este editor tan popular por ser un editor de texto muy potente,que ofrece muchas funcionalidades. Se ha utilizado la version *Emacs* 23.2.
- **GCC:** El compilador de *C++* empleado ha sido *g++* (de la familia de compiladores *gcc*), en su versión 4.4.5-1.
- **GDB:** Fue usado para depurar fragmentos de código. Es el depurador más popular usado en *GNU/Linux*. Se ha utilizado la versión 7.0.1.

[1] `http://code.google.com/p/amaavem/`

[2] *Mercurial* es un sistema de control de versiones multiplataforma, para desarrolladores de software.

- **Make:** Fue usado para crear el sistema de *Makefiles* de compilación que simplifica la compilación del proyecto. Se ha utilizado la versión es la 3.81.
- **IceGridAdmin:** Es la interfaz gráfica que ofrece *ZeroC Ice* para crear o desplegar fácilmente las aplicaciones distribuidas. Se ha utilizado la versión 3.4.1
- **Mysql-Admin:** Fue usado para crear los esquemas de la base de datos usados por el sistema Amaavem. Se ha utilizado la versión 5.0.
- **Mysql-Query-Browser:** Fue usado para realizar consultas *SQL* en la base de datos. Se ha utilizado la versión 5.0.
- **FFMPEG:** *FFmpeg* es una colección de software libre que puede grabar, convertir y hacer *streaming* de audio y vídeo. Fue usado para la composición de vídeos a partir de un conjunto de imágenes y para la descomposición de un vídeo en imágenes. Fue usado en su versión 5.0.

Documentación y gráficos

- **LibreOffice Draw:** Herramienta de dibujo vectorial que fue empleada para crear las figuras incluida en la presente documentación. Se ha utilizado la versión 3.2.
- **Dia:** Programa para generar todo tipo de diagramas, y fue utilizado para desarrollar los diagramas necesarios de la fase de análisis. Se ha utilizado la versión 0.97.
- **LibreOffice Calc:** Herramienta de creación y manipulación de hojas cálculo que fue utilizada para la generación de las estadísticas incluidas en el capítulo de análisis de resultados. Se ha utilizado la versión 3.2.
- **Latex:** Herramienta para generar documentos con aspecto técnico, usada para la generación de la presente documentación. Se ha empleado la versión 2009 de *TexLive*.
- `http://www.websequencediagrams.com/` Página web empleada para generar los diagramas de secuencia.

Bibliotecas

- **OpenCV:** Biblioteca que se ha empleado para hacer uso del *tracker* de vídeo auxiliar, en su versión 2.1. Para instalar la librería es necesario descargar los siguientes paquetes: *libcv2.1, libcv-dev, libcvaux-dev, libhighgui-dev*.
- **MySQLClient:** Librería necesaria para poder interactuar con la base de datos. Se empleó *MySQL* para realizar las consultas *SQL* debido a su velocidad, robustez y facilidad de uso. Se ha utilizado la versión 5.1.49. Para instalar la librería es necesario descargar los siguientes paquetes: *libmysqlclient16, mysql-common, libmysqlclient-dev*.
- **Build-Essential:** Es un contenedor de programas necesarios para poder crear archivos binarios. Incluye programas tales como: *gcc*+,*g*++,*make*,etc. Se ha utilizado en su

versión 11.5. Para instalar los programas es necesario descargar el siguiente paquete: *build-essential*.

- **Glibmm:** Es la interfaz *C++* oficial para la popular biblioteca de interfaz gráfica GTK+, en su versión 2.24.2. Se empleó para la gestión de hilos de los agentes así como la gestión de tiempo ocioso de los agentes empleando un *Timer* que provee la propia librería. Para instalar la librería es necesario descargar los siguientes paquetes: *libglibmm-2.4-dev, libglibmm-2.4-1c2a*.
- **ZeroC-ICE:** Es un moderno *middelware* para la computación distribuida con soporte para *C++,. NET, Java, Python, Objective-C, Ruby, PHP y ActionScript*. *ICE* se utiliza en proyectos muy importantes en muchas empresas de todo el mundo. Se empleó para este proyecto para alcanzar que el sistema fuera distribuido usando su versión 3.4. Para instalar el *middleware* es necesario descargar el siguiente paquete: *zeroc-ice34*.
- **Python-MySQLdb:** *MySQLdb* es una interfaz con el servidor de base de datos *MySQL* para *Python*. Fue usada para implementar las pruebas de vídeo, concretamente para acceder a la información de los objetos almacenada en la base de datos. Se ha utilizado la versión 5.1.49. Para instalar la librería es necesario descargar el siguiente paquete: *python-mysqldb*.
- **Python-PyGame:** Un kit de desarrollo de multimedia para *Python*. *Pygame* ofrece una cantidad de módulos para realizar múltiples acciones. En concreto, se empleó para modificar las imágenes de los *frames* de los vídeos de vigilancia para posteriormente, poder montar un vídeo modificado con los nuevos *frames*. Se ha utilizado la versión 1.8.1. Para instalar la librería es necesario descargar el siguiente paquete: *python-pygame*.

Capítulo 5

Arquitectura

En este capítulo se describe la arquitectura *multi-agente, adaptativa, modular* en la que está basada Amaavem. Esta arquitectura se explica siguiendo un enfoque *Top-Down*, en el que se comienza con una explicación funcional de cada modelo y componente hasta llegar a un nivel de detalle más especifico, abordando los aspectos más complejos y debatiendo las decisiones de diseño e implementación finales. De esto modo, se explica cada módulo independientemente, mostrando las relaciones con otros módulos de la arquitectura.

Cabe recordar que el objetivo de Amaavem es ofrecer una arquitectura, a los desarrolladores finales, para diseñar y desplegar un sistema de vigilancia inteligente en un entorno monitorizado.

La arquitectura está compuesta por cinco módulos principales que se enumeran a continuación (véase figura 5.1):

- **Módulo Generador de Eventos:** es el módulo encargado de generar los eventos a partir de los datos almacenados en archivos externos (una base de datos, un fichero XML, un fichero de texto, etc) para su posterior análisis. Se define evento como la información relevante proveniente de una fuente de información externa tal como una cámara de videovigilancia, un micrófono, un sensor de temperatura, un sensor de presión, etc. Una vez generados dichos eventos, serán enviados al monitor de vigilancia para su procesado.
- **Módulo Monitor de Vigilancia:** es el módulo encargado de procesar los eventos que llegan del generador de eventos. En caso de que en los eventos aparezcan objetos que antes no aparecían, es decir, objetos detectados por primera vez, este modulo será el encargado de avisar al módulo de generación de agentes para instanciar nuevos agentes. Asimismo, es el encargado de mandar los eventos a los agentes instanciados en todo el sistema empleando un canal de eventos implementado por un servicio de Ice, en concreto, el servicio de publicación-subscripción conocido como *IceStorm*.
- **Módulo Creación de Agentes:** es el módulo encargado de realizar la instanciación de los nuevos agentes en el sistema empleando bibliotecas dinámicas del sistema. Otra de sus funciones es comprobar que no haya agentes *zombies* activos en el sistema.

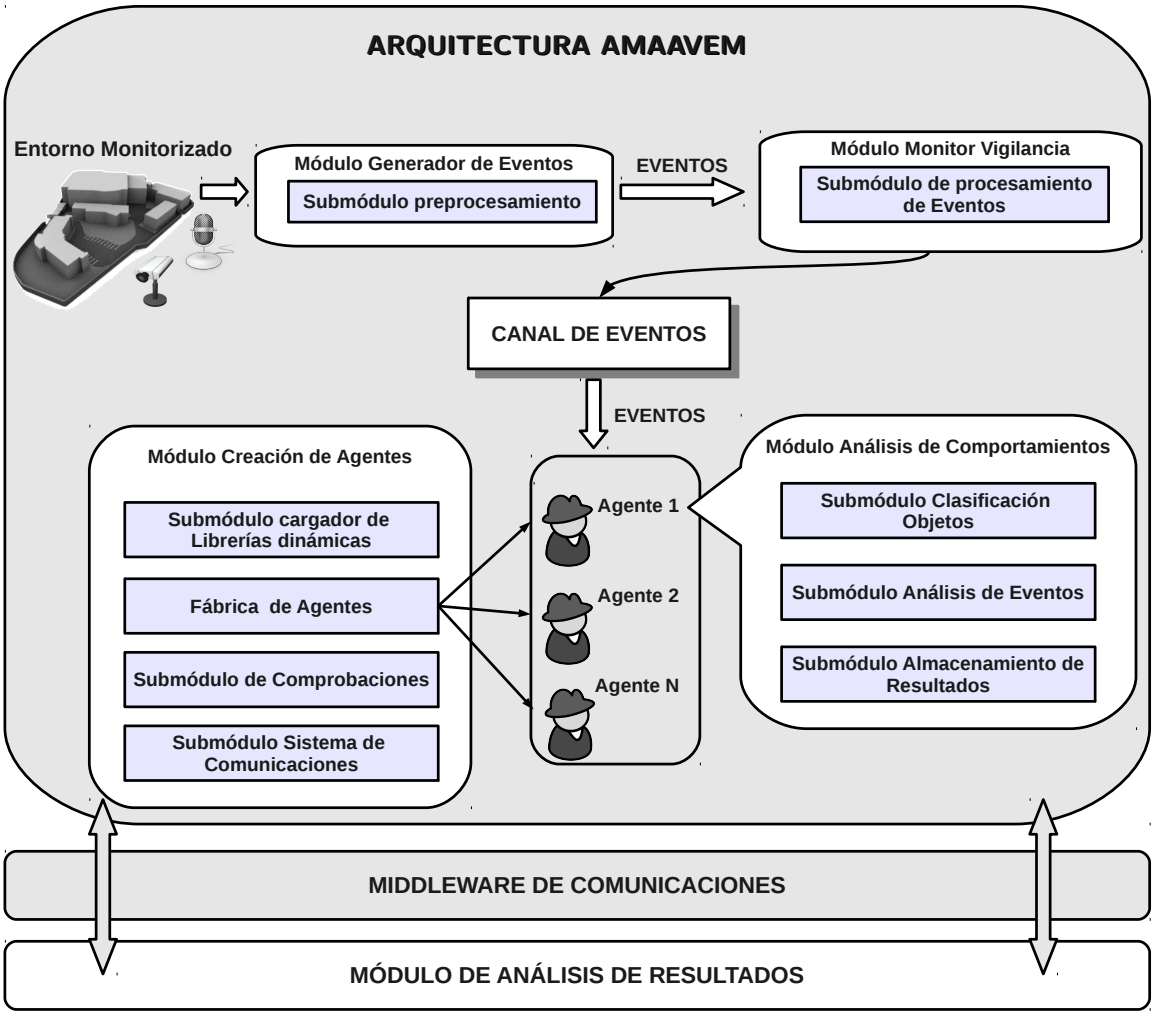


Figura 5.1: Esquema de la arquitectura de Amaavem.

- **Módulo Análisis de Comportamientos:** es el módulo encargado de realizar el análisis inteligente de los eventos. En este módulo también se lleva a cabo la clasificación de los objetos que aparecen en los eventos en un tipo concreto del dominio a monitorizar. Los resultados obtenidos son volcados en una base de datos auxiliar.
- **Módulo Análisis de Resultados:** es el módulo encargado de generar los informes de resultados tras el análisis de los eventos indicados por el usuario.

5.1. Descripción general

La arquitectura de Amaavem, por un lado, fue diseñada siguiendo una estructura adaptable y modular para que fuera fácilmente extensible. Por otro lado, dicha arquitectura está apoyada en el *middleware* de comunicaciones *ZeroC Ice* para conseguir un sistema distribuido. Se eligió *ZeroC Ice* debido a que ofrece una solución sencilla a las comunicaciones en una aplicación distribuida.

A continuación se detallan algunas de las **características** más destacadas de Amaavem:

- **Escalabilidad**. Se integran soluciones escalables a nivel de comunicación y despliegue de agentes. El sistema resultante es un sistema cómodamente escalable que puede dar soporte a tantos usuarios como sea posible. Se emplean técnicas distribuidas para poder analizar la mayor cantidad de objetos en una misma escena sin perder rendimiento.
- **Portabilidad**. La arquitectura se ha implementado usando bibliotecas portables y libres. Amaavem puede funcionar en tantas plataformas como sean soportadas por sus bibliotecas. Actualmente, la arquitectura soporta los sistemas operativos *GNU/Linux* y *Microsoft Windows*.
- **Adaptabilidad**. La arquitectura está dividida en módulos. Cada modulo ha sido pensado para que sea lo más independiente posible, con unas responsabilidades bien acotadas consiguiendo que el sistema gane en cohesión y no sufra de acoplamiento. Esto permite que se pueden agregar nuevas funcionalidades al sistema de manera sencilla modificando o agregando nuevos módulos. Para ello, se han empleado patrones de diseño que resuelven problemas típicos en la creación de Software.
- **Independencia del dominio**. La arquitectura es flexible a cualquier entorno monitorizado debido al diseño empleado en este proyecto. Para probar Amaavem, se ha realizado un caso de estudio en un dominio urbano. La arquitectura permite que el desarrollador tenga plena libertad para definir, no sólo los comportamientos propios de los objetos del dominio concreto sino también sus relaciones.
- **Tolerancia a fallos**. El sistema resultante está implementado por una arquitectura tolerante a fallos capaz de seguir trabajando en caso de fallos, como por ejemplo el hardware de soporte.
- **Balanceado de carga**. La arquitectura cuenta también con técnicas distribuidas que dan cabida a más usuarios definiendo una serie de servidores que trabajen conjuntamente en la resolución del problema.

Los **patrones de diseño** son una referencia sólida para solucionar problemas comunes en el desarrollo de software [GHJV94]. Por ello, Amaavem emplea patrones de diseño para resolver muchos de los problemas que surgen al desarrollar un software. En concreto, los patrones de diseño que se emplean en Amaavem son principalmente los siguientes:

- ***Abstract Factory***. Empleado para la instanciación de los diversos tipos de agentes.
- ***Singleton***. Empleado para la implementación de un único *Logger*.
- ***Adapter***. Empleado por *Ice* para transformar una interfaz en otra.
- ***Facade***. Empleado para acceder a los métodos de las clases especificadas en *Slice*.
- ***Proxy***. Empleado para mantener representantes de objetos *Ice*.

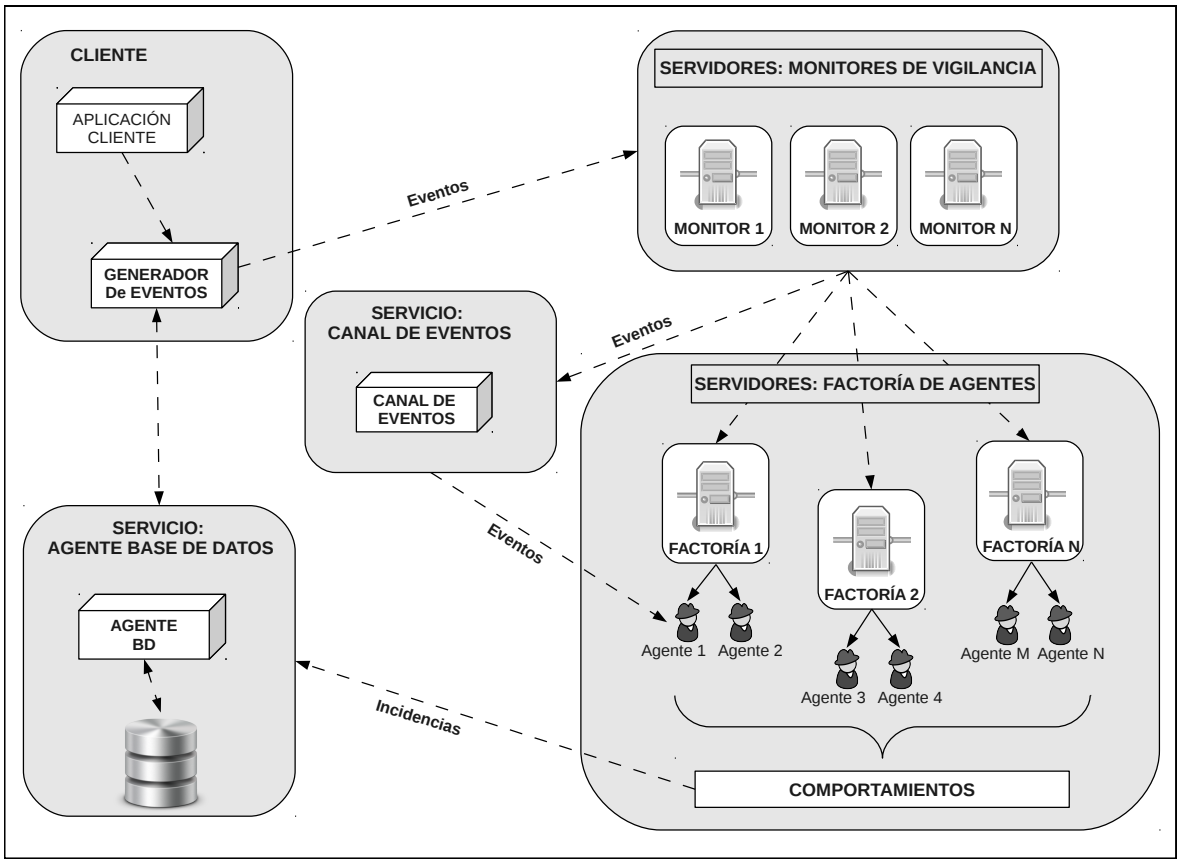


Figura 5.2: Diagrama de componentes de Amaavem

- ***Iterator***. Empleado para recorrer objetos.

Para comprender mejor el diseño de Amaavem, se incluye un **diagrama abstracto** de sus componentes así como sus relaciones entre ellos (véase figura 5.2). Cada servidor y servicio puede estar alojado en *hosts* diferentes para conseguir que el sistema sea altamente escalable y fácilmente extensible. Para que la aplicación sea capaz de localizar los distintos servicios y servidores, se emplea uno de los servicios más potentes implementados por *ZeroC Ice*, en concreto, el servicio de localización llamado **IceGrid**. Con *IceGrid*, se hace relativamente sencilla la localización de los diferentes servidores desplegados por la aplicación.

Seguidamente, se hace necesario explicar las distintas estructuras de datos empleadas por Amaavem. Existe un archivo de configuración (*config.ice*) definido en lenguaje *Slice*,[1] en el cual se definen las distintas constantes, estructuras, clases e interfaces. En la figura 5.3, se puede apreciar un **diagrama de las estructuras de datos** empleadas en Amaavem. Cabe citar que la expansibilidad de la arquitectura es relativamente sencilla, simplemente añadiendo nuevos campos a las estructuras ya existentes o añadiendo nuevas estructuras.

[1]Las interfaces, las operaciones, y los tipos de datos intercambiados entre clientes y el servidores se definen utilizando el lenguaje de definición de interfaces propuesto por ZeroC Ice, llamado Slice.

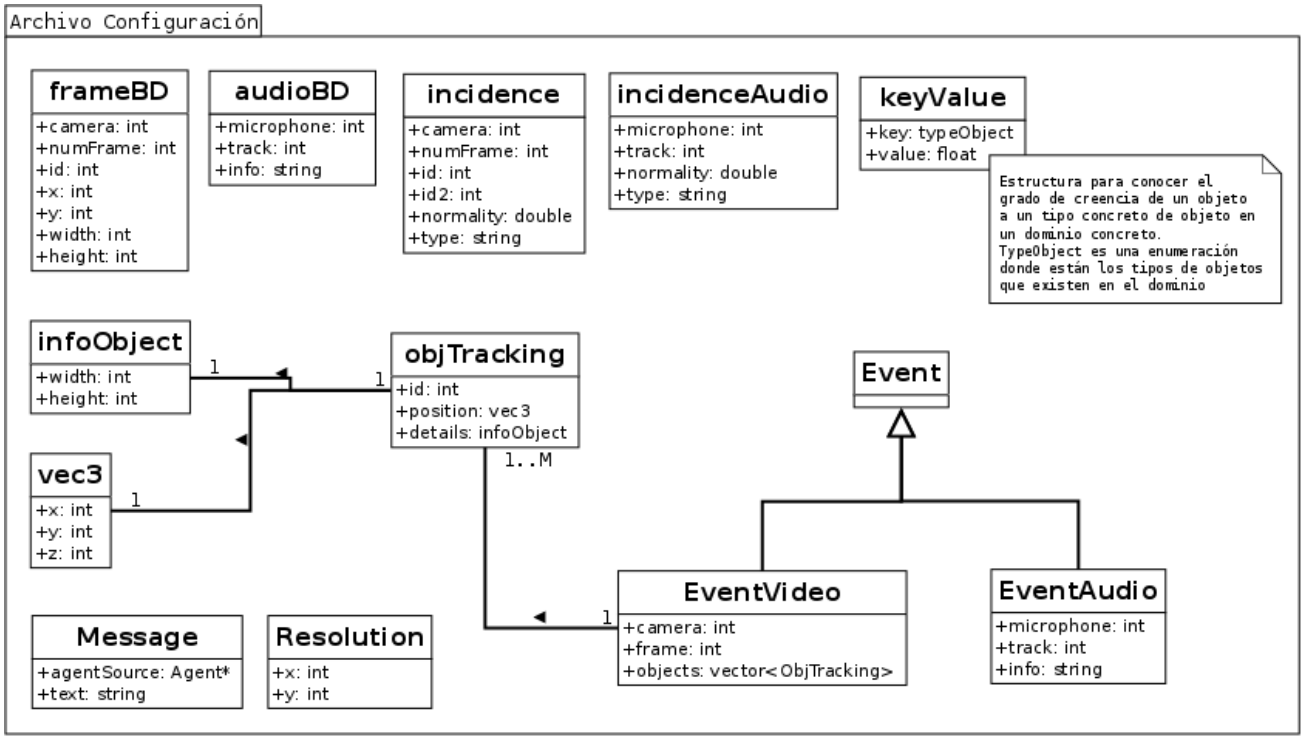


Figura 5.3: Diagrama del archivo de configuración de Amaavem

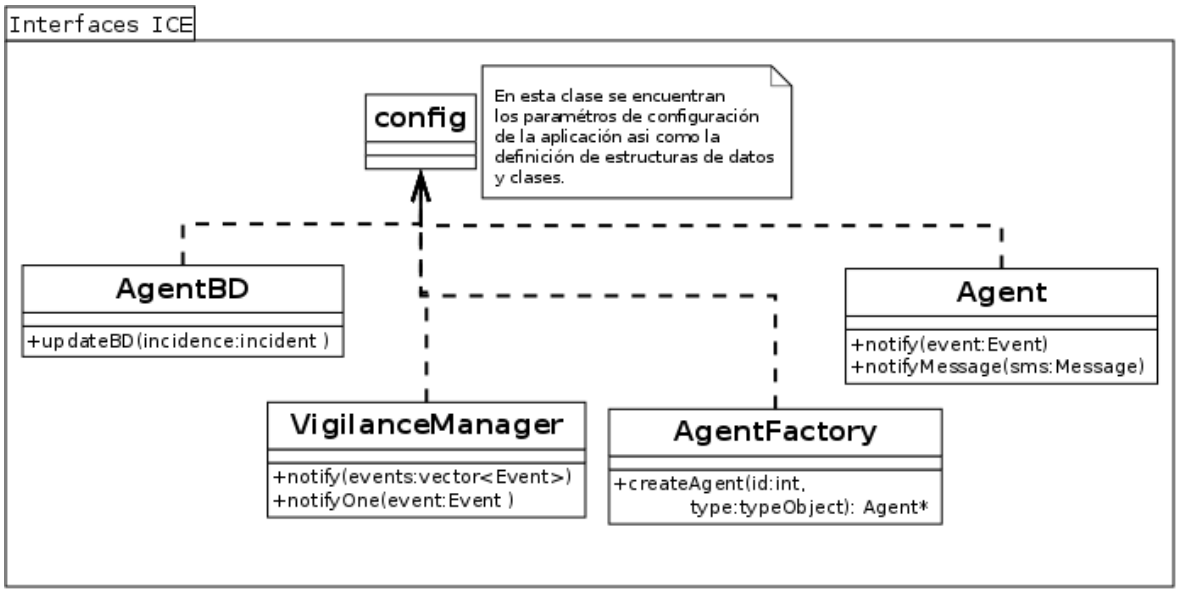


Figura 5.4: Diagrama de las interfaces de Ice para Amaavem.

Asimismo, es interesante mostrar las interfaces que se han definido con *Slice* para la arquitectura propuesta (véase figura 5.4 y anexo E). Las clases especializadas serán las encargadas de implementar dichas interfaces. *Slice* es el lenguaje de definición de interfaces de *ZeroC Ice*. Las definiciones *Slice* se compilan por un compilador a una API para un lenguaje de programación especifico, es decir, la parte de la API que es específica a las interfaces y los tipos que previamente han sido definidos, formando el código generado. Para conseguir el *mapping* de *Slice* a C++ se usó el compilador *slice2cpp*. Las implementaciones de las interfaces *Ice* son realizadas por las clases especializadas, tal como se muestra en el **diagrama de clases** de la arquitectura Amaavem (véase la figura 5.5).

En la figura 5.6 se muestra un **diagrama de secuencia** para comprender mejor como fluye la información a través del sistema. La información que fluye son los propios eventos, y pasan del generador de eventos al monitor de vigilancia. El monitor de vigilancia los procesa, y los envía al canal de eventos, encargado de suministrar los mismos a todos los agentes del sistema instanciados. Dichos agentes, procesan los eventos y determinan el grado de normalidad.

Este proyecto está escrito en C++ en casi su totalidad. Se decidió implementar la arquitectura en C++ debido a las siguientes ventajas que ofrece este lenguaje:

- **Portabilidad**: es un lenguaje estandarizado y un mismo código puede ser compilado para diferentes plataformas.
- **Eficiencia**: es un lenguaje muy potente en la ejecución de problemas complejos.
- **Versatilidad**: es un lenguaje de propósito general, por lo que se puede emplear para resolver cualquier tipo de problema.
- **Popularidad**: es uno de los lenguajes más empleados en la actualidad y existe gran cantidad de información, compiladores, depuradores y librerías para su uso.
- **Adaptabilidad**: es un lenguaje fácilmente adaptable a *middlewares* de comunicación, y en concreto a *ZeroC Ice*.

Para que el desarrollador final, que emplee Amaavem en la creación de un sistema de vigilancia inteligente, pueda configurar, utilizar y desplegar este proyecto se incluyen una serie de anexos en la parte final del documento:

- **Manual de Usuario**: Anexo A.
- **Guía de Desarrollo**: Anexo B.
- **Archivos de configuración**: Anexo C.
- **Esquema de la base de datos**: Anexo D.
- **Interfaces de comunicaciones**: Anexo E.

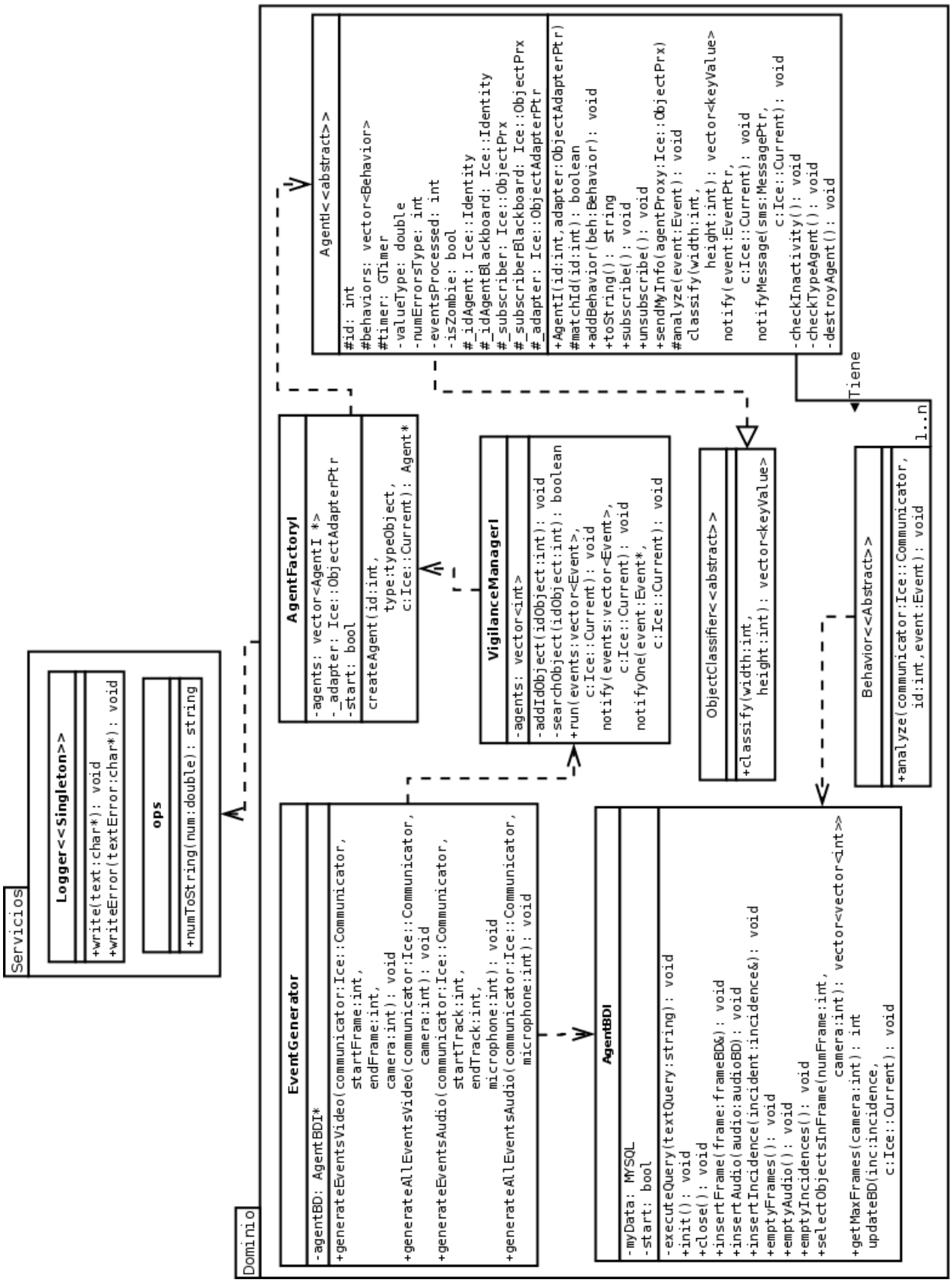


Figura 5.5: Diagrama de clases de Amaavem.

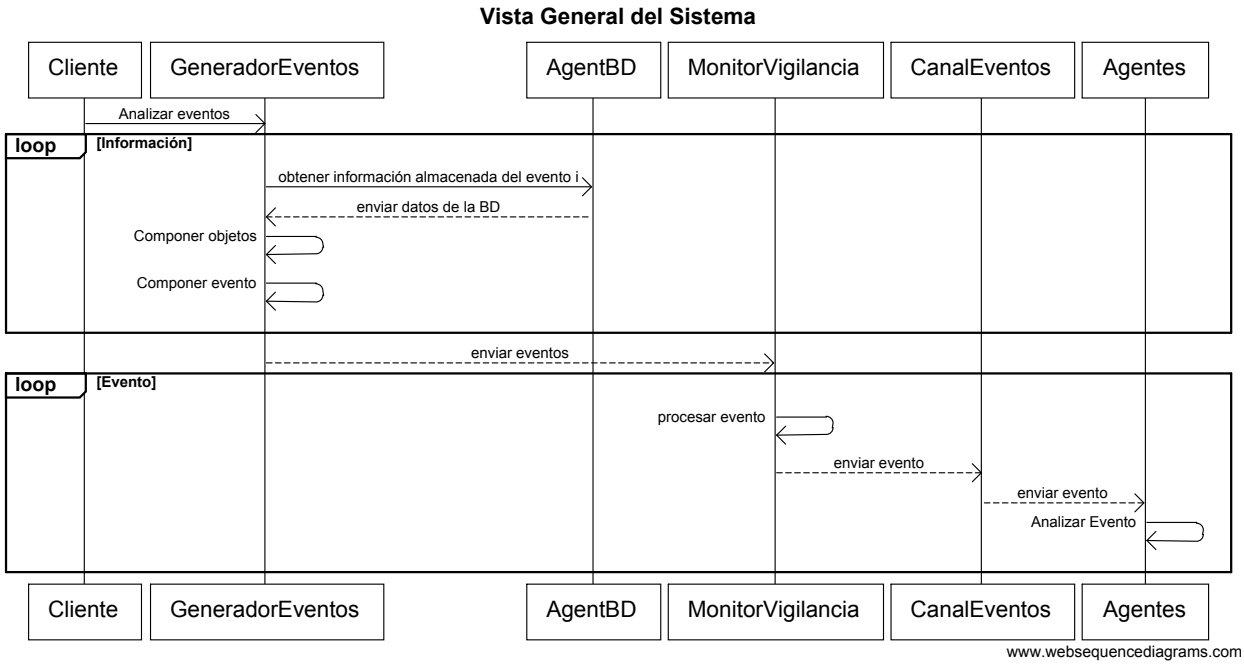


Figura 5.6: Diagrama de secuencia general de Amaavem.

En las próximas secciones se realizará un análisis en profundidad de cada uno de los módulos que componen Amaavem (véase figura 5.1), en primer lugar realizando una descripción funcional de cada módulo/submódulo, seguidamente detallando dependencias y requisitos con otros módulos y finalmente, analizando detalles de implementación relevantes.

5.2. Módulo Generador de Eventos

Este módulo es el encargado de generar los eventos, ya sean de audio, de vídeo o de otro tipo definido en el archivo de configuración. Dichos eventos son generados a partir de una información, proveniente de fuentes heterogéneas, tal como puede ser un sistema de *tracking* auxiliar de videovigilancia, un archivo de *XML*, un micrófono, un sensor, etc.

Este módulo puede ser empleado para enviar eventos por un lado, en tiempo real, utilizando la información recogida por algún sistema de *tracking* auxiliar, y por otro lado, de manera *offline*, extrayendo la información de una base de datos. El desarrollador final debe determinar como va a emplear la arquitectura y actuar en consecuencia. Si va a extraer información en tiempo real, se recomienda guardar la información en formato *XML*, ya que se ofrece también un parseador de datos *XML* a base de datos.

Dicho módulo está implementado por la **clase *EventGenerator***, que tiene una dependencia funcional de la clase *AgentBDI* (el broker de la base de datos encargado de realizar las consultas *SQL* y de abstraer las operaciones *CRUD* a los demás elementos del sistema). Cuando se instancia un generador de eventos, éste instancia a su vez, un agente de base de datos que

realiza las consultas necesarias, para obtener la información de la base de datos en la creación de los eventos. La clase *AgentBDI* es una especialización de la clase *AgentBD* (definida como una interfaz con *Slice*) y es la encargada de implementar la función distribuida *updateBD*, recibiendo como parámetro una incidencia. Esta clase será explicada en profundidad en la sección 5.2.1.

La clase *EventGenerator* tiene una única responsabilidad: brindar al monitor de vigilancia una serie de eventos, procesados a partir de la información externa al sistema. Dicha clase tiene cuatro métodos fundamentales, definidos como sigue a continuación:

```
1   void generateEventsVideo(Ice::Communicator& communicator,
2                            const int& startFrame,
3                            const int& endFrame,
4                            const int& camera);

6   void generateAllEventsVideo(Ice::Communicator& communicator,
7                               const int& camera);

9   void generateEventsAudio(Ice::Communicator& communicator,
10                           const int& startTrack,
11                           const int& endTrack,
12                           const int& microphone);

14  void generateAllEventsAudio(Ice::Communicator& communicator,
15                              const int& microphone);
```

Todos los métodos precisan de un parámetro llamado *communicator*. Este objeto es de tipo *Ice::Communicator*, el objeto central de Ice necesario para establecer la comunicación entre aplicaciones. Dicho objeto se emplea típicamente, no sólo para buscar los distintos objetos Ice instanciados en el sistema, sino también para agregar nuevos adaptadores de objetos al sistema. Cuando un cliente precise una generación de eventos, deberá pasar como parámetro un comunicador que entienda dónde está alojado el nodo principal y asimismo, que conozca los objetos que puede invocar.

El primero de ellos genera los eventos, comprendidos entre los *frames* definidos por los parámetros *startFrame* y *endFrame* para una cámara fijada, en caso de utilizar información ya almacenada. El pseudocódigo de este método es el siguiente:

```
1   evento = nuevo evento de video
2   Eventos = []

4   Desde frame inicial hasta frame final hacer
5       evento.frame = frame actual
6       evento.camera = camera indicada
7       eventos.objetos = seleccionar objetos del frame actual
8       Eventos.agregar al final(evento)
9       vaciar evento auxiliar

11  Buscar un Monitor de Vigilancia
12  Enviar Eventos al monitor de vigilancia buscado
```

En la sección 5.2.1, se explica cómo se realiza la selección de los objetos. La búsqueda de un monitor de vigilancia, instanciado en el sistema distribuido, se hace usando las funcionalidades que ofrece *ZeroC Ice*. Para ello, sólo es necesario indicar al comunicador que se necesita un *proxy* de un objeto bien conocido, llamado *vigilancemanager*, tal como sigue en el siguiente extracto de código del método estudiado de la clase *EventGenerator*:

```
1    //Obtain the VigilanceManager
2    VigilanceManagerPrx vigManager;
3    try {
4        vigManager = VigilanceManagerPrx::checkedCast(communicator.
             stringToProxy("vigilancemanager"));
5    }
```

El método *stringToProxy* definido en la interfaz del comunicador, obtiene un *proxy* al objeto remoto. Para ello, recibe como parámetro la identidad del objeto que es contrastada con los archivos de configuración definidos en el despliegue de la aplicación y de esta manera, *IceGrid* se encarga de localizar dicho objeto. Es decir, es como una especie de traducción de un texto a un objeto que está instanciado en una máquina remota. Posteriormente, se realiza una conversión del *proxy* obtenido por el método *stringToProxy*, que es de tipo *Ice::ObjectPrx* a un *proxy* del tipo *VigilanceManagerPrx* gracias a la operación *checkedCast*, que ofrece el *middleware*. Para enviar el evento al monitor de vigilancia, tan sólo es necesario invocar algunas de las operaciones definidas por dicho objeto. En concreto, se invoca a la operación *notify*, que precisa de un parámetro (un vector de eventos).

El segundo método permite generar todos los eventos de vídeo de la cámara indicada previamente parámetro. Los otros dos siguientes, son métodos que tienen las mismas funcionalidades que el explicado con anterioridad, pero para eventos de tipo audio.

5.2.1. Submódulo de preprocesamiento.

En el análisis de los comportamientos que suceden en las diferentes situaciones, se hace necesario un preprocesamiento de la información que se maneja en un sistema típico de vigilancia inteligente, ya que se emplean fuentes de información heterogéneas.

Para ello, Amaavem emplea una **base de datos relacional**, donde se almacena la información que emiten las diversas fuentes (cámara de videovigilancia, un micrófono, un sensor, etc). En la figura 5.7, se muestra un esquema de las tablas empleadas por el sistema Amaavem. En el anexo D está el código en *SQL* para generar las tablas expuestas previamente.

La **clase *AgentBDI*** es la encargada de implementar las funciones *CRUD*, que son las operaciones básicas de un almacenamiento persistente. Estas operaciones se explican en la tabla que sigue a continuación:

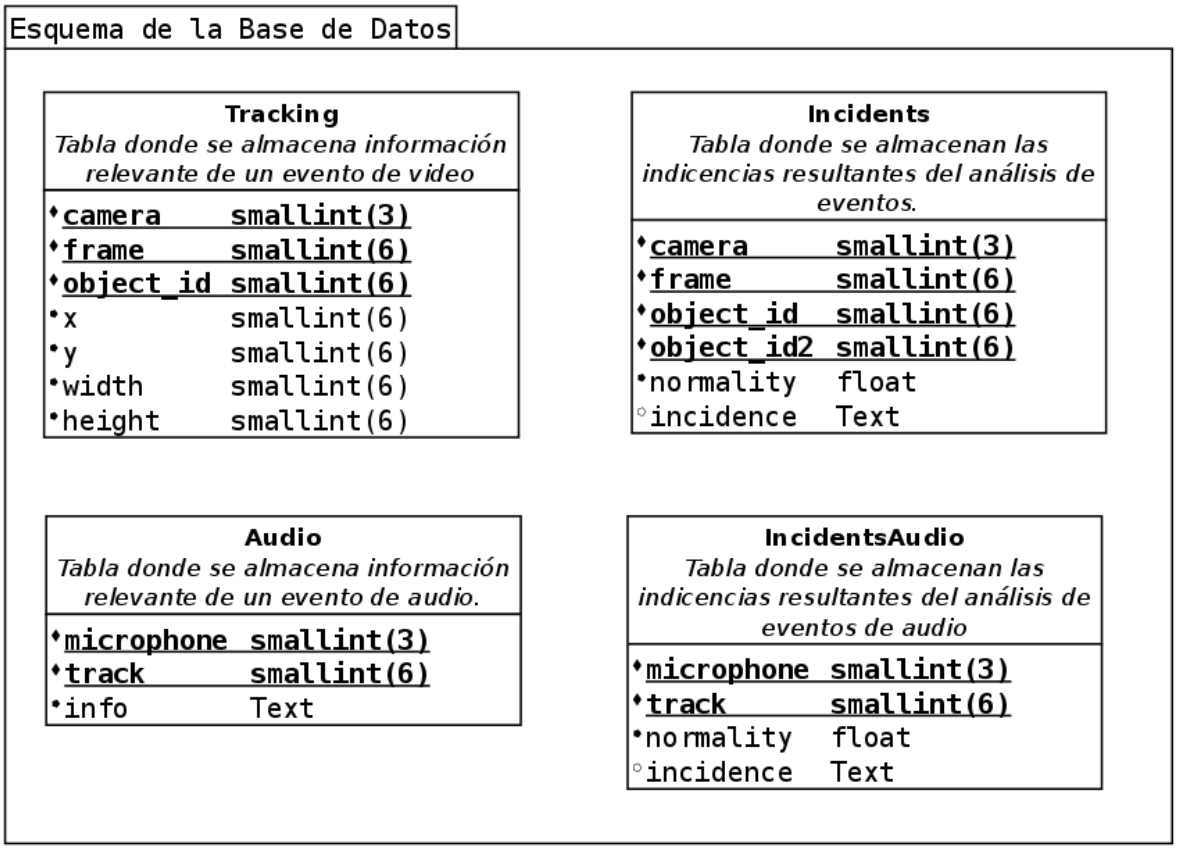


Figura 5.7: Esquema de la base de datos.

Operación	SQL	Amaavem
Create	INSERT	Insertar eventos o incidencias
Recover	SELECT	Seleccionar eventos
Update	UPDATE	Actualizar la base de datos
Remove	REMOVE	Vaciado de las tablas

La interacción con la base de datos se hace posible gracias al uso de la biblioteca que ofrece *MySQL* para *C++*, utilizándose el sistema *MySQL* por ser un sistema de gestión de base de datos consolidado en el área y sobre todo, por ser un sistema con licencia *GNU GPL*.[2]

En el diagrama de clases general del sistema Amaavem (véase figura 5.5) se pueden observar las funcionalidades que ofrece dicha clase. Siguiendo con la línea expuesta en los objetivos planteados en este Proyecto Fin de Carrera, se pretende crear una arquitectura lo más extensible posible. Por ello, se diseñó este módulo como algo totalmente independiente de las demás clases expuestas en la arquitectura. La idea general del sistema es que cada artefacto que componen el sistema sea lo más independiente posible del resto y así lograr

[2]Es una licencia creada por la *Free Software Foundation* en 1989, y está orientada principalmente a proteger la libre distribución, modificación y uso de software.

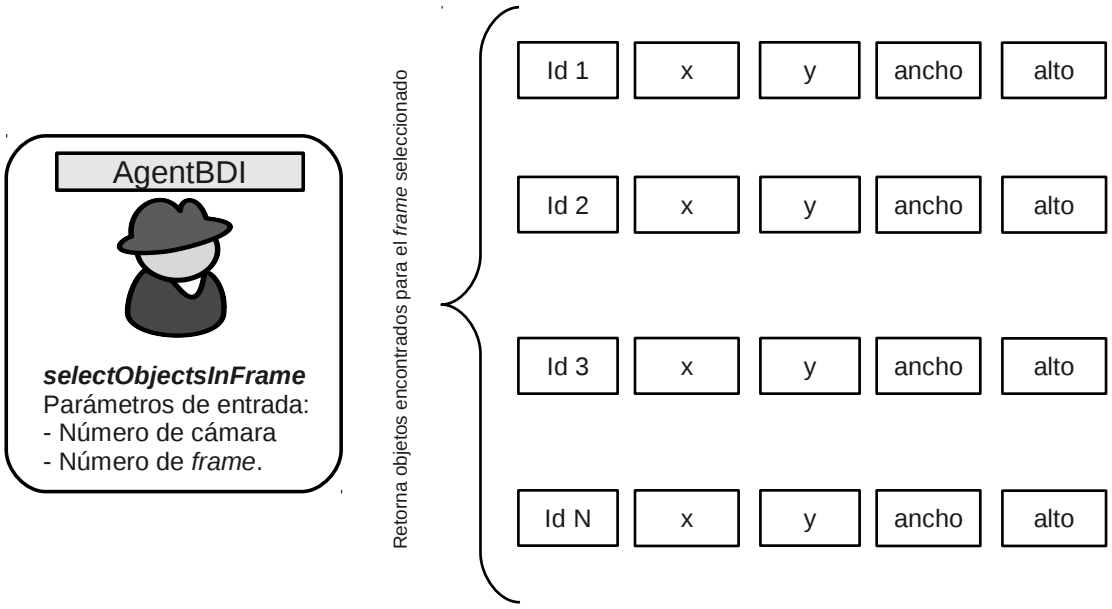


Figura 5.8: Esquema de la selección de objetos en la base de datos.

que el sistema gane en cohesión y no sufra un alto acoplamiento. Se pretende que cada modulo tenga unas responsabilidades bien definidas y no haga más de lo que debe hacer. Por consiguiente, extender las operaciones relacionadas con la base de datos es sencillo. Para agregar o modificar las operaciones relacionadas con la base de datos simplemente se deben implementar los cambios en la clase *AgentBDI*.

En cuanto a la **generación de los eventos de vídeo**, la clase *EventGenerator* solicita al agente de base de datos la información específica de un *frame* determinado. Para ello, el agente de la base de datos tiene un método con la siguiente definición:

```
1 vector< vector<int> > selectObjectsInFrame(const int& numFrame,
2                                            const int& camera);
```

Dicho método, por un lado recibe como parámetro un número de cámara y un número de frame y por otro lado, devuelve un vector de vectores de enteros. Debido a que un evento de vídeo está formado por una cámara, un frame y una lista de objetos, este método sirve para componer esta lista de objetos (véase la figura 5.8 para entender mejor como trabaja este método). Cuando el *EventGenerator* recibe dicha lista de enteros, compone el evento final de vídeo con su lista de objetos ya preprocesados por este submódulo.

Esta clase implementa una de las interfaces Ice definidas con *Slice*, en concreto , la interfaz *AgentBD* (véase figura 5.4), en la cual se define una única operación: *updateBD*, que recibe como parámetro una incidencia. Dicha operación tiene la responsabilidad de insertar la incidencia, recibida por parámetro, en la tabla de incidencias que ha sido definida en el esquema

de la base de datos. Para implementar la operación de actualización de la base de datos se ha hecho uso del servicio ofrecido por *ZeroC Ice*, conocido como *IceBox*. Gracias al empleo de *IceBox*, los servicios se desarrollan como componentes que se cargan dinámicamente y que pueden configurarse a través de un servidor de propósito general, según sea necesario.

En la creación de un servicio *IceBox*, es necesario implementar una de las interfaces *IceBox*. En concreto, se implementa la función *IceBox::Service*, que se muestra a continuación:

```
module IceBox {
    local interface Service {
        void start (string name,
            Ice::Communicator communicator,
            Ice::StringSeq args);
        void stop ();
    };
};
```

Como se puede apreciar, un servicio sólo necesita implementar dos operaciones, que son invocadas por el gestor del servicio. La primera de ellas, *start*, invocada por el servidor para arrancar el servicio, y la segunda, *stop* invocada cuando el servidor *IceBox* paraliza su ejecución.

En concreto, la implementación de dicho servicio se encuentra en el archivo *UpdateBDServiceI*, incluido en la carpeta (*/src*) del proyecto. Las configuraciones necesarias para lanzar el servicio se encuentran en el archivo de configuración *amaavem.xml*, incluido en la carpeta raíz del proyecto.

También puede darse el caso de que los **eventos de vídeo**, que deben ser analizados, se encuentren en un fichero *XML*. Dicho fichero debe ser parseado si se precisa un análisis previo. Así pues, se debe realizar un preprocesamiento del fichero y posteriormente, almacenarlo correctamente en la base de datos, concretamente en la tabla *Tracking*. La responsabilidad de esta labor recayó sobre un pequeño módulo de preprocesado de ficheros *XML* implementado en lenguaje *Python*. Para ello, se definió un archivo (*src/utils/parser.py*) que hace uso de una biblioteca para el parseado de ficheros *XML* con *Python*, en concreto, la biblioteca *minidom*. Dicho fichero parsea la información que se encuentra en el fichero *XML* y la almacena en la base de datos.

Para que el parseador pueda comprender el fichero *XML* correctamente, debe existir un fichero *XML* por cada objeto, que aparece en la escena, y cada fichero debe estar compuesto con las mismas etiquetas empleadas en el ejemplo que sigue en la página siguiente:

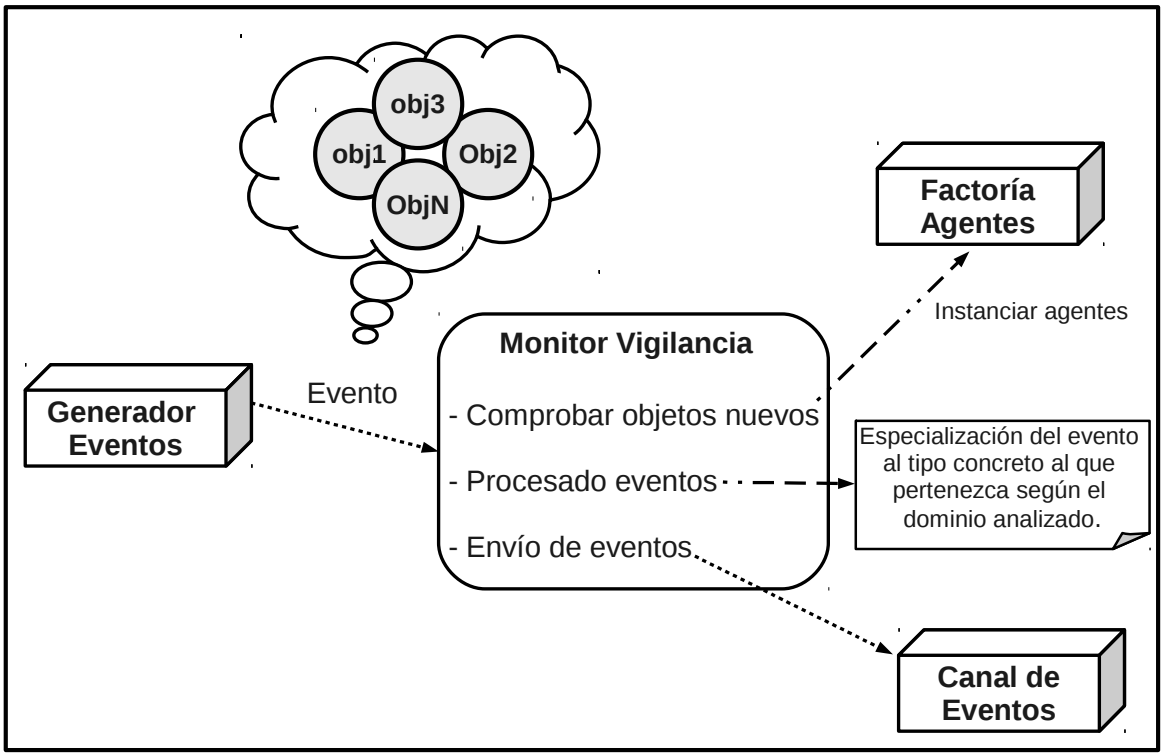


Figura 5.9: Diagrama de interacción del monitor de vigilancia.

```
<JPR_EVENT_SEQUENCE DESCRIPTION="Red-Coat Female" FRAME_CREATED="110"
   FRAME_DESTROYED="653" ID="0">
    <OBJECT TIME="110">
        <OBSERVATION>
            <BOUNDING_BOX BOTTOM="247" LEFT="1" RIGHT="15" TOP="287"/>
            <CENTROID CENTER_X="8" CENTER_Y="267"/>
            <POSITION BASE_X="8" BASE_Y="247"/>
        </OBSERVATION>
    </OBJECT>
    <OBJECT TIME="112">
        <OBSERVATION>
            <BOUNDING_BOX BOTTOM="233" LEFT="1" RIGHT="13" TOP="264"/>
            <CENTROID CENTER_X="6" CENTER_Y="273"/>
            <POSITION BASE_X="6" BASE_Y="253"/>
        </OBSERVATION>
    </OBJECT>
```

5.3. Módulo Monitor de Vigilancia

Este módulo es el encargado de procesar los eventos provenientes del módulo generador de eventos. Cuando se genera un evento del tipo que sea, se debe enviar al monitor de vigilancia correspondiente para que sea procesado y distribuido a los diferentes agentes instanciados en el sistema. El monitor de vigilancia es específico de cada dominio, ya que los eventos

recibidos son de tipo genérico. La función de este monitor de vigilancia es conocer de qué tipo de evento se trata en función del dominio que se esté monitorizando. Por ejemplo, si se monitoriza un paso de cebra, los eventos que se procesan serán eventos de vídeo. En cambio, si se quiere analizar los sonidos de una serie de animales del bosque, los eventos que se procesan serán eventos de audio. Por tanto, una de las responsabilidades de este monitor de vigilancia es la de identificar qué tipo de eventos se está procesando, dependiendo del dominio en el cual ha sido desplegado Amaavem. El desarrollador final ha de tener en cuenta el tipo de dominio en donde va a desplegar Amaavem, para ajustar este monitor de vigilancia, en función a las propias características del entorno.

Entonces, las responsabilidades de este monitor de vigilancia se describen a continuación:

- **Recepción de eventos**. Este módulo es el encargado de recibir los eventos, generados previamente por el módulo de generación de eventos.
- **Procesado de eventos**. Este módulo es el encargado de conocer que tipo de eventos se están tratando, en función del dominio donde se despliegue Amaavem. Realiza una especialización del evento general a un tipo de evento específico.
- **Distribución de eventos**. Este módulo es el encargado de enviar los eventos a todos los agentes instanciados en el sistema, haciendo uso del canal de eventos, implementado con *IceStorm*.

Este módulo está implementado por la **clase *VigilanceManagerI***, incluida en la carpeta (*/src*). Esta clase implementa una de las interfaces de *Ice* definidas previamente, en concreto, la interfaz *VigilanceManager* y sus operaciones *notify* y *notifyOne*. La definición de dicha clase es sencilla y se muestra a continuación:

```
1   class VigilanceManagerI : public VigilanceManager
2   {
3       private:
4           vector<int> idAgents; // list of the ids of agents
5           void addIdObject(const int& idObject);
6           bool searchObject(const int& idObject);
7
8       public:
9           VigilanceManagerI ();
10          ~VigilanceManagerI ();
11          void run(const vEvents& events,
12                   const Ice::Current& c);
13          void printAgents();
14          virtual void notify(const vEvents& events,
15                              const Ice::Current& c);
16          virtual void notifyOne(const EventPtr& event,
17                                 const Ice::Current& c);
18
19  };
```

los métodos *notify* y *notifyOne* son las operaciones remotas de este módulo, que son las encargadas de recibir los eventos procedentes del generador de eventos. Dichos métodos reciben los eventos y los pasan al método *run* para su procesado. Un punto importante en el desarrollo de Amaavem a tener en cuenta ha sido la eficiencia del sistema. Para conseguir la mejora de ésta, los métodos reciben referencias o punteros a memoria en lugar de referencias por valor.

El método *run* es la operación principal de este módulo. Este método es específico de cada dominio y recibe como parámetros una lista de eventos de tipo genérico. Para una mejor compresión del módulo planteado, se incluye un diagrama de secuencia en la figura 5.10 y un diagrama de interacción en la figura 5.9. Las responsabilidades de la operación *run* se enumeran a continuación:

- **Procesamiento de los eventos** incluyendo una conversión adecuada al tipo de evento concreto. Los eventos que se reciben son de tipo específico y es responsabilidad de este módulo convertir el evento al tipo específico, para ello se emplearon los moldes proporcionados por *C++*.
- **Obtención de una factoría de agentes** para la instanciación de nuevos agentes, cuando se detecta un objeto nuevo en el evento. Cada objeto está identificado por un número entero, el monitor de vigilancia mantiene una lista de enteros para comprobar si el objeto pertenece a la lista o no. En el caso de que el objeto se encuentre en dicha lista no es necesario instanciar un nuevo agente porque ya se instanció en su momento. Por el contrario, si el objeto que se está procesando no pertenece a dicha lista, entonces sí se hace necesario una instanciación de un nuevo agente. Para usar una factoría de agentes remota se debe obtener un *proxy* de una factoría desplegada en el sistema tal y como se muestra en el siguiente fragmento de código:

```
1     /* Obtain an AgentFactory */
2     /* Comm is a Ice::Communicator */
3     AgentFactoryPrx agentF ;
4     try {
5         agentF = AgentFactoryPrx::checkedCast(comm->stringToProxy
              ("agentfactory"));
6     }
```

  La instanciación de los agentes es responsabilidad de la fábrica de agentes (véase sección 5.5). El análisis de los eventos es tarea de los agentes instanciados en el sistema existiendo un agente por cada objeto detectado, con lo cual cada agente del sistema tiene como objetivo analizar únicamente un objeto.

- **Envío de los eventos** procesados a los diferentes agentes instanciados en el sistema. Los eventos han sido previamente convertidos al tipo concreto del dominio a monitorizar por el monitor de vigilancia, así el agente no tiene que preprocesar de nuevo dichos eventos. Esta acción se realiza enviando los eventos al canal de eventos. Dicho canal

de eventos se implementa usando el servicio de *Ice* conocido como *IceStorm*, que es el encargado de transmitir los eventos a los diferentes agentes subscritos al propio canal de eventos. La obtención del canal de eventos correspondiente se explica a partir del siguiente fragmento de código:

```
1   IceStorm::TopicManagerPrx topicManager;
2   IceStorm::TopicPrx topic;
3   string topicName = "eventchannel";

5   // PASO 1
6   // Obtain a TopicManager of IceStorm
7   topicManager = IceStorm::TopicManagerPrx::checkedCast(comm->
        propertyToProxy("TopicManager.Proxy"));

9   // PASO 2
10  // Retrieve the topic to sent the events
11  try{
12      topic = topicManager->retrieve(topicName);
13  }catch(const IceStorm::NoSuchTopic&){}

15  // PASO 3
16  // Get the topic's publisher object, and create a Agent proxy
17  Ice::ObjectPrx publisher = topic->getPublisher();
18  AgentPrx agent = AgentPrx::uncheckedCast(publisher);

20  // PASO 4
21  // Notify all the agents with the new event
22  agent->notify(newEvent);
```

Los *topics* son los temas a los que se pueden subscribir los objetos. En el caso concreto de los agentes, se subscriben a un *topic* que se conoce como *eventchannel*. Un *topic* es equivalente a una interfaz *Slice*: las operaciones del interfaz definen los tipos de mensajes soportados por el propio *topic*. Un publicador (en este caso, el monitor de vigilancia) usa un *proxy*[3] al interfaz *topic* para enviar sus mensajes, y un subscriptor (el agente) implementa la interfaz *topic* para recibir los mensajes. Realmente dicha interfaz representa el contrato entre el publicador y el subscriptor, excepto que *IceStorm* encamina cada mensaje a múltiples receptores de manera transparente.

El monitor de vigilancia (publicador) es el encargado de implementa el envío de los eventos a los agentes siguiendo el método de *IceStorm*. Para implementar un publicador *IceStorm*, se han de seguir los siguientes pasos (véase el código anterior para identificar los pasos):

1. Obtener un *proxy* para el *TopicManager*. Este el principal objeto *IceStorm*, que maneja los diferentes *topics* y los subscriptores. Es usado por publicador y subscriptor.

[3]Representante de un objeto remoto

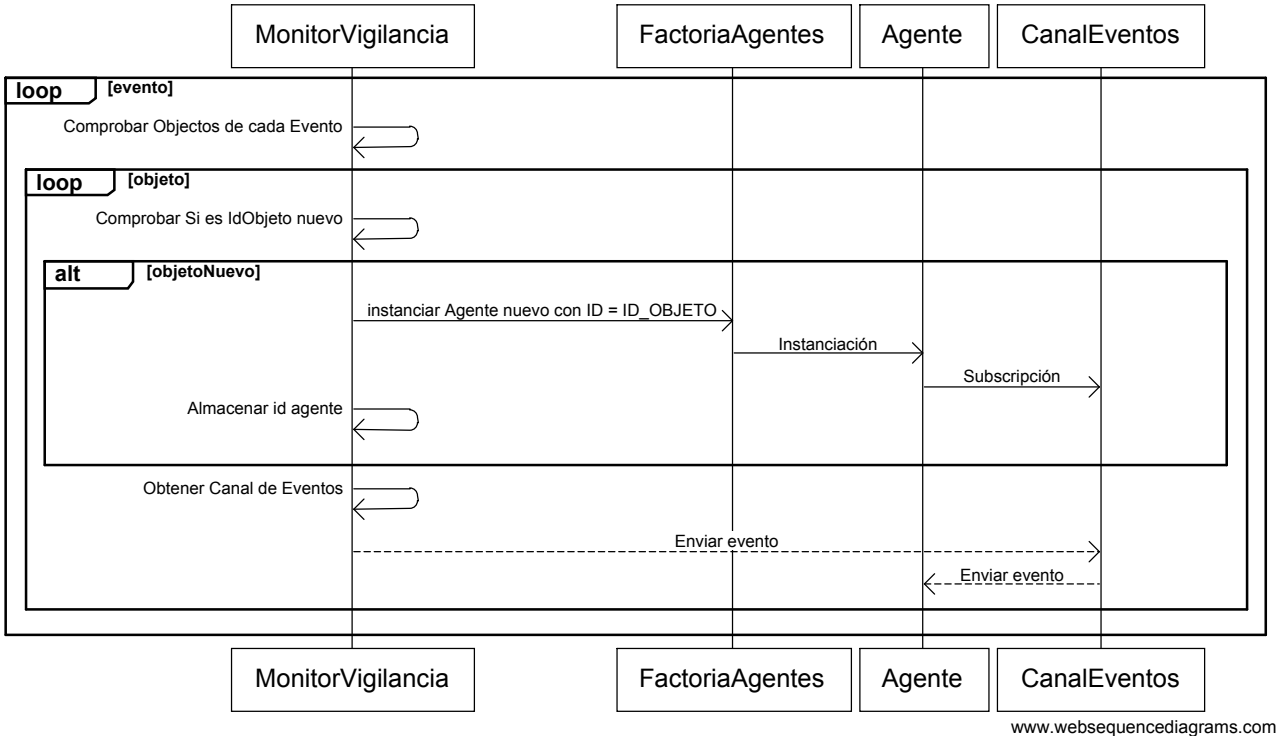


Figura 5.10: Diagrama de secuencia del módulo monitor de vigilancia.

2. Obtener un *proxy* para el *topic* conocido como *eventchannel*, creando el *topic* si no existe o recuperándolo.
3. Obtener un *proxy* para el objeto publicador. Se realiza una conversión a tipo *AgentPrx* para poder publicar el mensaje.
4. Enviar a los agentes el evento procesado.

Para concluir la explicación del módulo monitor de vigilancia, queda explicar cómo se logra que el objeto sea remoto y pueda ser accedido por los demás componentes del sistema. Primeramente, se implementó una clase llamada *VigilanceManagerServer*, incluida en la carpeta (*/src/Server*), que hereda de la clase *Ice::Application* y que contiene todo el código asociado a la instanciación de los recursos necesarios para obtener la parte servidora de la aplicación. La clase *Ice::Application* es proporcionada por *Ice* para encapsular todo el código asociado a las actividades de inicialización y finalización de un servidor. La idea de esta clase es abstraer al programador de la creación del servidor y centrarse en la implementación del método *run()*, en el que se debe agregar un sirviente[4], de la clase *VigilanceManagerI* al adaptador de objetos previamente creado por el propio servidor.

[4]El componente en la parte del servidor que proporciona el comportamiento asociado a la invocación de operaciones

El código de este método es muy sencillo y a continuación se muestra un fragmento de código que implementa dicha operación:

```
2    Ice::ObjectAdapterPtr adapter = communicator()->
        createObjectAdapter("VigilanceManager");
3    pfc::VigilanceManagerPtr vig = new pfc::VigilanceManagerI();
4    adapter->add(vig, id);
5    adapter->activate();
```

En primer lugar, se crea un adaptador de objetos con el nombre *VigilanceManager*, que a su vez crea un puerto de escucha, para recibir las peticiones de los clientes. Seguidamente, se crea una instancia (sirviente) de la clase *VigilanceManagerI* y finalmente, se agrega dicho sirviente al adaptador de objetos para servir las peticiones de los clientes que quieran invocar las operaciones definidas previamente en la cabecera *Slice* de la interfaz *VigilanceManager* (véase figura 5.11).

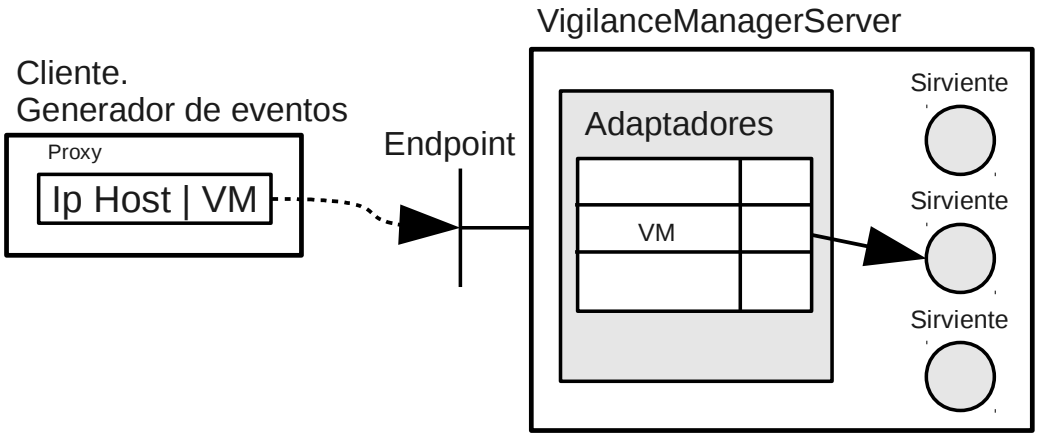


Figura 5.11: Diagrama de los adaptadores de objetos para el monitor de vigilancia.

5.4. Módulo Creación de Agentes

En este modulo, se produce la generación de agentes inteligentes, encargados de analizar los sucesos que aparecen en los diferentes eventos. La arquitectura planteada por Amaavem se diseñó usando una tecnología multi-agente para resolver el problema planteado. Se decidió emplear agentes racionales, para intentar reducir la intervención de los operadores humanos en los sistemas de videovigilancia. Lo que se pretende con esto es que cada agente inteligente, instanciando en el sistema, se encargue de un único objeto, que aparece en los eventos y que además, estos agentes determinen el nivel de normalidad de los acontecido en los eventos.

La responsabilidad de **generar instancias de agentes**, recae en la fábrica de agentes, implementada por la clase *AgentFactoryI*. Esta factoría se desarrolló siguiendo los patrones de diseño y en concreto, el patrón factoría abstracta, debido a que los agentes que se manejarán pueden ser de diversos tipos, en función del entorno a tratar. Por ello, cada factoría de agentes va asociada a un dominio concreto. Los agentes serán especializaciones de la clase *AgentI*, para cada dominio, donde se despliegue Amaavem. La clase *AgentI* es una clase abstracta de la que deben heredar los agentes concretos de los entornos que se quieran monitorizar. En las siguientes subsecciones, se explica cómo fueron implementadas dichas clases, así como decisiones de diseño relevantes.

5.4.1. Factoría de agentes

La factoría de agentes fue implementada siguiendo el diseño planteado por *ZeroC ICE* en el artículo [Ice06]. En dicho artículo, se plantea una factoría abstracta remota, que posibilita la creación de distintos tipos de objetos heterogéneos, la búsqueda de objetos ya instanciados e incluso la destrucción de objetos en el sistema.

En el diseño de una arquitectura soportada por *ICE* se debe tener en cuenta el ciclo de vida que poseen los diferentes objetos en el sistema. Se debe tener cuidado en la destrucción de las instancias debido a que los sirvientes, que se añaden a los adaptadores de objetos, tienen un ciclo de vida distinto a la propia instancia. Los servidores de las factorías de agentes se despliegan del mismo modo que los servidores de monitores de vigilancia, en los distintos computadores, que componen Amaavem.

Por tanto, se pensó en este aspecto de diseño debido a que la arquitectura Amaavem está pensada para ser empleada en cualquier tipo de entorno monitorizado. La heterogeneidad de los objetos es un punto a tener en cuenta en la creación de una arquitectura de tales características y por ello, emplear una factoría abstracta es una garantía de efectividad en cuanto a que el desarrollador final, sólo debe preocuparse en implementar la parte específica del dominio de esta factoría y abstraerse de otros aspectos de diseño. Entonces, la idea general es que cada factoría sea única a un dominio concreto, y todas ellas tengan las mismas carac-

terísticas y sólo difieran en la parte de instanciación del agente concreto, ya que los tipos de objetos serán diferentes en cada dominio. Todas las factorías concretas deben implementar el método *createAgent*, en función de las características del entorno. En el anexo B, se incluye el código de una factoría de agentes de un entorno urbano, para que el desarrollado final pueda observar cómo implementar una factoría concreta.

El **esqueleto de la factoría de agentes** de la arquitectura Amaaavem se muestra a continuación:

```
typedef AgentI* (*getAgent_t)(int, const Ice::ObjectAdapterPtr);

class AgentFactoryI : public AgentFactory
{
    public:
        AgentFactoryI();
        AgentFactoryI(const Ice::ObjectAdapterPtr& adapter);
        virtual ~AgentFactoryI();
        virtual AgentPrx createAgent(const int& id,
                                     const typeObject& type,
                                     const Ice::Current& c);
        void cleanAgents();
        AgentI* findAgent(const int& id);
    private:
        vector<AgentI *> agents;
        Ice::ObjectAdapterPtr _adapter;
        IceUtil::Mutex _fileMutex;
        bool start;
};
```

La función *cleanAgents* es la encargada de eliminar del sistema agentes que han quedado ociosos o que no son del tipo correcto. Un agente debe ser eliminado cuando el *flag zombie* está activado. Dicho *flag* se activa cuando sucede alguno de los hechos mencionados anteriormente. Puede darse el caso de que un objeto que aparece en un evento, no vuelva a aparecer más en el evento. Por tanto, es interesante no consumir recursos del sistema y así eliminar dicho agente ya que nunca va a recibir más eventos a analizar debido a que el objeto que está analizando ha desaparecido de los propios eventos.

La función *findAgent* busca en la factoría un agente instanciado con el identificador pasado por parámetro y devuelve un puntero a dicho agente si es encontrado.

La función *createAgent* recibe como parámetro un identificador y un tipo de objeto, para posteriormente devolver un *proxy* de un agente específico. Anteriormente, se habló de la singularidad de las factorías de agentes en función del entorno a monitorizar. Por ejemplo, si se esta monitorizando un entorno urbano entonces, lo normal es que existan objetos y agentes de tipo persona, vehículos y animales. Por consiguiente, se necesita una factoría concreta que instancie los diferentes tipos de agentes. Si se tratara de otro tipo de entorno, resultaría necesaria otra factoría concreta, que instancie correctamente los distintos agentes (véase diagrama conceptual de la figura 5.12 para comprender mejor este punto).

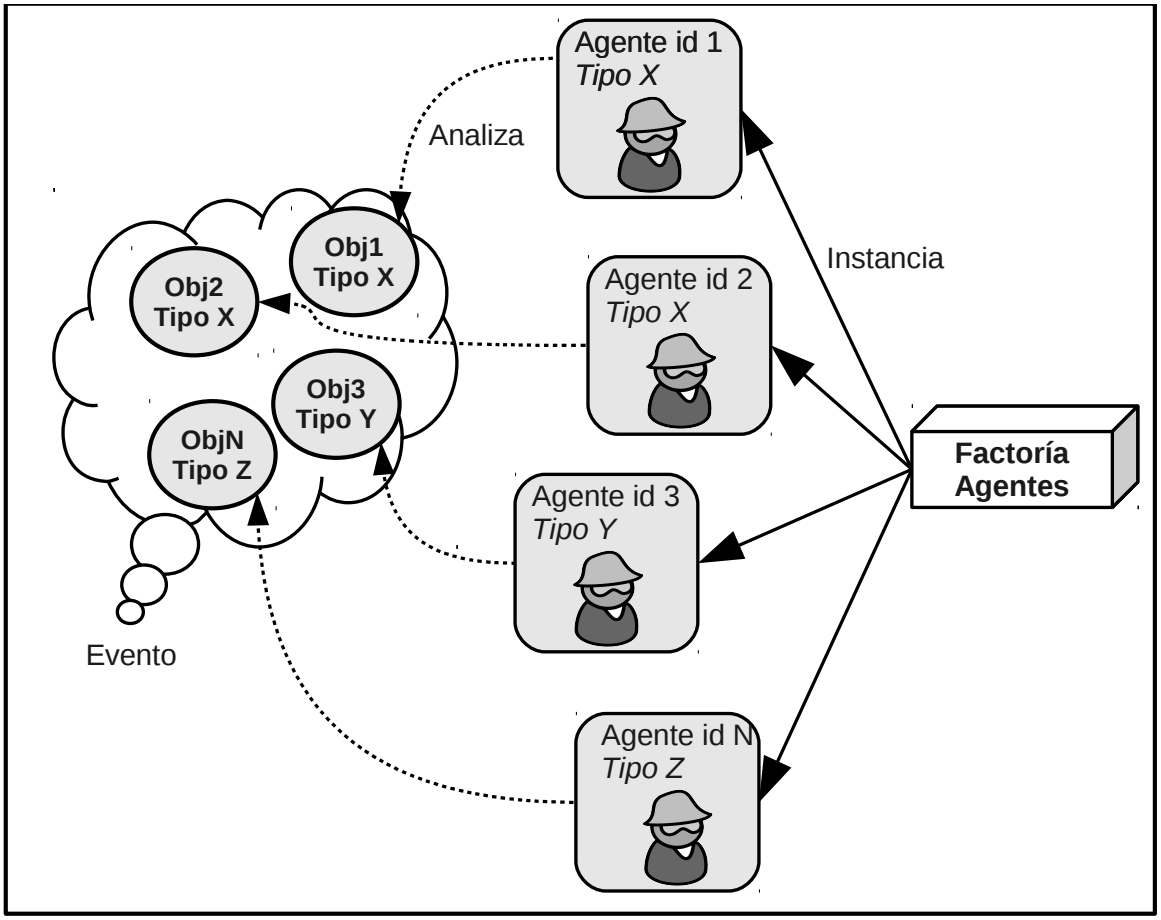


Figura 5.12: Diagrama conceptual de las factorías de agentes.

En referencia a la **instanciación de los agentes**, se emplearon **bibliotecas dinámicas** (*DLL*), que son otra forma de añadir funcionalidad a una aplicación. Se optó por el uso de bibliotecas dinámicas, debido a que aportaban expansibilidad y adaptabilidad a la arquitectura, en la funciones de creación de agentes. Las bibliotecas dinámicas son ficheros independientes que pueden ser invocados desde cualquier ejecutable, de modo que su funcionalidad puede ser compartida por varios ejecutables. Esto significa que sólo se necesita una copia de cada fichero de biblioteca(*DLL*) en el Sistema. Esta característica constituye la razón principal de su utilidad, ya que dota al sistema de mayor eficiencia y rendimiento. Una biblioteca dinámica puede ser cargada bajo demanda en el momento en que se necesita su funcionalidad, y en nuestro caso concreto, para la arquitectura Amaavem sólo son requeridas cuando se precisa la instanciación de un nuevo agente.

En realidad lo que caracteriza a una *DLL* es la forma en que es traída a ejecución; no directamente desde el *shell* del Sistema como un ejecutable *.exe* normal, sino desde otro ejecutable (que puede ser incluso otra *DLL*), de forma parecida a como se invoca una función (una especie de función externa al programa). Por esta razón no disponen de una función *main* o de un módulo de inicio en el sentido clásico.

Por consiguiente, en la arquitectura Amaavem, el empleo bibliotecas dinámicas para instanciar agentes ha dado al sistema las siguientes ventajas:

- **Reducen el tamaño de los archivos ejecutables**: Gran parte del código puede estar almacenado en bibliotecas y no en el propio ejecutable lo que redunda en una mejor modularización.
- **Pueden estar compartidas entre varias aplicaciones**: Si el código es suficientemente genérico, puede resultar de utilidad para múltiples aplicaciones
- **Facilitan la gestión y aprovechamiento de la memoria del sistema**: La carga dinámica permite al sistema operativo, aplicar algoritmos que mejoren el rendimiento del sistema, cuando se carguen estas bibliotecas. Además, al estar compartidas, basta con mantener una copia en memoria, para todos los programas que la utilicen.
- **Brindan mayor flexibilidad frente a cambios**: Es posible mejorar el rendimiento o solucionar pequeños errores, distribuyendo únicamente una nueva versión de la biblioteca dinámica. Nuevamente, ésta corrección o mejora será aprovechada por todas las aplicaciones que compartan la biblioteca.

Es importante resaltar otro punto de diseño referente a las bibliotecas dinámicas. Dicho punto es la manera de cargar y usar las propias bibliotecas. Para ello, existe una *API* para *C++* (*dlfcn.h*) que define las siguientes operaciones:

- *void** ***dlopen*** *(const char *nomfich, int indicador);*

  Esta función carga una biblioteca dinámica desde el fichero nombrado por la cadena que termina con el valor nulo *nomfich* y devuelve un manejador para la biblioteca dinámica. Si *nomfich* no es un camino absoluto (esto es, no comienza con */*), entonces el fichero se puede buscar en los siguientes lugares:

  - En una lista de directorios, separados por dos puntos, ubicados en la ruta que apunta la variable de entorno de usuario *LD_LIBRARY*.
  - En la lista de bibliotecas especificada en */etc/ld.so.cache*.
  - En la carpetas */usr/lib* y */lib*.

  El parámetro *indicador* puede ser de 2 tipos:

  1. *RTLD_LAZY*, que resuelve los símbolos no definidos según se vaya ejecutando código de la biblioteca dinámica.
  2. *RTLD_NOW*, que resuelve todos los símbolos sin definir antes de que *dlopen* retorne, y falle si no se puede realizar.

  Puede obtenerse una cadena legible, describiendo el error más reciente ocurrido desde cualquiera de las rutinas dll (*dlopen, dlsym o dlclose*), mediante la operación *dlerror()*.

- *void** ***dlsym****(void *manejador, char *símbolo);*

  Esta función toma un manejador de una biblioteca dinámica devuelto por *dlopen* y el nombre del símbolo terminado en nulo, devolviendo la dirección donde ese símbolo está cargado. Si no se encuentra el símbolo, *dlsym* devuelve NULL. Para mayor seguridad, el modo correcto de comprobar un error de *dlsym* es guardar el resultado de la operación *dlerror* en una variable, y comprobar si el valor guardado no es nulo.

- *int* ***dlclose*** *(void *manejador);*

  Esta función decrementa el contador de referencias al manejador de la biblioteca dinámica. Cuando el contador de referencias llegue a cero, entonces se descarga la biblioteca dinámica del sistema. Si la biblioteca dinámica exporta una rutina llamada *_fini*, entonces esa rutina se llama justo antes de que la biblioteca se descargue.

La compilación de las bibliotecas se realiza añadiendo el *flag -fPIC* a los objetos que se emplean en la misma y finalmente, construyendo la biblioteca (*.dll*) añadiendo el *flag -shared*. Dichas bibliotecas se deben guardar en la carpeta (*/libs/libsDll*) del proyecto y se debe configurar la variable de entorno *LD_LIBRARY_PATH* en caso de usar un sistema operativo *GNU Linux* o la variable *PATH* en *Microsoft Windows*.

Deben existir tantas bibliotecas como tipos de agentes haya en el sistema. Es decir, cada librería dinámica se asocia a un tipo de agente ayudando a que el sistema sea fácilmente extensible y escalable. Por cada objeto que exista en el entorno a monitorizar, del mismo modo se debe implementar un agente del tipo citado. Por tanto, si se desean añadir nuevos agentes al sistema, se deben crear también nuevas bibliotecas dinámicas que permitan su instanciación de manera sencilla y eficiente. Para ello, se debe incluir una pequeña función en la clase que implementa un agente especifico tal y como se muestra en el siguiente fragmento de código:

```
1   TYPE es el tipo especifico del agente. Ejemplo: AgentPerson

3   extern "C" pfc::AgentI* getAgentTYPE(int id, const Ice::
        ObjectAdapterPtr& adapter){
4       return new pfc::AgentTYPE(id,adapter);
5   }
```

Entonces, la factoría de agentes implementada, para un dominio concreto, carga la biblioteca dinámica, especificada previamente por el parámetro *typeObject* y después, llama al método *getAgentTYPE* para instanciar el agente. La parte del sistema, que implementa la carga de la bibliotecas dinámicas, también es específica de cada dominio a monitorizar. A modo de ejemplo, se estudia esta operación para un dominio urbano donde se han definido tipos de agente persona y vehículo. En la carpeta *libs/libsDll* se encuentran las bibliotecas dinámicas generadas tras compilar el proyecto. La constante *ROUTE_DLL* hace referencia a la ruta relativa de la carpeta donde están las bibliotecas dinámicas.

En el siguiente fragmento de código se muestra como se realiza la carga de las bibliotecas dinámicas:

```
1         AgentPrx AgentFactoryI::createAgent(int id, typeObject type,
              const Ice::Current& c){
2             ...
3             string libAgent;
4             string typeAgent;
5             ...
6             // SPECIFIED DOMAIN
7             // Select the dynamic library in function of type agent
8             switch(type){
9                 case PERSON:
10                    libAgent = ROUTE_DLL + "libAgentPerson.so";
11                    typeAgent = "getAgentPerson" ;
12                    text = "[AgentFactory] Instantiating Agent Person
                          with id: " +ops::numToString(id);
13                break;

15                case VEHICLE:
16                    libAgent = ROUTE_DLL + "libAgentVehicle.so";
17                    typeAgent = "getAgentVehicle";
18                    text = "[AgentFactory] Instantiating Agent Vehicle
                          with id: "+ops::numToString(id);
19                break;
20            }
21// Open the dynamic library
22            void *handle = dlopen(libAgent.c_str(), RTLD_LAZY);

24            if (!handle){
25                    textError = "[AgentFactory] Cannot open dynamic
                          library";
26                    _fileMutex.lock();
27                    Logger::getInstance()->write_error(textError.c_str
                          ());
28                    _fileMutex.unlock();
29            }
30            // Clean errors
31            dlerror();

33            getAgent_t getAgent = (getAgent_t) dlsym(handle, typeAgent
                  .c_str());
34            const char* dlsym_error = dlerror();

36            if (dlsym_error){
37                    textError = "[AgentFactory] Cannot open symbol '
                          getAgent()'";
38                    _fileMutex.lock();
39                    Logger::getInstance()->write_error(textError.c_str
                          ());
40                    _fileMutex.unlock();
41                    exit(1);
42            }
43        }
```

Para instanciar un agente, es necesario pasarle por parámetros un identificador y un adaptador de objetos, al que se registra el sirviente del agente, y así poder atender las invocaciones remotas. El constructor de la clase *AgentI* (del que heredan las agentes especializados) se muestra a continuación:

```
1    AgentI(const int& id,
2           const Ice::ObjectAdapterPtr& adapter );
```

Se optó por emplear herencia y polimorfismo en la representación de la jerarquía de los agentes y los comportamientos debido a la heterogeneidad de los mismos. Las ventajas más relevantes de usar polimorfismo se muestran a continuación:

- Evita la duplicación de código para funcionalidades que sean comunes para todos los tipos.
- Permite la creación de nuevos agentes o comportamientos especializados de manera sencilla.
- Permite una fácil expansibilidad del sistema.

Una vez realizada la instanciación del agente, éste debe suscribirse en el canal de eventos y avisar a los demás agentes de su existencia. En la siguiente subsección se explica detalladamente este último punto.

5.4.2. Submódulo sistema de comunicaciones

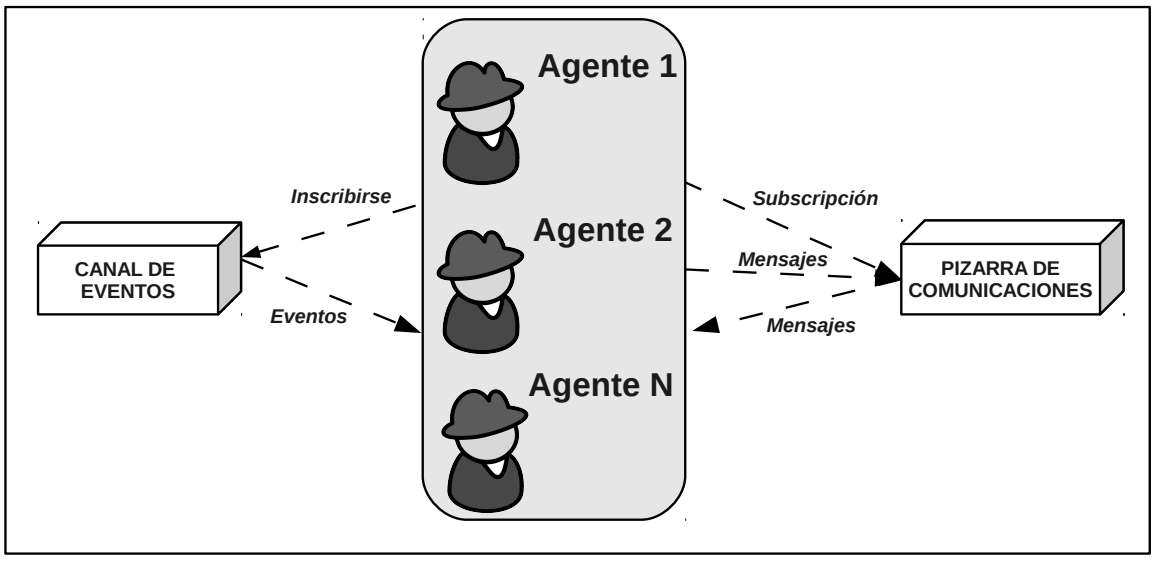


Figura 5.13: Diagrama del sistema de comunicaciones de Amaavem.

Este submódulo implementa un sistema de comunicaciones entre los componentes del sistema y los propios agentes. La principal ventaja de usar un sistema de comunicaciones

es abstraer a los demás componentes del sistema de las comunicaciones con los agentes. Con este sistema, comunicarse con los agentes resulta muy sencillo. Ningún componente necesita conocer donde se encuentran alojados los objetos remotos, que hacen referencia a los agentes.

La instanciación de un agente en el sistema involucra dos acciones:

1. **Suscribirse al canal de eventos del sistema**.

   El canal de eventos del sistema está implementado por el servicio de *ICE* conocido como *IceStorm*. Este canal de eventos hace de mediador entre el módulo monitor de vigilancia y los agentes del sistema, que analizan las situaciones que se producen en cada instante, consiguiendo que el monitor de vigilancia no tenga que conocer donde se encuentran los agentes instanciados.

   Gracias al uso de *IceStorm*, se gestiona el envío y la recepción de eventos de una manera sencilla, flexible y eficiente. La ventaja más importante de usar *IceStorm* es el hecho de poder centrarse en la aplicación en cuestión, minimizando el tiempo empleado en cuestiones ajenas a la naturaleza de la misma y obteniendo un sistema eficiente. El monitor de vigilancia simplemente envía los eventos que precise al canal de eventos y este último se encarga de repartir el mensaje a todos los agentes instanciados en el sistema de manera transparente.

   En el módulo monitor de vigilancia, se explicó cómo se lleva a cabo el servicio de publicación de eventos y a continuación, se detallan los pasos que se han de seguir para realizar una subscripción del agente al canal de eventos:

PASO 1 Obtener un *proxy* para el *TopicManager*.

PASO 2 Registrar en el adaptador de objetos (el mismo que el agente recibe cuando se instancia como un parámetro en su constructor) un sirviente asociado a la interfaz *Agent*.

PASO 3 Activar dicho sirviente en el adaptador de objetos.

PASO 4 Subscribir el agente al *topic* requerido, en este caso el *topic* es conocido con el nombre de *eventchannel*. Para ello, se emplea la operación ofrecida por la interfaz *IceStorm::Topic* llamada *subscribeAndGetPublisher*.

2. **Informar a los demás agentes de su presencia**.

   Del mismo modo que los agentes reciben eventos por un canal de eventos, también se implementó un sistema de comunicación entre los propios agentes. Para ello, se creó otro *topic* (*blackboard*) de *IceStorm* que realizase la función de pizarra de comunicación entre agentes colaborativos. Aunque la arquitectura planteada no precise de la colaboración de agentes para resolver el problema, puede darse un caso futuro en el que se desee dar soporte a este tipo de problemática.

Por ello, se implementó un sistema de mensajes entre agentes que resultase sencillo y fácilmente extensible. De la misma manera que los agentes se subscriben al canal de eventos, al mismo tiempo se subscriben a este sistema de paso de mensajes entre agentes. La comunicación entre agentes se realiza enviando un mensaje a la pizarra de comunicación, con una estructura previamente definida en los archivos de configuración. Dicha estructura contiene un *proxy* del agente que envía el mensaje y un texto donde se indica el mensaje que es enviado a los demás agentes. La estructura de datos puede ser ampliada cuando sea preciso y actualmente está definida como sigue a continuación:

```
1    // Class for the communication of Agents

3    class Message{
4        Object* agentSource;
5        string text;
6    };
```

Cuando se produce la instanciación de un nuevo agente, éste avisa a los demás agentes de su presencia por medio de la pizarra de comunicaciones. La pizarra es la encargada de promover el mensaje a los distintos agentes instanciados en el sistema. Dicha operación se implementa en la función *sendMyInfo* de la clase *AgentI*. En la figura 5.13 se muestra un diagrama del sistema de comunicaciones de Amaavem.

Una vez explicada la subscripción de los agentes al sistema de comunicaciones, es interesante explicar cómo se realiza la cancelación de la subscripción por parte de los agentes. Cuando un agente es suprimido del sistema, por motivos que se estudiarán más adelante, debe darse de baja del sistema de comunicaciones. Si no se diera de baja, surgiría un conflicto importante en el sistema, ya que el canal de eventos intentaría enviar mensajes a un agente que ya no existe y lanzaría excepciones difíciles de controlar por parte del cliente final. Por ello, cuando se precise eliminar algún agente del sistema, lo primero que debe hacer dicho agente, es darse de baja del sistema de comunicaciones para evitar que el sistema continue enviándole información. Para darse de baja del sistema de comunicaciones, tan sólo se debe suprimir el sirviente del adaptador de objetos, que está escuchando en el lado del servidor y avisar al sistema de comunicaciones con la noticia de la cancelación de la subscripción, mediante el método *unsubscribe*), que es ofrecido por la interfaz *IceStorm::Topic*.

Las operaciones de subscripción son implementadas por la función *subscribe* de la clase *AgentI* y las operaciones de darse de baja del sistema de comunicaciones, por la función *unsubscribe* (véase diagrama de clases de la figura 5.5.

5.4.3. Submódulo de comprobaciones

Este módulo es el encargado por un lado de comprobar el estado de los agentes, y por otro lado llevar a cabo acciones de eliminación en el caso de que sea necesario. Un agente instanciado en el sistema tiene una variable booleana llamada *zombie*, que define el estado del agente en el sistema. Cuando dicho *flag* se encuentre activado, significa que el agente ya no es un agente activo del sistema y debe ser eliminado para que no reciba más eventos y no ocupe memoria innecesariamente. Dicho *flag* se activa por dos motivos:

- **El agente instanciado no es del tipo correcto**. El monitor de vigilancia instancia tantos tipos de agentes, como tipos de objetos hay en el sistema debido a que el monitor no tiene como responsabilidad clasificar los objetos que procesa. Esta función de clasificación es responsabilidad de los agentes y se explicará en el módulo siguiente. Por tanto, existen agentes instanciados en el sistema que no son del tipo correcto y por lo que, sería adecuado destruir estos agentes del sistema cuando se tenga una certeza de que el agente no pertenece al tipo correcto. A modo de ejemplo, no tendría sentido analizar el comportamiento de un coche cuando ha sido instanciado un agente de tipo persona para llevar a cabo dicho análisis. Lo interesante sería que cada objeto esté vinculado con un agente del mismo tipo porque sino se obtendrían resultados no deseados.

  La implementación de esta operación, la realiza el método *checkTypeAgent* de la clase *AgentI*. Cada vez que el agente reciba un evento, debe clasificar al objeto que está analizando, ya que el objeto puede sufrir variaciones de tamaño a lo largo del tiempo. Además, el sistema de *tracking* auxiliar puede fallar, y en un instante concreto detectar un objeto como un tipo inadecuado. Por ello, el agente debe realizar la clasificación del objeto cada vez que procese un nuevo evento y comprobar si el objeto analizado coincide con la naturaleza del propio agente. Para esta causa, los agentes tienen una variable numérica, que cuenta el número de fallos de tipo, que ha sufrido el agente cuando realiza la clasificación del objeto. Es decir, cuando un agente de tipo X analiza un objeto de tipo Y, en cada evento que procesa, el agente hace una clasificación distinta al tipo de su naturaleza, por lo que debe incrementar este contador de fallos.

  Por ejemplo, póngase el caso de que un agente de tipo persona, le ha tocado analizar a un vehículo. Todas las veces, que dicho agente clasifique este objeto, incrementará el contador debido a que el objeto es de tipo diferente al agente y no tiene sentido analizarlo. Si el contador de fallos supera un umbral establecido previamente por una constante de configuración (*NUM_ERRORS_MAX*) el agente debe ser destruido.

El código de la operación planteada se muestra a continuación:

```
1   void AgentI::checkTypeAgent(){
2       string text ;
3       if ( this->numErrorsType >= NUM_ERRORS_MAX ){
4           text = "[Agent]"+ ops::numToString(getId()) +"
                deleted because wasn't the type correct";
5           cout << text << endl ;
6           Logger::getInstance()->write(text.c_str());
7           destroyAgent();
8       }
9   }
```

- **El agente deja de recibir eventos**. En el caso de que un agente deje de recibir eventos durante un periodo prolongado, puede significar que el objeto que estaba analizando deja de aparecer en los eventos y por ello, resulta adecuado eliminar este agente debido a su inactividad. El agente ocupa una porción de memoria en el computador, donde está desplegado y resulta provechoso eliminar ese agente, cuando ya no se requiera su servicio y así poder liberar esa porción de memoria.

  La implementación de esta comprobación corre a cargo de la función *checkInactivity* de la clase *AgentI*. Los agentes tienen asociados un temporizador, que cuenta el tiempo (segundos) entre la recepción del último evento y el siguiente. Cada vez que el agente recibe un evento se reinicia este temporizador. Si no llega un evento nuevo en un intervalo de tiempo determinado por una constante definida en los archivos de configuración (*TIME_FINISH*) el agente será destruido. El temporizador se implementa usando una biblioteca para desarrolladores de *GTK+* llamada *glibmm* [Gli12].

  En vez de realizar esta comprobación cada vez que llega un evento, se realiza cada vez que se han procesado un serie de eventos (por ejemplo, cada 50 eventos recibidos). Con ello, se logra que el sistema no se sature por las numerosas comprobaciones, ya que se supone que el agente va a quedar inactivo pasado un tiempo de ejecución considerable.

  El código de esta operación se muestra a continuación:

```
1   void AgentI::checkInactivity(){
2       gdouble elapsed;
3       gulong usec;
4       string text ;
5       elapsed = g_timer_elapsed (timer, &usec) ;
6       if ( elapsed >= TIME_FINISH ){
7           text = "[Agent]"+ ops::numToString(getId()) +"
                deleted for inactivity";
8           cout << text << endl ;
9           Logger::getInstance()->write(text.c_str());
10          destroyAgent();
11      }
12  }
```

El código de la recepción de los eventos, donde se invoca a los métodos que realizan las comprobaciones, también se muestra a continuación:

```
1   void AgentI::notify(const EventPtr& event, const Ice::Current& c){
2
3       if ( not isZombie){
4           (this->eventsProcessed)++;
5           if ( (this->eventsProcessed % 50 ) == 0)
6               checkInactivity();
7           checkTypeAgent();
8           analyze(*event);
9       }
10  }
```

5.5. Módulo Análisis de Comportamientos

Este módulo es el responsable de realizar el análisis concreto de los eventos, procesados en los módulos anteriormente estudiados. Está representado por la **clase *Behavior***, que es una clase abstracta de la que heredan los comportamientos concretos (véase figura 5.14). La cabecera de esta clase se muestra a continuación:

```
1   // Abstract class
2   class Behavior
3   {
4       public:
5           Behavior();
6           virtual ~Behavior();
7           virtual double analyze(Ice::Communicator& communicator,
8                                  const int& id,
9                                  const Event& event) = 0;
10
11  };
```

Los comportamientos específicos heredan de esta clase abstracta y están asociados a los agentes especializados. Los comportamientos son específicos para cada dominio concreto y el desarrollador tiene plena libertad para desarrollar los comportamientos que determine oportunos. Es decir, para cada dominio, donde se despliega Amaavem, se deben especificar los agentes específicos del dominio, así como sus comportamientos naturales de conducta.

A modo de ejemplo, si por ejemplo se quiere desplegar Amaavem en un entorno acuático, dado que se quieren estudiar los comportamientos de los peces de un río contaminado. Se podrían definir comportamientos típicos de conducta de los mismos peces, pero en ríos normales, y comprobar la diferencia entre ambos. Otro ejemplo podría ser en un entorno de carreteras peligrosas, se podría definir un comportamiento para vehículos que mida el nivel de peligrosidad de los coches en curvas peligrosas.

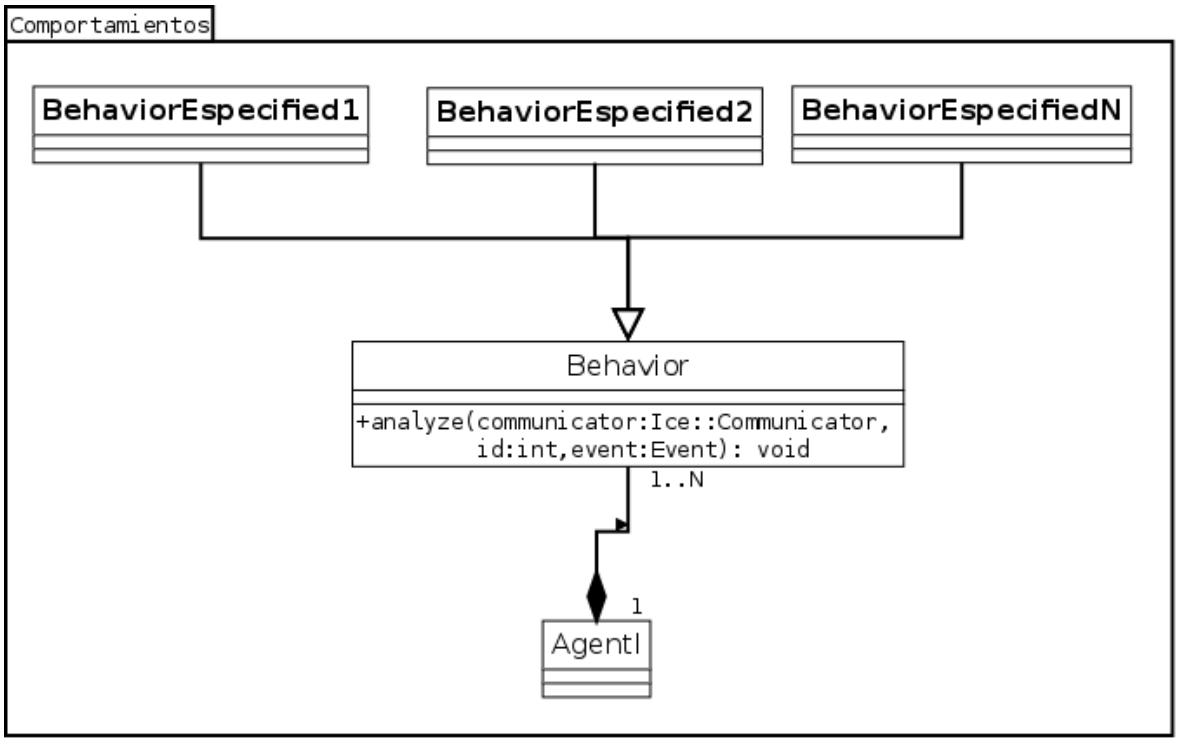


Figura 5.14: Diagrama de la especialización de comportamientos

Como se puede apreciar en la figura 5.14, todos los comportamientos específicos del dominio a monitorizar deben implementar el método virtual puro *analyze*, que será el método encargado de llevar a cabo el análisis de la información. En las siguientes subsecciones, se va a desglosar este módulo en pequeños submódulos, detallando aspectos de diseño e implementación. En la figura 5.15, se puede apreciar un diagrama de secuencia que ejemplifica las acciones de este módulo y sus relaciones con otros módulos.

5.5.1. Submódulo clasificación de objetos

Anteriormente se expuso que un agente sólo va a analizar el comportamiento de un objeto determinado por el monitor de vigilancia. El monitor de vigilancia solicita instanciar tantos agentes para un objeto concreto como tipos de objetos existan en el dominio. El agente al recibir un evento del canal de eventos, primeramente, comprueba si en el evento se encuentra el objeto que debe analizar. Para ello, simplemente hace una comprobación de identificadores. En el evento se encuentra una lista de objetos y el agente está identificado por el identificador del objeto, que debe analizar. Entonces, debe hacer una búsqueda de dicho identificador en la lista de objetos que contiene el propio evento.

Una vez que el agente comprueba que su objeto asignado se encuentra en el evento a procesar, éste tiene como responsabilidad determinar si el objeto que está analizando coincide con su propia naturaleza. Por razones obvias, no tendría sentido que existiera un agente de un tipo *Y* y esté analizando un comportamiento de una situación concreta de un objeto de tipo

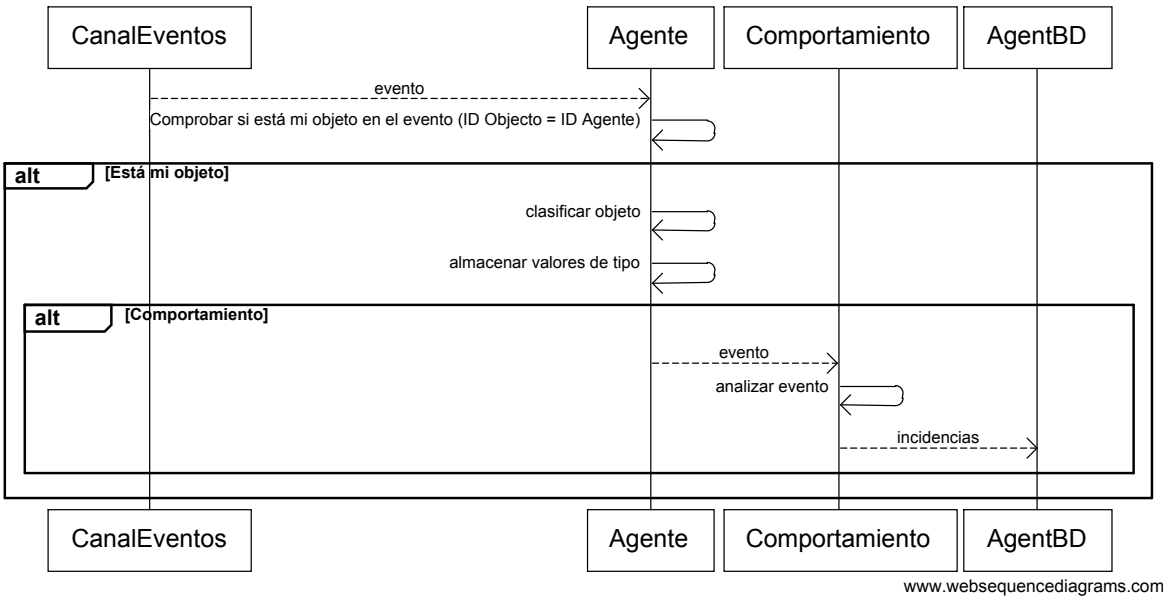


Figura 5.15: Diagrama de secuencia: análisis de los eventos

X, debido a que los resultados no van a ser lo esperados. Lo verdaderamente interesante, es que el agente clasifique el objeto que le ha sido asignado, para determinar si es de su misma naturaleza. Tal y como se explicó en la sección 5.4.3, en caso de que un agente determine que el objeto que está analizando no coincide con su naturaleza, el agente será removido inmediatamente.

La **clasificación del objeto** se implementa en el método *classify* de la clase abstracta *AgentI*. Dicho método es específico para cada dominio concreto (eventos de vídeo) y recibe como parámetros la altura y la anchura del objeto, retornando un vector de *keyValue*. La estructura *keyValue* está compuesta por dos atributos: un tipo de objeto y un número real que determina el valor de creencia del objeto al tipo especificado. En caso de que los eventos sean de audio, la clasificación no es un punto fundamental, debido a que clasificar eventos auditivos es una tarea muy complicada y se puede asumir que todos los eventos de audio son del mismo tipo. Para comprender mejor la estructura *keyValue*, se incluye una tabla de un ejemplo para un dominio urbano:

PERSON	0.3
VEHICLE	0.6
ANIMAL	0.1

Esta tabla hace referencia al vector que se retorna en el método estudiado. Es decir, el agente, cada vez que clasifica un objeto, determina el grado de creencia de dicho objeto a todos los tipos de objetos, que existan en el dominio que se está monitorizando. Después de que el agente haya realizado la clasificación del objeto, debe comprobar si el objeto que se está analizando coincide con su naturaleza:

Primero, se obtiene valor de creencia específico (en el que coincide con el tipo de agente concreto) del vector que retorna la función de *classify*. Seguidamente, se compara el valor de creencia del agente seleccionado al tipo concreto con un mínimo de valor de creencia (constante *MIN_VALUE_BELIEF* definida en el archivo de configuración *IceHeaders/config.ice*). En caso de que el valor sea superior a esta cota, simplemente se almacena el valor de creencia en el atributo *valueType* (que almacena el valor de creencia de dicho agente al tipo instanciado) de la clase *AgentI*. En caso de que el valor de creencia no supere la cota, se aumenta un error de fallo de tipo (variable *numErrorsType* de la clase *AgentI*) para que el submódulo de comprobaciones tome las medidas que se precisen.

Por ejemplo, supongamos un entorno urbano donde está instanciado un agente de tipo persona y está clasificando un objeto. La clasificación del objeto devuelve un vector con los siguientes valores: *Tipo persona con valor de 0.3 y tipo vehículo con valor de 0.7*. El tipo persona tiene un valor de 0.3 y la cota mínima ha sido determinada por 0.5 por lo que se debe incrementar un fallo de tipo. Si el tipo hubiera sido mayor a la cota mínima simplemente se hubiera almacenado dicho valor.

5.5.2. Submódulo análisis de eventos

Una vez realizada la clasificación de objetos en el sistema, se debe realizar el análisis del evento recibido. Los agentes especificados que heredan de la clase abstracta *AgentI* deben implementar el método *analyze*. En este método, el agente distribuye el evento a todos los comportamientos que tenga instanciados y cada comportamiento analiza la situación del entorno.

Cabe recalcar la fácil adaptabilidad y expansibilidad del sistema, ya que el desarrollador que use Amaavem puede implementar tantos comportamientos como sea preciso en función del dominio donde se despliegue la aplicación. Esta arquitectura ofrece un servicio para monitorizar inteligentemente eventos en múltiples dominios. La idea era ofrecer una arquitectura que resolviera muchos problemas típicos en sistemas de vigilancia inteligente y por ello, se optó por crear este sistema multi-agente, que defina multitud de comportamientos para el análisis de los eventos.

Por ejemplo, para un dominio urbano, se pueden implementar una serie de comportamientos para una persona o para un vehículo. Algunos de estos comportamientos en entornos urbanos podrían ser:

- Medición de la velocidad de un vehículo en una vía.
- Comprobación de accesos o paradas inadecuadas en zonas restringidas.
- Medición de la distancia entre personas y vehículos.
- Detección de peleas entre personas en la vía urbana.
- Detección de infracciones de tráfico por parte de vehículos.

- Detección de colisiones entre vehículos.
- Detección de objetos sospechosos abandonados en la vía urbana.

Para conseguir el análisis de normalidad de las condiciones del entorno, se determinan una serie de comportamientos asociados a la naturaleza de cada objeto del dominio, donde se despliega Amaavem. Entonces, para poder decidir si lo que está ocurriendo se encuentra en un grado de normalidad aceptable, los agentes analizarán dichos comportamientos y decidirán en cuanto difieren de lo acontecido. Para llevar a cabo el análisis de la situación, se desarrolla un análisis de normalidad en función de los comportamientos definidos. Es decir, cada comportamiento determina el grado de normalidad de la situación que se está analizando. El agente, a través de su comportamientos, normaliza la situación en un valor comprendido entre 0 y 1, asumiendo un 1 la normalidad total y un 0 la anormalidad total.

El comportamiento valora la situación en función de los datos, que ha almacenado hasta el momento, y su propia naturaleza. Realiza una estimación de la normalidad de la situación y genera una incidencia, donde se almacena el análisis de lo sucedido. La incidencia para eventos de vídeo está representada por una estructura de datos llamada *incidence* y se muestra a continuación:

```
1    class incidence{
2        int camera;
3        int numFrame;
4        int id ;
5        int id2;
6        double normality;
7        string type;
8    };
```

La incidencia para eventos de audio está representada por una estructura de datos llamada *incidenceAudio* y se muestra a continuación:

```
1    class incidenceAudio{
2        int microphone;
3        int track;
4        double normality;
5        string type;
6    };
```

Para almacenar la incidencia en el sistema, se debe obtener un *proxy* del objeto remoto *AgentBDI* y así poder invocar al método *updateBD* que actualiza la base de datos con la incidencia recibida. Se omite la explicación de cómo se realiza la llamada a objetos remotos *Ice* debido a que ya fue explicado en secciones anteriores, pero se muestra a continuación la porción de código que implementa la creación de la incidencia y el envío de la misma al agente de la base de datos.

```
    // BEHAVIORSPECIFIED es el comportamiento especifico implementado.
        Ejemplo: BehaviorCheckStops

double BEHAVIORSPECIFIEF::analyze(Ice::Communicator& communicator,
   const int& id, const Event& event){
    ....
    const EventVideo* eventV  = static_cast<const EventVideo*>(&event)
       ;
    incidence incident;
    // Save the results in the BD - Event VIDEO
    incident.camera = eventV->camera ;
    incident.numFrame = eventV->frame;
    incident.id = id ;
    incident.id2 = (*it).id ;
    incident.normality = normality ;
    // Call to AgentBD by ICE
    AgentBDPrx agent = AgentBDPrx::uncheckedCast(communicator.
       propertyToProxy("AgentBD.Proxy"));
    if(!agent){
        ...
    }
    agent->updateBD(incident);
```

5.6. Módulo Análisis de Resultados

Este módulo es el responsable de analizar los resultados generados en las etapas anteriores. En el módulo anterior se generaban las incidencias que ocurrían en el sistema y en este módulo se analizan. Aveces puede resultar complicado monitorizar un entorno y analizar una sucesión de eventos pero empleando un sistema como Amaavem, que ofrece un soporte adicional, se puede aumentar y facilitar la vigilancia de entornos monitorizados. Debido a que resultaba complicado estudiar los resultados arrojados por el sistema en la base de datos, se propusieron una serie de alternativas para estudiar y analizar los resultados, que se detallan a continuación.

El cliente final de Amaavem tiene **dos formas** de analizar los resultados que el sistema ha volcado en la base de datos. Estas formas se enumeran a continuación:

- Analizando los resultados **de forma textual** leyendo el **Log del sistema**. Los programas no triviales hacen muchas acciones entre que toman los datos de entrada y generan la salida. Por tanto, es interesante tener un registro detallado de lo que va sucediendo.

  Todo lo que va aconteciendo en la traza del sistema se escribe en un archivo de texto diario, almacenado en la carpeta del proyecto (*/Log*). El nombre del fichero sigue un patrón *Día_Mes_Año* (ejemplo 01_03_12). El volcado de la información en dichos ficheros se hace posible gracias a la clase *Logger*. La cabecera de esta clase (implementa el patrón *Singleton*) se muestra a continuación:

```
class Logger : public Singleton<Logger> {

public:
    Logger();
    ~Logger();
    void write(const char * text);
    void write_error(const char * text);
};
```

Esta clase implementa la funcionalidad descrita con anterioridad en su operación *write*, que simplemente abre o crea un fichero de texto con el formato previamente citado, escribe el texto que recibe por parámetro y cierra el fichero. A continuación se muestra el código de esta clase:

```
void Logger::write(const char* text) {

    time_t tiempo = time(0);
    struct tm *tlocal = localtime(&tiempo);
    char output[128];
    char hour[128];
    strftime(output,128,"./Log/%d_%m_%y",tlocal);
    strftime(hour,128,"<%H:%M:%S> ",tlocal);
    // Creates an output file
    // the name is the date of the day DD_MM_YEAR
    fstream file ;
    file.open(output, fstream::in | fstream::out | fstream::
       app);
    // Write the text
    file << hour ;
    file << text << endl;
    // Close the file
    file.close();
}
```

El formato de los mensajes que se almacenan en el log siguen el siguiente patrón:

`<HORA_LOCAL>[CLASE] Mensaje`

A continuación se muestra un pequeño fragmento de un log del sistema Amaavem:

```
<19:55:08> [EventGenerator] Generate all events
<19:55:09> [AgentFactory] Instantiating Agent Person with id: 0
<19:55:09> [AgentFactory] Instantiating Agent Vehicle with id: 0
<19:55:10> [Agent]0 deleted because wasn't the type correct
<19:55:11> [AgentFactory] Instantiating Agent Person with id: 1
<19:55:11> [AgentFactory] Instantiating Agent Vehicle with id: 1
<19:55:11> [BehaviorCheckDistances] Person ID:0 near to the Car
   ID:1 .NumFrame: 455 Value Normality:= 0.6
<19:55:11> [BehaviorCheckDistances] Person ID:0 near to the Car
   ID:1 .NumFrame: 456 Value Normality:= 0.6
```

También, se incluye otra función en el log llamada *write_error*, que realiza la misma funcionalidad que la anterior, pero escribe en distinto formato y en otro fichero de

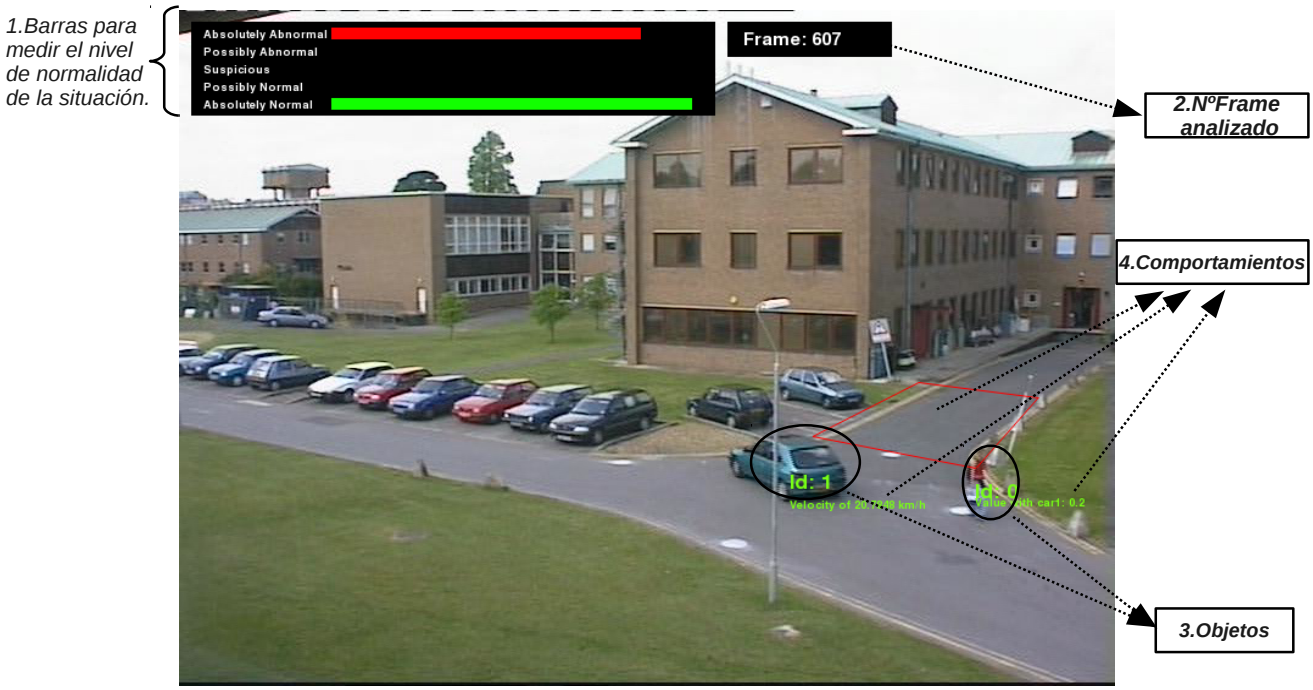


Figura 5.16: Ejemplo de un análisis de resultados

texto, en concreto en ficheros con el nombre *Día_Mes_Año_error*. Está función fue implementada para volcar los errores típicos que surgen en la ejecución de aplicaciones *Ice*.

- Analizando los resultados **de forma visual**, empleando el módulo incluido en la carpeta (*/Debug*) e implementado en el fichero *video.py*. Este módulo trabaja a partir de un conjunto de imágenes, que representan cada frame del evento de vídeo. Dichas imágenes son modificadas, empleando el módulo *PyGame*.[5]

  Dicho módulo recibe una serie de parámetros de entrada, modifica las imágenes de los *frames* añadiéndole información adicional, y finalmente, guarda estos *frames* modificados como imágenes en una carpeta. Los parámetros que recibe como entrada se enumeran a continuación:

  - Carpeta de origen, donde se encuentran las imágenes de los *frames* a estudiar.
  - Carpeta de destino, donde se almacenarán los *frames* modificados.
  - *Frame* inicial, para iniciar el análisis.
  - *Frame* final, para detener el análisis.
  - Número de cámara a analizar.

[5]Conjunto de módulos del lenguaje de programación Python que permiten la creación de videojuegos en dos dimensiones de una manera sencilla. Pygame también puede utilizarse para crear otros programas multimedia o interfaces gráficas de usuario.



Figura 5.17: Ejemplo de otro análisis de resultados

Para comprender mejor como trabaja este modulo, en la figura 5.16 y 5.17, se puede observar el resultado que arroja dicho módulo, para un frame específico. Los elementos que forman parte de este módulo se enumeran y describen a continuación:

1. **Barras que miden los comportamientos**. Estas barras miden la normalidad de la situación analizada. Previamente, en la fase de análisis de comportamientos, los agentes inteligentes miden el grado de normalidad de la situación estudiada. Entonces, con este módulo se lee la información, que fue guardada en la base de datos, y se plasma en la escena modificando las imágenes, gracias al uso de *PyGame*.

   En cuanto a cómo se distribuyen los valores de dichas barras, existen cinco tipos de niveles de barras, separados por un rango de 0.2 y sabiendo que el valor de normalidad está siempre comprendido entre 0 y 1.

   Dichos valores se enumeran y se describen a continuación (empezando por la barra superior hasta la barra inferior):

   - **Total normalidad**: Esta valor significa que el comportamiento tratado está en los límites de normalidad total. Este caso se da cuando el valor de normalidad, calculado por el agente inteligente, está comprendido entre 0.8 y 1.
   - **Posible normalidad**: Esta valor significa que el comportamiento tratado está un poco por debajo de la normalidad total. Este caso se da cuando el valor de normalidad está comprendido entre 0.6 y 0.8.

- **Sospechoso**: Esta valor significa que el comportamiento tratado es sospechoso. Este caso se da cuando el valor de normalidad está comprendido entre 0.4 y 0.6.
- **Posible anormalidad**: Este valor significa que el comportamiento ha superado los límites de sospecha y se encuentra en una posible anormalidad. Este caso se da cuando el valor de normalidad está comprendido entre entre 0.2 y 0.4.
- **Total anormalidad**: Este valor significa que el comportamiento tratado se encuentra en un índice elevado de peligrosidad y se da una situación de total anormalidad. Es el peor de los casos y se deben lanzar una serie de alarmas en caso de que sea preciso. Este caso se da cuando el valor de normalidad está comprendido entre 0 y 0.2.

Por cada comportamiento detectado en un frame específico, se dibuja en el diagrama de barras el valor de normalidad que representa a dicho comportamiento. Por ejemplo, si se da el caso que un vehículo circula a una velocidad de 110 Km/h en un territorio urbano, el agente inteligente analiza la situación y determinará que dicho valor de normalidad es de 0.1. Por tanto, se debería llenar la barra que hace referencia al rango de *total anormalidad*.

2. **Número de frame que se está analizando**. Este número simplemente hace referencia al número de *frame* que se está analizando en cada momento.
3. **Objetos en movimiento**. Los objetos en movimiento, previamente detectados por un sistema de *tracking* auxiliar, son modificados añadiéndoles una etiqueta con su identificador numérico. Por cada objeto detectado, existe un agente inteligente, que estudiará el comportamiento del mismo. El objeto detectado siempre tendrá su identificador, y debajo del mismo, la leyenda del comportamiento detectado en el frame actual.
4. **Comportamientos definidos**. A parte de rellenar las barras de normalidad, se escribe una pequeña explicación de los comportamientos, detectados en los *frames* estudiados, para que el cliente final pueda entender que está sucediendo en todo momento de la escena.

   Por ejemplo, en la figura que se está estudiando, el vehículo con id = 1, tiene una línea escrita, que hace referencia al comportamiento detectado, el cual dice *Velocity = 20.79 Km/h*. El cliente puede intuir que este coche tiene un comportamiento normal, en cuanto a su velocidad, debido a que está entre los límites de velocidad permitidos.

En resumen, este modulo fue creado para que el cliente tenga más facilidades a la hora de analizar una serie de eventos, en especial, para eventos de vídeo. El cliente tiene dos formas de analizar los eventos: de manera *textual* o *visual*.

Capítulo 6

Evolución, Resultados y Costes

En este capítulo se describen las fases e iteraciones de desarrollo de Amaavem, especificando las hitos característicos de cada una, y mostrando datos relacionados con la complejidad y el coste temporal. Asimismo, se aportan datos del *profiling* (rendimiento) del sistema, en el caso de estudio propuesto de la arquitectura empleada.

6.1. Evolución

En esta sección se muestran las diferentes iteraciones seguidas en el desarrollo de Amaavem, describiendo la información relevante a cada de ella tal como hitos alcanzados, complejidad, esfuerzo y otros detalles.

En primer lugar, se realizaron varios reuniones con el director del proyecto para determinar los requisitos previos del proyecto Amaavem. Se crearon una serie de documentos que describían de forma genérica la arquitectura basada en un sistema multi-agentes y algunas de las funcionalidades que debería cubrir. Se estudiaron los problemas que presentan arquitecturas similares a Amaavem y se definieron una serie de objetivos que debe cumplir la arquitectura. Estos objetivos iniciales fueron los siguientes:

- La arquitectura debe estar basada en bibliotecas libres y estándares abiertos para conseguir un sistema multiplataforma.
- La arquitectura debe ser adaptable a cualquier dominio monitorizado.
- La arquitectura debe ser modular y que resulte fácil modificar y agregar módulos para conseguir una alta expansibilidad.
- La arquitectura debe ser fácilmente escalable para dar soporte a los máximos usuarios posibles. En los sistemas de vigilancia inteligente, el rendimiento es un aspecto importante a tener en cuenta y por ello, se pretendió crear un sistema muy escalable.
- La arquitectura debe emplear patrones de diseño en la resolución de problemas planteados en situaciones similares.
- La arquitectura debe emplear técnicas distribuidas para aumentar su rendimiento en caso de que sea preciso.

En segundo lugar, tras la definición de dichos objetivos generales, se determinaron los componentes iniciales que iban a formar parte de Amaavem. En las sucesivas reuniones se determinaron los siguientes componentes:

- **Generador de eventos**: encargado de generar los eventos a partir del entorno monitorizado.
- **Agente de base de datos**: encargado de realizar las gestiones con la base de datos para abstraer a los demás componentes de manejar con bases de datos.
- **Monitor de vigilancia**: encargado de recoger los eventos que genera el generador de eventos, procesarlos y distribuirlos a los agentes del sistema.
- **Factoría de agentes**: encargada de instanciar los distintos agentes en el sistema.
- **Agentes inteligentes**: encargados de analizar los eventos generados en etapas anteriores y determinar el grado de normalidad de las situaciones estudiadas.

Finalmente, se comenzó con el desarrollo del proyecto (10 enero de 2012). En la siguiente sección se explican detalladamente las iteraciones que se siguieron en el desarrollo de Amaavem.

6.1.1. Iteraciones

En esta sección se muestra la división en iteraciones de desarrollo de Amaavem, especificando los hitos característicos de cada una, y mostrando datos relacionados con la complejidad y el coste temporal.

Iteración 1

En esta iteración inicial se creo la **estructura básica de directorios** que alojan Amaavem. Una de las premisas iniciales para este proyecto fue conseguir que el programa sea fácil de usar, por tanto, se construyó un archivo *Makefile* que abstrae al desarrollador de realizar las compilaciones complejas, que conllevan la creación de proyectos de envergadura considerable. Asimismo, se creo un pequeño manual, donde se indican los pasos necesarios para compilar Amaavem.

Tras las primeras reuniones con el director del proyecto, se decidió empezar a implementar los módulos más sencillos y acabar con los módulos más complejos en iteraciones posteriores. Entonces, en esta iteración se implementó el **agente de base de datos**, ya que era la parte más independiente de la arquitectura y la cual no debería causar muchos problemas. Previamente a la implementación final del agente de base de datos, se crearon las estructuras de datos necesarias para albergar la información que fluye por el sistema, así como los esquemas *SQL* de la base de datos concreta. Estas estructuras de datos se incluyeron en un archivo de configuración general para la arquitectura Amaavem (archivo *config.ice* de la carpeta */src*).

Como resultado, en esta iteración no se obtuvo ningún prototipo funcional pero se implementó la primera versión del agente de base de datos (*AgentBDI*), que implementa las operaciones *CRUD* de una base de datos relacional, y las estructuras de datos básicas de Amaavem.

En resumen, esta primera iteración sirvió para crear el esqueleto básico necesario para poder alojar las funcionalidades deseadas por el proyecto Amaavem.

Iteración 2

Una vez finalizada la primera iteración, en la que se obtuvieron las estructuras de datos básicas y un *broker* de base de datos, se dispuso a implementar la **lógica del generador de eventos**. El generador de eventos hace uso del agente de base de datos para obtener la información relevante de cada evento.

El generador de eventos no va a ser un objeto *Ice* debido a que va a representar la parte de un cliente en un aplicación *Ice*, pero debe contener un comunicador de *Ice* y así poder comunicarse con otros objetos remotos instanciados en el sistema.

La complejidad en esta iteración no fue muy elevada. Tan sólo hubo una complejidad y ésta estuvo en la conversión de los datos almacenados en la base de datos a las estructuras de datos definidas previamente en la iteración primera. En la base de datos simplemente se almacenan datos concreto, como por ejemplo un número entero o un texto, pero en las estructuras de datos de Amaavem los eventos pueden contener una lista de objetos. Entonces, la lista de eventos, que a su vez contiene una lista de objetos, debía ser compuesta por el generador de eventos, implementado en la presente iteración.

Para probar el prototipo obtenido por esta iteración, se realizó una sencilla batería de pruebas. Previamente, se volcó en la base de datos un archivo XML con información de una monitorización en un aparcamiento de una universidad de Nevada. Posteriormente, fue lanzado el generador de eventos para testear que la composición de la lista de eventos funcionaba correctamente y también, se probaron las distintas funcionalidades del generador de eventos en función de los parámetros fijados. Tras una serie de correcciones en la implementación, se obtuvo un prototipo totalmente funcional en la presente iteración.

En cuanto al volcado de datos XML a la base de datos, se implementó un pequeño modulo en *Python* llamado *parser.py*, que procesaba la información del fichero y la volcaba a la base de datos. El procesamiento de la información se realizó usando un *plugin* de *Python* de uso trivial, en concreto el *plugin minidom*.

En resumen, en esta iteración se creo el módulo de generación de eventos y un pequeño submódulo de inserción de información en la base de datos a partir de ficheros XML.

Iteración 3

El sistema de generación de eventos ya estaba implementado y los esfuerzos de esta iteración se centraron en implementar un monitor de vigilancia (*VigilanceManagerI*) que procesara los eventos, generados previamente por el módulo de la iteración 2.

El **monitor de vigilancia** tiene la responsabilidad de procesar los eventos que el cliente solicite analizar. Para ello, el flujo de datos de los eventos corre desde el generador de eventos al módulo que se implementa en la presente iteración.

El proceso de implementación de este módulo se produjo de manera trivial debido a la baja complejidad de este módulo. Este módulo simplemente recogía los eventos, los procesaba, y distribuía al canal de eventos. En esta iteración no se introdujo todavía el concepto de *IceStorm* para el envío de los eventos a los agentes instanciados y quedó indicado para una refinación del código en etapas futuras.

Debido a que toda la arquitectura local no estaba implementada, no sería adecuado introducir ya todos los servicios *Ice* cuando no se podían probar. Se decidió completar la arquitectura en vista local y posteriormente, refinar el código e introducir las funcionalidades que ofrece el *middleware ZeroC Ice*.

En resumen, en esta iteración se obtuvo un prototipo inicial del monitor de vigilancia. No pudo ser probado en su totalidad debido a las dependencias que tiene con otros módulos que serán implementados en iteraciones futuras.

Iteración 4

En esta iteración se centro la atención en la creación de diferentes tipos de agentes mediante el uso de una **fabrica abstracta de agentes** (*AgentFactoryI*). Dicha fábrica, diferente para dominio, tiene la responsabilidad de instanciar los diferentes agentes del sistema. Igual que en el módulo anterior, y hasta no tener la arquitectura local completada, se implementó este modulo de manera local y en una iteración futura se refinaría el código para conseguir su versión remota.

Tras varias reuniones con el director, se decidió emplear bibliotecas dinámicas para resolver el problema de la instanciación. El uso de las bibliotecas dinámicas era una complejidad importante en el desarrollo debido a que nunca había implementado algo similar. Se dedicaron varios días a estudiar y comprender cómo se trabaja con bibliotecas dinámicas en *GNU Linux* y en concreto, a la biblioteca ofrecida por el sistema operativo *GNU Linux* llamada *dlfcn.h*.

Una vez adquirido el conocimiento necesario, se implementó una factoría concreta para un entorno urbano, en la que se podían instanciar agentes de personas y agentes de vehículos.

El prototipo de esta iteración avanzaba un poco más, y la arquitectura ya podía generar eventos, procesarlos e instanciar agentes de diferentes tipos.

En resumen, la complejidad de esta etapa fue elevada por el tema de las bibliotecas dinámicas y se obtuvo un prototipo funcional de la fábrica de agentes. Posteriormente, en unas iteraciones futuras se añadiría la funcionalidad remota.

Iteración 5

Los esfuerzos de esta iteración se centraron en la creación de la **lógica del agente** (clase *AgentI*). En la iteración anterior, la factoría tenia la posibilidad de instanciar agentes, pero sin lógica. Por tanto, en esta iteración se implemento la lógica de los agentes del sistema.

Los agentes tenían asociados una serie de comportamientos, y en esta iteración también se creo la lógica de los comportamientos (clase *Behavior*).

Para poder probar la arquitectura local, que quedó concluida tras la implementación de estos módulos, se implementaron dos tipos de agentes (clase *AgentPerson* y *AgentVehicle*) para el entorno urbano donde fue probada la arquitectura de Amaavem, así como un comportamiento asociado a los agentes de personas que medía las distancias entre las personas y los vehículos de la escena (clase *BehaviorCheckDistances*). Los comportamientos también hacen uso del agente de base de datos (creado en la primera iteración) para poder almacenar los análisis que hacían de las situaciones.

Una vez concluida la implementación de la arquitectura local, se realizaron una serie de pruebas para probarla. En primer lugar, se almacenó información de un vídeo en la base de datos. Posteriormente, se lanzó el sistema para analizar todos los *frames* de este vídeo. El monitor de vigilancia mandaba la información de los eventos procesados a los distintos agentes, previamente instanciados por la factoría de agentes, y estos últimos analizaban la situación y volcaban las incidencias en la base de datos. Tras una serie de refinamientos en el código de los distintos módulos, se pudo dar por concluida la arquitectura local de manera satisfactoria.

Los objetivos planteados en la etapa inicial de análisis habían sido cubiertos casi en su totalidad, siendo especialmente cuidadosos en las características de la arquitectura, y logrando un producto satisfactorio.

En resumen, en esta iteración se obtuvo un prototipo totalmente funcional de la arquitectura local. El reto que se plantea a partir de ahora era lograr que esta arquitectura funcionara correctamente de manera remota, apoyándose en el soporte de un *middleware* de comunicaciones (*Ice*).

Iteración 6

Una vez concluida la arquitectura local, se decidió crear un **módulo de análisis de resultados** para poder analizar los resultados de manera visual o de manera textual.

Por un lado, para analizar los resultados de manera textual, se implementó un registro (clase *Logger*) de resultados. Todos los componentes del sistema escriban sus resultados o

sus errores según sea necesario. Los agentes volcaban tanto en la base de datos como en el *Logger* los análisis realizados en cada momento.

Por otro lado, para analizar los resultados de manera visual, se implementó un módulo de análisis visual (clase *video.py*). Dicho módulo fue implementado en *Python* y se usó el módulo *PyGame* para resolver el problema de modificación de imágenes. Con este módulo, se modificaban los frames que se debían analizar, añadiendo información adicional referente a las incidencias almacenadas en la base de datos.

La complejidad esta iteración residió en el estudio y uso del módulo *PyGame*. Personalmente, había trabajado con este módulo en asignaturas de años pasados, pero poco en la funcionalidad que se precisaba (modificación de imágenes). Por tanto, se dedicaron unos días al estudio de las diferentes funcionalidades que ofrece este módulo para poder alcanzar el objetivo citado.

En resumen, en esta iteración se obtuvieron módulos adicionales a la arquitectura para poder ver y analizar los resultados arrojados por la misma. La complejidad de esta iteración se puede considerar que fue de un nivel medio debido a las dificultados que surgieron al usar *PyGame*.

Iteración 7

Sin duda, esta fue la iteración mas compleja y densa en el desarrollo de Amaavem. Uno de los objetivos fundamentales de la arquitectura era obtener un sistema distribuido adaptativo y con balanceo de carga, en el caso de que sea preciso.

Tras la obtención de la arquitectura local en iteraciones anteriores, en esta iteración se centraron los esfuerzos en obtener las mismas funcionalidades locales pero de manera distribuida. Para obtener el sistema remoto se empleó el *middleware* de comunicaciones *ZeroC Ice* o *Ice*.

El trabajo con sistemas distribuidos fue abordado desde cero y supuso un gran esfuerzo para comprender dichos sistemas. Se dedicó cerca de un mes completo para analizar y estudiar todo el manual de usuario propuesto en [Ice11]. A medida que se iba estudiando el manual de usuario de *Ice*, se realizaban todos los ejemplos propuestos por el mismo, lo cual supuso un gran gasto de tiempo y una parada en el desarrollo de Amaavem.

Tras este mes de estudio y trabajo, se dispuso a implementar los diferentes servidores y servicios para resolver los problemas de comunicación. En primer lugar, se implementó el agente de base de datos como una biblioteca dinámica y cargada como un servicio *IceBox*. En segundo lugar, se implementaron el servidor para el monitor de vigilancia y para la factoría de agentes. Para resolver el balanceado de carga, se optó por tener tantas factorías distribuidas, determinadas previamente por el desarrollador, distribuidas por diferentes *hosts* de una red. Finalmente, se uso *IceStorm* en la creación de un canal de eventos y un canal de comunicaciones para resolver la comunicación de los agentes con los diferentes módulos.



Figura 6.1: Ejemplo de caso de estudio. Calle de una universidad de USA.

El código fuente de cada módulo tuvo que ser finamente refinado para ofrecer las mismas funcionalidades pero de manera remota.

Tras una serie de largas pruebas de la arquitectura en diferentes *hosts*, el desarrollo de Amaavem fue finalizado, logrando los objetivos planteados en la etapas de análisis. A partir de este punto, se empezó a desarrollar la documentación y el manual de usuario.

A continuación se muestra un cuadro con el resumen de las fechas de cada iteración.

Iteración	**Fecha de entrega**
Iteración 1º	20 de Enero de 2012
Iteración 2º	3 de Febrero de 2012
Iteración 3º	17 de Febrero de 2012
Iteración 4º	2 de Marzo de 2012
Iteración 5º	16 de Marzo de 2012
Iteración 6º	2 de Abril de 2012
Iteración 7º	4 de Mayo de 2012

Cuadro 6.1: Calendario de las iteraciones

6.2. Caso de estudio de la arquitectura

En el desarrollo de Amaavem, era necesario la creación de uno o varios casos de estudio que sirvieran como una base de explotación de la arquitectura, así como un banco de pruebas y un sistema de depuración adecuado para corregir los problemas típicos en el desarrollo de software.

El caso de estudio de la arquitectura planteada, se realizó pensando en un entorno urbano común, tal como podría ser una universidad, una puerta de un colegio, una calle concurrida de una ciudad, un paso de cebra peligroso, una zona de salida nocturna, etc.

Para este caso en concreto, se tomaron como ejemplos una serie de vídeos de una calle de una universidad de los Estados Unidos (véase figura 6.1 para ver un frame ejemplo de estos vídeos). En cuanto al *tracking* de los objetos en las escenas, se obtuvo gracias a una serie de archivos *XML* definidos en la página web de un grupo de investigadores del área visión por computador [CVG08]. A partir de dichos archivos *XML*, se ejecuto el *parser*, creado en etapas anteriores, y los resultados fueron volcados en la base de datos.

Una vez se obtuvo un *tracking* correcto de todos los objetos que aparecían en los vídeos, se crearon los distintos tipos de agentes para analizar los diferentes tipos de objetos de la escena. En entornos urbanos, los objetos predominantes son las personas y los vehículos. Por ello, se decidió crear agentes para monitorizar personas y agentes para monitorizar vehículos.

En la figura 6.2, se puede observar un diagrama de clases de dichos agentes. Además, se observa una clase auxiliar de apoyo (*urbanOperations*, incluida en la carpeta */src*), empleada por dichos agentes para resolver operaciones típicas relacionadas con un entorno urbano.

En concreto para esta clase, se definieron una serie de operaciones para determinar si un objeto es de tipo persona o de tipo vehículo. En caso de que el desarrollador desee agregar nuevas operaciones, tan solo debe modificar esta clase o añadir una nueva clase. El agente especifico que desee emplear esta clase debe seguir el siguiente patrón, tal y como se muestra en el siguiente fragmento de código:

```
1 // Object is a car...
2 if (urbanOperations::checkVehicle(width ,height))
3     ...
4 else
5     ...
```

Siguiendo con el modelo modular, extensible y adaptativo de la arquitectura general planteada, el objetivo de esta clase es por un lado, evitar repetir código cuando se comprueban los tipos de objetos y por otro lado, la creación de otra capa de abstracción entre la arquitectura general y un caso concreto donde se despliegue Amaavem.

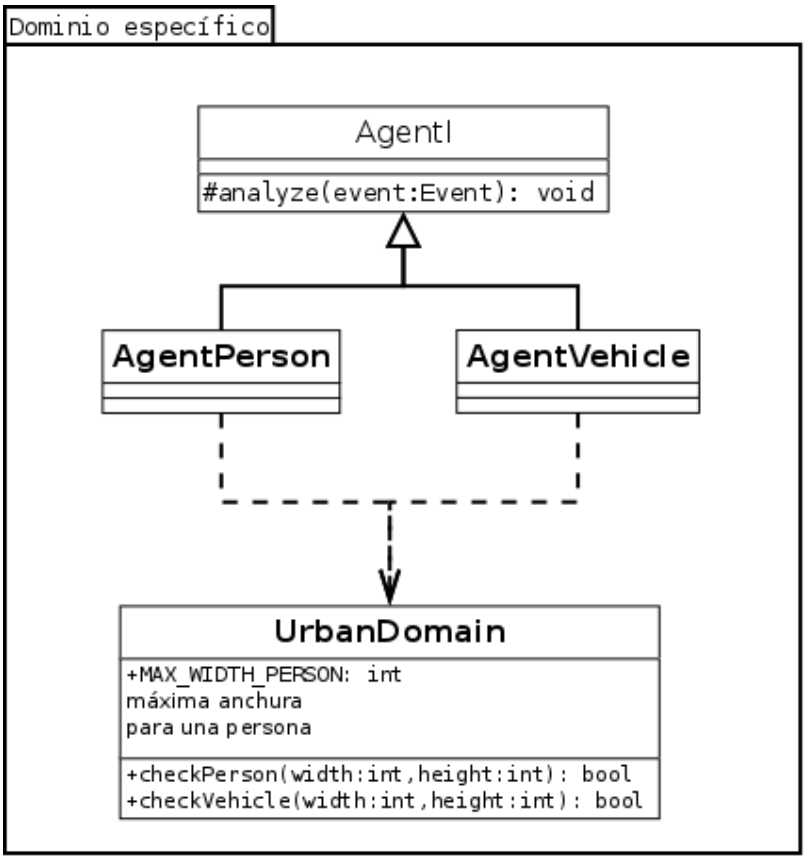


Figura 6.2: Diagrama de los agentes para un dominio concreto.

El código completo de la clase *urbanOperations* se muestra a continuación:

```
#include <UrbanDomain/urbanOperations.h>
namespace pfc {
    namespace urbanOperations {
        // Return true is the object is a vehicle
        bool checkVehicle(const int& width, const int& height){
            if (( width >= height) || width > MAX_WIDTH_PERSON)
                return true ;
            return false ;
        }
        // Return true is the object is a person
        bool checkPerson(const int& width, const int& height){
            if (( width < height) && width <= MAX_WIDTH_PERSON)
                return true ;
            return false ;
        }
    }
}
```

Una vez explicada la funcionalidad de las clases de apoyo para los agentes específicos, es importante explicar cómo los agentes implementan la operación de análisis de eventos. Los agentes especializados deben implementar la operación *analyze*, que es la operación donde se analizan los comportamientos, especificados previamente por el desarrollador. Estos agentes

tienen asociados una serie de comportamientos, que serán los encargados de medir el nivel de normalidad de las situaciones, que sucedan en los eventos.

La implementación de los agentes especializados, que heredan de la clase *AgentI*, deben seguir un mismo patrón. Para comprender mejor como realizar implementaciones de agentes específicos, se incluye la cabecera de uno de ellos (ej. AgentVehicle).

```
#ifndef AGENTVEHICLE_H
#define AGENTVEHICLE_H
#include <config.h>
#include <AgentI.h>
#include <BehaviorCheckStops.h>
#include <BehaviorCheckVelocity.h>

using namespace std;
namespace pfc {
    class AgentVehicle : public AgentI
    {
        public:
            AgentVehicle(int id, const Ice::ObjectAdapterPtr& adapter)
                : AgentI(id,adapter){
                this->addBehaviorsVehicle();
            }
            void analyze(const Event& event) ;
            void addBehaviorsVehicle();
    };
};
#endif
```

Los agentes en su inicialización, deben crear las instancias de los comportamientos, que van a necesitar para analizar las situaciones. Por ejemplo, se estudia el caso para los vehículos. La clase *AgentVehicle* tiene un método (*addBehaviorsVehicle*), que instancia los diferentes comportamientos, que son requeridos por este tipo de agentes. A continuación, se incluye el código de este método, para que el desarrollador final de Amaavem comprenda como se instancian los comportamientos:

```
   // Method that adds behavior of the Vehicle
    void AgentVehicle::addBehaviorsVehicle(){
        // BehaviorCheckStops
        // to instantiate the behavior : Resulution (x, y) and frames
           to check the time stopped
        Behavior* b1 = new BehaviorCheckStops(768,576,160) ;
        static_cast<BehaviorCheckStops*>(b1)->addZone(512,214,774,255)
           ;
        this->addBehavior(b1);
    }
```

Una vez explicada la arquitectura de estos agentes concretos, el siguiente paso es explicar que **comportamientos** se definieron para el dominio en concreto. Éstos se enumeran a continuación:

- Comportamiento que mide el nivel de peligrosidad de una persona, en cuanto a la **distancia entre las personas y los vehículos**. Este comportamiento se asoció a los agentes personas y medía la distancia euclídea con los vehículos de la escena, determinando el nivel de normalidad en función de dicha distancia. Cuanto más cerca están los objetos, se obtiene menos grado de normalidad.
- Comportamiento que mide la **velocidad de los vehículos**. Este comportamiento se asoció a los vehículos y mide la velocidad de los vehículos de la escena.
- Comportamiento que detecta **paradas ilegales de los vehículos**. Este comportamiento se asoció a los vehículos y detectaba cuando un coche se paraba por un tiempo, definido por el desarrollador, en un zona, definida también por el desarrollador.

Los posibilidades son diversas y queda a gusto del desarrollador la creación de tantos comportamientos como se necesiten. El objetivo primordial de este proyecto no era crear un caso de estudio muy complejo, sino dar los medios necesarios para que un desarrollado pueda desplegar Amaavem en el dominio que precise, y posteriormente, definir los agentes y comportamientos que necesite para sus intereses.

Para que el desarrollador final tenga una visión mejor de cómo crear comportamientos, se muestra él código de uno de ellos, en concreto, del comportamiento que mide las distancias entre personas y vehículos. La cabecera de dicha clase es la siguiente (debido a que el código fuente completo es demasiado largo, se puede observar en la guía de desarrollo, incluida en el anexo B.

```
2     class BehaviorCheckDistances : public Behavior
3     {
4         private:
5             resolution res; //Resolution of the image
6         public:
7             BehaviorCheckDistances(const int& xMax, const int& yMax):
                  Behavior(){
8                 res.x = xMax;
9                 res.y = yMax;
10            }
11            ~BehaviorCheckDistances();
12            double getDistance(const int& myX, const int& myY, const
                  int& x, const int& y);
13            double getValueNormality(const double& distance);
14            int getXmax();
15            int getYmax();
16            double analyze(Ice::Communicator& communicator,const int&
                  id, const Event& event);
17    };
18 };
```

Este comportamiento en concreto, mide las distancias entre las personas y los vehículos. Para ello, cuando dicho comportamiento recibe el evento, a través del agente asociado, comprueba si su identificador está incluido en el evento recibido. En tal caso, debe comprobar si en la información del evento se encuentra algún vehículo. Entonces, cuando en el evento se encuentra el objeto que debe analizar y además, se encuentran vehículos en la escena, el comportamiento medirá la distancia entre la persona y el vehículo. Una vez medida dicha distancia, el comportamiento normalizará este valor entre 0 y 1, siendo 0 la anormalidad total y 1 la normalidad total. Tras dicha medición, el comportamiento generará una incidencia y la almacenará en la base de datos, haciendo uso del agente de base de datos.

El código de esta operación se incluye en la operación *analyze* de la clase *BehaviorCheckDistances*, tal y como se muestra a continuación:

```
2  double BehaviorCheckDistances::analyze(Ice::Communicator& communicator
       ,const int& id, const Event& event){
3          // Analyze the specific behavior of the an event video
4          const EventVideo* eventV  = static_cast<const EventVideo*>(&
               event);
5          vector<objTracking> auxObjs = eventV->objects ;
6          vector<objTracking>::const_iterator it;
7          int myX = 0, myY = 0;
8          double euclideanDistance = 0.0, normality =0.0;
9          string text ;
10         string textError;
11         bool iAmPerson = false ;
12         incidence incident;
13         incident.type = "DistancePersonVehicle";

15         // Save our X and our Y
16         for ( it = auxObjs.begin(); it != auxObjs.end() ; ++it){
17             if ( (*it).id == id){
18                 myX = (*it).position.x ;
19                 myY = (*it).position.y ;
20                 iAmPerson = urbanOperations::checkPerson ( (*it).
                       details.width , (*it).details.height)  ;
21                 break;
22             }
23         }


26         // Search the vehicles and get the distance
27         for ( it = auxObjs.begin(); it != auxObjs.end() ; ++it){
28             // If the other object is a Vehicle and the id = My Id and
                   I am a person -> Analyze

30             if( urbanOperations::checkVehicle((*it).details.width , (*
                   it).details.height) && (id != (*it).id) && iAmPerson )
                   {
31                 // Obtain the euclidean distance
32                 euclideanDistance = getDistance(myX,myY,(*it).position
                       .x,(*it).position.y) ;
33                 // Normalize this value
```

```
34                  normality = getValueNormality(euclideanDistance);

36                  text = "Person ID:" +ops::numToString(id)+" near to
                        the Car ID:"+ops::numToString((*it).id);
37                  text = text +" .NumFrame: "+ops::numToString(eventV->
                        frame)+" Value Normality:= "+ ops::numToString(
                        normality);

39                  // Print the results in the Log
40                  Logger::getInstance()->write(text.c_str());

42                  // Save the results in the BD
43                  incident.camera = eventV->camera ;
44                  incident.numFrame = eventV->frame;
45                  incident.id = id ;
46                  incident.id2 = (*it).id ;
47                  incident.normality = normality ;
48                  // Call to AgentBD by ICE
49                  AgentBDPrx agent = AgentBDPrx::uncheckedCast(
                        communicator.propertyToProxy("AgentBD.Proxy"));
50                  if(!agent)
51                  {
52                       textError = "[BehaviorCheckDistances][analyze]
                            Invalid to obtain the proxy AgentBD";
53                       Logger::getInstance()->write_error(textError.c_str
                            ());
54                  }
55                  agent->updateBD(incident);
56              }
57          }

59          return incident.normality;

61      }
```

Además, la creación de un caso de estudio, supuso un avance importante en los problemas de la arquitectura, debido a que sirvió como un sistema de depuración de la misma. Según se iban implementando agentes o comportamientos y ejecutando el caso de estudio, se iban realizando casos de pruebas para comprobar la funcionalidad del sistema.

A medida que se iba implementando la arquitectura, surgían muchos problemas típicos al afrontar un proyecto de tal envergadura. Estos problemas se resolvían a medida que iban surgiendo. El programa era lanzado en diferentes situaciones para probar el código en sus totalidad.

Los problemas más relevantes que surgieron desarrollando Amaavem se describen a continuación:

- **Problemas de comunicación**. Debido a que la curva de aprendizaje del *middleware ZeroC Ice* era elevada, surgían bastantes problemas al ejecutar el caso de estudio. Estos problemas se iban resolviendo observando problemas similares resueltos el página oficial y el foro de *ZeroC Ice* [Ice11].

- **Problemas con los comportamientos**. Desde la implementación de un comportamiento concreto hasta el funcionamiento funcional del mismo, pasaban una serie de test para verificar que el comportamiento almacenaba información adecuada. Por ejemplo, para medir la velocidad de los coches a partir de una trayectoria, almacenada en la base de datos, se tenían que realizar muchas pruebas para optimizar el valor de un *pixel* en metros.
- **Problemas en la destrucción de agentes**. Los agentes específicos eliminados del sistema, seguían recibiendo eventos del canal de eventos. Esto surgía debido a que la instancia del agente y el sirviente, que escucha las invocaciones remotas, tienen ciclos de vida distintos. Entonces, se corrigió este error eliminando ambas instancias.

Como **conclusión**, destacar las facilidades que ofrece Amaavem en la creación de un sistema de vigilancia inteligente. Si se hubiera querido desarrollar este sistema desde cero, para este caso de estudio concreto, el trabajo a realizar hubiese sido bastante elevado. Empleando la arquitectura Amaavem, muchos de los problemas típicos en la creación de este tipo de sistemas han sido solventados. El desarrollador final sólo se debe preocupar de decisiones de dominio y olvidarse de problemas de comunicaciones, escalabilidad, adaptabilidad y otros problemas. Estas decisiones del desarrollador final se describen a continuación:

- Implementación de los **tipos de agentes** que existirán en el sistema, en función de los tipos de objetos que hay en el entorno.
- Implementación de los **comportamientos** de dichos agentes, para comparar con sus líneas de conducta definidas y decidir cuanto difieren de lo acontecido.
- Establecer las **opciones de pertenencia** de un objeto a un tipo, en función de sus características. (operación classify de la clase *AgentI*).
- Definir los **grados de normalidad** de las situaciones, para que los agentes puedan cuantificar cuanto difiere lo que esté pasando con sus comportamientos predefinidos.

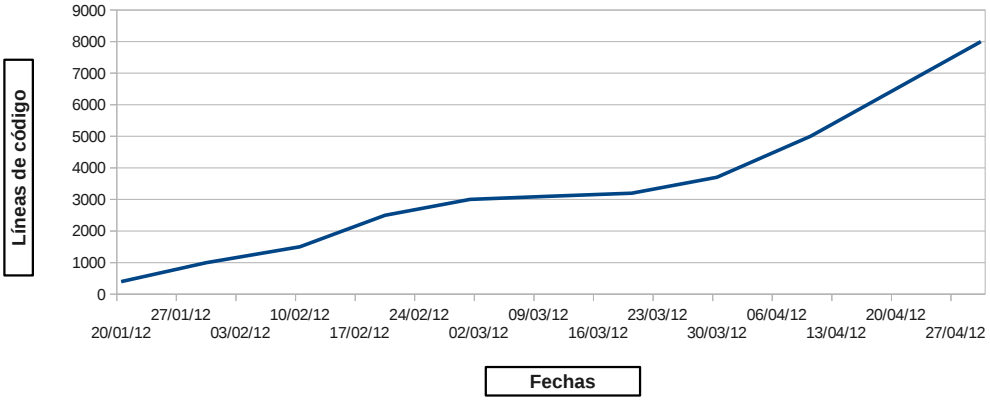


Figura 6.3: Gráfico de actividad del repositorio de Amaavem.

6.3. Recursos y Costes

En este apartado se describen los diferentes recursos empleados en la realización de Amaavem. También se ofrece una estimación de lo que hubiera costado desarrollar un proyecto de estas características. Finalmente, se incluye un gráfico donde se puede apreciar la evolución del proyecto en lineas de código a lo largo del periodo de desarrollo.

6.3.1. Coste

El tiempo de implementación del presente proyecto final de carrera estuvo comprendido aproximadamente entre el 9 de Enero de 2012 y el 4 de Mayo de 2012, aunque el tiempo total del proyecto abarcó más tiempo debido a las reuniones previas para afianzar los requisitos de la arquitectura. Suponiendo un trabajo diario de 7 horas diarias y asumiendo 5 días de trabajo a la semana, suman 101 días de trabajo y unas 770 horas de trabajo aproximadamente. Considerando un precio típico de un programador C++ de 25€/hora, en el cuadro 6.2 se muestra el desglose económico de todos los recursos utilizados en el desarrollo de Amaavem.

Recurso	**Cantidad**	**Coste**
Sueldo programador	1	17.675 €
Portatil HP	1	600 €
Computadores Zotacs	3	1.500 €
Total		19.775 €

Cuadro 6.2: Coste total estimado del desarrollo de Amaavem.

6.3.2. Estadísticas

En cuanto al control de versiones, se ha utilizado un repositorio *Mercurial* ofrecido el servicio de *Google Code*. En la figura 6.3 se muestra un gráfico de la evolución de las líneas

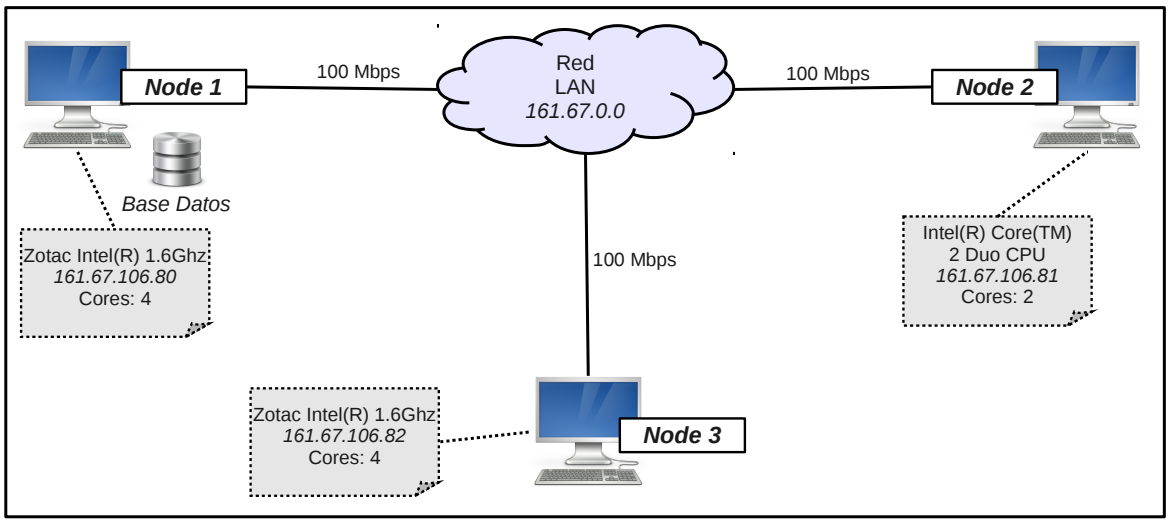


Figura 6.4: Despliegue para 3 nodos de la arquitectura Amaavem.

de código fuente durante toda la etapa de desarrollo. Según un informe de *Google Code*, se han realizado 130 actualizaciones del repositorio a lo largo del desarrollo.

Para controlar las líneas de código escritas, se ha empleado el programa *cloc*[1], que contabiliza las líneas de código fuente de cada lenguaje ligado a un proyecto. Cabe citar que *ZeroC Ice* genera código automáticamente (alrededor de 3.000 líneas de código). Por eso, se aprecia tanta subida en la etapa final del desarrollo, etapa en la que se añadió la funcionalidad remota a la arquitectura Amaavem.

En el cuadro 6.3 se puede apreciar un informe de la envergadura en líneas de código de este proyecto final de carrera, generado con el programa *cloc*.

Lenguaje	**Archivos**	**Espacios en blanco**	**Comentarios**	**Líneas de código**
Cabeceras C++	22	922	139	4166
C++	24	714	318	4300
Python	2	59	29	165
XML	1	4	7	94
Make	2	23	13	52
Total	**51**	**1823**	**497**	**8777**

Cuadro 6.3: Número de líneas de código fuente de Amaavem.

6.3.3. Rendimiento

En ingeniería de software el análisis de rendimiento, comúnmente llamado ***profiling***, es la investigación del comportamiento de un programa usando información del análisis dinámico

[1] http://sourceforge.net/projects/cloc/

del programa (cuando éste está ejecutándose con una configuración determinada previamente) en oposición al análisis estático (análisis de código).

El objetivo de realizar un análisis de rendimiento es determinar qué partes del código son cuellos de botella y consumen más recursos, y por tanto, son más susceptibles de ser optimizadas para poder ganar en rendimiento.

Ahora bien, el sistema Amaavem es un sistema distribuido y por ello, se ha medido el rendimiento del propio sistema en diferentes situaciones de configuración. En este caso, se midió el sistema en dos situaciones:

- En modo **local**, empleando solo un *host*, en concreto, un portatil HP Intel Core Duo.
- En modo **remoto**, empleando concretamente tres *hosts* de la misma red tal y como sigue en el diagrama de despliegue (véase la figura 6.4). En la figura 6.5, se puede observar el diagrama de componentes que surge al desplegar los distintos servidores y servicios en los diferentes *hosts*. El *host registry* es el que debe albergar la base de datos. Los demás *hosts* conocen la dirección *ip* del *registry* permitiendo la comunicación con los demás elementos del sistema, gracias al uso de las funcionalidades ofrecidas por *IceGrid*.

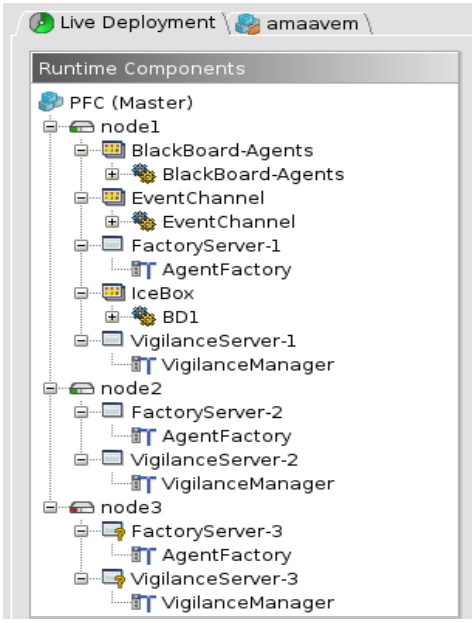


Figura 6.5: Diagrama de componentes Ice para 3 nodos.

Para este caso concreto de mediciones de rendimiento, se ha usado el caso de estudio propuesto en secciones anteriores. El entorno empleado es una calle de acceso a una universidad. Se observan objetos de tipo persona y de tipo vehículo. En el vídeo analizado, aparecen alrededor de 15 objetos y se instanciaron 3 tipos de comportamientos.

Los tiempos de análisis, para medir el sistema Amaavem en las dos situaciones previamente explicadas, se pueden observar en la figura 6.6. Los tiempos de comunicación entre los diferentes componentes que componen Amaavem se puede apreciar en la figura 6.7.

TIEMPOS DE ANÁLISIS

GENERADOR de EVENTOS

	1 NODO	3 NODOS
GenerateEventsVideo // aprox. 400 frames	0.14 seg	0.15 seg
GenerateAllEventsVideo // aprox. 3000 frames	2.82 seg	3.2 seg

AGENTE BASE de DATOS

	1 NODO	3 NODOS
Insert	0.01 seg	0.1 seg
Select	0.01 seg	0.1 seg
Delete	0.01 seg	0.1 seg
Init	0.01 seg	0.1 seg

MONITOR de VIGILANCIA

	1 NODO	3 NODOS
run	0.41 seg	0.44 seg

FACTORÍA de AGENTES

	1 NODO	3 NODOS
createAgent	0.02 seg	0.02 seg

AGENTES

	1 NODO	3 NODOS
analyze	0.001 seg	0.001 seg
subscribe	0.02 seg	0.1 seg

COMPORTAMIENTOS

	1 NODO	3 NODOS
analyze	0.001 seg	0.001 seg

Figura 6.6: Tiempos de análisis de la arquitectura Amaavem.

En cuanto al tiempo de despliegue, que hace referencia al tiempo que pasa entre que se despliega el primer servidor hasta el último, se enumera a continuación:

- Para *1 Nodo*: el tiempo de despliegue es aproximadamente *1.1 segundos*.
- Para *3 Nodos*: el tiempo de despliegue es aproximadamente *4.1 segundos*.

A continuación, se va a realizar un estudio de los tiempos obtenidos. A priori, la conclusión más relevante obtenida es que *ZeroC Ice* no supone ningún problema de tiempo adicional

TIEMPOS DE COMUNICACIÓN

Comunicación con	**1 NODO**	**3 NODOS**
Monitor de Vigilancia	0.0001 seg	0.1 seg
Agente de Base de Datos	0.0001 seg	0.1 seg
Factoría de Agentes	0.0001 seg	0.1 seg
Agentes	0.0001 seg	0.1 seg
Canal de Eventos	0.0001 seg	0.1 seg

Figura 6.7: Tiempos de comunicación de la arquitectura Amaavem.

en las comunicaciones de la arquitectura. Se puede apreciar cierta diferencia en ejecutar el caso de estudio, para un nodo o para tres nodos, pero la diferencia de tiempos es totalmente sostenible y no supone un cuello de botella para el sistema.

Como se puede apreciar en la figura 6.7, los **tiempos de comunicación** entre componentes son insignificantes, y se puede concluir que *ZeroC Ice* ha sido un buen recurso para plantear la arquitectura remota.

En cuanto a los **tiempos de ejecución** de las operaciones más importantes de la arquitectura (figura 6.6), se puede observar una diferencia clara entre analizar pocos eventos a analizar muchos, y es totalmente normal, debido a la cantidad de información que se debe procesar. La diferencia entre usar un nodo o usar tres nodos se considera totalmente irrelevante.

Y para concluir, el **tiempo de despliegue** aumenta cuantos más nodos se pretendan desplegar, y este aumento de tiempo es totalmente estándar debido a la cantidad de componentes aumenta, y la aplicación debe ser desplegada en los mismos.

En definitiva, los resultados han sido satisfactorios y se puede concluir, que los tiempos son totalmente aceptados y que el uso de *ZeroC Ice* ha sido una buena alternativa para ofrecer las funcionalidades remotas.

Capítulo 7

Conclusiones y Propuestas

En esta sección se hace una valoración de las conclusiones obtenidas en la realización del Proyecto Fin de Carrera. Amaavem ha cumplido con los objetivos planteados inicialmente de manera previa a su desarrollo. Asimismo, en este capítulo también se plantean una serie de líneas de trabajo futuras para completar la funcionalidad y el soporte de Amaavem y finalmente, se incluye una conclusión personal del aporte que ha supuesto en mi vida el desarrollo de este proyecto.

7.1. Objetivos alcanzados

Los objetivos planteados en el capítulo 2 han sido alcanzados tal y como se describe a continuación:

- **Independencia del dominio**. Se ha creado una arquitectura lo más independiente posible del dominio a monitorizar. Para cumplir con este objetivo, la arquitectura se ha basado en el uso de estructuras de datos abstractas e interfaces neutras independientes del dominio a monitorizar. También se han empleado agentes inteligentes, como gestores del conocimiento de un dominio determinado. Estos agentes son adaptables a cualquier entorno monitorizado, añadiendo o modificando sus funcionalidades en función del medio donde se encuentren. El desarrollador tiene la posibilidad de crear agentes de todo tipo basándose en un sistema jerárquico, como el planteado en este proyecto, y además, tiene la posibilidad de agregar tantos comportamientos como el agente precise.

- **Escalabilidad**. La arquitectura final tiene un diseño que fomenta la escalabilidad. Para cumplir este objetivo, se diseño la arquitectura de manera modular, adaptable y con funcionalidades remotas ofreciendo un análisis de información eficiente. En caso de que el desarrollador quiera ampliar el sistema, tan sólo debe desplegar la aplicación en tantos *hosts* como desee. El sistema permite desplegar nuevos componentes al sistema de manera sencilla. El *middleware* de comunicaciones, empleado para resolver los problemas de comunicación entre los distintos componentes que forman Amaavem, fue de *ZeroC Ice*. La arquitectura se apoya en *Ice* para lograr el despliegue de los

diferentes servidores y servicios, logrando una aplicación distribuida y permitiendo que el desarrollador final pueda analizar la mayor información posible. Si se diera el caso de que algún *host* quede sobrecargado debido a una cantidad masiva de información, se da la posibilidad de agregar nuevos *hosts* aumentando el objetivo planteado, la escalabilidad.

- **Adaptabilidad**. La arquitectura fue diseñada siguiendo algunos de los patrones de diseño estudiados. Además ofrece la posibilidad de expandir dicha arquitectura de manera relativamente sencilla. Por ejemplo, la creación de agentes en el sistema es responsabilidad de una fábrica de agentes específica para el dominio concreto, que hereda de una factoría abstracta general (patrón fábrica abstracta). La agregación de nuevos agentes al sistema, es tan sencilla como crear un agente específico, que hereda de una clase general de agentes llamada *AgentI*, implementar una serie de métodos abstractos de la clase general y añadir los comportamientos que dicho agente va a necesitar para analizar la información.
- **Modularidad**. Para cumplir este objetivo, se ofrecieron una serie de servicios, cada uno de los cuales fue implementados por un módulo, siendo lo más independiente posible de los otros módulos. Cada módulo tiene unas responsabilidades bien acotadas consiguiendo un sistema que gana en cohesión y no sufre de acoplamiento. En Amaavem, existen una serie de módulos bien diferenciados tal y como se estudió en el capítulo 5. Agregar nuevos módulos al sistema sería una tarea sencilla y la comunicación entre los mismos es directa gracias al uso de *Ice*.
- **Robustez**. La arquitectura diseñada es tolerante a fallos capaz de seguir trabajando en caso de fallos en algunos de los módulos, gracias al servicio de replicación, ofrecido por *Ice*. Con *Ice*, resulta sencillo desplegar una serie de servicios con las mismas funciones y responsabilidades. En caso de fallo en algunos de los *hosts*, *Ice* se encarga de asumir estas perdidas y distribuir el trabajo entre los *hosts* en funcionamiento. Cabe citar también, que en la creación de Amaavem se intentó crear una arquitectura lo más independiente posible de los diferentes sistemas auxiliares de captación de información, ya que en los sistemas de vigilancia inteligente la mayoría de los errores surgen al realizar un mal *tracking* de los objetos que aparecen en las escenas. Por ello, Amaavem parte a partir de una información de unos eventos y no tiene por qué preocuparse de dónde fue captada dicha información.
- **Balanceo de carga**. En caso de que el tiempo de cómputo sea muy elevado debido a una afluente información o por la insuficiencia de nodos de cálculo, Amaavem ofrece una solución a este problema. El desarrollador tiene la posibilidad de agregar tantas factorías de agentes como precise, en nodos nuevos donde se despliegue Amaavem. Por tanto, en caso de que algún nodo tenga instanciados demasiados agentes, se ofrece dicha posibilidad e intentar distribuir el trabajo entre más nodos de trabajo. Por ejem-

plo, supóngase el caso de una configuración de Amaavem dónde solo existe un nodo, en el que están instanciados todos los módulos del sistema. Si dicho nodo tuviera instanciados 100 agentes, es muy posible que el tiempo de cómputo sea elevado debido a la cantidad de agentes trabajando en el mismo nodo. Entonces, para estos casos se da la posibilidad de desplegar más factorías de agentes y distribuir el trabajo entre todos los nodos, consiguiendo un aumento del rendimiento y la robustez.

- **Portabilidad**. El desarrollo de Amaavem se ha realizado siguiendo estándares y tecnologías libres, multiplataforma y consolidados, con el objetivo de que pueda ser portado al mayor número de plataformas posibles.
- **Facilidad de uso**. Una premisa importante en el desarrollo de Amaavem fue la creación de un sistema orientado a usuarios con un nivel medio en programación.

  El desarrollador final tan sólo debe tener claro el dominio donde quiere instalar Amaavem, los tipos de agentes que habrá en el sistema, los comportamientos que tendrán asociados dichos agentes y un sistema de clasificación para los objetos. Con estas premisas, podrá desplegar Amaavem de manera sencilla y rápida en cualquier dominio monitorizado. Debe fijarse en las partes del código que son exclusivas para cada dominio y modificarlas para que los resultados sean los deseados (véase manual de usuario en el Anexo A)

  Para cumplir con este objetivo, también se elaboró un manual de usuario (Anexo A) y una guía de desarrollo (Anexo B), para que el desarrollador final pueda entender cómo emplear Amaavem de manera correcta, y conseguir crear un sistema de vigilancia inteligente soportado por Amaavem.

En resumen, el objetivo principal de la creación de este Proyecto Fin de Carrera, que era ofrecer una arquitectura multi-agente adaptativa para la creación de sistemas de vigilancia inteligente en entornos monitorizados, ha sido cubierto, así como todos los subjetivos que surgieron a raíz de este objetivo principal.

7.2. Propuestas de trabajo futuro

Amaavem no pretende ser sólo un proyecto final de carrera, sino una aplicación robusta en el ámbito de la vigilancia inteligente, y para ello se debe trabajar duramente y alcanzar nuevos objetivos.

A continuación, se van a enumerar una serie de líneas de trabajo futuro para lograr una sustancial mejora del sistema Amaavem:

- **Replicación de comportamientos**. Actualmente, los comportamientos son parte de los agentes. En la instanciación de los agentes, éstos añaden a su esqueleto los diferentes comportamientos que precisan para analizar los eventos. En una linea futura, se pretende que dichos comportamientos sean objetos *Ice* independientes de los agentes, y sean estos últimos los que llamen a dichos comportamientos, pasándole como parámetros la información pertinente para que los comportamientos puedan analizar la información. De este modo, existirían sólo las instancias que el desarrollador estime oportuno. Con este cambio en la arquitectura, se permitiría tener replicados comportamientos en cada nodo del sistema consiguiendo una mejora en la eficiencia y el rendimiento.
- **Creación de un paquete de comportamientos básicos**. El objetivo principal de este proyecto no era crear un caso de estudio que ofreciera resultados potentes, sino ofrecer los medios necesarios a un desarrollador final para poder crear sistemas de vigilancia inteligente en un dominio monitorizado. Muchos desarrolladores que vayan a usar Amaavem será para controlar situaciones cotidianas que se dan en algunos de los entornos. La idea es ofrecer una serie de paquetes de comportamientos que midan situaciones típicos tales como:
  - Accidentes en dominios de carreteras.
  - Objetos abandonados en estaciones públicas.
  - Infracciones de tráfico.
  - Allanamientos de espacios privados.
  - Control de acceso a zonas determinados.
- **Creación de comportamientos más complejos**. Ofrecer comportamientos más complejos que impliquen la colaboración entre varios agentes. Un ejemplo podría ser la detección de peleas en la vía publica o la detección de incendios en bosques con alta probabilidad de los mismos.
- **Mejoras en el sistema de mensajes**. La arquitectura actual ofrece un sistema de comunicaciones básicas. En la arquitectura existen dos módulos: por un lado, uno para comunicar todos los componentes del sistema con los agentes y por otro lado, un sistema para gestionar las comunicaciones entre los propios agentes. Este último puede

ser mejorado tratando que dichos agentes puedan colaborar de manera conjunta en la resolución de problemas, tal como podría ser el caso de medir algún comportamiento complejo (una pelea entre seres humanos por ejemplo). Entonces, lo interesante sería definir un coordinador entre los agentes que sea el encargado de gestionar estas tareas complejas y usar el sistema de comunicaciones para la comunicación entre agentes. La idea es que el sistema multi-agente sea capaz de resolver problemas complejos cooperando entre los agentes instanciados. Tal y como se estudio en el apartado del estado del arte, sería interesante implementar un sistema de pizarra distribuida que gestione todo lo descrito anteriormente.

- **Uso de técnicas de manejo de información**. Los datos tratados a veces son imprecisos y vagos. Por ello, resulta interesante aplicar técnicas de lógica difusa para determinar los grados de normalidad de una situación o los grados de pertenencia de un objeto a un tipo determinado.
- **Optimización del espacio en disco**. Se pretende diseñar un sistema de optimización, en lo referente al almacenamiento de información. Actualmente, la información que fluye en el sistema se almacena por diferentes vías: en base de datos o ficheros a través de los comportamientos, y en imágenes a través del sistema de análisis de resultados. Debido a que la información de un vídeo ocupa demasiada información (2 minutos de vídeo equivalen aproximadamente a 10.000 filas en una base de datos o 3000 imágenes si se separa en *frames*), resulta interesante un sistema de optimización para resolver este problema.
- **Despliegue de la arquitectura en más entornos de monitorización**. Resulta interesante poder probar la arquitectura en más entorno de monitorización, así como la creación de comportamientos que puedan analizar situaciones cotidianas en diferentes entornos tal como una tienda, un bosque, el mar, etc.

7.3. Conclusión personal

Sinceramente, cuando empecé a realizar este proyecto, lo consideraba un reto importante debido a mi desconocimiento en la materia y la tecnología que se maneja en el ámbito de la Vigilancia Inteligente. Pero, trabajando a diario creo que he conseguido unos resultados aceptables y las esperanzas por este ámbito han despertado en mi interior. Pienso que este proyecto me ha servido para aumentar mis conocimientos en Informática.

He progresado en materias tan desconocidas para mí, como por ejemplo los sistemas multi-agentes, los sistemas distribuidos o lenguajes de programación, como *C++*. He afrontado situaciones caóticas, típicas en la realización de proyectos con envergadura considerable, y las he superado adecuadamente. He luchado por mejorar el software día a día hasta conseguir los objetivos planteados. He realizado múltiples reuniones con mi director de proyecto, de las cuales he aprendido mucho también. He estudiado técnicas inexploradas por mi experiencia. He preguntado en foros especializados y he sabido desenvolverme en situaciones en las que nunca había estado. He aprendido un lenguaje tan potente como es *C++*, que es usado por infinidad de empresas de desarrollo software del mundo. He peleado con sistemas distribuidos hasta conseguir que mi aplicación pueda ser considerada como un buen sistema distribuido. Por tanto, este proyecto ha superado mis expectativas previas y he progresado cuantiosamente como Ingeniero Informático.

En definitiva, creo que este Proyecto Fin de Carrera es una última prueba de cara al mundo laboral y pienso que me ha servido para sentirme preparado para afrontar la vida en un mundo laboral tan competitivo. He cerrado otra etapa importante en mi vida, de la cual estoy muy contento por lo resultados. Aprender y progresar como Ingeniero Informático han sido mis retos principales desde que comencé con la carrera, y en un futuro estoy convencido que estos objetivos seguirán vigentes.

ANEXOS

Anexo A

Manual de usuario de Amaavem

En este anexo, se muestra el manual de usuario básico para instalar, desplegar y ejecutar Amaavem. Debido a que Amaavem es un arquitectura distribuida y compleja, es necesario realizar una configuración adicional para poder desplegarla y emplearla en la construcción de un sistema de vigilancia inteligente. Por tanto, en este anexo se describen los pasos necesarios para usar correctamente Amaavem.

A.1. Consideraciones iniciales

Para poder comprender el sistema, se va a explicar cómo se despliegan los distintos servidores y servicios ofrecidos por Amaavem empleando el *middleware* de comunicaciones *ZeroC Ice*:

- **Agente de Base de Datos**. Es el *broker* encargado de manejar la base de datos para conseguir que otros componentes se abstraigan totalmente de interactuar con bases de datos y lenguajes de consultas. Se ha usado *MySQL* porque es un sistema libre de gestión de bases de datos relacional, multihilo, multiusuario y con funcionalidades distribuidas. Para lograr que el servicio del agente de base de datos esté disponible a los distintos nodos de Amaveem, se ha empleado el servicio de *ZeroC Ice* conocido como **IceBox**.

  IceBox es un único servidor de aplicaciones que permite gestionar el arranque y la parada de un determinado numero de componentes de aplicación. Dichos componentes se pueden desplegar como una biblioteca dinámica en lugar de un proceso. Este hecho reduce la sobrecarga global del sistema. El servicio de actualización de la base de datos se implementa como una librería dinámica del sistema y con *IceBox* se logra que esté disponible a los demás componentes.

- **Monitor de Vigilancia**. Es el encargado de gestionar los eventos que el cliente desea analizar. Para cubrir este servicio se ha implementando un servidor *ZeroC Ice*, que incluye un monitor de vigilancia. Como se explicó en la figura 5.2, pueden existir tantos servidores como el desarrollador indique en la configuración inicial del sistema. Cada servidor puede estar en un ordenador distinto y es *ZeroC Ice* el encargado de realizar

las comunicaciones entre los distintos servidores y servicios del sistema, empleando el servicio conocido como **IceGrid**. *IceGrid* es el servicio de activación y localización de aplicaciones ICE (véase sección 3.3.2 para comprender mejor las funcionalidades de los servicios implementados por *ZeroC Ice*).

- **Factoría de Agentes**. Es la encargada de instanciar los agentes en el sistema. Para implementar este servicio se ha empleado un servidor *ZeroC Ice* con la misma filosofía que la explicada en el punto anterior.
- **Canal de Eventos**. Es el encargado de recibir los eventos procesados por el monitor de vigilancia y enviarlo a los diferentes agentes del sistema. Para cubrir este servicio, se ha empleado el servicio ofrecido por *ZeroC Ice* conocido como **IceStorm**. *IceStorm* es un eficiente servicio de publicación-subscripción para aplicaciones ICE. Actúa de mediador entre el publicador y el subscriptor. Los agentes del sistema se subscriben al *topic* (canal de eventos), y éste cuando recibe nuevos eventos se los envía a todos los agentes subscritos al canal de manera transparente.

A.2. Ajustar los archivos de configuración

En los primeros pasos previos a la ejecución del software, se han de configurar los distintos archivos de configuración empleados por Amaavem.

En primer lugar, se deben ajustar las constantes y variables de configuración, incluidas en el archivo *Slice config.ice*, que se encuentra en la carpeta (*/src*). En dicho archivo se encuentran parámetros de configuración importantes, tal como la *ip* del *host* donde se encuentra alojada la base de datos, el nombre de usuario y contraseña de acceso a la base de datos, las rutas que hacen referencia a la ruta relativa donde se almacenan las bibliotecas dinámicas o incluso, los tipos de objetos que se emplean en el entorno monitorizado concreto. De ahí la importancia de que el desarrollador final modifique este archivo para ajustarlo a sus exigencias.

A continuación, se muestra el fragmento de código que ha de ser modificado por el desarrollador:

```
1     // route of the dynamic libraries
2     const string ROUTE_DLL = "./libs/libsDll/";
3
4     // route of the config ICE
5     const string ROUTE_CONFIG_CLIENT = "./configIce/config.client";
6
7     // MYSQL. Information of access
8     const string IP_BD = "localhost";
9     const string USER_BD = "root";
10    const string PASS_BD = "admin";
11    const string BD = "pfc";
12
13    // Maximum of changes of type for an agent
```

```
14      const int NUM_ERRORS_MAX = 50;

16      // Minimum value of belief for one type of agent
17      const double MIN_VALUE_BELIEF = 0.5 ;

19      // Maximum time idle for an agent (seg)
20      const double TIME_FINISH = 90 ;

22      // SPECIFIED OF DOMAIN
23      // enum for structify objets
24      enum typeObject { PERSON, VEHICLE };
```

En segundo lugar, el desarrollador debe modificar el archivo *amaavem.xml*, que se encuentra en la carpeta raíz del proyecto. En dicho archivo se indican cuantos servidores y servicios se van a emplear y su localización, así como otros aspectos de configuración relevantes, empleados por *ZeroC Ice*. Esta configuración se puede editar de manera sencilla con un programa que proporciona el propio *middleware* de *ZeroC Ice*, conocido con el nombre de *Icegrid-Gui*. Con dicha aplicación se puede editar información relevante al número de nodos (*hosts* con *ips* distintas) que se despliegan, cantidad de replicas de los servidores, información de localización de los *proxys*, las bibliotecas dinámicas que van a ser lanzadas como servicios y demás características. En general, el desarrolador deberá modificar la dirección *ips* de los *endpoints* (direcciones donde se encuentren instanciados los componentes). En caso de que el programa se esté ejecutando en modo local, las direcciones *ips* de los diferentes componentes deben quedar como sigue a continuación:

```
1       tcp -h localhost
```

En caso de ejecutar el programa en modo remoto, se deben agregar las diferentes *ips* de los componentes, sustituyendo el *localhost*, por la dirección *ip* específica donde se encuentre el componente.

Una vez modificadas las direcciones *ips* de los componentes de Amaavem, es necesario indicar en este mismo archivo cómo van a estar distribuidos dichos componentes. En caso de que se trate de modo local, sólo debe existir un nodo, que debe estar descrito tal y como sigue a continuación:

```
1   <!-- HOST 1 -->
2       <node name="node1">
3         <!-- Service of AgentBD -->
4         <icebox id="IceBox" activation="on-demand" exe="icebox">
5           <description>A sample IceBox server</description>
6           <properties>
7                <property name="Ice.Admin.Endpoints" value="tcp -h
                     localhost"/>
8                <property name="IceBox.InstanceName" value="${server}"/
                     >
9                <property name="IceBox.Trace.ServiceObserver" value="1"
                     />
```

```
          </properties>
          <service-instance template="ServiceBDTemplate" name="BD1"/>
        </icebox>
        <!-- Servers of FactoryServer -->
        <server-instance template="FactoryServer" index="1"/>
        <!-- Service of EventChannel (IceStorm) -->
        <server-instance template="IceStorm" instance-name="
           EventChannel"/>
        <!-- Service of BlackBoard-Agents (IceStorm) -->
        <server-instance template="IceStorm" instance-name="BlackBoard
           -Agents"/>
        <!-- Servers of VigilanceManagerServer -->
        <server-instance template="VigilanceServer" index="1"/>

      </node>
```

Cuando se trate el programa en modo remoto, se debe indicar el esquema de despliegue de los componentes en los N nodos que conformen la aplicación. A continuación, se muestra un ejemplo de configuración con 3 nodos:

```
<!-- HOST 1 -->
   % IGUAL QUE EL ANTERIOR

<!-- HOST 2 -->
    <node name="node2">
      <!-- Servers of FactoryServer -->
      <server-instance template="FactoryServer" index="2"/>
      <!-- Servers of VigilanceManagerServer -->
      <server-instance template="VigilanceServer" index="2"/>
    </node>

<!-- HOST 3 -->
    <node name="node3">
      <!-- Servers of FactoryServer -->
      <server-instance template="FactoryServer" index="3"/>
    </node>
```

En tercer lugar, se precisa modificar los archivos de configuración *Ice*, que son necesarios para que *IceGrid* realice la localización de los *proxys* de los componentes de Amaavem. Estos archivos se encuentran en la carpeta (*/configIce*) y pueden ser de diversos tipos:

- **config.grid**: En este archivo se define la dirección *ip* del nodo principal (también conocido como *registry*), así como otros parámetros de configuración necesarios. Cabe citar que el nodo 1 siempre hace referencia al nodo principal (*registry*). El desarrollador debe modificar la dirección *ip* del *proxy* localizador de *IceGrid*, añadiendo la *ip* del nodo 1, tal y como se muestra a continuación:

```
# The IceGrid locator proxy.
# - h localhost = IP OF THE NODE 1 !!

Ice.Default.Locator=PFC/Locator:default -h localhost -p 4061
```

- **config.nodeN**: Este archivo hace referencia al archivo de configuración del nodo en cuestión, siendo N el número de nodo. Es decir, si por ejemplo Amaavem se despliega para 3 nodos, debe existir un archivo *config.grid* (nodo 1 = *registry*), un archivo *config.node2* (nodo 2) y otro archivo *config.node3* (nodo 3). En estos archivos, se deben definir también la dirección *ip* del nodo principal, de la misma manera que se definió en el archivo anterior (*config.grid*).
- **config.client**: Este archivo será empleado por la aplicación cliente. En él, se debe definir también la dirección *ip* del nodo principal, así como los *endpoints* de los *proxys*, empleados en Amaavem (pues ya está configurado y no es necesario modificar nada). El desarrollador sólo debera modificar esta dirección *ip*, y añadir la *ip* donde este desplegado el *registry*.

  En resumen, se debe añadir la *ip* del nodo principal (*registry*) en todos los archivos de configuración previamente explicados.

En cuarto y último lugar, se debe agregar la ruta, donde se encuentran las librerías dinámicas generadas por Amaavem (*libs/libsDll*), a la variable de entorno *LD_LIBRARY_PATH*. En caso de usar un sistema operativo *GNU/Linux*, se deben seguir dos sencillos pasos:

1. Abrir como administrador el fichero *bashrc*, que se encuentra en la carpeta raíz del sistema.
2. Agregar al final del fichero la siguiente línea:

```
1     # $RUTA_AMAAVEM = Ruta donde este instalado Amaavem
2     # ejemplo /home/javi/amaavem
3 export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/$RUTA_AMAAVEM/libs/
    libsDll/
```

En caso de usar *Microsoft Windows*, seguir el manual propuesto por *Microsoft* en la página `http://support.microsoft.com/kb/310519/es`

A.3. Configurar la base de datos

Para la gestión de la base de datos, se recomienda el uso de *MySQL Administrator*, debido a su sencillez de uso. Este manual se ha realizado usando el programa recomendado.

Una vez instalado el *MySQL Administrator*, se debe crear la base de datos del proyecto. Para ello, se deben seguir las siguientes instrucciones:

1. Abrir *MySQL Administrator*. Introducir los datos de acceso para conectar con el sistema de administración.
2. Ir al anexo D y copiar el código SQL en un archivo que se ha llamado con el nombre *base_datos_amaavem.sql*.

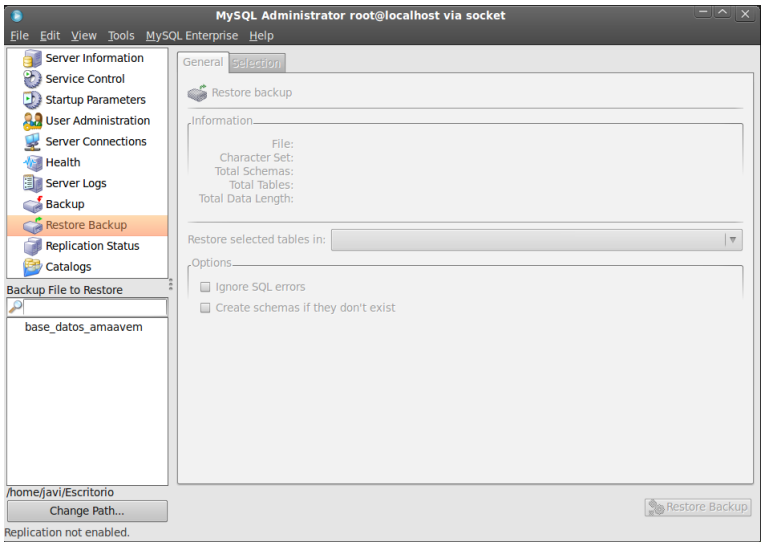


Figura A.1: Manual usuario: configuración BD 1

3. En la pestaña *Restore Backup* de *MySQL Administrator*, buscar y seleccionar el archivo anterior (véase figura A.1). Al seleccionar el fichero el programa pedirá un formato para la base de datos. Elegir formato *UTF-8*.

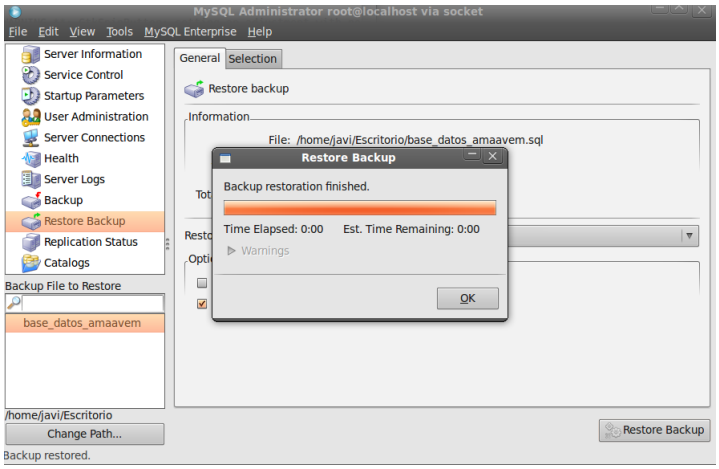


Figura A.2: Manual usuario: configuración BD 2

4. Pinchar en el botón que está en la parte inferior derecha, llamado *Restore Backup*, y esperar al volcado de los resultados (véase figura A.2).
5. Reiniciar el programa. Dirigirse a la pestaña *Catalogs*, observar si aparece una base de datos con el nombre *pfc*. En caso afirmativo, se ha instalado correctamente la base de datos de Amaavem (véase figura A.3).

Tras la instalación de la base de datos correctamente, el siguiente paso es habilitar el acceso remoto a la base de datos. Para ello, se han de seguir los siguientes pasos (omitir estos pasos si el programa se va a usar únicamente en modo local):

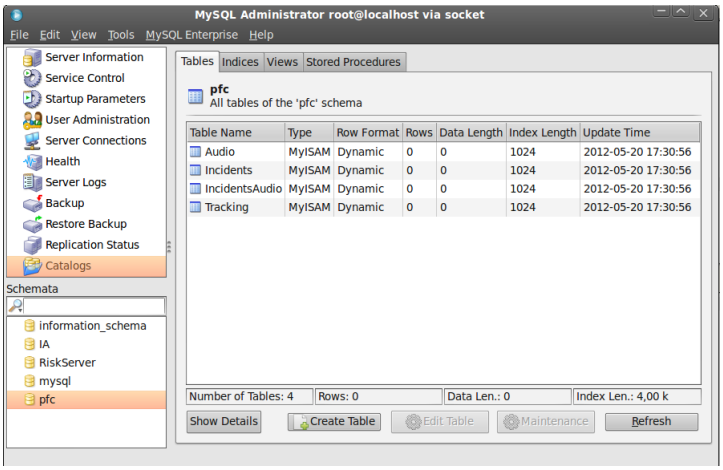


Figura A.3: Manual usuario: configuración BD 3

1. Modificar el archivo de configuración de *MySQL* llamado *my.cnf*. En *GNU/Linux*, se encuentra en la ruta */etc/mysql/* y en *Microsoft Windows*, en la ruta raíz del sistema. Abrir este fichero, buscar la siguiente línea:

```
1 bind-address = localhost
```

   Modificar esta línea, sustituyendo *localhost*, por la dirección *ip* de la computadora, que alberga la base de datos de Amaavem.

2. Asignar privilegios de acceso remoto a la base de datos. Para ello, se procederá de la siguiente manera:

   a) Abrir *MySQL Administrator*. Ir a la pestaña *User Administration*. Hacer *click* derecho sobre el usuario de la base de datos y pinchar en la opción *Add Host* (véase figura A.4).

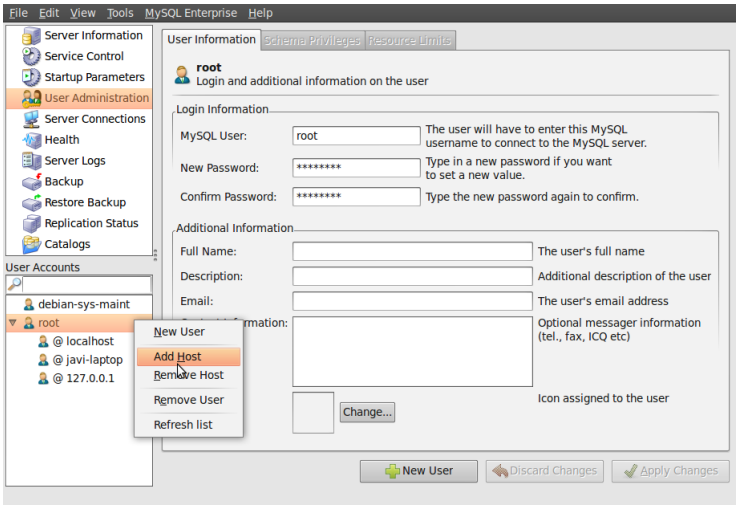


Figura A.4: Manual usuario: acceso remoto BD 1

b) A continuación, saltará una ventana: seleccionar la opción *Any Host* y pinchar en el botón de *OK* (véase figura A.5).

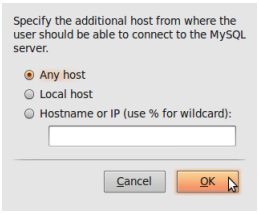


Figura A.5: Manual usuario: acceso remoto BD 2

c) Posteriormente, iremos a la lista de *hosts* para nuestro usuario, pinchar en el que pone @ %. Escoger la base de datos *pfc*. Seleccionar todos los privilegios disponibles y pinchar en en la flecha, tal y como se muestra en la figura A.6.

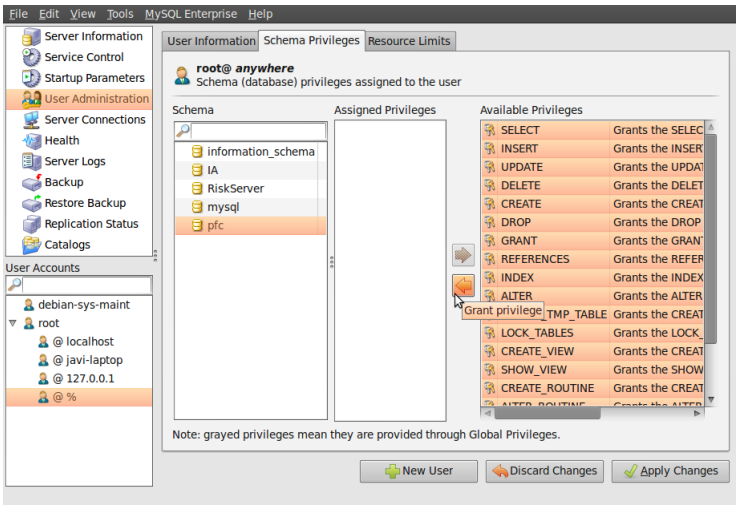


Figura A.6: Manual usuario: acceso remoto BD 3

d) Finalmente, si todo ha salido correctamente, el resultado será igual al mostrado en la figura A.7.

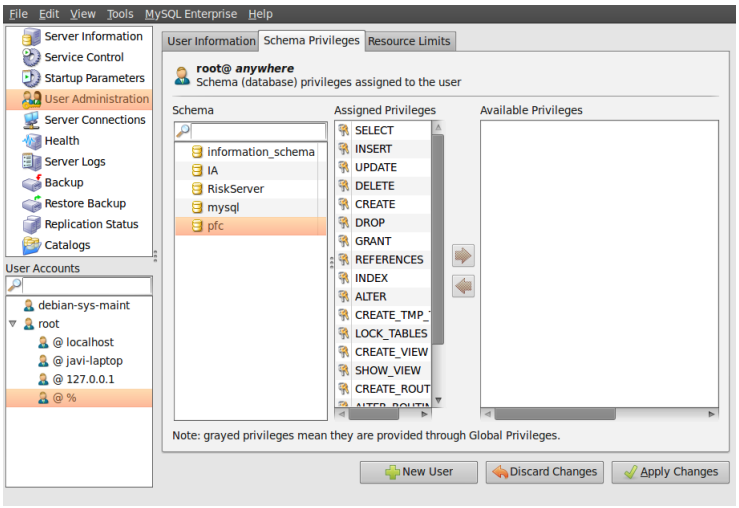


Figura A.7: Manual usuario: acceso remoto BD 4

A.4. Configuración del archivo Makefile

El empleo de un archivo *Makefile* es un manera sencilla de compilar proyectos de envergadura considerable. El *Makefile* de Amaavem ha de ser configurado por el desarrollador final antes de empezar a compilar el proyecto.

Amaavem puede ser usado en diferentes entornos monitorizados, en los que se definen una serie de agentes específicos. Por ello, es necesario indicar en este *Makefile* los tipos de agentes que existen en el sistema. Tal y como se explicó en la sección 5.5, por cada agente especifico, se crea una librería dinámica. Por tanto, el desarrollador que emplee Amaavem debe modificar una pequeña regla del *Makefile*, en función del dominio y los agentes que haya creado.

En concreto, se ha de modificar la regla *dll* de dicho archivo. Por cada agente específico del sistema, se debe agregar la siguiente línea (siendo *TYPE* el tipo concreto y *OBJETOS* la lista de objetos que deben ser conocidos por la librería):

```
1 dlls:
2   g++ -shared -o $(DIRHEA)$(DIRDLL)libAgentTYPE.so $(DIROBJ)OBJETOS.o
      $(LDLIBS)
```

A modo de ejemplo, supóngase el caso en el que Amaavem ha sido desplegado en un entorno urbano, y se han definido dos tipos de agentes: personas y vehículos.

Dicha regla debe estar formada tal y como se muestra a continuación:

```
1 dlls:
2   g++ -shared -o $(DIRHEA)$(DIRDLL)libAgentPerson.so $(DIROBJ)OBJETOS.
      o $(LDLIBS)
3   g++ -shared -o $(DIRHEA)$(DIRDLL)libAgentVehicle.so $(DIROBJ)OBJETOS
      .o $(LDLIBS)
```

A.5. Compilación de Amaavem

Una vez terminada la fase de configuración, se deben crear o modificar las clases específicas para el dominio concreto donde se va a desplegar Amaavem (véase Anexo B, para seguir con la guía de desarrollo).

Tras la realización de los pasos previos, el siguiente paso es compilar el proyecto Amaavem. En primer lugar, se deben instalar las librerías necesarias para poder compilar Amaavem. En la sección 4.2.3, se detallan todas las librerías que se han de instalar.

La compilación del proyecto Amaavem se realiza ejecutando dos sencillos pasos:

1. Abrir un terminal para usar la línea de comandos del sistema operativo. Situarse en la carpeta raíz del proyecto.
2. Ejecutar la siguiente orden en el terminal:

```
make amaavem
```

Esta orden genera todos los archivos necesarios para ejecutar Amaavem. Por un lado, genera los ejecutables, tanto de los servidores como de la aplicación principal, y por otro lado, genera las librerías dinámicas. Cuando se termine la compilación, ya está disponible el sistema Amaavem para su ejecución.

Cuando se desee hacer una limpieza de los ejecutables, las librerías dinámicas y los *logs* del sistema bastará con ejecutar la siguiente orden:

```
make clean
```

A.6. Ejecución del programa

Si ya se han realizado todos los pasos previos a este punto, significa que el programa está listo para ser empleado. En el archivo *amaavem.xml*, debe estar definida la estructura de componentes, que se han de desplegar, en los distintos nodos a desplegar.

Antes de ejecutar la aplicación, se deben lanzar **los nodos de *IceGrid*** .Un nodo *IceGrid* es un proceso que activa, monitoriza, y desactiva los procesos del servidor registrado. Cada nodo debe estar ejecutándose en cada uno de los *hosts*, y todos los nodos están ligados a un nodo principal (*registry*).

Entonces, primeramente el desarrollador final debe lanzar el **nodo principal (*registry*)** del sistema. Para ello, se debe abrir un terminal de consola en el *host* donde se va a desplegar el nodo *registry*. Situarse en la carpeta raíz del proyecto y escribir el siguiente comando:

```
icegridnode --Ice.Config=configIce/config.grid
```

Una vez desplegado el nodo registry, se lanzan los **demás nodos del sistema**, en caso de que sea preciso. Si la configuración indica que solo existe un nodo, no hace falta realizar más pasos. En caso de que el sistema cuente con más nodos, éstos deben ser lanzados uno a uno. Por tanto, se debe abrir un terminal de consola en cada uno de los *hosts*, excepto en el *host* que alberga el nodo *registry* (debido a que ya fue desplegado en un paso anterior), y escribir el siguiente comando (siendo *N* el número concreto del *host*):

```
icegridnode --Ice.Config=configIce/config.nodeN
```

El siguiente paso, tras el lanzamiento de todos los nodos que forman el sistema, es **desplegar la aplicación generada usando Amaavem**, descrita en el fichero *amaavem.xml*. Para este paso, se va a emplear el programa *IceGrid-Gui*, ofrecido por el *middleware ZeroC Ice*. Los pasos que se han de seguir para desplegar la aplicación se describen a continuación:

1. En el host donde fue desplegado el nodo *registry*, abrir un terminal de consola y escribir el siguiente comando:

```
icegrid-gui
```

2. Loguearse en el nodo principal. Pulsar en el botón de la esquina superior izquierda e introducir los siguientes datos (véase figura A.8 y A.9).

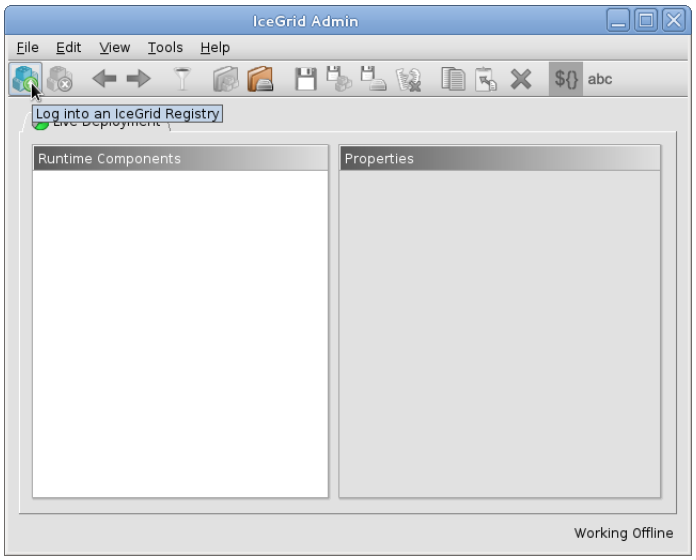


Figura A.8: Manual usuario: Despliegue Amaavem 1

3. Si el paso anterior ha funcionado, se debe ver algo parecido a lo que se muestra en la figura A.10 (este manual está hecho con una configuración de un sólo nodo).

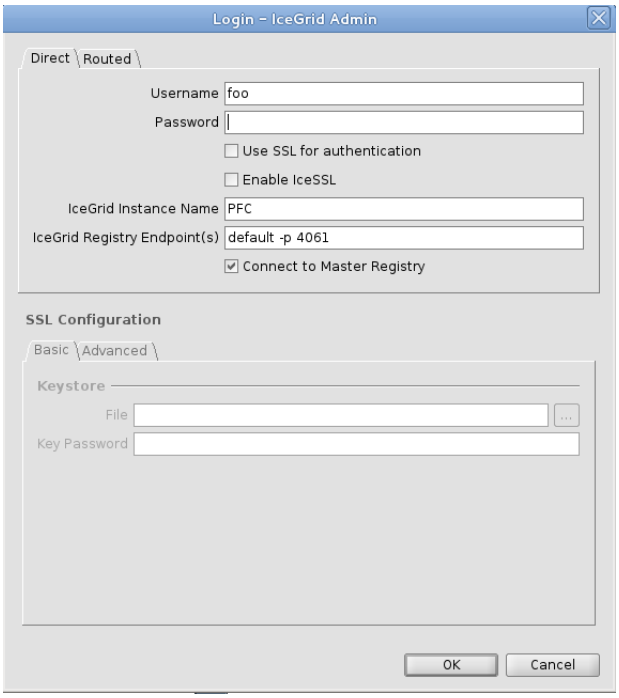


Figura A.9: Manual usuario: Despliegue Amaavem 2

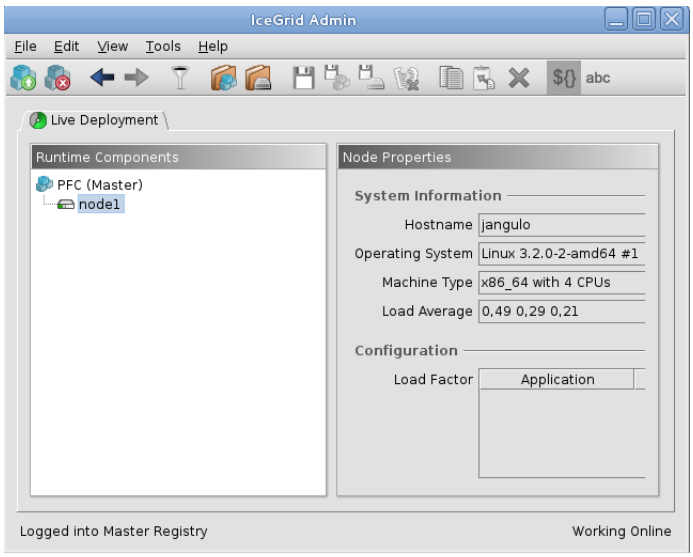


Figura A.10: Manual usuario: Despliegue Amaavem 3

4. Abrir la aplicación *amaavem.xml*. Para ello, pulsar en el botón que se indica en la figura A.11. Elegir el archivo *amaavem.xml*.
5. Guardar la aplicación en el *registry*. Para ello, pulsar en el botón que se indica en la figura A.12.
6. ¡Ya está la aplicación desplegada en el sistema! (véase figura A.13).

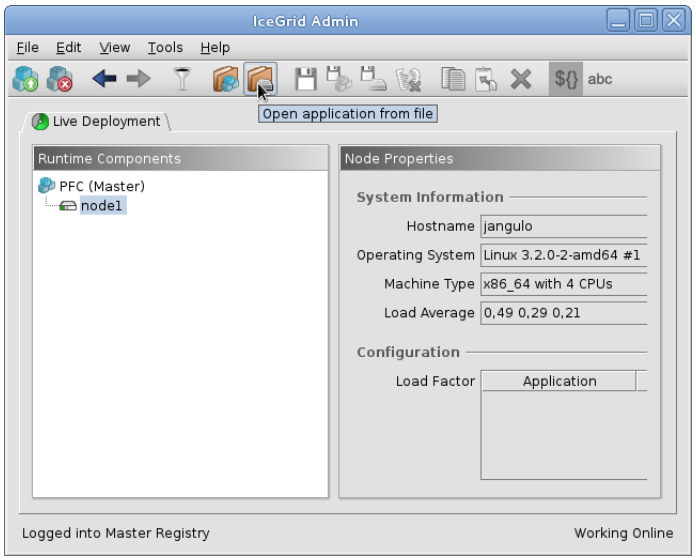


Figura A.11: Manual usuario: Despliegue Amaavem 4

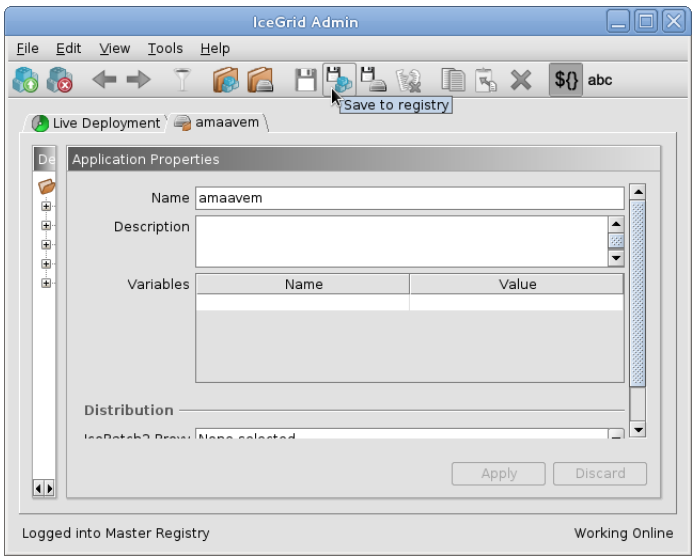


Figura A.12: Manual usuario: Despliegue Amaavem 5

El punto final es **ejecutar la aplicación**. Para ello, abrir un terminal de consola y situarse en la carpeta raíz del proyecto. El ejecutable se encuentra en la carpeta (*/exec/*) y tiene el nombre de *Main*. Este programa tiene varias opciones de entrada dependiendo de los eventos, que se desean analizar. Estas opciones se describen a continuación:

- **Para eventos de vídeo.**

  Si se desean analizar eventos de vídeo, escribir el siguiente comando en el terminal:

```
1 ./exec/Main -f OPTIONS_FRAME -c OPTION_CAMERA
```

  OPTION_CAMERA, es el número entero de la cámara a analizar.

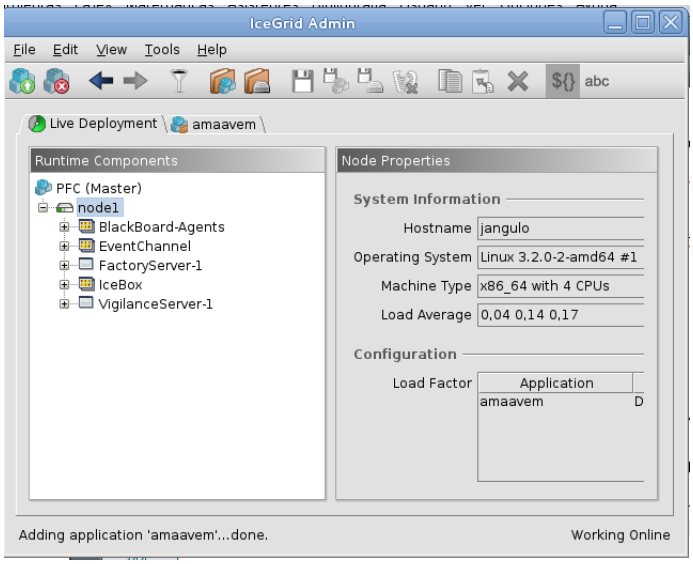


Figura A.13: Manual usuario: Despliegue Amaavem 6

OPTIONS_FRAME, es un parámetro que puede tomar dos valores:

1. *all*: Si se desean analizar todos los *frames* almacenados en la base de datos, de una cámara fijada por el usuario. Ejemplo:

```
./exec/Main -f all -c 2
```

2. *X Y*: Siendo X e Y dos números enteros con $X < Y$. Dichos números hacen referencia al *frame* inicial y al *frame* final para el análisis. Ejemplo:

```
./exec/Main -f 20 1324 -c 1
```

- **Para eventos de audio.**

  Si se desean analizar eventos de audio, escribir el siguiente comando en el terminal:

```
./exec/Main -audio -m OPTION_MICROPHONE
```

  OPTION_MICROPHONE, es un número entero que hace referencia al micrófono, que se desea analizar. Ejemplo:

```
./exec/Main -audio -m 1
```

Al ejecutar el programa, se debe esperar un tiempo hasta que el mismo termine de realizar los análisis. Una vez finalizado el análisis, se pueden observar de tres formas distintas los resultados:

- Revisando la base de datos.
- Revisando los *logs* arrojados en la carpeta (/Log).
- Utilizando el módulo de análisis de resultados tal y como se muestra en la sección que sigue a continuación.

A.7. Manual de uso del módulo análisis de vídeo

Tras la ejecución del programa, el cliente tiene la posibilidad de analizar eventos de vídeo con este módulo, que se encuentra implementado en la carpeta (*/Debug*) con el nombre *video.py*. Para usar este modulo, se debe abrir un terminal, situarse en la carpeta mencionada con anterioridad y proceder a ejecutar el siguiente comando:

```
./video.py [DIR_SOURCE_FRAMES] [DIR_MODIFIED_FRAMES] [INIT_FRAME] [
    END_FRAME] [NUM_CAMERA] [RESOLUTION]
```

Las opciones de los argumentos de entrada se describen a continuación:

- *[DIR_SOURCE_FRAMES]*: Directorio donde se encuentran los *frames* originales de la escena que se precise analizar.
- *[DIR_MODIFIED_FRAMES]*: Directorio donde se van a guardar los *frames* modificados.
- *[INIT_FRAME]*: Número de *frame* inicial desde donde se inicia el análisis.
- *[END_FRAME]*: Número de *frame* final de detección del análisis.
- *[NUM_CAMERA]*: Número de cámara que se desea analizar.
- *[RESOLUTION]*: Resolución de la imagen (x,y) con formato *NUMxNUM*.

A.8. Manual de uso del parseador de ficheros XML

El *parseador* de ficheros *XML*, empleado para volcar datos desde un fichero *XML* a la base de datos, se encuentra implementado en la carpeta (*/src/utils*) con el nombre *parser.py*. Este módulo es muy sencillo de usar. Tan sólo hay que abrir un terminal de consola, situarse en la carpeta mencionada y ejecutar el siguiente comando:

```
./parser.py [DIR_SOURCE_FILES] [NUM_OBJECTS] [NUM_CAMERA]
```

Las opciones de los argumentos de entrada se describen a continuación:

- *[DIR_SOURCE_FILES]*: Directorio donde se encuentran los ficheros *XML*, todos ellos con el formato (*ObjectXML %d.xml*, siendo *%d = número de objeto*).
- *[NUM_OBJECTS]*: Número de objetos que se quieren volcar a la base de datos.
- *[NUM_CAMERA]*: Número de la cámara.

Anexo B

Guía de desarrollo con Amaavem

Para utilizar la arquitectura Amaavem e implementar un sistema de vigilancia inteligente en un entorno monitorizado, el desarrollador final debe implementar algunas partes específicas, en función del dominio donde se despliegue Amaavem.

En **primer lugar**, se debe implementar el monitor de vigilancia. Un ejemplo de un monitor de vigilancia se muestra a continuación:

```
2  ---------
3  .h
4  ---------
5  #ifndef VIGILANCEMANAGER_H
6  #define VIGILANCEMANAGER_H
7
8  #include <VigilanceManager.h>
9  #include <config.h>
10 #include <ops.h>
11 #include <Logger.h>
12 #include <AgentFactoryI.h>
13
14 #include <IceUtil/IceUtil.h>
15 #include <Ice/Ice.h>
16 #include <IceStorm/IceStorm.h>
17 #include <IceUtil/Mutex.h>
18
19 using namespace std;
20
21 namespace pfc {
22
23     // Class specific to each particular environment
24     // Manager responsible for directing the system
25     // Keep a list of ids of agents instantiated
26     // Meet the factory of agents and the event channel
27     // Each event comes the new processes and sends to all agents
           subscribing to topic
28     // For each object id is instantiating an agent with the same id
29
30     class VigilanceManagerI : public VigilanceManager
31     {
32
33         private:
34             vector<int> idAgents; // list of the ids of agents
35             void addIdObject(const int& idObject);
```

```
36              bool searchObject(const int& idObject);

38          public:
39              VigilanceManagerI ();
40              ~VigilanceManagerI ();
41              void run(const vEvents& events,const Ice::Current& c);
42              void printAgents();
43              virtual void notify(const vEvents& events, const Ice::
                    Current& c);
44              virtual void notifyOne(const EventPtr& event, const Ice::
                    Current& c);

46      };
47  };

49  #endif

51  ---------
52  .cpp
53  ---------

55  #include <VigilanceManagerI.h>


58  namespace pfc {


61      VigilanceManagerI::VigilanceManagerI (){}

63      VigilanceManagerI::~VigilanceManagerI (){}


66      void VigilanceManagerI::run(const vEvents& events, const Ice::
            Current& c){

68          // Obtain the communicator Ice
69          Ice::CommunicatorPtr comm = c.adapter->getCommunicator();

71          vEvents::const_iterator it;
72          EventPrx auxEvent ;
73          vector<objTracking> auxObjs ;
74          vector<objTracking> ::const_iterator it2 ;
75          AgentFactoryPrx agentF;
76          IceStorm::TopicManagerPrx topicManager;
77          IceStorm::TopicPrx topic;
78          string topicName = "eventchannel";
79          string textError ;

81          // Obtain a TopicManager of IceStorm
82          topicManager = IceStorm::TopicManagerPrx::checkedCast(comm->
                propertyToProxy("TopicManager.Proxy"));

84          if(!topicManager){
85              textError = "[VigilanceManager][run] Invalid proxy
                    TopicManager";
86              Logger::getInstance()->write_error(textError.c_str());
87          }
```

```
89          // We cross the events to see new items
90          for ( it = events.begin(); it != events.end() ; ++it){
91              EventPtr newEvent = EventPtr::dynamicCast(*it);
92              EventVideoPtr eventV = EventVideoPtr::dynamicCast(newEvent
                    );

94              auxObjs = eventV->objects;
95              for ( it2 = auxObjs.begin(); it2 != auxObjs.end() ; ++it2)
                    {
96                  //In case of new object
97                  //Store its maximum height and maximum width to
                        correctly classify
98                  if (! searchObject((*it2).id)){

100                     /* Obtain an AgentFactory */
101                     try {
102                         agentF = AgentFactoryPrx::checkedCast(comm->
                                stringToProxy("agentfactory"));
103                     }
104                     catch(const Ice::NotRegisteredException&){
105                         textError = "[VigilanceManager][run] Invalid
                                proxy agentfactory";
106                         Logger::getInstance()->write_error(textError.
                                c_str());
107                     }
108                     // Add the new object and create the new Agent for
                            this type object
109                     addIdObject((*it2).id);
110                     // The factory create the agent
111                     agentF->createAgent( (*it2).id , PERSON);
112                     agentF->createAgent( (*it2).id , VEHICLE);
113                 }
114             }


117             // Retrieve the topic to sent the events

119             try {
120                 topic = topicManager->retrieve(topicName);
121             }catch(const IceStorm::NoSuchTopic&){
122                 try{
123                     topic = topicManager->create(topicName);
124                 }catch(const IceStorm::TopicExists&){
125                     textError = "[VigilanceManager][run] Invalid topic
                            ";
126                     Logger::getInstance()->write_error(textError.c_str
                            ());
127                 }
128             }
129             // Get the topic's publisher object, and create a Agent
                    proxy

131             Ice::ObjectPrx publisher = topic->getPublisher();
132             AgentPrx agent = AgentPrx::uncheckedCast(publisher);

134             // Notify all the agents with the new event
135             agent->notify(newEvent);
136         }
```

```
138     }


141     void VigilanceManagerI::addIdObject(const int& idObject){
142         //Add the new id to the list of agents
143         idAgents.push_back(idObject);

145     }

147     // Search an id in the list of agents
148     bool VigilanceManagerI::searchObject(const int& idObject){
149         vector<int>::const_iterator it;
150         for ( it = idAgents.begin(); it != idAgents.end() ; ++it){
151             if ( (*it) == idObject) return true;
152         }
153         return false ;

155     }

157     void VigilanceManagerI::printAgents(){
158         vector<int>::const_iterator it;
159         for ( it = idAgents.begin(); it != idAgents.end() ; ++it){
160             cout << "[VigilanceManager] Agent instanciated with id : "
                    << (*it) <<  endl ;
161         }

163     }

165     // Operation ICE that receive all the events
166     void VigilanceManagerI::notify(const vEvents& events,const Ice::
            Current& c){
167         run(events,c);
168     }

170     // Operation ICE that receive one event
171     void VigilanceManagerI::notifyOne(const EventPtr& event,const Ice
            ::Current& c){
172         vEvents events;
173         events.push_back(event);
174         run(events,c);
175     }

177 };
```

En **segundo lugar**, se deben crear los agentes específicos del dominio. El código de un agente específico se muestra a continuación:

```
AGENTE EJEMPLO: TIPO PERSONA

---------
.h
---------
#ifndef AGENTPERSON_H
#define AGENTPERSON_H

#include <config.h>
#include <AgentI.h>
#include <BehaviorCheckDistances.h>
#include <UrbanDomain/urbanOperations.h>

using namespace std;

namespace pfc {

    class AgentPerson : public AgentI
    {
        public:
            AgentPerson(int id, const Ice::ObjectAdapterPtr& adapter )
                : AgentI(id,adapter){
                this->addBehaviorsPerson();
            }
            void analyze(const Event& event) ;
            void addBehaviorsPerson();

        private:
            bool checkVehicle(const Event& event);
    };

};
#endif

---------
.cpp
---------

#include <AgentPerson.h>

    extern "C" pfc::AgentI* getAgentPerson(int id, const Ice::
       ObjectAdapterPtr& adapter){
        return new pfc::AgentPerson(id,adapter);
    }
namespace pfc {

    void AgentPerson::analyze(const Event& event){

        vector<objTracking> ::const_iterator it ;
        vector<Behavior*> bes ;
        vector<keyValue> values ;
        bool analyzed = false ;
        const EventVideo* eventV = static_cast<const EventVideo*>(&
```

```
            event);

54          for ( it= eventV->objects.begin(); it != eventV->objects.end()
                ; ++it){
55              // If matchID with our ID - analyze the frame
56              if ( matchId((*it).id)) {
57                  analyzed = true ;

59                  // Reset timer
60                  g_timer_reset(this->getTimer());

62                  // Clasiffy the object !
63                  values = this->classify((*it).details.width, (*it).
                        details.height );
64                  this->setValueType(this->getMyValue(values, PERSON));

66                  // Search cars in the frame
67                  if (checkVehicle (event) ) {
68                      bes = this->getBehaviors();
69                      vector<Behavior*>::const_iterator it2;
70                      for ( it2 = bes.begin(); it2 != bes.end()  ; ++it2
                            ){
71                          (*it2)->analyze(*(this->getAdapter()->
                                getCommunicator()),this->getId(),event);
72                      }
73                  }
74              }
75              if (analyzed) break;
76          } // end for
77      }

79      // Method that adds behavior of the person
80      void AgentPerson::addBehaviorsPerson(){
81          // BehaviorCheckDistances
82          // to instantiate the behavior : Resulution (x, y)
83          Behavior* b1 = new BehaviorCheckDistances(768,576) ;
84          this->addBehavior(b1);
85      }

87      // Method that looks any car on the scene
88      bool AgentPerson::checkVehicle(const Event& event){
89          const EventVideo* eventV =  static_cast<const EventVideo*>(&
                event);
90          vector<objTracking> auxObjs = eventV->objects ;
91          vector<objTracking> ::const_iterator it;

93          for ( it = auxObjs.begin(); it != auxObjs.end() ; ++it){
94              if ( (*it).id == this->getId() )
95                  continue ;
96              else if (urbanOperations::checkVehicle((*it).details.width
                    , (*it).details.height))
97                  return true ;
98          }
99          return false ;
100     }
101 }
```

En **tercer lugar**, se deben crear la factoría específica del entorno a monitorizar. A continuación se muestra un ejemplo de una factoría para un entorno urbano:

```
---------
.h
---------

#ifndef AGENTFACTORYI_H
#define AGENTFACTORYI_H

#include <config.h>
#include <AgentI.h>
#include <AgentFactory.h>
#include <dlfcn.h>
#include <Logger.h>
#include <IceUtil/IceUtil.h>
#include <Ice/Ice.h>
#include <IceStorm/IceStorm.h>
#include <IceUtil/Mutex.h>

using namespace std;

namespace pfc {

        typedef AgentI* (*getAgent_t)(int, const Ice::ObjectAdapterPtr
           );
        class AgentFactoryI : public AgentFactory
        {
                public:
                        AgentFactoryI();
                        AgentFactoryI(const Ice::ObjectAdapterPtr&
                           adapter);
                        virtual ~AgentFactoryI();
                        virtual AgentPrx createAgent(int id,
                           typeObject type,const Ice::Current& c);
                        void cleanAgents();
                        AgentI* findAgent(const int& id);

                private:
                        vector<AgentI *> agents;
                        Ice::ObjectAdapterPtr _adapter;
                        IceUtil::Mutex _fileMutex;
                        bool start;
        };
};
#endif

---------
.cpp
---------

#include <AgentFactoryI.h>


namespace pfc {
```

```
52          AgentFactoryI::AgentFactoryI(const Ice::ObjectAdapterPtr&
               adapter){
53              _adapter = adapter;
54              start = true ;
55          }

57          AgentFactoryI::AgentFactoryI(){
58              start = false ;
59          }

61          AgentFactoryI::~AgentFactoryI(){

63          }

65          void AgentFactoryI::cleanAgents(){
66              for (unsigned int i = 0 ; i < this->agents.size() ; i++){
67                  if ( this->agents.at(i)->getZombie() ){
68                      //delete this->agents.at(i);
69                      this->agents.erase(this->agents.begin() + i);
70                  }
71              }
72          }

74          AgentI* AgentFactoryI::findAgent(const int& id){
75              for (unsigned int i = 0 ; i < this->agents.size() ; i++){
76                  if ( this->agents.at(i)->getId() == id ){
77                      return this->agents.at(i);
78                  }
79              }
80              return NULL;
81          }

83          AgentPrx AgentFactoryI::createAgent(int id, typeObject type,
               const Ice::Current& c){
84              if (!start){
85                  start = true ;
86                  _adapter = c.adapter ;
87              }
88              AgentI* agent ;
89              AgentPrx agentProxy;
90              string libAgent;
91              string typeAgent;
92              string text, textError ;
93              vector<AgentI*>::iterator it;

95              // SPECIFIED DOMAIN
96              // Select the dynamic library in function of type agent
97              switch(type){
98                  case PERSON:
99                      libAgent = ROUTE_DLL + "libAgentPerson.so";
100                     typeAgent = "getAgentPerson" ;
101                     text = "[AgentFactory] Instantiating Agent Person
                           with id: " +ops::numToString(id);
102                 break;
103                 case VEHICLE:
104                     libAgent = ROUTE_DLL + "libAgentVehicle.so";
105                     typeAgent = "getAgentVehicle";
106                     text = "[AgentFactory] Instantiating Agent Vehicle
```

```
                    with id: "+ops::numToString(id);
107             break;
108         }

110         //Save the changes in the logger
111         _fileMutex.lock();
112         Logger::getInstance()->write(text.c_str());
113         cout << text  << endl ;
114         _fileMutex.unlock();

116         // Open the dynamic library
117         void *handle = dlopen(libAgent.c_str(), RTLD_LAZY);

119         if (!handle){
120                 textError = "[AgentFactory] Cannot open dynamic
                        library";
121                 _fileMutex.lock();
122                 Logger::getInstance()->write_error(textError.c_str
                        ());
123                 _fileMutex.unlock();
124                 exit(1);

126         }
127         // Clean errors
128         dlerror();

130         getAgent_t getAgent = (getAgent_t) dlsym(handle, typeAgent
                .c_str());
131         const char* dlsym_error = dlerror();

133         if (dlsym_error){
134                 textError = "[AgentFactory] Cannot open symbol '
                        getAgent()'";
135                 _fileMutex.lock();
136                 Logger::getInstance()->write_error(textError.c_str
                        ());
137                 _fileMutex.unlock();
138                 exit(1);
139         }


142         // Add the servant for the agent
143         try
144         {
145             // Create the agent with the dynamic library
146             agent = getAgent(id,_adapter);

148             // Add the agents to the list
149             this->agents.push_back(agent);

151             // Subscribe the agent to the topics of Icestorm
152             agent->subscribe();
153         }

155         catch(const Ice::AlreadyRegisteredException&) {}

157         // Eliminates the agents are in zombie state
158         cleanAgents();
```

```
160             // Close the handle of the Dll
162             dlclose(handle);
164             return agent ;
166         }
167 }
```

En **último lugar**, sólo sería necesario implementar los comportamientos específicos, que van a estar asociados a los agentes específicos del entorno a monitorizar.

Anexo C

Archivos de configuración de Amaavem

```
#
# The IceGrid locator proxy.
# -h localhost = IP OF THE REGISTRY !

Ice.Default.Locator=PFC/Locator:default -h localhost -p 4061

#
# This property is used by the clients to connect to IceStorm.
#
TopicManager.Proxy=EventChannel/TopicManager
TopicManager.Proxy2=BlackBoard-Agents/TopicManager

#
# Servers. Property of the Proxy
#
Agent.Endpoints= tcp:udp

EventChannel.Endpoints=tcp:udp

AgentBD.Proxy=agentbd
```

Figura C.1: Archivo de configuración de IceGrid para las aplicaciones cliente.

```
#
# The IceGrid instance name.
#
IceGrid.InstanceName=PFC

#
# The IceGrid locator proxy.
#
Ice.Default.Locator=PFC/Locator:default -h localhost -p 4061

#
# IceGrid registry configuration.
#
IceGrid.Registry.Client.Endpoints=default -p 4061
IceGrid.Registry.Server.Endpoints=default
IceGrid.Registry.Internal.Endpoints=default
IceGrid.Registry.Data=db/registry
IceGrid.Registry.PermissionsVerifier=PFC/NullPermissionsVerifier
IceGrid.Registry.AdminPermissionsVerifier=PFC/NullPermissionsVerifier
IceGrid.Registry.SSLPermissionsVerifier=PFC/NullSSLPermissionsVerifier
IceGrid.Registry.AdminSSLPermissionsVerifier=PFC/
   NullSSLPermissionsVerifier


#
# IceGrid node configuration.
#
IceGrid.Node.Name=node1
IceGrid.Node.Endpoints=default
IceGrid.Node.Data=db/node
IceGrid.Node.CollocateRegistry=1
IceGrid.Node.Output=db
IceGrid.Node.RedirectErrToOut=1

#
# Trace properties.
#
IceGrid.Node.Trace.Activator=1
#IceGrid.Node.Trace.Adapter=2
#IceGrid.Node.Trace.Server=3

#
# username and password for icegridadmin.
#
IceGridAdmin.Username=foo
IceGridAdmin.Password=bar
```

Figura C.2: Archivo de configuración de IceGrid para el registry (nodo principal).

C.1. Archivo XML para desplegar Amaavem

Y a continuación se muestra el código del archivo *amaavem.xml*, donde se definen las opciones para desplegar Amaavem.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!-- This file was written by IceGrid Admin -->
<icegrid>
   <application name="amaavem">
      <service-template id="IceStorm">
         <parameter name="instance-name"/>
         <parameter name="topic-manager-endpoints" default="tcp -h
            localhost"/>
         <parameter name="publish-endpoints" default="tcp -h localhost
            "/>
         <parameter name="flush-timeout" default="1000"/>
         <service name="${instance-name}" entry="IceStormService,34
            :createIceStorm">
            <properties>
               <property name="${service}.InstanceName" value="${
                  instance-name}"/>
               <property name="${service}.Flush.Timeout" value="${
                  flush-timeout}"/>
            </properties>
            <adapter name="${service}.Publish" endpoints="${publish-
               endpoints}" id="${instance-name}.Publish" register-
               process="true"/>
            <adapter name="${service}.TopicManager" endpoints="${topic
               -manager-endpoints}" id="${instance-name}.TopicManager
               " register-process="true">
               <object identity="${instance-name}/TopicManager" type="
                  ::IceStorm::TopicManager"/>
            </adapter>
            <dbenv name="${service}"/>
         </service>
      </service-template>
      <service-template id="ServiceBDTemplate">
         <parameter name="name"/>
         <service name="${name}" entry="UpdateBDService:create">
            <description>Simple that Update the BD - ${name}</
               description>
            <properties>
               <property name="AgentBD.Identity" value="agentbd"/>
            </properties>
            <adapter name="AgentBD-${name}" endpoints="default" id="
               AgentBD-${name}" replica-group="ServiceBDGroup" server
               -lifetime="false"/>
         </service>
      </service-template>
      <server-template id="FactoryServer">
         <parameter name="index"/>
         <server id="FactoryServer-${index}" activation="on-demand"
            exe="exec/AgentFactoryServer">
            <properties>
               <property name="Identity" value="agentfactory"/>
            </properties>
            <adapter name="AgentFactory" endpoints="tcp" id="${server
```

```
                }.AgentFactory" replica-group="FactoryGroup"/>
39          </server>
40       </server-template>
41       <server-template id="IceStorm">
42          <parameter name="instance-name" default="${application}.
               IceStorm"/>
43          <parameter name="topic-manager-endpoints" default="tcp -h
               localhost"/>
44          <parameter name="publish-endpoints" default="tcp -h localhost
               "/>
45          <parameter name="flush-timeout" default="1000"/>
46          <icebox id="${instance-name}" activation="on-demand" exe="
               icebox">
47             <properties>
48                <property name="IceBox.InstanceName" value="${server}"/
                     >
49                <property name="IceBox.ServiceManager.Endpoints" value=
                     "tcp -h localhost"/>
50                <property name="Ice.Trace.Network" value="1"/>
51                <property name="IceStorm.Trace.Subscriber" value="1"/>
52             </properties>
53             <log path="Log/IceStormServer.err" property="Ice.StdErr"/>
54             <log path="Log/IceStormServer.out" property="Ice.StdOut"/>
55             <service-instance template="IceStorm" instance-name="${
                  instance-name}" topic-manager-endpoints="${topic-
                  manager-endpoints}" publish-endpoints="${publish-
                  endpoints}" flush-timeout="${flush-timeout}"/>
56          </icebox>
57       </server-template>
58       <server-template id="VigilanceServer">
59          <parameter name="index"/>
60          <server id="VigilanceServer-${index}" activation="on-demand"
               exe="exec/VigilanceManagerServer">
61             <properties>
62                <property name="Identity" value="vigilancemanager"/>
63             </properties>
64             <adapter name="VigilanceManager" endpoints="tcp" id="${
                  server}.VigilanceManager" replica-group="
                  VigilanceGroup"/>
65          </server>
66       </server-template>
67       <replica-group id="FactoryGroup">
68          <load-balancing type="random" n-replicas="2"/>
69          <object identity="agentfactory" type="::pfc::AgentFactory"/>
70       </replica-group>
71       <replica-group id="ServiceBDGroup">
72          <load-balancing type="round-robin" n-replicas="1"/>
73          <object identity="agentbd" type="::pfc::AgentBD"/>
74       </replica-group>
75       <replica-group id="VigilanceGroup">
76          <load-balancing type="round-robin" n-replicas="1"/>
77          <object identity="vigilancemanager" type="
               ::pfc::VigilanceManager"/>
78       </replica-group>

80       <!-- HOST 1 -->
81       <node name="node1">
82         <!-- Service of AgentBD -->
```

```
83          <icebox id="IceBox" activation="on-demand" exe="icebox">
84            <description>A sample IceBox server</description>
85            <properties>
86                 <property name="Ice.Admin.Endpoints" value="tcp -h
                      localhost"/>
87                 <property name="IceBox.InstanceName" value="${server}"/
                      >
88                 <property name="IceBox.Trace.ServiceObserver" value="1"
                      />
89            </properties>
90            <service-instance template="ServiceBDTemplate" name="BD1"/>
91          </icebox>
92          <!-- Servers of FactoryServer -->
93          <server-instance template="FactoryServer" index="1"/>
94          <!-- Service of EventChannel (IceStorm) -->
95          <server-instance template="IceStorm" instance-name="
              EventChannel"/>
96          <!-- Service of BlackBoard-Agents (IceStorm) -->
97          <server-instance template="IceStorm" instance-name="BlackBoard
              -Agents"/>
98          <!-- Servers of VigilanceManagerServer -->
99          <server-instance template="VigilanceServer" index="1"/>

101       </node>


104    </application>
105 </icegrid>
```

Anexo D

Esquema de la base de datos empleada por Amaavem

```
--
-- Create schema pfc
--

CREATE DATABASE IF NOT EXISTS pfc;
USE pfc;

--
-- Definition of table `pfc`.`Audio`
--

DROP TABLE IF EXISTS `pfc`.`Audio`;
CREATE TABLE  `pfc`.`Audio` (
  `microphone` smallint(6) NOT NULL,
  `track` int(11) NOT NULL,
  `info` text NOT NULL,
  PRIMARY KEY (`microphone`,`track`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;


--
-- Definition of table `pfc`.`Incidents`
--

DROP TABLE IF EXISTS `pfc`.`Incidents`;
CREATE TABLE  `pfc`.`Incidents` (
  `camera` smallint(6) NOT NULL,
  `frame` smallint(6) NOT NULL,
  `object_id` smallint(6) NOT NULL,
  `object_id2` smallint(6) NOT NULL,
  `normality` float DEFAULT NULL,
  `incidence` text CHARACTER SET latin1,
  PRIMARY KEY (`camera`,`frame`,`object_id`,`object_id2`)
) ENGINE=MyISAM DEFAULT CHARSET=utf8;

--
-- Definition of table `pfc`.`IncidentsAudio`
--

DROP TABLE IF EXISTS `pfc`.`IncidentsAudio`;
CREATE TABLE  `pfc`.`IncidentsAudio` (
  `microphone` smallint(6) NOT NULL,
  `track` smallint(6) NOT NULL,
  `normality` float DEFAULT NULL,
```

```
45    `incidence` text CHARACTER SET latin1,
46    PRIMARY KEY (`microphone`,`track`)
47  ) ENGINE=MyISAM DEFAULT CHARSET=utf8;


50  --
51  -- Definition of table `pfc`.`Tracking`
52  --

54  DROP TABLE IF EXISTS `pfc`.`Tracking`;
55  CREATE TABLE  `pfc`.`Tracking` (
56    `camera` smallint(3) unsigned NOT NULL,
57    `frame` smallint(6) unsigned NOT NULL,
58    `object_id` varchar(6) NOT NULL,
59    `x` smallint(6) NOT NULL,
60    `y` smallint(6) NOT NULL,
61    `width` smallint(6) NOT NULL,
62    `height` smallint(6) NOT NULL,
63    PRIMARY KEY (`camera`,`frame`,`object_id`)
64  ) ENGINE=MyISAM DEFAULT CHARSET=utf8;
```

Anexo E

Interfaces de comunicación de Amaavem

El código slice del archivo *config.ice*:

```
#ifndef CONFIGICE_H
#define CONFIGICE_H


module pfc {

    // route of the dynamic libraries
    const string ROUTE_DLL = "./libs/libsDll/";

    // route of the config ICE
    const string ROUTE_CONFIG_CLIENT = "./configIce/config.client";

    // MYSQL. Information of access
    const string IP_BD = "localhost";
    const string USER_BD = "root";
    const string PASS_BD = "admin";
    const string BD = "pfc";

    // Maximum of changes of type for an agent
    const int NUM_ERRORS_MAX = 50;

    // Minimum value of belief for one type of agent
    const double MIN_VALUE_BELIEF = 0.5 ;

    // Maximum time idle for an agent (seg)
    const double TIME_FINISH = 90 ;

    // SPECIFIED OF DOMAIN
    // enum for structify objets
    enum typeObject { PERSON, VEHICLE };

    // Structure to save the frames
    struct frameBD{
        int camera;
        int numFrame ;
        int id;
        int x;
        int y;
        int width ;
        int height ;
    };
```

```
43      // Structure to save the tracks audio
44      struct audioBD{
45          int microphone;
46          int track;
47          string text;
48      };

50      // Structure to save the incidences
51      struct incidence{
52          int camera;
53          int numFrame;
54          int id ;
55          int id2;
56          double normality;
57          string type;
58      };

60      struct incidenceAudio{
61          int microphone;
62          int track;
63          double normality;
64          string type;
65      };

67      // position of an object
68      struct vec3 {
69          int x;
70          int y;
71          int z;
72      };

74      // extra information of an object
75      struct infoObject{
76          int width;
77          int height;
78      };

80      // information of an object. id, position, details
81      struct objTracking {
82          int id ;
83          vec3 position;
84          infoObject details ;
85      };

87      sequence<objTracking> lObjects;

89      // generic event
90      class Event {};

92      // video events
93      class EventVideo extends Event{
94          int camera;
95          int frame ;
96          lObjects objects ;
97      };

99      // audio events
100     class EventAudio extends Event {
```

```
101         int microphone;
102         int track ;
103         string info;

105     };

107     // struct key - value
108     // key - type object
109     // value - Degree of belief of an object type
110     struct keyValue{
111         typeObject key;
112         float value;
113     };

115     // Class for the communication of Agents
116     class Message{
117         Object* agentSource;
118         string text;

120     };

122     // resolution of an image
123     struct resolution{
124         int x;
125         int y;
126     };

128 };

130 #endif
```

El código slice del archivo *Agent.ice*:

```
2  #ifndef AGENT_ICE
3  #define AGENT_ICE

5  #include <config.ice>

7  module pfc
8  {
9      interface Agent
10     {
11         idempotent void notify(Event ev);
12         idempotent void notifyMessage(Message sms);
13     };
14 };

16 #endif
```

El código slice del archivo *AgentBD.ice*:

```
#ifndef AGENTBD_ICE
#define AGENTBD_ICE

#include <config.ice>

module pfc
{
    interface AgentBD
    {
        idempotent void updateBD(incidence inc);
    };

};
#endif
```

El código slice del archivo *AgentFactory.ice*:

```
#ifndef AGENTFACTORY_ICE
#define AGENTFACTORY_ICE

#include <config.ice>
#include <Agent.ice>

module pfc
{
    interface AgentFactory
    {
        Agent* createAgent(int id, typeObject type);
    };
};
#endif
```

El código slice del archivo *VigilanceManager.ice*:

```
#ifndef VIGILANCEMANAGER_ICE
#define VIGILANCEMANAGER_ICE

#include <config.ice>

module pfc
{
sequence<Event> vEvents;

    interface VigilanceManager
    {
        idempotent void notify(vEvents events);
        idempotent void notifyOne(Event ev);
    };
};

#endif
```

Anexo F

Código fuente

Dada la extensión del código fuente, se entrega el mismo en su versión electrónica, en el *DVD* adjunto a la parte anterior de este libro.

Dicho código fuente se estructura en los siguientes directorios:

- **/src:** contiene el cuerpo de las clases.
- **/src/utils:** contiene el código del parseador de ficheros *XML*.
- **/libs:** contiene la cabecera de las clases.
- **/Debug:** contiene el código del módulo de análisis de resultados.
- **/configIce:** contiene los archivos de configuración de *ZeroC Ice*.

F.1. Código del parseador de ficheros XML

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import sys
import xml.dom.minidom as minidom
import MySQLdb

db=MySQLdb.connect(host="localhost",user="root", passwd="admin",db="
    pfc")
cursor=db.cursor()
# FORMAT OF THE XML FILES
# -----------------------
# NAME = ObjectXML[NUM].xml - Example ObjectXML1.xml
# FORMAT OF THE BODY =
# <OBJECT TIME="110">
#   <OBSERVATION>
#     <BOUNDING_BOX BOTTOM="247" LEFT="1" RIGHT="15" TOP="287"/>
#     <CENTROID CENTER_X="8" CENTER_Y="267"/>
#     <POSITION BASE_X="8" BASE_Y="247"/>
#   </OBSERVATION>
#   <OCCLUSION>
#       NOTOCCLUDED
#   </OCCLUSION>
# </OBJECT>
```

```
24 #
       ----------------------------------------------------------------------

25 def saveFrames(xml,idObject, camera):

27     doc = minidom.parse(xml)
28     node = doc.documentElement
29     frames = doc.getElementsByTagName("OBJECT")

31     events = []
32     for frame in frames:

34         numFrame = frame.getAttribute('TIME')
35         frameNum = frame.getElementsByTagName("OBSERVATION")[0]
36         bounding = frameNum.getElementsByTagName("BOUNDING_BOX")[0]
37         centroid = frameNum.getElementsByTagName("CENTROID")[0]
38         position = frameNum.getElementsByTagName("POSITION")[0]

40         bottom = bounding.getAttribute('BOTTOM')
41         top = bounding.getAttribute('TOP')
42         left = bounding.getAttribute('LEFT')
43         right = bounding.getAttribute('RIGHT')

45         x = centroid.getAttribute('CENTER_X')
46         y = centroid.getAttribute('CENTER_Y')

48         width = (int)(right) - (int)(left)
49         height = (int)(top) - (int)(bottom)

51         events.append([camera,(int)(numFrame),idObject,(int)(x),(int)(
               y),width,height])

53     for event in events:
54         sql = "INSERT INTO Tracking values(%s,%s,%s,%s,%s,%s,%s)" %(
               event[0],event[1],event[2],event[3],event[4],event[5],
               event[6])
55         cursor.execute(sql)


58 if __name__ == "__main__":
59     if(len(sys.argv) != 4 ):
60         print "Parameters are invalids."
61         print "Usage ./parser.py [DIR_SOURCE_FILES] [NUM_OBJECTS] [
               NUM_CAMERA]"
62         print "Format of the files = \"ObjectXML%d.xml\", %d is a
               integer \n"
63         sys.exit(0)

65     dirFiles = sys.argv[1]
66     objects = int(sys.argv[2])
67     camera = int(sys.argv[3])
68     for i in range(objects):
69         idObject = (int) (i)
70         document = dirFiles+"/ObjectXML%d.xml" %(idObject)
71         saveFrames(document, idObject, camera)
```

F.2. Código del analizador de vídeos

```
#!/usr/bin/python
# -*- coding: utf-8 -*-

import MySQLdb
import sys, pygame
import time

factor_add = 0.125
min_barra = 150

def draw_absolutely_abnormal(screen,frame,normality):
    if beh_totalAnor[frame] > 1:
        normality = normality + factor_add * beh_totalAnor[frame]
        if (normality > 1): normality = 1

    pygame.draw.rect(screen, (255,0,0), (125,15,min_barra+normality
       *150,10))
    #font = pygame.font.Font(None, 20)
    text = font.render(str(beh_totalAnor[frame]), True, (0,0,0))
    screen.blit(text, (126,10))
    pygame.display.flip()

def draw_possibly_abnormal(screen,frame,normality):
    if beh_posAnor[frame] > 1:
        normality = normality + factor_add * beh_posAnor[frame]
        if (normality > 1): normality = 1

    pygame.draw.rect(screen, (255,102,0), (125,30,min_barra+normality
       *150,10))
    text = font.render(str(beh_posAnor[frame]), True, (0,0,0))
    screen.blit(text, (126,25))
    pygame.display.flip()

def draw_suspicious(screen,frame,normality):
    if beh_susp[frame] > 1:
        normality = normality + factor_add * beh_susp[frame]
        if (normality > 1): normality = 1

    pygame.draw.rect(screen, (255,216,0), (125,45,min_barra+normality
       *150,10))
    text = font.render(str(beh_susp[frame]), True, (0,0,0))
    screen.blit(text, (126,40))
    pygame.display.flip()

def draw_possibly_normal(screen,frame,normality):
    if beh_posNor[frame] > 1:
        normality = normality + factor_add * beh_posNor[frame]
        if (normality > 1): normality = 1

    pygame.draw.rect(screen, (192,255,0), (125,60,min_barra+normality
       *150,10))
    text = font.render(str(beh_posNor[frame]), True, (0,0,0))
    screen.blit(text, (126,55))
    pygame.display.flip()
```

```
53 def draw_absolutely_normal(screen,frame,normality):
54     if beh_totalNor[frame] > 1:
55         normality = normality + factor_add * beh_totalNor[frame]
56         if (normality > 1): normality = 1
57
58     pygame.draw.rect(screen, (18,255,0), (125,75,min_barra+normality
           *150,10))
59     text = font.render(str(beh_totalNor[frame]), True, (0,0,0))
60     screen.blit(text, (126,70))
61     pygame.display.flip()
62
63 def check_normality(screen,frame,normality):
64     # 1 - NORMALITY TOTAL
65     # 0 - ABNORMALITY TOTAL
66
67     if (1 >= normality > 0.8):
68         beh_totalNor[frame]+=1
69         #value_normalized = (normality - 0.8) / 0.2
70         draw_absolutely_normal(screen,frame,normality)
71     if (0.8 >= normality > 0.6):
72         beh_posNor[frame]+=1
73         value_normalized = (normality - 0.6) / 0.2
74         draw_possibly_normal(screen,frame,value_normalized)
75     if (0.6 >= normality > 0.4):
76         beh_susp[frame]+=1
77         value_normalized = (normality - 0.4) / 0.2
78         draw_suspicious(screen,frame,value_normalized)
79     if (0.4 >= normality > 0.2):
80         beh_posAnor[frame]+=1
81         value_normalized = (normality - 0.2) / 0.2
82         draw_possibly_abnormal(screen,frame,value_normalized)
83     if (0.2 >= normality >=0):
84         beh_totalAnor[frame]+=1
85         #value_normalized = (normality - 0.0) / 0.2
86         draw_absolutely_abnormal(screen,frame,normality)
87
88 def objects_near(frame, x , y ):
89     max_distance = 30
90     if (positions_num[frame] > 0 ):
91         xObj = positions[frame][0]
92         yObj = positions[frame][1]
93         if ( (xObj + max_distance >= x) and (xObj-max_distance <= x)
94                 and (yObj + max_distance >= y) and (yObj -
                       max_distance <= y)):
95             return True
96         return False
97
98 def getNewPosition(frame,x,y):
99     factor = 20;
100    if (positions_num[frame] <=3):
101        return [x, y + positions_num[frame] *  factor]
102    else:
103        return [x, y - (positions_num[frame]-3) *  factor]
104
105
106 if __name__ == "__main__":
107     pygame.init()
```

```
109     #CONSTANTS COLORS

111     BLACK = (0,0,0)
112     RED = (255,0,0)
113     GREEN = (0,255,0)
114     BLUE = (0,0,255)
115     WHITE = (255,255,255)


118     if(len(sys.argv) != 6 ):
119         print "Parameters are invalids.\n"
120         print "Usage ./video.py [DIR_SOURCE_FRAMES] [
                DIR_DESTINATION_MODIFIED_FRAMES] [INIT_NumFRAME] [
                END_NumFRAME] [NumCAMERA]\n"
121         sys.exit(0)

123     # Variables
124     dirSource = sys.argv[1]
125     dirDestination = sys.argv[2]
126     initFrame = int(sys.argv[3])
127     endFrame = int(sys.argv[4])
128     camera = int(sys.argv[5])
129     maxX = 768
130     maxY = 576

132     # Zone restricted for the cars
133     zone = [(520,363),(607,317),(705,330),(654,390)]

135     # Behaviors counters
136     beh_totalAnor = {}
137     beh_posAnor = {}
138     beh_susp = {}
139     beh_posNor = {}
140     beh_totalNor = {}
141     positions = {}
142     positions_num = {}

144     db=MySQLdb.connect(host="localhost",user="root", passwd="admin",db
            ="pfc")
145     cursor=db.cursor()

147     screen = pygame.display.set_mode((maxX, maxY))
148     pygame.display.set_caption("Amaavem: Intelligent Surveillance in
            an Urban Environment")

150     # For each frame show the incidents
151     for iFrame in range(initFrame,endFrame):
152         #Init behaviors counters
153         beh_totalAnor[iFrame] = 0
154         beh_posAnor[iFrame] = 0
155         beh_susp[iFrame] = 0
156         beh_posNor[iFrame] = 0
157         beh_totalNor[iFrame] = 0

159         #Init dictionary of positions
160         positions[iFrame] = [0,0]
161         positions_num[iFrame] = 0
```

```
163         # SQL query
164         sql = "Select * from Tracking where frame="+str(iFrame)+" and
                camera="+str(camera)

166         actualFrame = "Frame: "+ str(iFrame)
167         cursor.execute(sql)
168         resultado=cursor.fetchall()

170         image = pygame.image.load(""+dirSource+""+str(iFrame)+".png")
171         screen.blit(image, (0, 0))

173         pygame.draw.rect(screen, (0,0,0), (10,10,430,80))
174         pygame.draw.rect(screen, (0,0,0), (450,10,135,30))

176         font = pygame.font.Font(None, 22)
177         text = font.render(actualFrame, True, (255,255,255))
178         screen.blit(text, (463,15))
179         font = pygame.font.Font(None, 14)

181         text = font.render('Absolutely Abnormal', True, (255,255,255))
182         screen.blit(text, (20,15))

184         text = font.render('Possibly Abnormal', True, (255,255,255))
185         screen.blit(text, (20,30))

187         text = font.render('Suspicious', True, (255,255,255))
188         screen.blit(text, (20,45))

190         text = font.render('Possibly Normal', True, (255,255,255))
191         screen.blit(text, (20,60))

193         text = font.render('Absolutely Normal', True, (255,255,255))
194         screen.blit(text, (20,75))

196         pygame.draw.aalines(screen, RED, True, zone, 1)
197         pygame.display.flip()

199         #For each object in the frame
200         for registro in resultado:
201             myId = registro[2]
202             myX = registro[3]
203             myY = registro[4]

205             # Search objects near
206             if (positions_num[iFrame] == 0):
207                 positions[iFrame] = [myX,myY]
208                 positions_num[iFrame] = 1
209             else:
210                 positions_num[iFrame] += 1
211                 if (objects_near(iFrame,myX,myY)):
212                     newPositions = getNewPosition(iFrame,myX,myY)
213                     myX = newPositions[0]
214                     myY = newPositions[1]


217             textId = "Id: "+str(myId)
218             font = pygame.font.Font(None, 20)
```

```
219             text = font.render(textId, True, (120,255,21))
220             screen.blit(text, (myX,maxY-myY))
221             pygame.display.flip()

223             sql = "Select * from Incidents where frame="+str(iFrame)+"
                    and object_id='"+myId+"'"+ " and camera="+str(camera)
224             cursor.execute(sql)
225             resultado3=cursor.fetchall()
226             cadena_clasificacion = ""
227             skip = 15 ;
228             for regNormality in resultado3:
229                 normality = regNormality[4]
230                 check_normality(screen,iFrame,normality)
231                 objectId2 = regNormality[3]
232                 incident = regNormality[5]
233                 # Print the distances between cars and persons
234                 if ( incident.find("Distance") >= 0 ):
235                     textNormality = "Value with car"+str(objectId2)+":
                            "+str(normality)
236                     font = pygame.font.Font(None, 14)
237                     text = font.render(textNormality, True,
                           (120,255,21))
238                     screen.blit(text, (myX,maxY-myY + skip))


241                 # Print the velocity of the cars
242                 elif ( incident.find("Velocity") >= 0 ):
243                     font = pygame.font.Font(None, 14)
244                     text = font.render(incident, True, (120,255,21))
245                     screen.blit(text, (myX,maxY-myY + skip + 10))

247                 pygame.display.flip()
248                 skip +=10



252             # Search the car stopped in zones restricteds
253             sql = "Select * from Incidents where frame="+str(iFrame)+"
                    and camera="+str(camera)+ " and incidence = 'Stop
                   Vehicle In Zone'";
254             cursor.execute(sql)
255             resultado3=cursor.fetchall()

257             for regStop in resultado3:
258                 objectId = regStop[2]
259                 normality = regStop[4]
260                 check_normality(screen,iFrame,normality)
261                 incident = regStop[5]
262                 textNormality = "Car:"+str(objectId)+" in zone. Value=
                       "+str(normality)
263                 font = pygame.font.Font(None, 14)
264                 text = font.render(textNormality, True, (255,0,0))
265                 screen.blit(text, (600,300))
266                 pygame.display.flip()

268         num = "%05d" %(iFrame)
269         pygame.image.save(screen, dirDestination+"/img"+str(num)+".png
               ")
```

Anexo G

GNU Free Documentation License

Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "freeïn the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The “Document”, below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as “you”. You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document

that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

The "publisher" means any person or entity that distributes copies of the Document to the public.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or non-commercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on

explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all sections Entitled "Endorsements".

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document,

then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, or distribute it is void, and will automatically terminate your rights under this License.

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, receipt of a copy of some or all of the same material does not give you any rights to use it.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present

version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation. If the Document specifies that a proxy can decide which future versions of this License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Document.

11. RELICENSING

"Massive Multiauthor Collaboration Site" (or "MMC Site") means any World Wide Web server that publishes copyrightable works and also provides prominent facilities for anybody to edit those works. A public wiki that anybody can edit is an example of such a server. A "Massive Multiauthor Collaboration" (or "MMC") contained in the site means any set of copyrightable works thus published on the MMC site.

"CC-BY-SA" means the Creative Commons Attribution-Share Alike 3.0 license published by Creative Commons Corporation, a not-for-profit corporation with a principal place of business in San Francisco, California, as well as future copyleft versions of that license published by that same organization.

"Incorporate" means to publish or republish a Document, in whole or in part, as part of another Document.

An MMC is "eligible for relicensing" if it is licensed under this License, and if all works that were first published under this License somewhere other than this MMC, and subsequently incorporated in whole or in part into the MMC, (1) had no cover texts or invariant sections, and (2) were thus incorporated prior to November 1, 2008.

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.

Bibliografía

[Ber96] P. Bernstein. Middleware: A Model for Distributed System Services. *Communications of the ACM, 39:2*, 1996.

[Boo96] Barry Boone. *Java Essentials for C and C++ Programmers*. Addison-Wesley, 1996.

[Cor03] Collaborating Software: Blackboard and Multi-Agent Systems & the Future, 2003. `http://dancorkill.home.comcast.net/~dancorkill/pubs/ilc03.pdf`.

[CVG08] Computational Vision Group. [En línea], 2008. `http://www.cvg.cs.rdg.ac.uk/`.

[DGT10] Control inteligente en las grandes ciudades. [En línea], 2010. `http://www.dgt.es/revista/archivo/pdf/num201-2010-vigilancia.pdf`.

[DLC87] EH Durfee, VR Lesser, y DD Corkill. *Coherent cooperation among communicating problem solvers*. IEEE Transactions on Computers, 1987.

[DVD08] H. Detmold, A. Van Den Hengel, A. Dick, K. Falkner, D.S.Munro, y R. Morrison. Middleware for Distributed Video Surveillance. *IEEE Distributed Systems Online, 9(2):1–11*, 2008.

[Eck04] Bruce Eckel. *Thinking in C++: Practical programming*. Prentice Hall, 2004.

[Fer09] David Vallejo Fernández. *Tesis Doctoral: Arquitectura Multi-Agente basada en Servicios para un Sistema de Vigilancia Cognitiva*. PhD thesis, Universidad de Castilla-La Mancha,, 2009.

[GHJV94] E. Gamma, R. Helm, R. Johnson, y J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[Gli12] Glib::Timer Class Reference [En línea], 2012. `http://developer.gnome.org/glibmm/unstable/classGlib_1_1Timer.html`.

[Hen04] M.Henning. Massively Multiplayer Middleware. *ACM Queue, 1(10):38:45*, 2004.

[Hen09] M.Henning. Choosing Middleware: Why Performance and Scalability do (and do not) Matter. Technical report, ZeroC Inc. *ZeroC Inc*, 2009.

[Ice06] Next Generation Grid Computing. [En línea], 2006. `http://www.zeroc.com/newsletter/issue14.pdf`.

[Ice11] Ice Manual [En línea], 2011. http://doc.zeroc.com/display/Ice/Ice+Manual.

[NMW04] C.Ñorris, M. McCahill, , y D. Wood. The growth of CCTV: A global perspective on the international diffusion of video surveillance in publicly accessible space. *Surveillance & Society, 2(2/3):110–135*, 2004.

[Obj08] Object Management Group (OMG). CORBA Specifications (Document Access Page), 2008. http://www.omg.org/technology/documents/index.htm.

[RN04] S. Russell y P. Norvig. *Inteligencia Artificial: Un Enfoque Moderno.* Pearson Educación S.A, 2004.

[SMI04] G.J.D. Smith. Behind the screens: Examining constructions of deviance and informal practices among cctv control room operators in the UK. *Surveillance and Society*, 2004.

[TBH04] Yingli Tian, Arun Hampapur, Lisa Brown, Jonathan Connell, Hans Merkl Max Lu, Sharat Pankanti, Andrew Senior, y Chiao-Fe Shu. The IBM Smart Surveillance System. *19 Skyline Drive, Hawthorne, NY 10532 (914)-784-7440*, 2004.

[VBL05] S.A. Velastin, B.A. Boghossian, B.P.L. Lo, J. Sun, y M.A.Vicencio-Silva. Prismatica: Toward ambient intelligence in public transport environments. *IEEE Transactions on Systems, Man and Cybernetics, Part A, 35(1):164–182*, 2005.

[VV05] M. Valera y S.A. Velastin. *Intelligent distributed surveillance systems: a review*. In IEE Proceedings Vision, Image and Signal Processing, 2005.

[WD88] E. Wallace y C. Diffley. *CCTV control room ergonomics.Technical report, UK Home Office*. Police Scientific Development Branch (PSDB), 1988.

[Wei99] G. Weiss. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, 1999.

[Woo01] Michael Wooldridge. *An Introduction to MultiAgent Systems.* John Wiley y Sons, Inc. New York, 2001.

[WS004] W3C Consortium. Web Services Architecture. [En línea], 2004. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/#whatis.