Universidad Complutense
Facultad de Informática

Sistemas Informaticos 2009/2010

Icaro-D: Infraestructura Multiagente Distribuida

Autores:
Andrés Picazo Cuesta
Arturo Mazon Tribiño
Alejandro Fernández Carrión

Supervisores:
Juan Pavón Mestras
Francisco Garijo

11 Junio 2010

Índice

Autorización Legal

Los abajo firmantes, matriculados en la asignatura de Sistemas Informáticos de la Facultad de Informática, autorizan a la Universidad Complutense de Madrid a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a los autores el presente trabajo de proyecto de Sistemas Informáticos: "Icaro-D: Infraestructura Multiagente Distribuida", realizado durante el curso académico 2009-2010, bajo la dirección de Juan Pavón Mestras (UCM) y Francisco Garijo, incluyendo la propia memoria, el código, la documentación y el prototipo desarrollado.

También autorizan a la biblioteca de la Universidad Complutense de Madrid, a depositarlo en el archivo institucional E-Prints Complutense, con el objeto de incrementar la difusión, uso e impacto del trabajo en internet y garantizar su preservación y acceso a largo plazo.

Andrés Picazo **Arturo Mazón** **Alejandro Fernández**

Resumen

El presente trabajo parte de la infraestructura Icaro para sistemas multi-agente y pretende ampliarlo incluyendo un sistema de comunicación entre agentes que facilite la distribución de los componentes de Icaro (agentes y recursos) en una red, simplificando el problema de las comunicaciones. Con este objetivo se ha incorporado un nuevo componente, un gestor de comunicaciones, que realiza esta tarea de forma transparente a los desarrolladores. También se ha incorporado tres nuevas clases que reducen la carga de trabajo de un desarrollador, simplificando el uso de la infraestructura como puede comprobarse en los ejemplos incluidos. Adicionalmente se han introducido en paralelo diversas mejoras en el codigo original de Icaro, entre las que destacan la simplificación y depuración de métodos, y la implementación de un nuevo sistema de trazas más completo que el original.

Para validar el presente trabajo se han desarrollado dos aplicaciones, un chat, que sirva como demostrador de todas las funcionalidades implementadas, y una aplicación que sirva para ilustrar la guia de ejemplo incluida en esta documentación. Finalmente se ha usado una aplicación ya existente, masterIA, para mostrar el proceso de migración de un modelo centralizado a uno distribuido.

Actualmente las aplicaciones distribuidas son muy utilizadas e importantes, y este proyecto aporta una herramienta que facilita la tarea de crear aplicaciones distribuidas eliminando la necesidad de partir de cero.

Palabras clave: Agente, Multiagente, Distribuido, Java, Icaro, RMI, Infraestructura, Organización, Sistemas multiagente, Aplicaciones inteligentes.

Abstract

The developed work is based on the software multi-agent system Icaro. Its purpose is extending it with the comunication layer between agents facilitating the distribution for Icaro components (agents and resources), in a computer network. With this objective it has been incorporated a new component, the comunications manager, to implement this task. Also, three new clases have been implemented to reduce the workload of developers, simplifying the use of the platform as can be seen in the validation chapter. Aditionally, several upgrades has been introduced in the original Icaro source code, highlighting the simplification and grouping of methods, and the implementation of a new visual tracing system.

To validate this work it has been developed two applications, a chat, to serve as a testbench of the new functionality, and another to illustrate the tutorial present in this documentation. Finally, it has been used an existing application, masterIA, to demostrate the migration from centralized to distributed.

Distributed applications today are very important and widespread, and this project provides a tool which facilitates the task of creating distributed applications, removing the need to start from zero.

Keywords: Agent, Mullti-agent, Java, Distributed, RMI, Icaro, Organization, Infraestructure, Multi-agent systems, Inteligent aplications.

1. Introducción

1.1. Motivación

El proyecto Icaro-D, surgie a raíz del trabajo realizado en la asignatura de ingeniería del software, donde se desarrollo una aplicación (gestion de una PYME) para una empresa utilizando la infraestructura ICARO-Mini. Durante el desarrollo de ese proyecto surgieron ideas de como mejorar la infraestructura Icaro, que se implementan en este trabajo.

La principal carencia que se encuentra en Icaro durante este periodo es la ausencia de un sistema que permita distribuir los componentes de Icaro en distintos nodos de una red de ordenadores, detalle que se considera, según algunas definiciones (por ejemplo, la que puede encontrarse en pagina de wikipedia), como algo imprescindible para los sistemas multi agente. Dado que la distribución de aplicaciones es además una solución empleada comúnmente para resolver problemas durante el desarrollo de algunas aplicaciones que pueden ser empleadas por varios usuarios a la vez, y que necesiten establecer una forma de asegurar la sincronización de los datos, se considera que esta aportación resulta una importante mejora sobre la versión original de Icaro.

Las alternativas existentes a Icaro, como por ejemplo Jade o Semantic Agent ya implementan este modelo, sin embargo son aplicaciones que requieren una mayor curva de aprendizaje, y en muchos casos, resultan más pesadas y complejas de utilizar. Por ello se considera necesario incluir esta funcionalidad en Icaro, más ligero y basado en conceptos más familiares a la mayoria de desarrolladores, para poder convertirse en una alternativa atractiva, e incluso didáctica, para sus potenciales usuarios.

1.2. Qué es Icaro

Icaro-T (nombre original) es una infraestructura software ligera, basada en el lenguaje de programación Java, para el desarrollo de sistemas multiagente mediante organizaciones de agentes. Estas organizaciones suponen una de las grandes diferencias entre Icaro y otras plataformas, como podría ser Jade. Cada aplicación que se implementa con Icaro utiliza una descripción de organización, escrita en formato xml, en la que se detallan que agentes y recursos componen la aplicación, sus modelos y sus instancias, asi como su integración dentro de una organización de control (gestores).

Mientras que otras plataformas multi agentes se centran en seguir un estándar para las comunicaciones, como KQML o FIPA, Icaro se centra en ofrecer componentes de alto nivel que faciliten el desarrollo de agentes capaces de coordinarse entre sí. Esto simplifica el desarrollo de los propios agentes y recursos, pero en cambio no implementa ningún estándar de comunicación como hacen la mayoría de plataformas, cuestión que se aborda en este trabajo.

Icaro esta modelado en tres capas: control, recursos e información. En este modelo la capa de control, formada por gestores y agentes especializados, con idéntica estructura a un agente normal, pero con disferente rol en la infraestructura, estos gestores son responsables de las tareas de gestión de una aplicación, tales como configuración,

activación y monitorización. Los agentes especializados son los encargados de llevar a cabo la funcionalidad propia de la aplicación para lo cual son capaces de colaborar entre ellos. La capa de recursos contiene los elementos que proveen a la aplicación de funcionalidades e información para alcanzar sus objetivos. Finalmente la capa de información contiene las entidades que modelan los datos de las aplicaciones. Puede verse en la figura siguiente esta estructura de capas de forma gráfica.

Además incluye ejemplos de utilización en proyectos software concretos para ayudar en el aprendizaje del uso de esta infraestructura. Estas aplicaciones se encuentran dentro del paquete "aplicaciones", como se detalla en el apartado "organizacion del código".

Un sistema multiagente podria definirse como un sistema distribuido en el cual los nodos o elementos son sistemas de inteligencia artificial, o bien un sistema distribuido donde la conducta combinada de dichos elementos produce un resultado en conjunto inteligente. (http://es.wikipedia.org/wiki/Sistema_multi_agente).

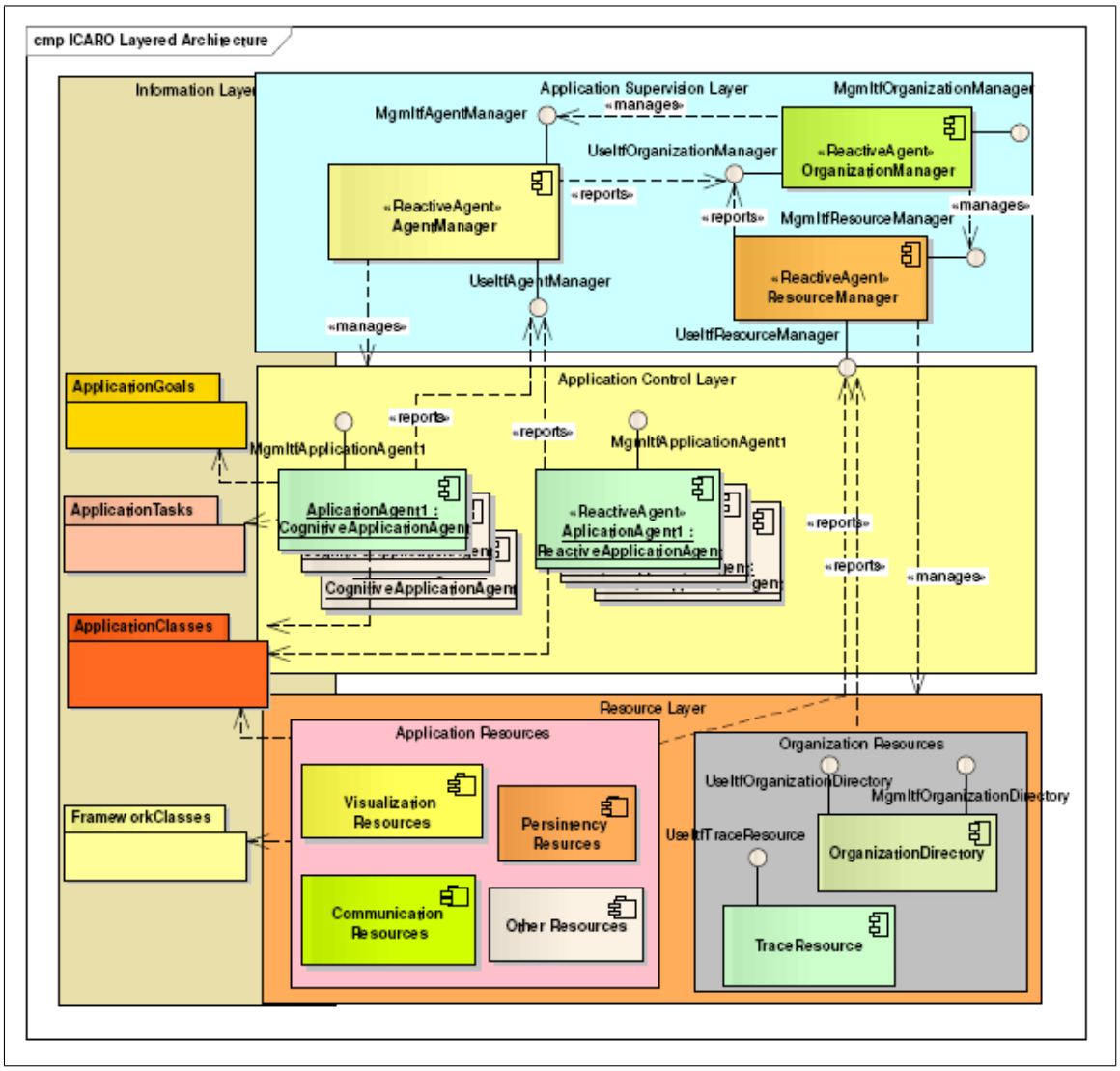


Figura 1: Modelo de capas de Icaro

1.3. Objetivos

El objetivo principal planteado para este proyecto es integrar en Icaro la capacidad de distribuir sus componentes (agentes), entre los distintos nodos de una red de ordenadores, convirtiendo Icaro en una plataforma atractiva para futuros desarrolladores de aplicaciones distribuidas.

Se pretende aislar la infraestructura del programador de forma que solo tenga que preocuparse de desarrollar su aplicación, y no de la infraestructura en la que se apoya.

Promover el uso de esta infraestructura para el desarrollo de todo tipo de aplicaciones futuras. Ofreciendo una base de software sencilla de utilizar y que permita una depuración sencilla de las aplicaciones desarrolladas sobre esta. Como objetivo adicional, se pretende además complementar la formación de los autores en materia de sistemas distribuidos.

- Hacer que las aplicaciones hechas con ICARO funcionen de manera distribuida.
- Permitir el uso de distintos protocolos de comunicación.
- Permitir al programador definir protocolos nuevos y/o editar los existentes.
- Representar gráficamente la comunicación entre agentes.
- Implementar un recurso para la comunicación entre agentes genérico transparente al programador.
- Implementar la posibilidad de ejecutar acciones de los agentes desde la ventana de trazas.
- Implementar un sistema de seguimiento de errores de comunicación.
- Implementación de una aplicación de simulación de redes de computadores como demostrador tecnológico.
- Mejorar la visualización de todos los agentes y recursos de las aplicaciones así como de la infraestructura.
- Mejorar el sistema de trazas de ICARO haciéndolo más claro y detallado para que sea más fácil analizar y depurar el software.

Para el desarrollo de este proyecto se dispuso de un tiempo de 8 meses, con una dedicación media de 12 horas semanales por cada uno de los tres integrantes del grupo de proyecto.

1.4. Publico y beneficios

Dado que este proyecto es en realidad la actualización y mejora de Icaro, seguirá estando disponible al mismo público que antes: programadores Java y grupos de desarrollo de aplicaciones en Java.

ICARO-D proporciona a los desarrolladores de aplicaciones distribuidas una infraestructura que evite tener que controlar cada detalle de comunicaciones y despliegue distribuidos de sus aplicaciones. Para esto se añade una capa de abstración que se encargua de realizar el trabajo más complejo, proporcionando métodos sencillos para realizar las tareas más complejas inherentes a los sistemas distribuidos, de modo que se puedan centrar en el desarrollo aunque no tengan mucha experiencia en este tipo de trabajo. Además se intenta mejorar en lo posible otros aspectos de la infraestructura, como la ventana de trazas, o la organización y limpieza del código.

El proyecto se distribuye bajo una licencia de software libre GNU-GPL, y se publicará en la página ofical de la forja morfeo, donde podrán encontrarse, además, versiones anteriores y la documentación completa de uso y desarrollo de Icaro. La dirección de la página es: http://icaro.morfeo-project.org/preguntas-frecuentes.

La licencia GNU-GPL puede consultarse en las siguientes direcciones:

- Versión oficial (Ingles): http://www.gnu.org/copyleft/gpl.html
- Versión español (traducción no oficial): http://www.viti.es/gnu/licenses/gpl.html

1.5. Plan de proyecto

Para este proyecto de sistemas informáticos se dispuso de ocho meses de trabajo y un equipo de tres personas. durante este tiempo se han dedicado una media de 12 horas semanales por cada miembro del equipo. Con estas limitaciones en cuenta se plantea la planificación que se muestra en la tabla que se encuentra más adelante.

La organización del tiempo de trabajo se ha basado en periodos de puesta en común del trabajo realizado, estudio individual de los aspectos desconocidos (como RMI) y reparto de trabajo en función del tiempo disponible, ya que dependiendo de la fecha hay integrantes del grupo con más tiempo y otros con menos. Así la mayor carga de trabajo se ha conseguido repartir de forma uniforme a lo largo de todo el año.

Para alojar el proyecto, asi como para proporcionar otros recursos de colaboración, como listas de correo y un repositorio de subversion, se ha seleccionado la plataforma Kenai, propiedad de Sun, ya que nos ofrece todas los recursos necesarios, e incluso más, como wiki, varios repositorios y todas las listas de correo que necesitemos. Se puede acceder a esta plataforma desde la direccion http://www.kenai.com

La coordinación se ha realizado mediante las listas de correo de Kenai y un sistema de control de versiones, ambos de la plataforma Kenai. Teniendo muy en cuenta que todos los días de trabajo se realizará una reunión para tomar decisiones y discutir sobre un plan común de desarrollo. Además se cuenta con hacer reuniones periódicas con los coordinadores para revisar el estado del trabajo realizado, asi como para discutir las opciones disponibles para resolver los problemas que se encuentren.

Para el objetivo de distribuir ICARO se utiliza la API de Java RMI como middleware. Se elige esta opción sobre otras alternativas como CORBA o ICE ya que el proyecto se implementa en Java y RMI minimiza las dependencias del proyecto, a pesar de ser algo más limitado, por ejemplo, siendo válido unicamente para comunicar

objetos Java.

El proyecto se distribuye bajo una licencia de software libre GNU-GPL. La planificación del proyecto se ha dividido en las siguientes fases:

1. Especificación del proyecto	Se deciden las funcionalidades que debía ofrecer la implementación de este proyecto, decidiendo finalmente pasar por las fases especificadas a continuación.
2. Integración de RMI en Icaro	Se integra RMI en los patrones de agente existentes en Icaro. En esta fase se incluye la decisión de la infraestructura a utilizar para las comunicaciones, decidiendo finalmente el uso de RMI.
3. Desarrollo de nueva ventana de trazas	Se implementa la nueva ventana de trazas que se ha comentado más arriba, con el objetivo de tener una herramienta lo más completa posible para ayudar a la depuracion de la integración de RMI que se realiza en la fase siguiente
4. Implementación de aplicación de chat	Con RMI integrado y una ventana de trazas más completa, se desarrolla durante esta fase la aplicación de chat especificada anteriormente, el objetivo es probar el funcionamiento de las comunicaciones entre dos ordenadores a traves de una red utilizando los elementos básicos que proporciona RMI. También durante esta fase se implementan la clase controlRMI, para agrupar los métodos más habituales en una única clase.
5. Creación del gestor de comunicaciones	Se diseña y se implementa el gestor de comunicaciones que se encargará de proporcionar el servicio de comunicaciones a los demás agentes, este gestor utiliza las controlRMI y comunicacionAgentes, para ofrecer funcionalidad adicional a las comunicaciones, como son los mensajes a grupos.
6. Paso de centralizado a distribuido	Se coge una aplicación existente desarrollada en Icaro, masterIA, y se prueba a cambiar su modelo centralizado por uno distribuido.
7. Implementación del gestor de nodo	Se desarrolla el gestor de nodo, encargado de esperar por el despliiegue de aplicaciones. También se prueban métodos para desplegar aplicaciones que compartan un mismo fichero de descripción de organización. Se realizan las pruebas usando la aplicación de chats.
8. Implementación de la aplicación de ejemplo	Se desarrolla una sencilla aplicación, con el objetivo de usarse como tutorial para ilustrar la guia de ejemplo incluida en esta documentación en el apartado del mismo nombre
9. Redacción de la documentación	En esta fase, con todos los componentes nuevos implementados, integrados y probados, pasamos a redactar la presente documentación

1.5.1. Recursos necesarios

En base a la experiencia de desarrollo, los recursos hardware necesarios para llevar a cabo los objetivos del proyecto son los siguientes :

- 3 Pcs de media potencia como sistema operativo Windows o Linux: Dado que el proyecto trata de un sistema distribuido se ha considerado necesario un mínimo de dos equipos para realizar las pruebas del desarrollo, que finalmente han sido tres, uno por cada integrante del equipo.
- Conexión a internet: Para poder hacer las pruebas con más de un equipo es imprescindible disponer de una conexión a traves de la cual mandar los mensajes habituales para un sistema distribuido.

Además, se han utilizado los siguientes recursos software:

- Plataforma de desarrollo: Dado que se trata de un proyecto de cierta envergadura consideró necesario el uso de un entorno que facilite el trabajo, por ello se seleccionó el entorno Netbeans IDE versión 6.8, ya que se encuentra integrado con el servidor usado para alojar el proyecto.
- Alojamiento en un servidor: Con el fin de coordinar al grupo se empleó un repositorio en el que alojar el proyecto para poder trabajar de forma independiente. Se ha utilizado la plataforma Kenai (http://www.Kenai.com), de Sun, para esta tarea, que proporciona tanto el repositorio de subversion como el acceso a listas de correo, servidor ftp y cualquier otro recurso que pudiera haber resultado necesario. La versión de trabajo del proyecto se encuentra alojada con el título "Infraestructura Distribuida Icaro".
- Listas de distribución de correo: Para poder comunicarnos entre nosotros y con nuestros coordinadores de proyecto se ha utilizado una lista de correo, proporcionada también por la plataforma Kenai.

Los conocimientos necesarios de los desarrolladores han sido:

- Experiencia en desarrollo con el lenguaje de programación Java y en el uso de la API RMI.
- Conocimientos en ingeniería del software.
- Uso y gestión de redes de computadores, al menos al nivel necesario para comprobar el estado de las conexiones de la red.
- Diseño de sistemas distribuidos.
- Experiencia en el uso de la plataforma Icaro.
- Entender los conceptos de agente, recurso y sistema multiagentes.

1.6. Cuestiones legales

El proyecto es totalmente libre, de código abierto y por lo tanto es público para el uso de cualquier persona. En concreto se autoriza a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a

sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

La licencia sobre la que se distribuye es GNU-GPL, que puede consultarse (en Inglés) en la página: http://www.gnu.org/copyleft/gpl.html

1.7. Plan de Validación

Para comprobar las mejoras que se han implementado, asi como su funcionalidad, se han implementado dos aplicaciones: la principal, un sencillo chat que sirviera tanto de banco de pruebas para los diferentes métodos de comunicaciones que se iban desarrollando, como un punto único donde puedan encontrarse ejemplos de cómo utilizar dichos métodos, comentados en el capítulo de implementación. Adicionalmente se ha incluido otra aplicación, que sirviera de tutorial a los nuevos desarrolladores, esta aplicación tiene lo mínimo necesario para mostrar, al menos en líneas generales, todos los pasos necesarios y las recomendaciones de los autores para crear una aplicación con esta infraestructura. Esta aplicación se utiliza como ejemplo en el apartado "guia de ejemplo", de esta documentación. Finalmente se incluye una última aplicación, MasterIA, que sigue el modelo centralizado, para demostrar como se puede migrar una aplicación ya existente, desarrollada con versiones anteriores de Icaro, al modelo distribuido. Estos ejemplos pueden consultarse en el capítulo "validación".

Utilizando estas aplicaciones se ha probado su correcto funcionamiento en distintas redes (cable y wifi, tanto en red local como en internet), cada vez que se completa algún objetivo del proyecto, para asegurar que se mantiene el mismo comportamiento, concretamente la aplicación "chat" ha sido la utilizada en todas las pruebas, utilizando las otras dos únicamente para probar su propio comportamiento, contando ya con garantías del comportamiento de la infraestructura.

Adicionalmente se han realizado revisiones de código, tanto del nuevo como del original. Durante este proceso se han eliminado errores nuevos y antiguos, existentes en la versión de Icaro sobre la cual se han implementado las mejoras de este proyecto, y se ha intentado organizar el código lo mejor posible, encapsulando todo lo posible las clases del proyecto, con el objetivo de tener un código legible y claro. Este trabajo está orientado a los interesados en desarrollar aplicaciones con Icaro, así como a aquellos que deseen realizar otras mejoras sobre esta infraestructura. En el capítulo "conclusiones y lineas abiertas" se detallan posibles mejoras sobre el presente trabajo.

Finalmente, se puede observar al trabajar con Icaro que se generan una gran cantidad de trazas durante la ejecución, incluso en el caso de aplicaciones de escasa complejidad, como pueden ser las desarrolladas para en apartado de validación. Icaro ya disponía de un recurso gráfico, la ventana de trazas, para permitir la depuración mediante las trazas, sin embargo resultaba algo farragoso y poco flexible, por lo que adicionalmente se ha implementado una nueva ventana, basada en tablas de Java, en la que se organiza mejor la información y que hace más sencillo el seguimiento de trazas, mediante funciones de ordenación y filtrado.

Como ya se ha comentado, los resultados de esta validación pueden consultarse en el capítulo "Validación", más adelante en esta documentación. También pueden

encontrarse referencias a las aplicaciones usadas como ejemplo en el apartado de “implementación”, ya que se han usado para poner ejemplos de uso de las nuevas funcionalidades.

2. Requisitos y Especificaciones

2.1. Requisitos del usuario

Este proyecto como se ha mencionado antes, esta orientado a programadores con los siguientes requisitos:

- Encapsulamiento de funcionalidad para poder utilizar una infraestructura multi-agente transparente.
- Uso de Java para el desarrollo de su trabajo.
- Opcionalmente el desarrollador puede necesitar que la aplicación funcione de manera distribuida en redes de ordenadores como por ejemplo Internet.
- Utilizar agentes reactivos para el control de la aplicación y recursos.
- Utilizar tecnología RMI para la distribución de la aplicación y su integración con otras aplicaciones que ya esten usando RMI.
- Proyectos de Ingeniería del Software.

2.2. Especificación de Requisitos de software

Las aplicaciones desarrolladas con ICARO-D deberán poder funcionar en redes de ordenadores, tanto locales como en Internet. La base de la comunicación requiere que los terminales(Nodos) tengan sus respectivas direcciones IP y todas la infraestructura arrancada.

Para el uso de Icaro-D hay que definir tres tipos de actores: el primero es el usuario estándar de las aplicaciones desarrolladas con la infraestructura, el segundo es el administrador, responsable del despliegue del sistema, nodos y aplicaciones, y el tercero es interno a la infraestructura, un agente o recurso (ajeno al usuario en sí) que se llamará componente. Estos son los actores presentes en los casos de usos que a continuación se describen.

Los siguientes casos de uso corresponden a las acciones realizadas por los actores físicos en relación al arranque y terminación del sistema, y al uso de las trazas.

2.2.1. Arranque del sistema

CU-01	Arrancar infraestructura
Objetivo en contexto	Arrancar en un nodo una aplicación entera o parte de una aplicación distribuida.
Precondiciones	Hay que tener bien configurado el fichero de descripción de organización XML que se vaya a utilizar.
Postcondiciones si éxito	Se arranca la aplicación según el descriptor de organización utilizado como parámetro. También arranca la infraestructura y se empieza a trazar en el recurso de trazas.
Postcondiciones si fallo	Se avisa del fallo por la salida (estándar y la ventana de trazas) mostrando una traza con la información correspondiente de dicho fallo.
Actores	Usuario, Administrador
Secuencia normal	1. Se carga el recurso de trazas. 2. Se carga el recurso de configuración. 3. Se crea el gestor de organización y se arranca, asumiendo el control de la ejecución. Al crearse el gestor de comunicaciones se activa el registro RMI. 4. Se muestra la ventana de trazas. 5. Se crean los gestores (agentes, recursos y comunicaciones) . 6. Se arrancan en orden (comunicaciones,agentes,recursos). El gestor de organización espera el éxito del arranque de cada uno de los gestores anteriores antes de pasar al arranque del siguiente. En el caso del gestor de agentes se irán arrancando en orden los agentes descritos en la descripción de organización y se registran en el registro RMI. 7. Estado de monitorización en espera de eventos.
Secuencias alternativas	▪ Si hay fallo al cargar el sistema en alguno de los puntos anteriores, se informa del error. ▪ Una vez mostrada la ventana de trazas se muestran los errores por ella.

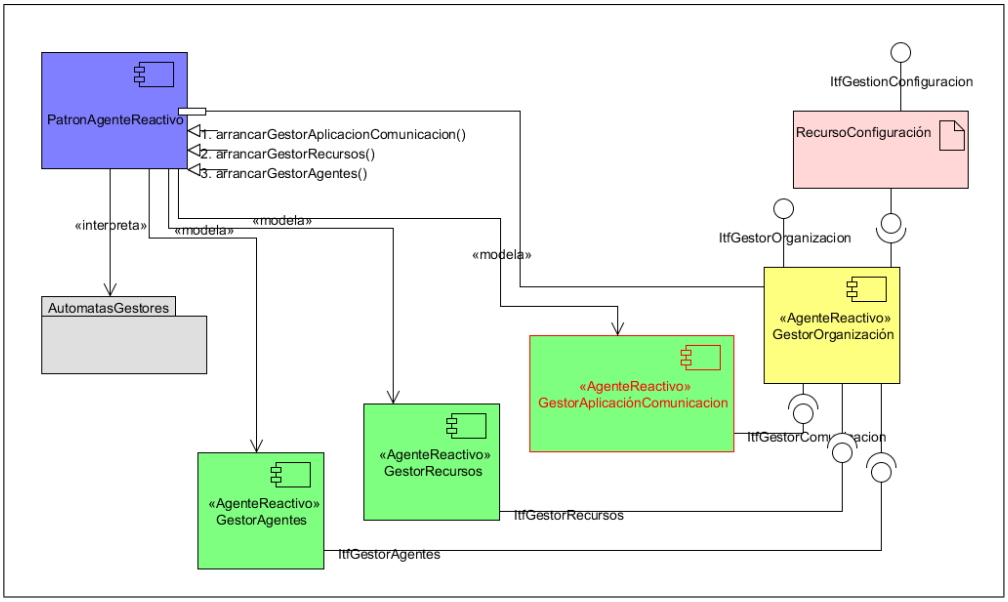


Figura 2: Arranque de los gestores. Diagrama de colaboración.

CU-02	Terminar infraestructura
Objetivo en contexto	Cerrar de forma ordenada las aplicaciones de un nodo y toda la infraestructura.
Precondiciones	La infraestructura tiene que estar arrancada y activa.
Postcondiciones si éxito	Se cierran todos los componentes de la aplicación o parte de la aplicación en caso distribuido (Agentes, Recursos) y toda la infraestructura por debajo.
Postcondiciones si fallo	Se visualiza en el recurso de trazas todos los acontecimientos y errores que hayan podido llevar al fallo del sistema.
Actores	Usuario, Administrador
Secuencia normal	1. El actor activa la terminación haciendo click en el botón "TERMINAR" o el botón de cierre genérico de la aplicación. 2. Le llega al gestor de organización el evento "peticion_terminar_todo" , momento en el que envía al resto de gestores la orden de terminar. 3. Los gestores terminan cada uno de sus componentes gestionados y cierran ordenadamente toda la aplicación y la infraestructura. 4. Se cierra el registro de RMI cuando el gestor de agentes ha finalizado todos sus componentes gestionados. 5. Desaparecen todas las visualizaciones de la pantalla.
Secuencias alternativas	▪ Si hay fallos en el cierre del sistema se trazan dichos errores en la ventana de trazas. Y se fuerza el cierre del sistema.

CU-03	Lanzar aplicación
Objetivo en contexto	Ejecutar la aplicación deseada con la descripción de organización especificada.
Precondiciones	En caso de usar comunicaciones remotas, contar con conexión a la red en la que se encuentran desplegados los nodos a usar en la aplicación.
Postcondiciones si éxito	Se ejecuta la aplicación deseada mostrando en primer lugar la ventana de trazas, en la que se monitorizan las trazas generadas tanto por la aplicación como por la infraestructura, y posteriormente el recurso de visualización (si lo hay) de dicha aplicación.
Postcondiciones si fallo	Se visualiza en el recurso de trazas todos los acontecimientos y errores que hayan podido llevar al fallo del sistema.
Actores	Usuario, Administrador
Secuencia normal	1. El actor lanza la aplicación con la descripción de organización de la aplicación deseada como parámetro. 2. Se arranca el sistema (CU-01). 3. Se ejecuta la aplicación mostrando los componentes de visualización si los hubiera. 4. La aplicación se queda en espera de las acciones del actor en cuestión.
Secuencias alternativas	▪ Si hay fallos se trazan dichos errores en la ventana de trazas.

CU-04	Despliegue de nodo
Objetivo en contexto	Arrancar un nodo para la posterior ejecución de una aplicación en él.
Precondiciones	No debe haber sido desplegado anteriormente un nodo en la máquina en la que se quiera desplegar.
Postcondiciones si éxito	Se arranca el nodo en la máquina deseada y se queda en estado de espera .
Postcondiciones si fallo	Se visualiza en el recurso de trazas todos los acontecimientos y errores que hayan podido llevar al fallo del sistema.
Actores	Administrador
Secuencia normal	1. El actor ejecuta Icaro-D con el parámetro "-nodo". 2. El gestor de nodo arranca un registro RMI y se incluye así mismo en él. 3. La aplicación se queda en espera de una petición de arranque de una aplicación en dicho nodo.
Secuencias alternativas	▪ Si hay fallos se trazan dichos errores en la ventana de trazas.

CU-05	Despliegue de aplicación
Objetivo en contexto	Arrancar una aplicación en los nodos ya desplegados que están a la espera.
Precondiciones	Deben existir los nodos requeridos y estar a la espera, y además contener una copia de Icaro-D y de la descripción de aplicación ejecutada.
Postcondiciones si éxito	Se arranca la aplicación según el despliegue deseado.
Postcondiciones si fallo	Se visualiza en el recurso de trazas todos los acontecimientos y errores que hayan podido llevar al fallo del sistema.
Actores	Administrador
Secuencia normal	1. El actor ejecuta Icaro-D con "-desplegar" y la descripción de organizacióon de la aplicación deseada como parámetros. 2. Se arranca el sistema (CU-01). 3. Se muestra en el nodo del actor una ventana que muestra el progreso del despliegue de la aplicación, en la que se informa del estado de cada nodo y de los posibles errores. 4. Se envía a todos los gestores de nodos remotos una petición de arranque con la descripción de aplicación deseada. 5. Se despliega la aplicación en los nodos especificados. 6. Cada uno de los nodos ejecuta la aplicación según la descripción de organización, y notifican el resultado de dicha operación.
Secuencias alternativas	▪ Si hay fallos se trazan dichos errores en la ventana de trazas de cada uno de los nodos, y se notifica el error al nodo del administrador quien lo visualizará en la ventana de progreso del desplieque.

CU-06	Filtrado de las trazas según el tipo de componente
Objetivo en contexto	Visualizar las trazas de un sólo tipo de componente (gestor, agente, recurso) en vez de todas a la vez.
Precondiciones	El sistema debe estar arrancado y la ventana de trazas debe estar visible y activa.
Postcondiciones si éxito	Se muestran las trazas del tipo de componente elegido en la pestaña correspondiente.
Postcondiciones si fallo	Cierre del sistema.
Actores	Usuario, Administrador
Secuencia normal	1. El actor elige el tipo de componente que quiere filtrar (Gestores, Agentes,Recursos o Errores). 2. Pulsa sobre la pestaña correspondiente a dicho tipo de componente. 3. Visualiza en el panel de esa pestaña solo las trazas relacionadas con el tipo de componente requerido.
Secuencias alternativas	▪ Este proceso también puede ser realizado usando el textField “Filtro” de la ventana de trazas y escribir en él el tipo de componente deseado.

CU-07	Visualización de las trazas de un componente en concreto.
Objetivo en contexto	Visualizar las trazas de un sólo componente.
Precondiciones	El sistema debe estar arrancado y la ventana de trazas debe estar visible y activa.
Postcondiciones si éxito	Se muestran las trazas del componente elegido en la pestaña correspondiente.
Postcondiciones si fallo	Cierre del sistema.
Actores	Usuario, Administrador
Secuencia normal	1. El actor pulsa en la pestaña del tipo correspondiente al componente en cuestión. 2. En esa pestaña aparece un panel lateral con el nombre de los componentes del tipo elegido. El actor pulsa sobre el nombre del componente deseado. 3. Se muestran sólo las trazas del componente elegido.
Secuencias alternativas	▪ Este proceso también puede ser realizado si el actor pulsa dos veces sobre el nombre del componente. El resultado de esta acción es la visualización de una nueva ventana solamente con las trazas del componente elegido.

CU-08	Guardar todas las trazas en un fichero de texto.
Objetivo en contexto	Guardar las trazas en su totalidad en un fichero txt.
Precondiciones	El sistema debe estar arrancado y la ventana de trazas debe estar visible y activa.
Postcondiciones si éxito	Se guardan las trazas en la carpeta log del sistema.
Postcondiciones si fallo	No se guarda ningún fichero y se notifica el fallo.
Actores	Usuario, Administrador
Secuencia normal	1. El actor pulsa en la pestaña principal de la ventana de trazas, en la que se muestran todas las trazas generadas por el sistema. 2. El actor pulsa sobre el menú "Archivo" y al desplegarse pulsa sobre el submenú "Guardar Todas las Trazas" . 3. Se crea el fichero "ficheroTrazas.txt" en la carpeta log del proyecto y se guardan todas las trazas en dicho fichero. 4. Se muestra un diálogo que informa del éxito de la operación.
Secuencias alternativas	4a Se muestra un diálogo que informa del fallo de la operación.

CU-09	Guardar las trazas de un determinado tipo de componente en un fichero de texto.
Objetivo en contexto	Guardar las trazas de un determinado tipo de componente (Gestores, Recursos o Agentes) en un fichero txt.
Precondiciones	El sistema debe estar arrancado y la ventana de trazas debe estar visible y activa.
Postcondiciones si éxito	Se guardan las trazas en la carpeta log del sistema.
Postcondiciones si fallo	No se guarda ningún fichero y se notifica el fallo.
Actores	Usuario, Administrador
Secuencia normal	1. El actor pulsa en la pestaña correspondiente al tipo de componente deseado en la ventana de trazas, en la que se sólo se muestran las trazas generadas el tipo de componente elegido. 2. El actor pulsa sobre el menú "Archivo" y al desplegarse pulsa sobre el submenú "Guardar Trazas de X " donde X es el tipo de componente elegido (Gestores, Recursos o Agentes). 3. Se crea el fichero "ficheroTrazasX.txt" en la carpeta log del proyecto donde X es el tipo de componente elegido (Gestores, Recursos o Agentes) y se guardan todas las trazas en dicho fichero. 4. Se muestra un diálogo que informa del éxito de la operación.
Secuencias alternativas	4a Se muestra un diálogo que informa del fallo de la operación.

Los siguientes casos de uso corresponden a acciones internas a la infraestructura transparentes totalmente al usuario.

CU-010	Envío de mensaje a un agente ya sea remoto o no
Objetivo en contexto	Enviar un mensaje desde un componente origen a un componente cuyo nombre sea conocido, pero no necesariamente su ubicación.
Precondiciones	El mensaje a enviar debe estar bien formado por medio de la clase Mensaje (crearInputBasico).
Postcondiciones si éxito	Se envía el mensaje al destino y se notifica el éxito.
Postcondiciones si fallo	Se visualiza en el recurso de trazas todos los acontecimientos y errores que hayan podido llevar al fallo del sistema.
Actores	Componente
Secuencia normal	1. El componente envía el mensaje a su gestor de comunicaciones por medio de ComunicacionAgentes. 2. Se busca el destino en el nodo local. Si se encuentra, se envía el mensaje al destino local por medio de ComunicacionAgentes. 3. Si el destino es conocido y remoto, se envía por medio de ControlRMI desde el gestor de comunicaciones del nodo origen al gestor de comunicaciones del nodo destino. 4. Si el destino es desconocido y remoto, ControlRMI buscará por medio de buscaInterfazRemota por todos los nodos registrados en el registro RMI hasta encontrar el destino. Cuando lo hace,se envía por medio de ControlRMI desde el gestor de comunicaciones del nodo origen al gestor de comunicaciones del nodo destino. 5. Se notifica el envío del mensaje. 6. El gestor de comunicaciones del nodo destino envía localmente al componente destino el mensaje por medio de su ComunicacionAgentes. 7. Se notifica la recepción del mensaje.
Secuencias alternativas	▪ Si hay fallos se trazan dichos errores en la ventana de trazas de cada uno de los nodos. 4a Si no se encuentra el destino en ninguno de los nodos, se notifica por medio de la ventana de trazas en el nodo origen.

La figura 12 del punto 3.4.4 representa el caso de uso anterior mediante un diagrama de secuencia.

CU-11	Envío de mensaje a un grupo de agentes ya sean remotos o no
Objetivo en contexto	Enviar un mensaje desde un componente origen a un grupo de componentes.
Precondiciones	El mensaje a enviar debe estar bien formado por medio de la clase Mensaje (crearInputParaGrupo) y el grupo destino debe estar especificado en la descripción de aplicación.
Postcondiciones si éxito	Se envía el mensaje al grupo de destino y se notifica el éxito de cada uno de los componentes del grupo.
Postcondiciones si fallo	Se visualiza en el recurso de trazas de cada uno de los nodos todos los acontecimientos y errores que hayan podido llevar al fallo del sistema.
Actores	Componente
Secuencia normal	1. El componente busca en la descripción de aplicación los componentes pertenecientes al grupo destino. 2. Una vez identificados, realiza las acciones del caso de uso anterior CU-06 para cada uno de los componentes del grupo destino.
Secuencias alternativas	▪ Si hay fallos se trazan dichos errores en la ventana de trazas de cada uno de los nodos.

2.2.2. Notas sobre las especificaciones

Característica	Descripción
Idioma	Castellano
Plataforma de desarrollo	Netbeans IDE 6.8
Plataforma Java	JDK 1.6
Dominio	Sistemas Multiagentes
XML	Versión 1.0, encoding UTF-8
Autómatas	Todos las maquinas de estados están implementadas como Autómatas Finitos Deterministas.
Interfaces gráficas	java.swing, java.awt
Modelo	Recursos-Agentes

3. Análisis y Diseño

3.1. Introducción

ICARO-D añade una capa de funcionalidad adicional a ICARO que le permite gestionar las comunicaciones entre agentes de una aplicación a través de RMI. Se podrán realizar fácilmente aplicaciones distribuidas según un modelo de despliegue establecido y todas las comunicaciones serán controladas por la infraestructura.

Se ha utilizado una arquitectura basada en gestores que se encargan de controlar todas las capas de las aplicaciones y un gestor de organización que se encarga de controlar los demás gestores.

3.2. Arquitectura Icaro

La infraestructura tiene la estructura de directorios que puede verse en la figura 3, que debe mantenerse para el correcto funcionamiento de la infraestrcutura, la estructura concreta y las restriccionesde nombrado pueden verse en el apartado "organización del código", en el capítulo de "Implementación". La estructura de paquetes del código fuente contenido de la carpeta src se muestra desplegado en la figura 4 . Refleja el modelo de diseño, que está inspirado en una organización cuyo objetivo consiste en implementar la funcionalidad de la aplicación, como se ha explicado en el apartado 1.2 ("Que es Icaro").

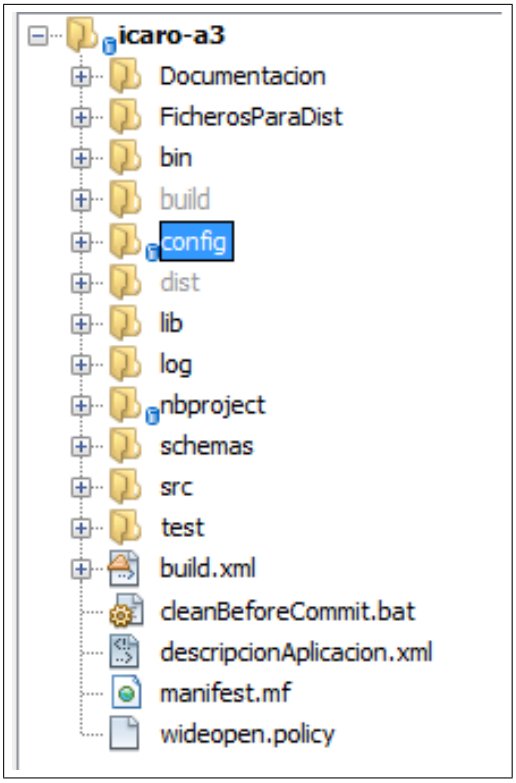


Figura 3: Estructura de directorios Icaro

El código está estructurado en dos capas:

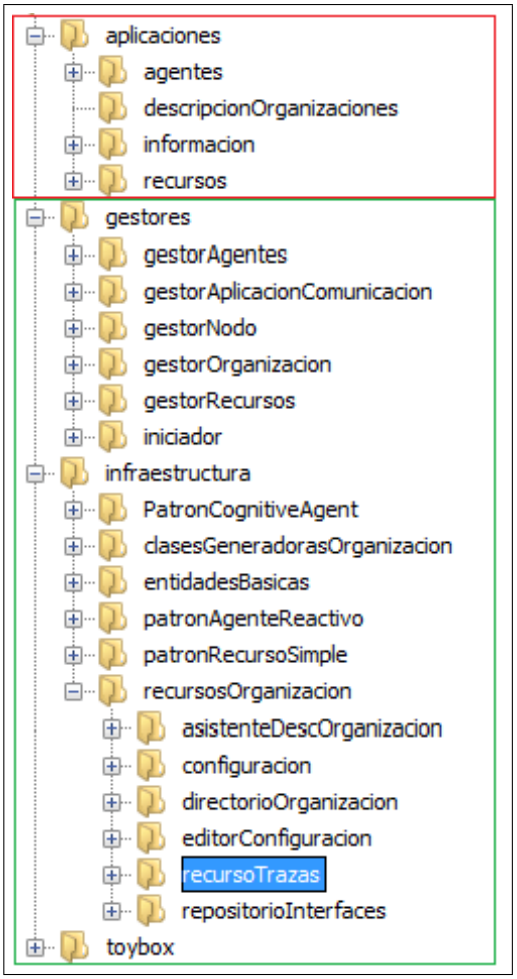


Figura 4: Estructura del directorio src de Icaro

- **La capa de aplicación**: Contiene los elementos –código y ficheros- que modelan las aplicaciones como organizaciones de agentes y recursos. La capa de aplicaciones contiene los siguientes tipos de paquetes:

  - **Agentes**: Contienen la descripción del comportamiento de los agentes que pueden utilizarse en las aplicaciones. Un agente puede tener varios comportamientos definidos en los paquetes correspondientes.
  - **Información**: contiene clases que modelan diferentes entidades de los dominios de aplicación. El paquete esta estructurado en tres paquetes para almacenar clases de dominio, objetivos y tareas.
  - **Recursos**: Contienen el código de los recursos. Hay dos tipos de recursos comunes a gran parte de las aplicaciones: los recursos de visualización o de forma más genérica de interfaces gráficas y los recurso de persistencia. Pueden añadirse otros recursos según el tipo de aplicaciones.
  - **Descripciones**: Contiene los documentos que describen las aplicaciones como organizaciones de agentes, recursos y entidades computacionales comunes.

- **La capa del marco de trabajo**: Contiene la infraestructura computacional necesaria para que el marco de trabajo funcione. Esta formada por los **gestores**

- agentes con un comportamiento ya definido - y por la **infraestructura** que contiene componentes y clases para la implementación de las organizaciones. La figura 5 representa las dos capas y las dependencias entre los componentes de cada capa. Los paquetes de la capa de marco de trabajo son:

- **Gestores**: Contiene agentes predefinidos para gestionar los componentes que implementan las organizaciones o aplicaciones construidas por los usuarios. Se proporcionan tres agentes gestores: Gestor de Organización (GO) Gestor de Agentes (GA) y Gestor de Recursos (GR).
- **Infraestructura**: De la organización, donde se encuentran los patrones que sirven para generar los agentes y los recursos de las aplicaciones, recursos de la organización y componentes básicos que hacen posible la implementación de los agentes y de los recursos.

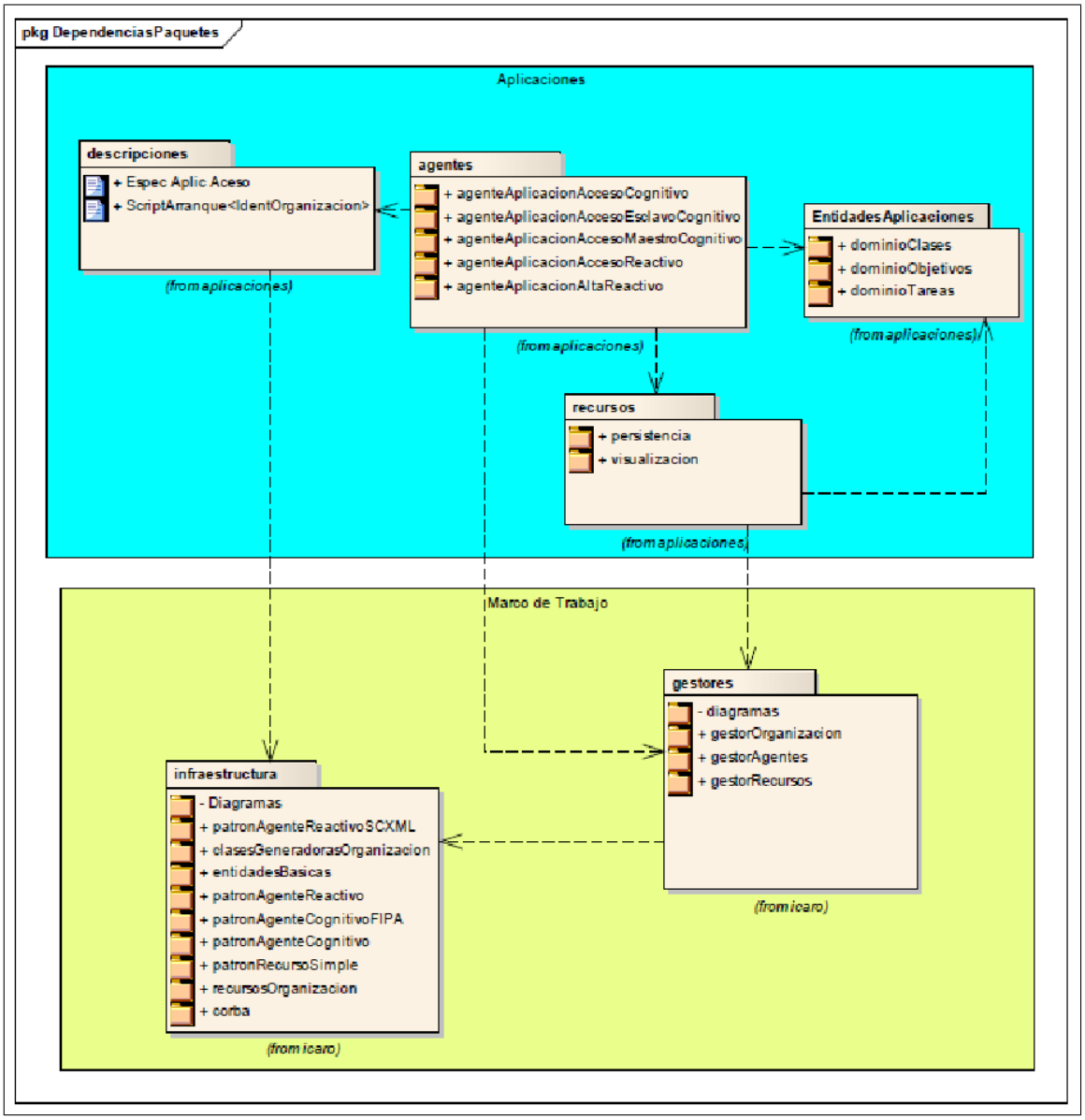


Figura 5: Estructura de Icaro

3.3. Arquitectura RMI: Java Remote Method Invocation

Es un mecanismo ofrecido por java para invocar un método de manera remota. Forma parte del entorno estándar de ejecución de Java y provee de un mecanismo simple para la comunicación de servidores en aplicaciones distribuidas basadas exclusivamente en Java. Proporciona paso de objetos por referencia , recolección de basura distribuida y paso de tipos arbitrarios.

Por medio de RMI, un programa Java puede exportar un objeto, lo que significa que éste queda accesible a través de la red y el programa permanece a la espera de peticiones en un puerto TCP. A partir de este momento, un cliente puede conectarse e invocar los métodos proporcionados por el objeto. La invocación se compone de los siguientes pasos:

- Encapsulado (marshalling) de los parámetros (utilizando la funcionalidad de serialización de Java).
- Invocación del método (del cliente sobre el servidor). El invocador se queda esperando una respuesta.
- Al terminar la ejecución, el servidor serializa el valor de retorno (si lo hay) y lo envía al cliente.
- El código cliente recibe la respuesta y continúa como si la invocación hubiera sido local.

3.3.1. Modelo de Capas RMI

La arquitectura RMI puede implementa un modelo de cuatro capas, que puede verse en la figura 6:

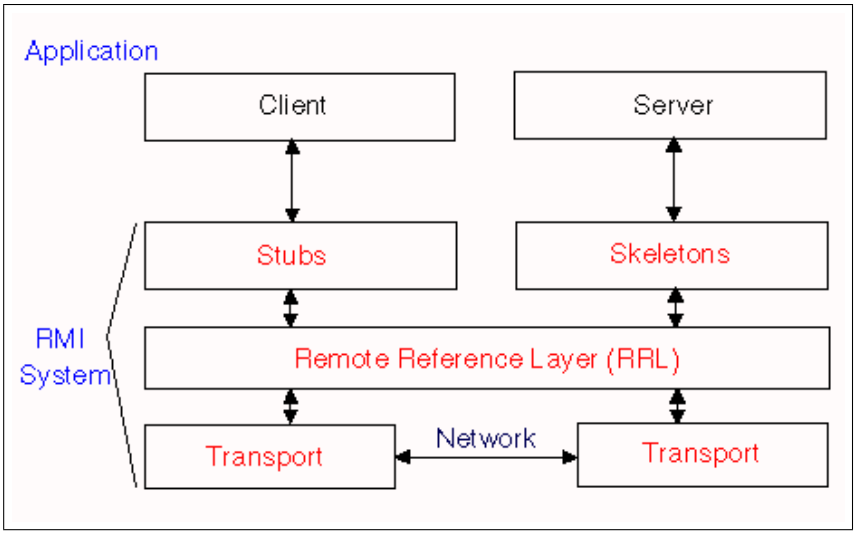


Figura 6: Arquitectura de capas RMI

1. La primera capa es la de aplicación y se corresponde con la implementación real de las aplicaciones cliente y servidor. Aquí tienen lugar las llamadas a alto nivel para acceder y exportar objetos remotos. Cualquier aplicación que quiera que sus métodos estén disponibles para su acceso por clientes remotos debe declarar dichos métodos en una interfaz que extienda java.rmi.Remote. Dicha interfaz se usa básicamente para "marcar" un objeto como remotamente accesible. Una vez que los métodos han sido implementados, el objeto debe ser exportado. Esto puede hacerse de forma implícita si el objeto extiende la clase UnicastRemoteObject (paquete java.rmi.server), o puede hacerse de forma explícita con una llamada al método exportObject() del mismo paquete.

2. La capa 2 es la capa proxy, o capa stub-skeleton. Esta capa es la que interactúa directamente con la capa de aplicación. Todas las llamadas a objetos remotos y acciones junto con sus parámetros y retorno de objetos tienen lugar en esta capa.

3. La capa 3 es la de referencia remota, y es responsable del manejo de la parte semántica de las invocaciones remotas. También es responsable de la gestión de la replicación de objetos y realización de tareas específicas de la implementación con los objetos remotos, como el establecimiento de las persistencias semánticas y estrategias adecuadas para la recuperación de conexiones perdidas. En esta capa se espera una conexión de tipo stream (stream-oriented connection) desde la capa de transporte.

4. La capa 4 es la de transporte. Es la responsable de realizar las conexiones necesarias y manejo del transporte de los datos de una máquina a otra. El protocolo de transporte subyacente para RMI es JRMP (Java Remote Method Protocol).

Pueden verse ejemplos de cómo implementar el uso de RMI en una aplicación en el apartado 4.2, "utilizando Java RMI", en el capítulo de "Implementación". En dicho apartado se muestran ejemplos comentados sobre cómo realizar las tareas comunes asociadas a RMI, como crear el registro o acceder a los objetos remotos.

3.4. Arquitectura para la distribución

3.4.1. Introducción

La ampliación de la arquitectura de ICARO para hacer que las aplicaciones puedan ser distribuidas sigue las mismas pautas que ICARO normal. Se ha añadido un nuevo gestor capaz de gestionar la redirección de los mensajes (gestorAplicacionComunicacion) entre nodos, con el objetivo de simplificar las comunicaciones entre agentes y entre nodos. Este gestor trabaja con objetos de la clase Mensaje detallada más adelante. Estos objetos son los que se envían entre nodos para las comunicaciones. Estamos llamando nodo a una maquina en la que se ejecuta una o más aplicaciones Icaro, puede verse la estructura de un nodo en la figura 8, que obviamente coincide con la arquitectura Icaro. También es importante señalar que cada nodo despliega una infraestructura Icaro completa, que se encargará de gestionar el nodo con normalidad, de la misma manera que si fuera centralizado.

También se han simplificado las comunicaciones locales a través de la clase ComunicacionAgentes y se ha encapsulado toda la funcionalidad de RMI con la clase ControlRMI. Puede verse un diagrama de la parte de de comunicaciones en la figura 8.

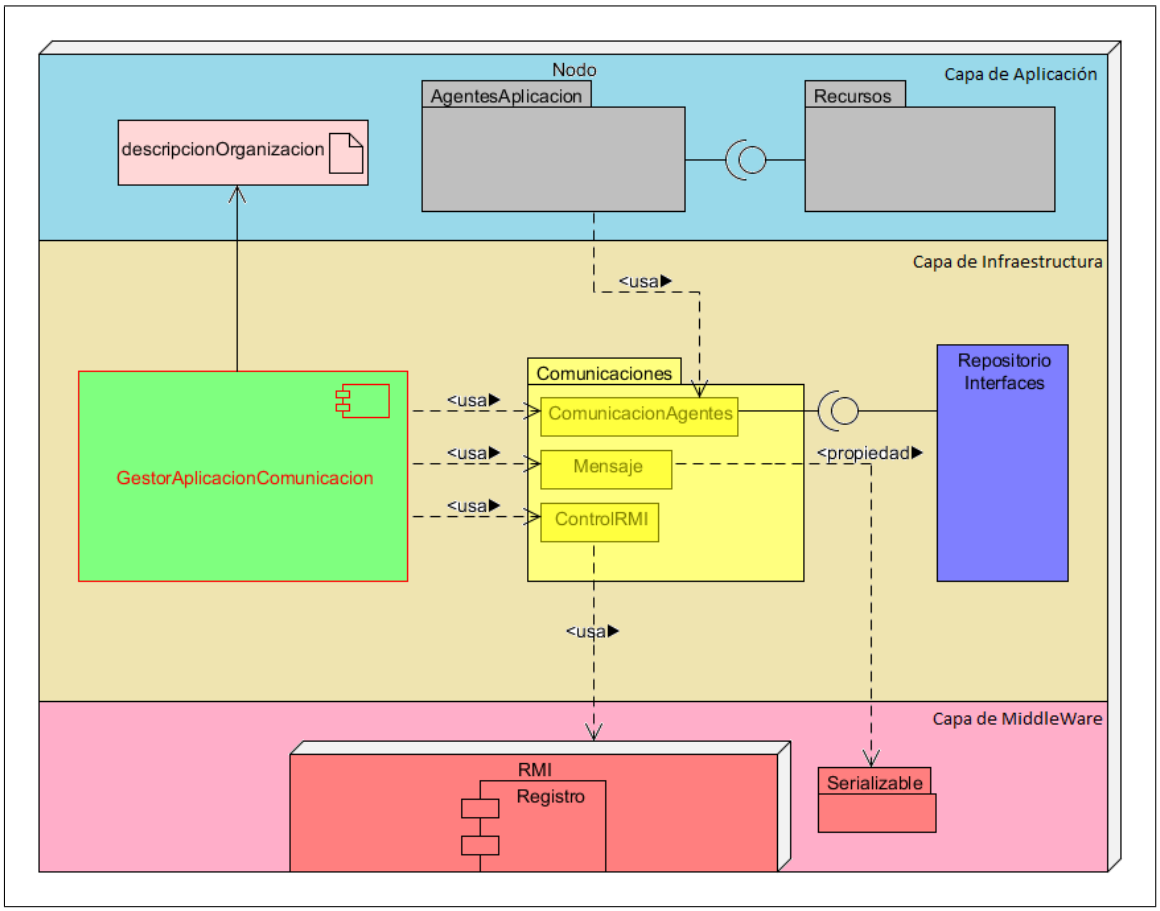


Figura 7: Estructura de un nodo en Icaro

3.4.2. Comunicación Agentes y Directorio

Ubicación: icaro.infraestructura.entidadesBasicas. La clase comunicaciónAgentes sirve para comunicar a agentes de un nodo(locales). El envío de inputs locales pasa a través de esta clase que busca al destinatario en el repositorio de interfaces y le envía el input que se pasa. En esta funcionalidad están incluidas todas las comunicaciones locales tanto entra agentes, entre gestores o entre agentes y gestores. Aunque esta clase puede utilizarse libremente, se recomienda que se deje su uso al gestor de agentes, empleando el método "enviarMensaje", presente el patrón de agente reactivo.

En cuanto a la clase Directorio, su funcionalidad es sencillamente agrupar en una sola clase el acceso a recursos de la organización, como el repositorio, las trazas y la configuración Icaro. Esta clase se presenta como una solución intermedia a la reestructuración de los recursos de organización. Para esto dispone de métodos que permitan acceder de forma sencilla a los servicios más frecuentes que ofrecen estos recursos, además de permitir el acceso a los propios recursos desde aqui.

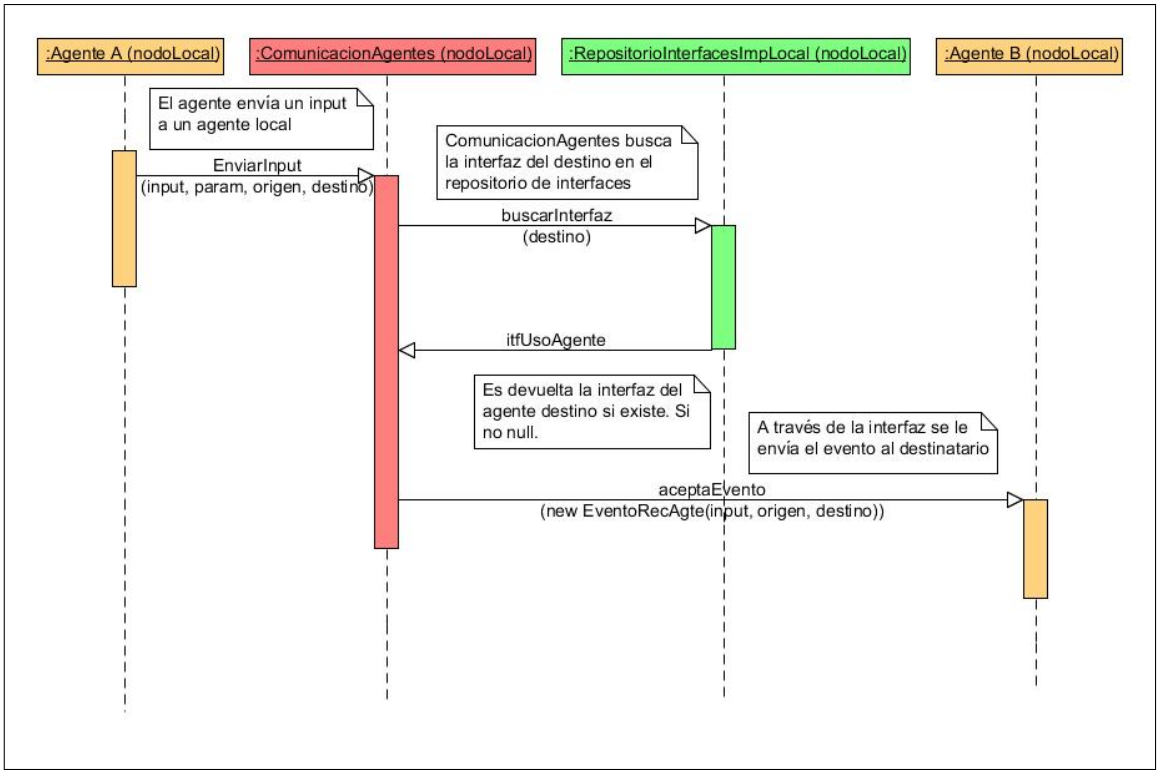


Figura 8: Diagrama de envio de un mensaje local

3.4.3. ControlRMI

Ubicación: icaro.infraestructura.entidadesBasicas. Esta clase está para encapsular toda la funcionalidad de RMI, de forma que se instancia cada vez que se quiera utilizar RMI para relizar las siguientes acciones:

- Asignar un puerto de comunicaciones.
- Inicializar RMI localizando el registro de RMI en el puerto indicado.
- Exportar objetos al registro RMI.
- Eliminar objetos del registro RMI.
- Obtener la dirección IP del nodo local.
- Enviar inputs remotos.

Cuando se inicia RMI, se crea un icono en la barra de estado para poder visualizar todo lo que haya en el registro de RMI, como puede verse en la figura 9. Al igual que la clase ComunicacionAgentes, se recomienda no emplear directamente esta clase, empleando en su lugar el gestor de comunicaciones.

Abajo a la derecha se puede observar el icono en la barra de estado. Si se hace click en alguno de los elementos listados, se saca del registro de RMI. Cualquier agente puede utilizar ControlRMI para enviar inputs remotos, pero la arquitectura que se detalla más arriba está pensada para que todo pase a través del gestor de comunicaciones. Esto

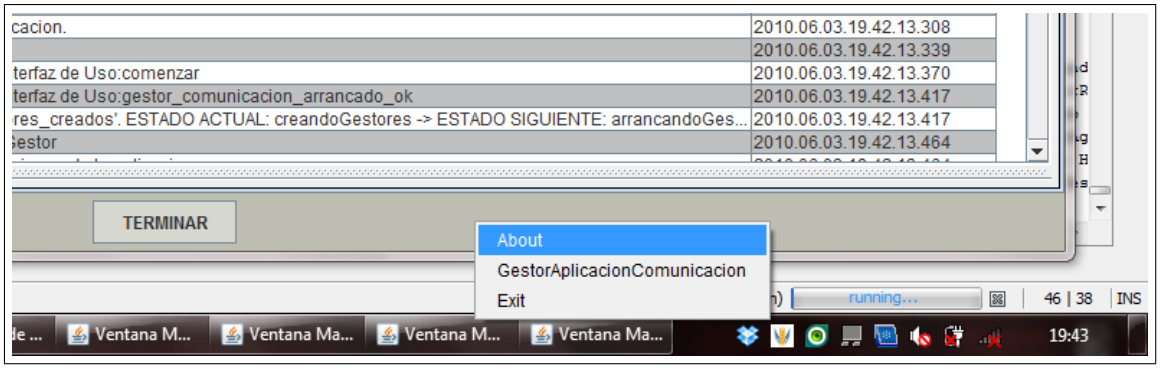


Figura 9: Icono de RMI

está hecho así para dejar libertad a los desarrolladores por si quieren enviar inputs remotos sin tener que utilizar los gestores de comunicaciones.

En el diagrama de secuencia que viene a continuación se muestra el envío de un input remoto de un agente a otro. Normalmente estos agentes serán los gestores de comunicación. Notese también que por lo menos el agente destinatario ha de estar localizado en el registro de RMI para recibir el mensaje.

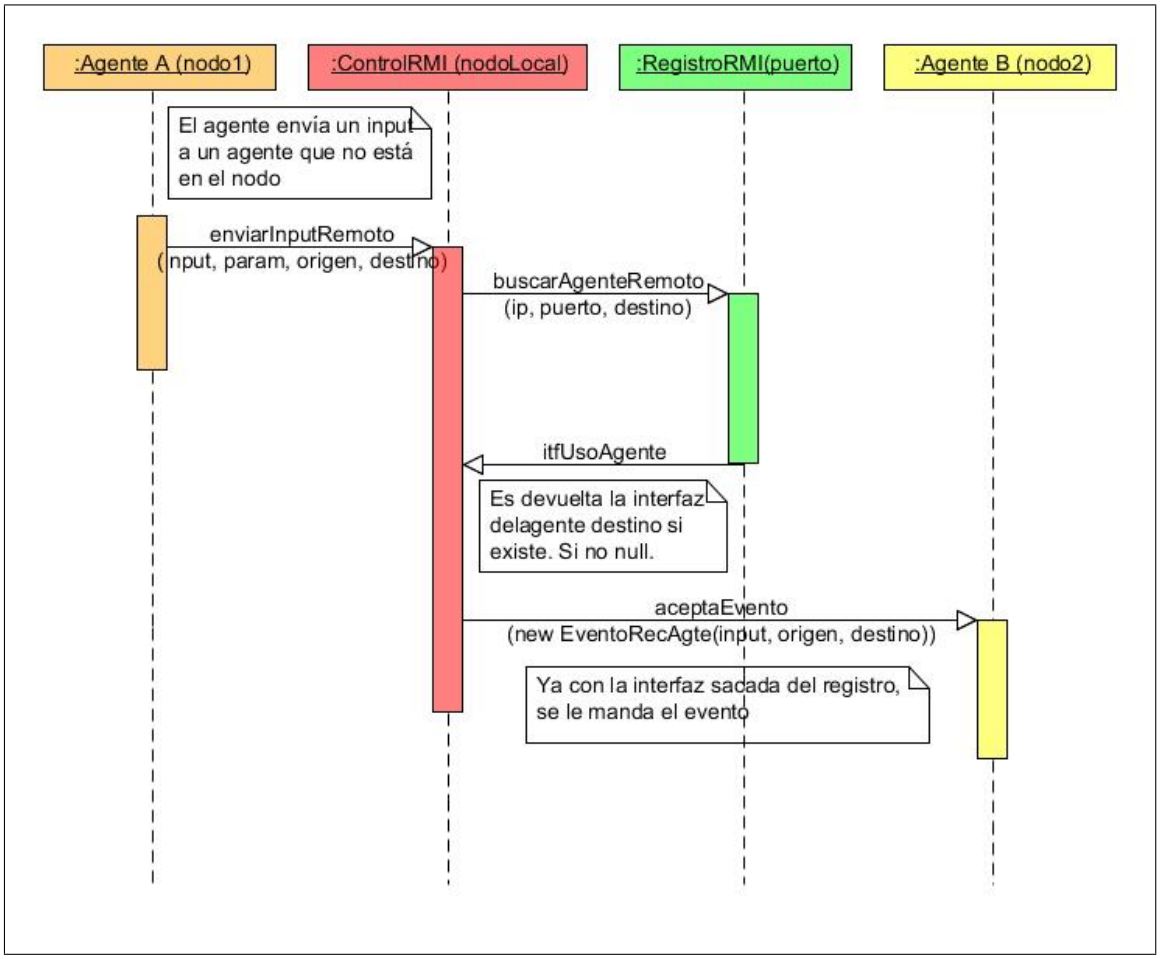


Figura 10: Envio de un input mediante control RMI

3.4.4. Gestor de Comunicaciones

Ubicación: icaro.gestores.gestorAplicacionComunicacion.comportamiento. El gestor de comunicaciones sirve para controlar el flujo de mensajes que circulan entre los nodos. Principalmente esta clase implementada como un agente reactivo trabaja con objetos de tipo Mensaje, en los cuales entraremos en detalle más adelante. Se recomienda utilizar esta clase de forma exclusiva para realizar las comunicaciones, ya que simplifica el código y permite la redirección de mensajes de forma automática, liberando al desarrollador de saber si el destinatario es local o remoto. Además simplifica enormemente la migración de una aplicación del modelo centralizado al distribuido, como puede verse en el capítulo de validación, en el apartado en el que se migra la aplicación masterIA a distribuido.

Como los otros gestores, este también utiliza el fichero de descripción de organización, que es el fichero utilizado por Icaro para declarar la estructura de la aplicación, localizado en la carpeta "config" de la infraestructura. Para más información acerca de cómo editar este archivo, consultar el manual de uso de Icaro. En concreto el gestor de comunicaciones utilizará el contenido de este archivo, contenido en tiempo de ejecución en el recurso "configuracion" de Icaro, para consultar en que nodo se encuentra el agente destinatario, y donde puede encontrar dicho nodo. Toda la información del despliegue de la aplicación viene escrita este fichero. Viene información de nodos de la aplicación, direcciones IP de cada nodo, agentes y recursos desplegados en cada nodo y otras propiedades como la pertenencia a un grupo.

La arquitectura que hemos desarrollado está pensada para que todas las comunicaciones pasen a través de este gestor. Tanto comunicación local como remota. La idea es que cuando un agente quiera enviar un input a otro, se lo pasa a su gestor de comunicaciones local y este con la información del mensaje ya es capaz de redireccionarlo a su destinatario, ya esté en el mismo nodo o en otro. Esto es lo que hace que la infraestructura ICARO-D haga que sea fácil de utilizar para desarrollar aplicaciones distribuidas, este nivel de transparencia para el programador, le permite poder hacer aplicaciones distribuidas sin tener que preocuparse de las comunicaciones entre agentes. Puede verse el autómata del gestor de comunicaciones en la figura 11. En este diagrama puede verse cómo despues del arranque el gestor, éste simplemente espera peticiones, aplicando en cada caso el método apropiado para redirigir el mensaje, quedandose en estado de espera únicamente en los casos en los que el mensaje requiera una respuesta por parte del destinatario y bloqueandose mientras atiende una petición, permitiendo que los mensajes se encolen y se vayan atendiendo según disponibilidad. Esta caracteristica de bloqueo durante una operación es la razón por la que se implementa esta funcionalidad como un gestor en lugar de como un recurso, el encolado de mensajes ya está implementado en el patron de agentes, y además la existencia de un autómata permite que más adelante se pueda refinar el funcionamiento de la redirección de mensajes de forma sencilla.

En esta maquina de estados se representa el gestor con todas sus acciones y posibles inputs. El método que hay que usar para el redireccionamiento de inputs es el de enviarMensaje(). Hay otros tantos métodos que también sirven, pero trabajan con otro tipo de información, utilizan una clase mensaje, que fue el una primera versión para realizar la comunicación, simplemente ese código se ha dejado por si sirviera a

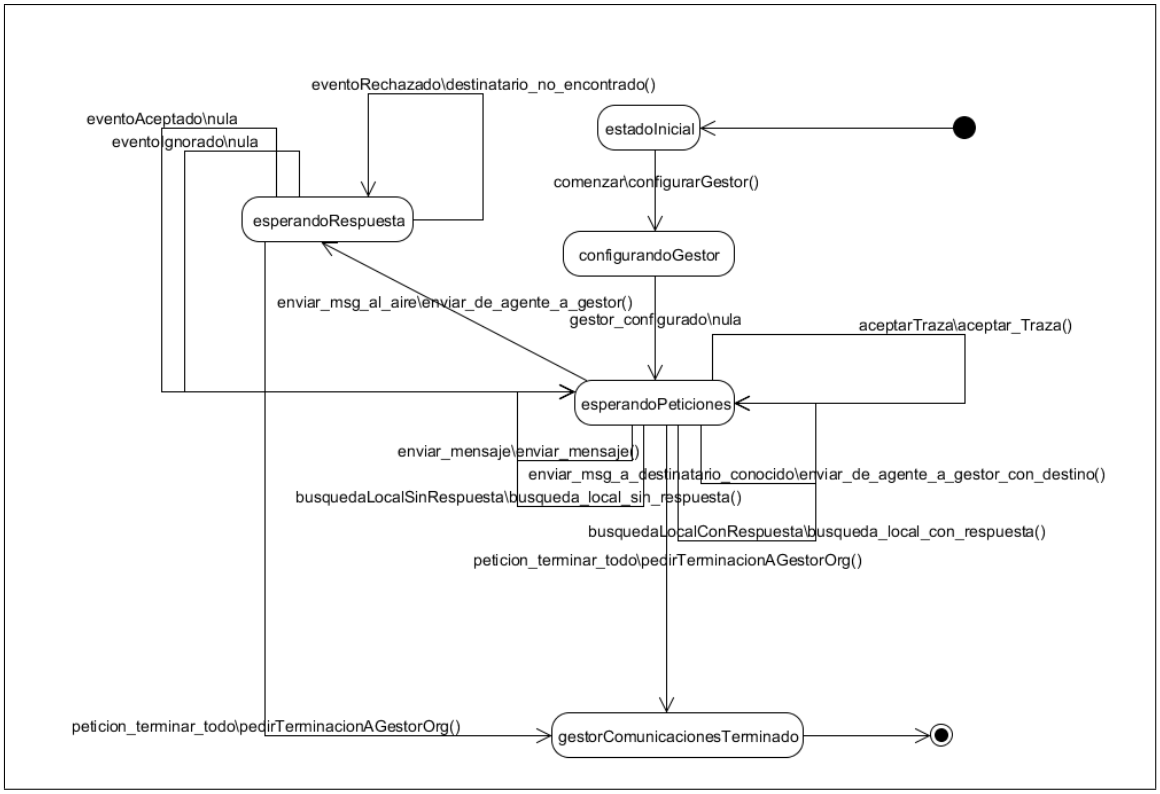


Figura 11: Maquina de estados del gestor de comunicaciones

algún programador en el futuro, pero nosotros utilizamos mensajes para el traspaso de información y de eso se encarga el método de enviarMensaje().

En caso de necesitar comunicar recursos, será necesario (o mejor dicho, está recomendado) crear un agente que lo gestione, quedando como única alternativa a esto que el desarrollador cree el recurso encargándose él de que el objeto que se instancie sea remoto, como puede verse en el apartado "Arquitectura RMI". Esto se debe al modelo de uso del propio RMI, en el caso de un recurso, su interfaz no tiene porque seguir un modelo estandarizado, como si lo hacen los agentes, lo que imposibilita integrar su adaptación remota dentro de la infraestructura, por lo que debe hacerse "a mano" dentro de la aplicación que se esté desarrollando. Tener un agente controlando a un recurso nos ofrece a cambio un mejor control sobre el recurso, pudiendo gestionar, por ejemplo, errores en el mismo, de forma automática.

El proceso habitual que se sigue para enviar un mensaje es el siguiente:

- Un agente crea el mensaje que quiere enviar utilizando alguna de las constructoras de la clase Mensaje
- Con el mensaje listo, solicita envialo al gestor de comunicaciones, utilizando como se recomienda, el metodo *enviarMensaje(Mensaje m)*.
- El método *enviarMensaje(Mensaje m)* utiliza la clase comunicacionAgentes para hacer llegar al gestor de comunicaciones del nodo el mensaje que se quiere enviar.
- El gestor de comunicaciones recibe un nuevo mensaje y evalúa que debe hacer con el, su destinatario y su contenido.

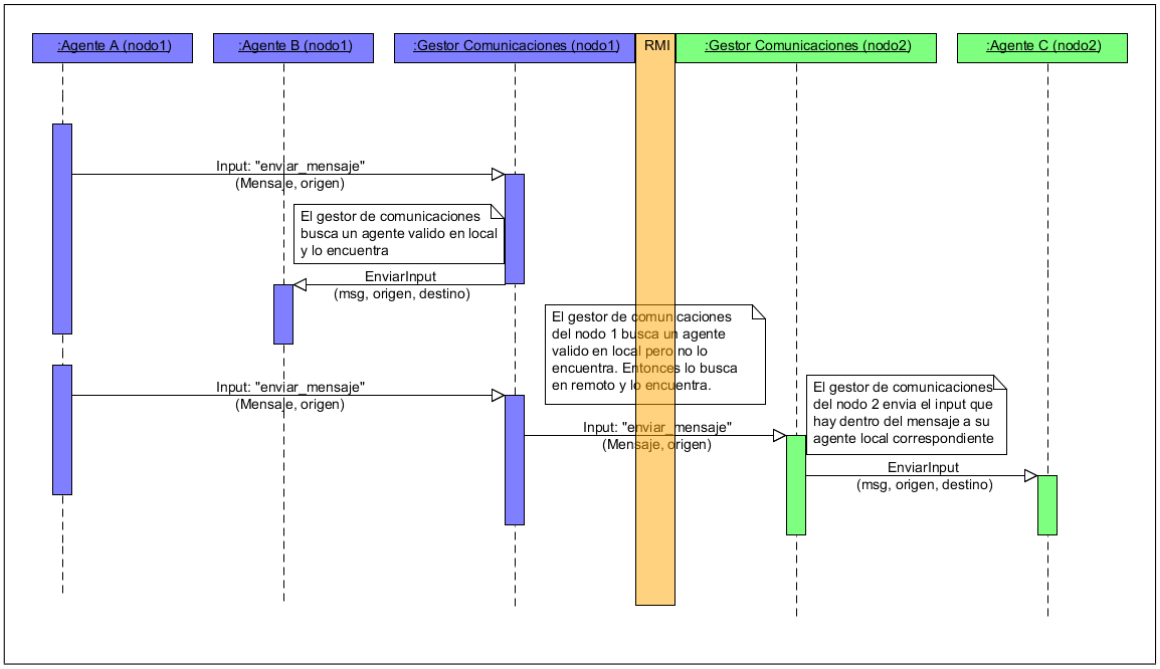


Figura 12: Diagrama de secuencia para el envio de un Mensaje

- Tomada la decisión sobre como redirigir el mensaje, el gestor utiliza, o bien comunicacionAgentes, si el destinatario es local, o bien controlRMI, si es remoto, para enviar el mensaje al destinatario correspondiente, quedando libre para atender otras peticiones

3.4.5. Mensajes

Ubicación: icaro.gestores.gestorAplicacionComunicacion.comportamiento. Los objetos creados a partir de la clase Mensaje son los que nos sirven para el envío de información remota. Cuando varios nodos de una aplicación están conectados en red y correctamente configurados en los descriptores de organización, agentes de diferentes nodos(gestores de comunicación) se comunican mediante el envío de objetos de la clase Mensaje. Estos son lo que circulan por RMI. Los mensajes a parte de contener el input normal, contienen una seria de datos adicionales, como direcciones IP o fecha de creación, esto es para facilitar todo tipo de datos a la hora de distribuir el mensaje.

3.4.6. Gestor de Nodo

Ubicación: icaro.gestores.gestorNodo.comportamiento: Este gestor se implementa para el caso en que queramos tener algún nodo en espera, para poder lanzar la aplicación completa desde otro nodo, en lugar de lanzar manualmente cada nodo en el momento. Aún seguira siendo necesario lanzar manualmente el gestor de nodo, pero no será necesario indicarle qué debe ejecutar, puesto que la petición llega desde otro lado.

3.5. Interfaz del usuario, las trazas

La interfaz de usuario de Icaro es el recurso de trazas. Parte del proyecto ha sido el desarrollo e implementación de un nuevo recurso de trazas que mejorara el ya existente. Para que el lector aprecie las diferencias adjuntamos en primer lugar una imagen con la antigua visualización de este recurso, que puede verse en la figura 13.

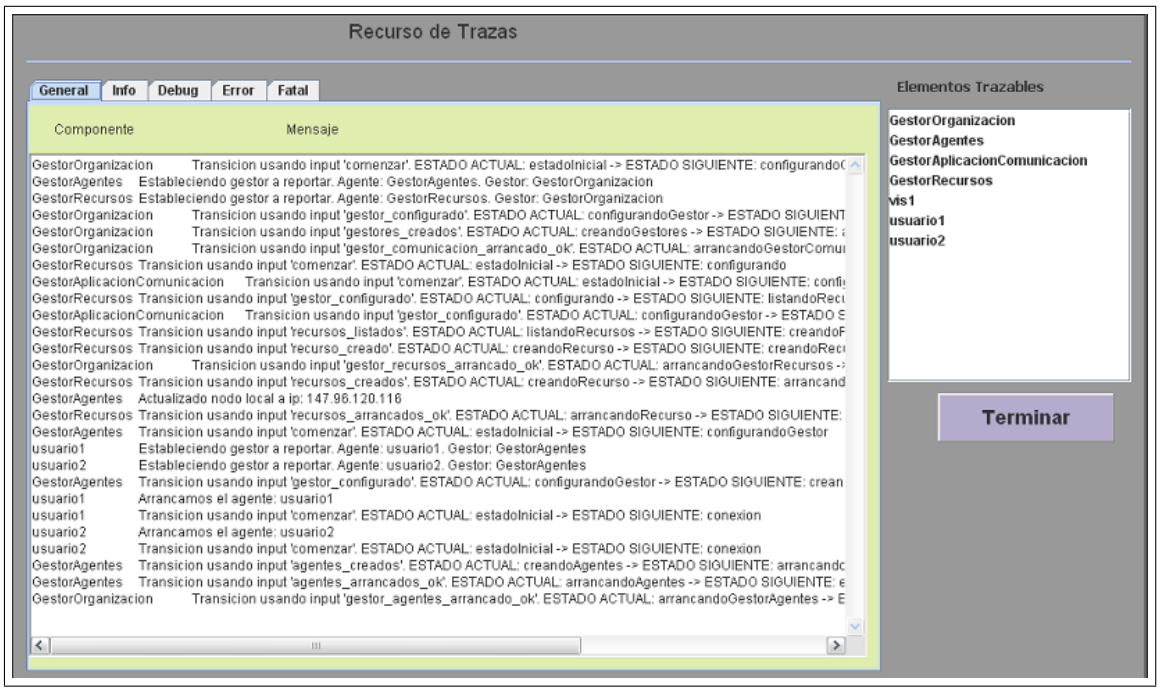


Figura 13: Visualizacion de trazas original

El grupo del proyecto tuvo que trabajar con estas trazas para un proyecto anterior y se detectaron una serie de debilidades que se han querido subsanar. Por ello, se decidió desarrollar un nuevo recurso de trazas que a continuación se describirá.

El gran defecto de las antiguas trazas es el excesivo tiempo que tarda en generar y mostrar las trazas. En este aspecto, el nuevo recurso es sensiblemente mejor. Otro aspecto a mejorar era el formato de visualización de las trazas. Como se puede observar en la figura 13 hay un marcado desorden. Además, se agrupan las trazas de todos los componentes (gestores,agentes,recursos) en la misma ventana. Todo esto hace de la tarea de seguir una simple línea de trazas de una aplicación más complicado de lo que debiera.

Como tareas añadidas, se han incluido funcionalidades tales como guardar las trazas en ficheros .txt y acelerar el proceso de terminación de una aplicación por medio del botón terminar. Con estas bases, se dispuso a desarrollar el nuevo recurso partiendo de cero, y el resultado ha sido el que se puede observar en las figuras 14 y 15. En el apartado de “Implementación de las trazas” en el punto 4, se describen en detalle todos los nuevos componentes y funcionalidades.

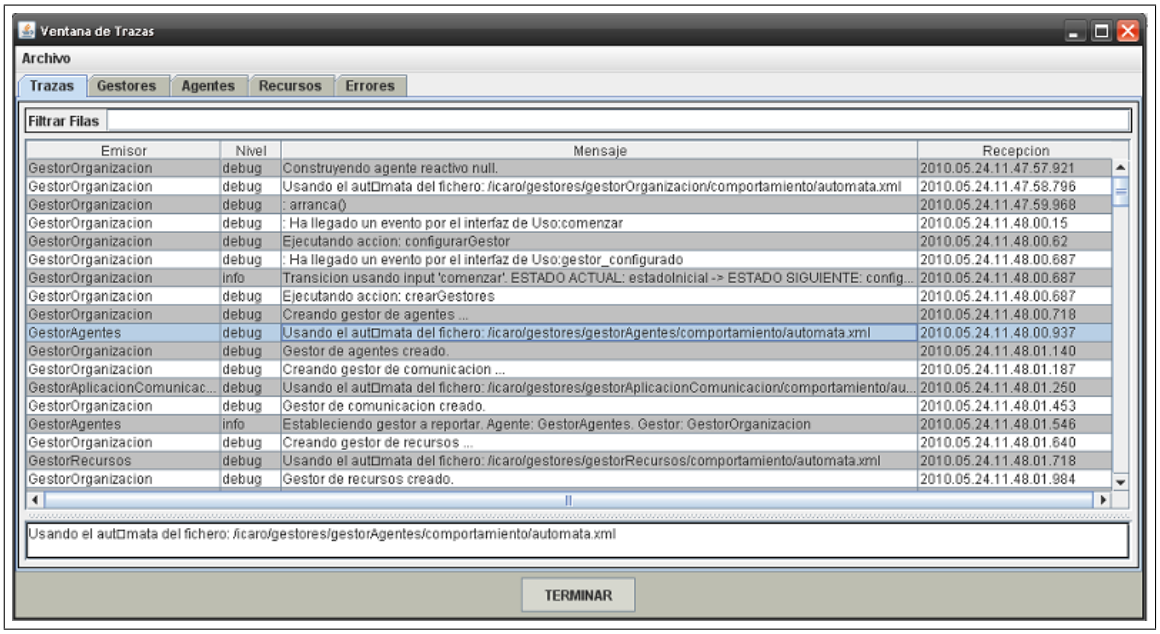


Figura 14: Visualizacion de trazas nueva, panel principal

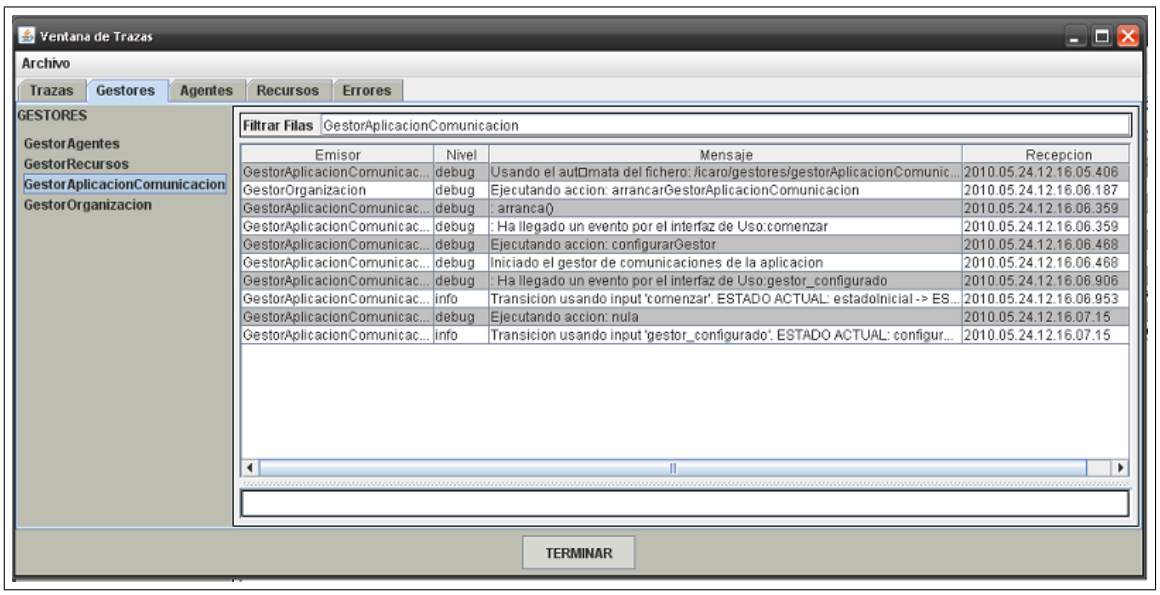


Figura 15: Visualizacion de trazas nueva, panel especificol

4. Implementacion

En este apartado se tratará la implementación específica de los principales componentes que se han desarrollado, no se incluyen todas las modificaciones que se han hecho en los patrones dado que en su mayoria se trata de cambios pequeños, en otro caso se tratará en el punto "Detalles de Implementación", mas adelante, se recomienda consultar previamente el manual de Icaro-T, ya que el contenido de este manual debe tratarse en muchos aspectos, como una ampliación del anterior, y no como un sustituto del mismo.

4.1. Organización del código

La organización del código completa puede verse en la documentación general de Icaro-T, que puede encontrarse en la web:

http://icaro.morfeo-project.org/documentacion/lng/es.

También puede encontrarse una descripción de como se han implementado los componentes principales diseñados para establecer la comunicación en el apartado *Implementación de los componentes diseñados*, más adelante en este mismo capítulo.

La organización general del código el la siguiente

- **Aplicaciones**: Contiene las aplicaciones basadas en la infraestructura. Nótese que no esta divido en aplicaciones, en el caso en que el mismo proyecto contenga varias. Se incluyen aqui los ejemplos desarrollados para la fase de validación.
  - **Agentes**: Contiene la implementación de los agentes, cada agente debe estar contenido en un paquete llamado, obligatoriamente, "agenteAplicacion" + <nombreAgente> + "Reactivo", por ejemplo "agenteAplicacionAccesoReactivo". La implementación de sus acciones semánticas y su autómata, en el caso de los agentes reactivos deberá estar en un paquete, llamado por defecto "comportamiento". En cualquier caso se puede utilizar un nombre distinto si se especifica en la descripción de la organización, aunque no se recomienda. Consultar el manual de uso de Icaro para más información.
  - **Recursos**: Contiene la implementación de los recursos, al contrario que los agentes el nombre del paquete en que debe ir cada recurso no es fijo, pero se recomienda que su contenido sea una interfaz, cuyo nombre debe ser "ItfUso" + <nombreRecurso>, y un paquete "imp", que contenga la implementación del interfaz, cuyo nombre debe ser "claseGeneradora" + <nombreRecurso>.
  - **Informacion**: Contiene las clases de dominio para la aplicación, es decir, que información se envía.
- **Gestores**: Contiene los agentes que se encargan de gestionar la infraestructura
  - **GestorOrganizacion**: Se encarga de lanzar y gestionar a los demás gestores, y realiza el arranque y parada de la aplicación.

  - **GestorAgentes**: Gestiona los agentes de la aplicación
  - **GestoresRecursos**: Gestiona los recursos de la aplicación
  - **GestorAplicacionComunicacion**: Este es el gestor implementado para simplificar las comunicaciones distribuidas entre nodos. Contiene además la implementación de la clase Mensaje.
  - **GestorNodo**: Se encarga de lanzar instancias de aplicaciones por petición remota. Es el único que, en principio, puede funcionar de manera independiente, a la espera de peticiones de arranque remotas.

- **Infraestructura**: Contiene los patrones y clases de apoyo de la infraestructura. Mencionar sólo dos paquetes importantes:
  - **entidadesBasicas**: Contiene las clases ComunicacionAgentes , ControlRMI y Directorio, entre otras.
  - **clasesGeneradorasOrganizacion**: Contiene las clases que arrancan la aplicación, es decir preparan las trazas, leen la descripción de organización y arrancan el gestor de organización.

A continuación expondremos brevemente la forma en que se ha utilizado Java RMI.

4.2. Uso de Java RMI

El mecanismo RMI de Java ha sido el medio elegido para implementar las comunicaciones entre distintos agentes, este mecanismo permite tratar con objetos Java que estan ejecutandose en un entorno remote como si fueran locales, ocultando gran parte del trabajo tedioso que supone establecer conexiones TCP. Java RMI ha sido elegido por encima de otras alternativas, como CORBA o ICE por los siguientes motivos:

- RMI esta integrado en Java, por lo que no son necesarias infraestructuras a parte, lo que evidentemente facilita su integración con el resto de codigo Java, usado en la implementación de Icaro-D
- No se requiere software externo. Aunque versiones anteriores de Java RMI necesitaban que los usuarios ejecutaran programas ajenos a la aplicación, como eran rmic o rmiregistry, Estos se encuentran ahora disponibles como parte de la API de RMI, y por lo tanto pueden controlarse programaticamente, lo que hace mas compacta la implementación.
- No se requieren skeletons ni stubs: Estos dos conceptos, muy utilizados en CORBA, ICE, y versiones anteriores de RMI requerian de un esfuerzo adicional por parte de los programadores e incrementaban el numero de clases Java requeridas. Java RMI realiza este trabajo de forma transparente, sin necesidad de intervención explicita por parte del usuario, ni tampoco precompilarlos para poder usarlos.
- El envio de objetos Java con RMI es muy sencillo, solo es necesario asegurarse de que los mismos sean serializables, lo que permite que la ejecución de métodos de

objetos remotos mediante reflexión (el mecanismo usado por los agentes reactivos de Icaro para realizar acciones).

A pesar de estas ventajas, existe una dificultad con Java RMI que hace que valga la pena dedicar un espacio aqui a explicar su funcionamiento, y es que prácticamente no existe documentación actualizada acerca del mismo, haciendo referencia la mayor parte de la documentación encontrada a procedimientos obsoletos para emplear RMI, incluida gran parte de la documentación que puede encontrarse en la propia página de Sun. Por esta razón, vamos a explicar brevemente como debe usarse RMI con un ejemplo sencillo:

Arquitectura RMI

La arquitectura básica usada en RMI puede verse en la figura 16, teniendo en cuenta que el propio sistema RMI gestionará los skeletons y stubs, por lo que podremos considerar RMI como una caja negra. Su funcionamiento depende de dos componentes basicos: los clientes y el registro RMI. El registro RMI es donde se van a encontrar aquellos objectos Java que queramos que estén disponibles para otros objetos, locales o remotos. Este registro se encargará de gestionar las conexiones y de mantener los enlaces entre el registro de nombres (NameRegistry) y los objetos que se registran en el, su propósito a efectos practicos es establecer un puente para poder acceder a los métodos de los objetos remotos.

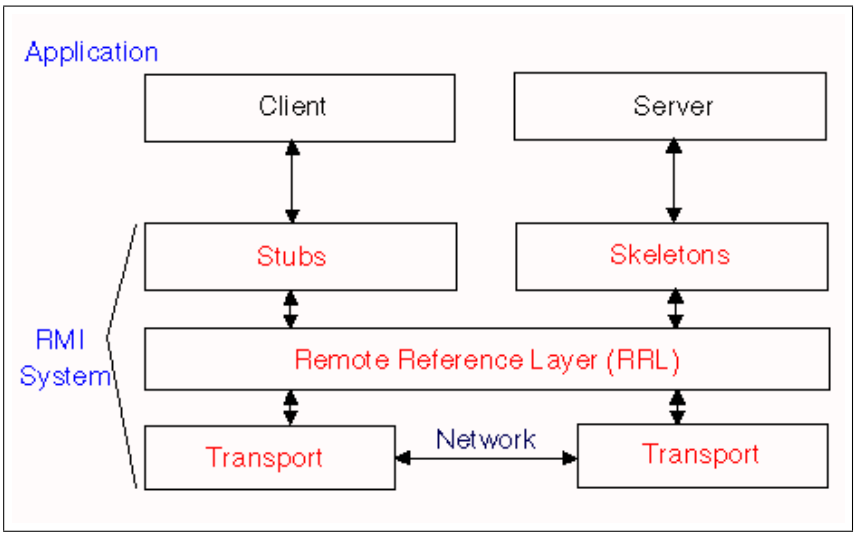


Figura 16: Arquitectura básica RMI

En cuanto a los clientes hay que tener en cuenta que no todos sus metodos estarán disponibles de forma remota, si no sólo aquellos marcados para lanzar excepciones remotas, del tipo *java.rmi.RemoteException*. Para marcar un método de una clase Java como accesible de forma remota debe, a parte de lanzar las excepciones de tipo *RemoteException* antes mencionadas la clase debe implementar un interfaz que extienda *Remote*, lo que etiqueta el método como accesible remotamente. La secuencia puede verse en la figura 17. Una vez hecho esto la secuencia para acceder a un metodo remoto es:

- Buscar el registro RMI: Un ejemplo de como se hace esto seria:

```
Registry regCliente = LocateRegistry.getRegistry(ip, puerto);
```

Nótese que esta es solo una forma de hacerlo, se puede omitir el puerto, la ip, o incluso ambos. El puerto RMI reservado oficialmente es el 1099.

- Obtener el objeto remoto que queramos, por ejemplo, en el caso de Icaro, para pedir un agente:

```
ItfUsoAgenteReactivo agenteRemoto = (ItfUsoAgenteReactivo)
   regCliente.lookup(nombreAgente);
```

Nótese que el registro RMI devuelve Objetos, lo que hace necesario hacer un casting a la clase que esperamos. Además cuanto pedimos al registro un objeto lo hacemos indicando all registro el nombre con el cual fue registrado el objeto (Ver más adelante como se registran objetos).

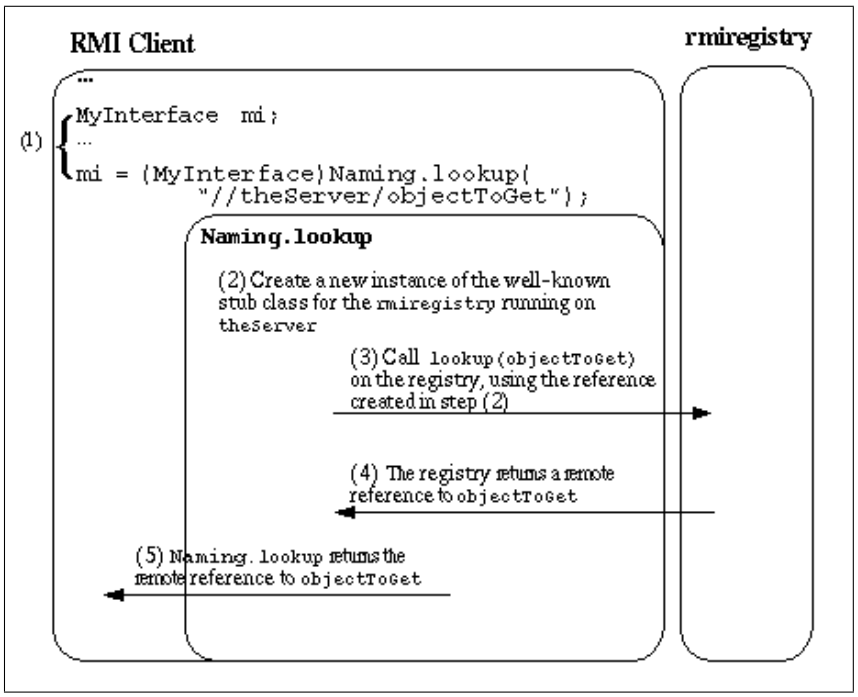


Figura 17: Ciclo de uso RMI

En cuanto al registro de objetos, el procedimiento a seguir es el siguiente:

- Si no se ha creado el registro RMI, crearlo, por ejemplo, con la siguiente sentencia:

```
int puertoRMI = 1099; /* Puerto por defecto */
Registry registroRMI = LocateRegistry.createRegistry(puertoRMI);
```

- Con el registro disponible, registrar los objetos

```
String nombreDeRegistro = "objetoRemotoA";
ItfObjetoRemoto objetoPorRegistrar = new objetoRemoto();
registroRMI.rebind(nombreDeRegistro, objetoPorRegistrar);
```

Nota: Estamos usando el método *rebind* en lugar de *bind*, esto sacará del registro a cualquier objeto que pudiera estar registrado previamente

- Para eliminar un elemento del registro se usa la sentencia:

```
registroRMI.unbind(nombreElemento);
```

4.3. Implementación de los componentes diseñados

Comentaremos aqui los detalles de implementación de los componentes diseñados para implementar las comunicaciones y la distribución, lo referente a la ventana de trazas se tratará en un punto a parte ya que es independiente del resot del desarrollo. También se tratará en un punto a parte un ejemplo de como se puede desarrollar una aplicación distribuida desde cero utilizando las mejoras implementadas en este proyecto.

4.3.1. GestorAplicaciónComunicación

Este nuevo gestor se integra para facilitar la comunicación entre agentes y simplificar de esta manera la distribución de los elementos del software. Para la implementación de cualquier aplicación candidata a ser distribuida se recomienda el uso de este gestor para TODAS las comunicaciones, ya que él mismo se encargará de buscar los agentes solicitados y entregarles los mensajes que se les envíen. A continuación se detallan algunos elementos de su implementación, el diagrama general de la clase que lo implementa puede verse en la figura 18 y sus clases directamente relacionadas en la figura 19. Además puede verse las relaciones completas de esta clase en la figura 8, mostrada anteriormente. Esta clase requiere de las clases de ComunicacionAgentes, ControlRMI y Mensaje.

Este gestor es el primero que arrancará después del gestor de organización, esto es necesario para poder utilizar este gestor para las comunicaciones de los gestores de agentes y/o de recursos. Además evita posibles errores en caso de que al arrancar los agentes supervisados por otro gestor estos intenten enviar algún mensaje y se genere una excepción al no encontrar al gestor de comunicaciones. Este gestor implementa distintos métodos para gestionar la comunicaciones, pensados para amoldarse a la manera particular que tengan los desarrolladores para escribir su código fuente, sin embargo se recomienda hacer uso únicamente del método `enviar_mensaje`, ya que es el más sencillo de entender y el que más transparencia ofrece a los desarrolladores, este método utiliza como mensajes objetos de la clase *mensaje*, que contiene datos del origen de la comunicación, asi como otros detalles para facilitar su gestión (esta clase se detallará más adelante). Nótese que, dado que tanto el gestor como el agente destinatario tienen su propio autómata, y por tanto, su propio conjunto de eventos reconocidos, aunque pretendamos mandar un evento "E1" a un agente cualquiera, este evento deberá estar encapsulado dentro de otro evento "E2", diferente, que es el que enviaremos realmente al gestor, por ello se recomienda el uso del método `enviar_mensaje`, cuyo funcionamiento se detalla a continuación:

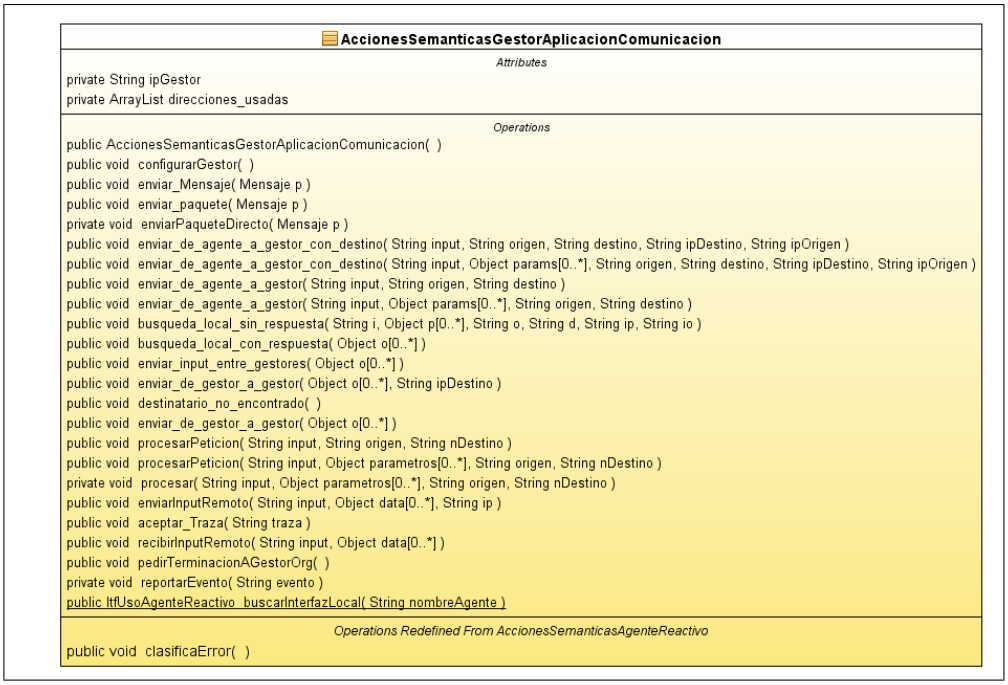


Figura 18: Diagrama de clase del gestor de comunicaciones

public void enviar_mensaje(Mensaje p): Este método es el recomendado para la gestión de las comunicaciones, trabaja de forma conjunta con el método *private void enviarPaqueteDirecto(Mensaje p)*, que se encargará de la comunicación efectiva, mientras que el primero gestionara la redirección del mensaje. De esta forma, el metodo `enviar_mensaje` recibe un objeto de la clase mensaje, creado previamente como se explicará mas adelante, y determinará que hacer con el usando el campo *tipoMensaje*, este campo contendrá un objeto de tipo String que identificará el propósito del mensaje, se ha preferido usar el tipo String sobre los habituales integers para que sea mas fácil mostrar los errores. Hay tres tipos principales de mensajes y según el tipo detectado se realizarán las siguientes redirecciones:

- *TIPO_INPUT_DESTINO_CONOCIDO*: La forma básica de comunicación, se usará el método "enviarPaqueteDirecto" para hacerlo llegar a su destino.
- *TIPO_INPUT_A_GRUPO*: Implementado para simplificar el envio de un mensaje a un grupo de agentes definido durante el desarrollo, al detectar este tipo de mensaje el gestor buscará todos los agentes definidos en la configuración cuyo grupo coincida con el especificado en el campo "destinoDatos" del mensaje, exceptuando por supuesto al que ha originado la comunicación. Para hacer esto por cada agente que encuentre creará un clon del mensaje a enviar y lo manipulará para que crea ser un mensaje "normal" con un único destinatario y lo redirigirá a su destino.

  Para poder usar esta característica, los agentes deberan tener definido su grupo en su lista de propiedades de la forma "grupo" "nombreDeGrupo", puede consultarse esta funcionalidad con más detalle en los ejemplos que se incluyen en el apartado de validación.
- *TIPO_INPUT_A_LISTA_DE_AGENTES*: Cuando no sepamos los componentes

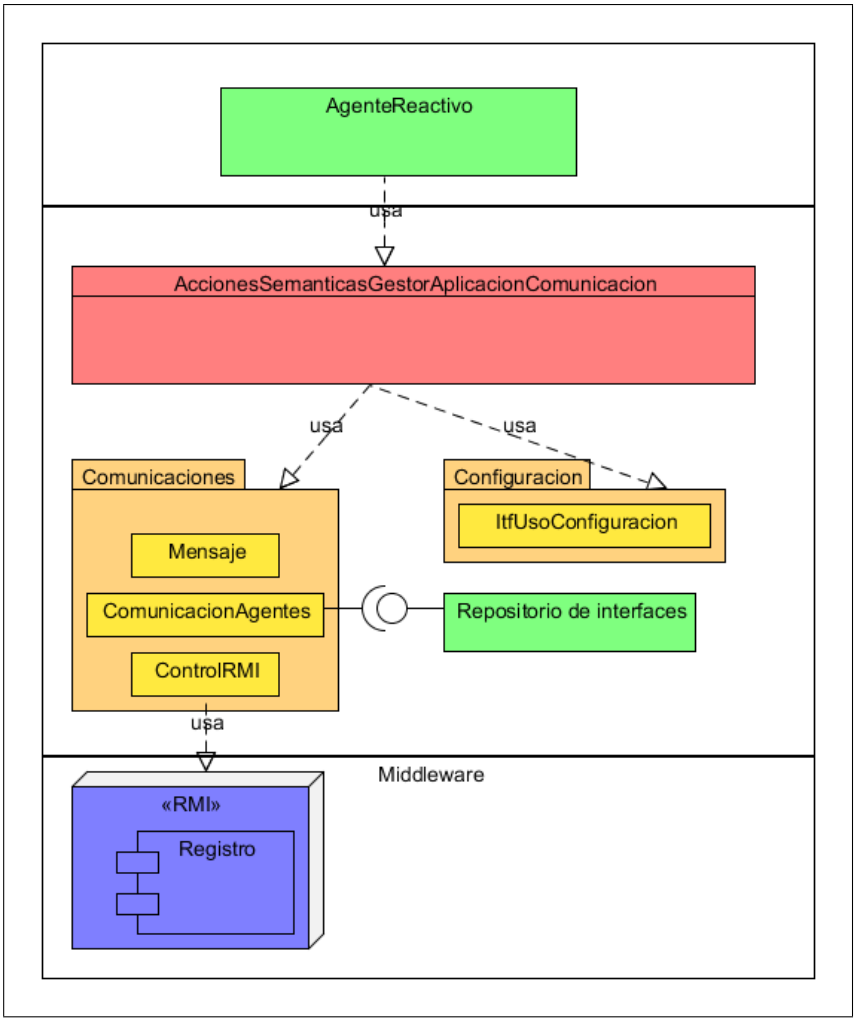


Figura 19: Relaciones de la clase AccionesSemanticasGestorAplicacionComunicaciones

de un grupo en tiempo de desarrollo o estos vayan variando mucho podemos hacer uso de este tipo de mensaje, al que pasaremos una lista de destinatarios para que el gestor les haga llegar un mensaje.

private void enviarPaqueteDirecto(Mensaje p): Este método es el que se encarga de hacer efectiva la entrega de mensajes, a un único destinatario, primero buscará el destinatario del mensaje en su propio nodo (la maquina donde se esté ejecutando), en caso de no encontrarlo se ocupará de buscarlo en todos los nodos que conozca, es decir, todos aquellos nodos que hayan sido declarados en las propiedades generales de la descripción de la aplicacion (fichero xml). Es posible añadir nuevos nodos en tiempo de ejecución pasándoselos a la configuración existente (Objeto), accesible, por ejemplo, mediante la clase `Directorio`.

Otras formas de comunicación

Como hemos dicho el método enviar_mensaje es el más recomendado, pero no el único, a continuación vamos a indicar otros métodos para poder enviar mensajes entre agentes. El ejemplo del chat incluido en la parte de validación contiene muchas formas distintas de realizar la comunicación, asi que haremos referencia aqui a dicho ejemplo. Los elementos se listan según el mensaje que debe recibir el gestor de comunicaciones

para interpretar los datos pasados correctamente, no el nombre de los métodos que se ejecutan, estos pueden consultarse en el fichero "automata.xml" en el paquete de comportamiento del gestor de comunicaciones.

- *enviar_msg_a_destinatario_conocido*: Este método y sus demas métodos sobrecargados se encargan de enviar mensajes a un destinatario especificado de forma parecida a enviar_mensaje, pero especificando los parámetros manualmente. Lo ilustraremos primero con un ejemplo, tomado del fichero AccionesSemanticasAgenteAplicacionClienteChat.java, de la carpeta aplicaciones del proyecto. Este método requiere la ip del destinatario, luego intentará enviar el mensaje directamente, el siguiente método no requiere este información.

```
public void enviarMensaje (String mensaje) {
String gestorComunicaciones = NombresPredefinidos.
   NOMBRE_GESTOR_APLICACION_COMUNICACION;
Object [] params = {miIP, mensaje};
Object [] input = {"enviarMensaje", params, nombreAgente,
   agenteServidor, sala, miIP};
 trazar ("Enviando mensaje: " + nombreCliente + ":" + mensaje);
ComunicacionAgentes.enviarInput("
   enviar_msg_a_destinatario_conocido", input, nombreAgente,
   gestorComunicaciones);
 chat.limpiarMensaje();
}
```

  En este ejemplo puede verse que es necesario encapsular el mensaje a entregar al destinatario final dentro de otro mensaje, que enviaremos al gestor de comunicaciones. Primero encapsulamos los parámetros que deseamos hacer llegar al destinatario final, con la instrucción *Object [] params = miIP, mensaje;*, esto es, enviamos el mensaje que queremos hacer llegar y nuestra IP como identificador del origen del mensaje. A continuacion encapsulamos el mensaje completo que queremos hacer llegar en otro objeto, mediante la instrucción *Object [] input = "enviarMensaje", params, nombreAgente, agenteServidor, sala, miIP*, téngase en cuenta que los eventos recibidos son del tipo EventoRecAgente, consulte el manual de Icaro si desea sabe más acerca de ellos. Finalmente, teniendo ya el mensaje a enviar al gestor se lo hacemos llegar a traves de un método de la clase estática ComunicaciónAgentes, con el formato apropiado, que es "enviar_msg_a_destinatario_conocido", como petición al gestor, esto informa al gestor de que es lo que queremos que haga con los datos contenidos en el objeto "input", finalmente le pasamos la cadena de texto "nombreAgente" como origen de los datos y gestorComunicaciones como destinatario, ya que ComunicacionAgentes no tiene sino otra forma de saber donde debe entregar el mensaje.

- *enviar_msg_al_aire*: Se comporta de forma parecida al enterior con una importante salvedad, no necesita que se le indique donde se encuentra el destinatario (aunque si tendremos que indicarle quien), sino que se encargará él mismo de buscar el destino, el método que se ejecutará es "enviar_input_entre_gestores(...)", éste primero intentará buscar el destinatario entre los agentes locales a él mismo, es decir, que están ejecutando dentro de su misma máquina virtual, en caso de encontrarlo en este contexto enviará directamente el mensaje y esperará un nuevo mensaje que entregar. En caso contrario sacará la descripción del agente

destinatario y buscará en que nodo se ha indicado que debe ejecutarse, si lo hay, con este dato utiliza la clase controlRMI para enviar el input remoto al destinatario final del mensaje. Un ejemplo de uso sería el siguiente:

```
public void enviarMensaje (String mensaje) {
String gestorComunicaciones = NombresPredefinidos.
   NOMBRE_GESTOR_APLICACION_COMUNICACION;
Object [] params = {miIP, mensaje};
/* Este es el mensaje que debe procesar el gestor */
Object [] input = {"enviarMensaje", params, nombreAgente,
   nombreServidor};
 trazar ("Enviando mensaje: " + nombreCliente + ":" + mensaje);
/* Esto es lo que llega al gestor */
ComunicacionAgentes.enviarInput("enviar_msg_al_aire", input,
   nombreAgente, gestorComunicaciones);
 chat.limpiarMensaje();
}
```

- *busquedaLocalSinRespuesta*: Este método se utiliza de forma análoga a *ComunicacionAgentes*, envia un mensaje a otro agente local (del mismo nodo). Un ejemplo de uso sería el siguiente:

```
String mensaje = "estoyListo";
Object [] parametros = {"noHayProblemas",0};
ComunicacionAgentes.enviarInput("busquedaLocalSinRespuesta",
   mensaje, parametros, nombreAgente, gestorComunicaciones,
   ipOrigen, ipDestino);
```

- *busquedaLocalConRespuesta*: Es mensaje sirve para indicar que debemos entregar un mensaje localmente, pero que el origen es remoto y esta esperando que le indiquemos que ha ocurrido con el mensaje. Primero intentará entregar el mensaje local, y a continuación consultará el origen del mensaje e intentará enviar una respuesta al origen del mensaje. Las respuestas enviadas seran mensajes de tipo 'eventoAceptado" en caso de éxito y "eventoRechazado" en caso de fallo, ambos sin parámetros. Su uso es idéntico al anterior, cambiando únicamente el texto del mensaje enviado al gestor de "busquedaLocalSinRespuesta" a "busquedaLocalConRespuesta". Para que la respuesta tenga se envíe correctamente hay que ajustar correctamente el origen y la ipOrigen del mensaje, además con esto se pueden desviar las respuestas a un destinatario diferente al origen real de los mensajes, para poder gestionar a parte los sucesos en las comunicaciones.

- *Otros metodos*: Existen otros métodos para enviar mensajes entre agentes, pero no requieren del gestor de comunicaciones, ver las secciones "ComunicacionAgentes" para envio de mensajes locales y "ControlRMI" para los remotos, más adelante en este mismo capítulo.

4.3.2. Mensaje

Esta clase implementa los objetos tipo mensaje que se han mencionado en el apartado anterior, básicamente sirven para englobar los datos que se quieran enviar, y se encargan de recoger algunos datos extra de información acerca del origen del mensaje.

Existen múltiples métodos para crear mensajes y modificar sus datos, además se incluyen métodos para crear automáticamente los tipos de mensajes más comunes para implementar la comunicación entre agentes reactivos de Icaro. De esta forma, se indica a continuación las acciones más comunes y como crear (y enviar), los mensajes típicos usando mensajes. Puede verse una descripcion de la clase mensaje completa en la figura 20, mientras que las clases que dependen de ella pueden verse en la figura 19.

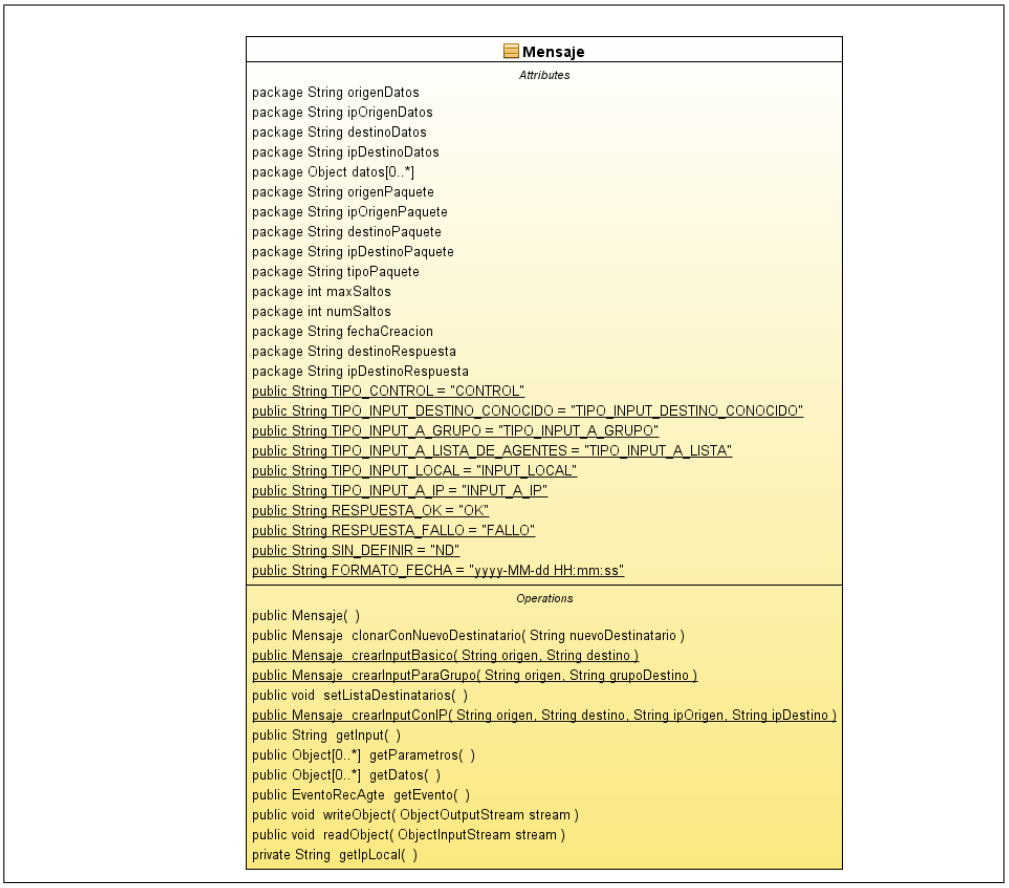


Figura 20: Diagrama de la clase Mensaje

- *Enviar un mensaje a un agente*: El más habitual, crearemos un mensaje que contenga un evento para un agente, asi como los parametros que pudieran ser necesarios, si los hay, el método para crear un mensaje de este tipo es *crearInputBasico*, a continuación se indica un ejemplo de su uso, nótese sin embargo que este método admite un número variable de parámetros, el primero de los cuales DEBE ser el nombre dele evento que deseamos hacer llegar (Ejemplo correspondiente al ejemplo chat del apartado Validación):

```
/* Primera forma: utilizando comunicacionAgentes */
Mensaje p;
p = Mensaje.crearInputBasico(
 nombreAgente, agenteServidor, "peticionConexion", nombreCliente,
     miIP, nombreAgente );
/* Le pido a comunicacion agentes que envie el mensaje a mi
   gestor */
ComunicacionAgentes.enviarInput(
 "enviar_mensaje", p, nombreAgente, gestorComunicaciones);
```

También se puede obviar el paso de enviarInput al gestor si se usa el método enviarMensaje(Mensaje p), incluido en las acciones semánticas existentes por defecto en cualquier tipo de agente, quedando el código asi:

```
/* Segunda forma: utilizando metodo incluido */
Mensaje p = Mensaje.crearInputBasico(
 nombreAgente, agenteServidor, "peticionConexion", nombreCliente,
     miIP, nombreAgente
);
enviarMensaje(p);
```

- *Enviar un mensaje a un grupo*: Cuando el destino es un grupo de agentes descrito en la descripción de la aplicación, podemos crear el mensaje de la forma siguiente (Ejemplo correspondiente al ejemplo MasterIA, del apartado de Validación)

```
/* Segunda forma: utilizando metodo incluido */
Mensaje p = Mensaje.crearInputParaGrupo(
 nombreAgente, "ROBOTS", mensaje, parametro1
);
               enviarMensaje(p);
```

- *Enviar un mensaje a una lista*: Cuando queramos enviar un mismo mensaje a un grupo de agentes no definido previamente, usaremos este método, que difiere un poco de los demas, ya que primero se crea un mensaje normal y después se le indica su nuevo cometido (destinatarios), un ejemplo seria:

```
Mensaje p = Mensaje.crearInputBasico(
 nombreAgente, "", "peticionConexion", nombreCliente, miIP,
     nombreAgente);
/* Ponemos  lista de destinatarios, aqui solo 2, pero puede ser
   cualquier numero*/
p.setListaDestinatarios (destinatario1, destinatario2);
enviarMensaje(p);
```

Nótese que el método setListaDestinatarios admite un número variable de parámetros, y que cualquier destinatario que pudieramos haber establecido previamente durante la creación del mensaje sera ignorado, a menos que volvamos a incluirlo dentro de los parametros del método setListaDestinatarios.

4.3.3. Comunicación Agentes

Esta clase está pensada meramente para hacer un poco más transparente el envio de mensajes a un agente (sin usar el gestor), contiene métodos estáticos, es decir, que no es necesario instanciar un objeto de esta clase para usar sus métodos, no es necesario usarla directamente si se utiliza el método sugerido con anterioridad (uso de mensajes), sin embargo no debe eliminarse, ya que el gestor de comunicaciones depende de ella. El ejemplo del chat incluido en el apartado de validación incluye muchos ejemplos de como se utiliza esta clase, en cualquier caso, un ejemplo sencillo seria el siguiente (ya puesto anteriormente):

```
/* Primera forma: utilizando comunicacionAgentes */
```

```
Mensaje p = Mensaje.crearInputBasico(nombreAgente, agenteServidor, "
    peticionConexion", nombreCliente, miIP, nombreAgente);
// Le pido a comunicacion agentes que envie el mensaje a mi gestor
ComunicacionAgentes.enviarInput("enviar_mensaje", p, nombreAgente,
    gestorComunicaciones);
```

Nótese que los metodos de esta clase no admiten un número variable de parámetros, lo que puede obligar a incluir los parámetros dentro de un objeto previamente a enviar el mensaje, como, por ejemplo, en el método enviarMensaje, incluido en el fichero AccionesSemanticasAgenteAplicacionClienteChat, que reproducimos a continuación:

```
Object [] params = {miIP, mensaje};
Object [] input = {"enviarMensaje", params, nombreAgente,
    agenteServidor, sala, miIP};
trazar ("Enviando mensaje: " + nombreCliente + ":" + mensaje);
ComunicacionAgentes.enviarInput("enviar_msg_a_destinatario_conocido",
    input, nombreAgente, gestorComunicaciones);
```

4.3.4. Control RMI

De forma análoga a *ComunicacionAgentes*, ControlRMI contiene un conjunto de métodos estáticos para ayudar a simplificar el uso de RMI, no es necesario usar esta clase en absoluto si se utiliza el gestor de comunicaciones, pero debe tenerse en cuenta que el gestor de comunicaciones, asi como otros gestores, como por ejemplo el gestor de organizacion, si utilizan esta clase para realizar distintas tareas, algunas de ellas durante el arranque o parada del sistema, por lo que no debe eliminarse aunque no se piense utilizar. Los métodos que ofrece son los siguientes:

- *public static boolean startRMI()*: Se encarga de crear, si es necesario, un nuevo registro en el puerto 1099, si este no se ha modificado previamente a realizar la llamada a este método. Cabe destacar que esta llamada provacará un intento por parte de controlRMI de añadir un icono a la barra de tareas, si se trata de un entorno windows, para indicar que se ha activado RMI, pulsando sobre el icono se obtendrá un menu con la lista de objetos registrados en RMI, ademas se tendra la opcion de cerrar RMI, lo que provocara que TODOS los elementos se eliminen del registro RMI (lo que no los elimina de la maquina virtual), finalmente desaparecerá el icono que indica que RMI esta activo. Para hacer pruebas a un sistema pueden añadirse acciones a realizar si se selecciona algún elemento del menu que proporciona, esto puede hacerse modificando la clase interna *MenuItemIcaro*, contenida en el mismo fichero que controlRMI.

- *setPuertoRMI(int puerto)*: Cambia el puerto donde debe crearse el registro RMI. Esta característica debe usarse con sumo cuidado, ya que el puerto que esta marcado por defecto, el 1099 esta reservado de forma oficial para el servicio RMI, y por lo tanto muchas aplicaciones, e incluso parte de la API de Java solo consideran este puerto como válido, lo que puede provocar fallos difíciles de depurar si se modifica.

- *public static boolean export(String element, Remote obj)*: Exporta un objeto, lo que equivale a registrar el objeto "obj" en el registro RMI bajo el identificador "element"
- *public static boolean fire(String element)*: "despide" un objeto del registro, esto elimina la asociacion con el objeto que existe en la maquina virtual, pero no lo elimina.
- *public static boolean fireAll ()*: Parecido a *fire*, solo que este método eliminara todas las asociaciones que hubiera en RMI, limpiandolo por completo. Este es otro método que debe usarse con cuidado, ya que el registro RMI puede estar compartido por varias aplicaciones, lanzar este método puede provocar inestabilidad en otras aplicaciones.
- *public static void createTray()*: Crea el icono para la bandeja de entrada explicado con anterioridad.
- *protected static Image createImage(String path, String description)*: Se encarga de crear la imagen que se usa en el icono para la barra de tareas.
- *public static String getIPLocal ()*: Devuelve la dirección IP de la máquina en que se ejecuta
- *private static class MenuItemIcaro extends MenuItem*: Esta clase privada se encarga de implementar los elementos que se mostrarán en el menú activado desde el icono creado en la barra de tareas, contiene un ActionListener, activado cuando se pulsa sobre el elemento que representa, y que puede modificarse para que realize multiples acciones, por defecto eliminar el agente de RMI.
- *public static ItfUsoAgenteReactivo buscarAgenteRemoto(String ip, String nombreAgente)*: Este método se encarga de buscar un agente registrado con el identificador "nombreAgente" en la dirección ip especificada, que puede ser la misma en que se ejecuta. De encontrarlo, devolverá una referencia con un interfaz ItfUsoAgenteReactivo, que podrá usarse para acceder a los métodos marcados como remotos previamente (Ver capitulo acerca de RMI, previamente en este manual), en otro caso devolverá null. Este método tiene otro más sobrecargado, al que tambien se le puede indicar el puerto donde estará escuchando el registro RMI del otro extremo, pero por las mismas razones explicadas antes sobre el cambio del puerto RMI, no se aconseja utilizar esta opción.
- *public static boolean enviarInputRemoto(String input, Object[] param, String nombreRecurso, ItfUsoAgenteReactivo remoto)*: Una vez obtenido un interfaz de uso de agente válido, puede usarse este método u otro de los demas métodos sobrecargados sobre este, para enviar un mensaje de forma remota, de la misma manera que se hacia mediante la clase estática *ComunicacionAgentes*.

4.3.5. Gestor Nodo

Este gestor arranca de manera diferente al resto de descripciones, gestor nodo es capaz de arrancar por si mismo, sin necesidad de lanzar primero un gestor de organización, ni tampoco necesita un fichero de descripcion xml que le diga lo que tiene que

hacer. Esto se hace debido a que su funcionalidad es muy sencilla y a que normalmente es un proceso que queremos tener en espera de que se le indique que se desea realizar alguna tarea, o lo que es lo mismo, que se le indique que aplicación icaro-d se desea arrancar.

Para realizar su trabajo, el nodo (máquina) donde se arranca este gestor debe disponer de todo lo necesario para lanzar sus aplicaciones por si mismo, ya que la única información que le va a llegar para iniciar su trabajo es el nombre de la descripción de aplicación que debe ejecutarse, por lo que debe disponer localmente tanto de los ficheros de descripción xml como del código fuente necesario.

Existen dos formas de pedir el arranque remoto, *arrancarAplicacion*, que lanza el resto de elementos dentro de la misma maquina virtual donde esta ejecutándose el gestor de nodo, y *executeApp*, que lanzará un proceso separado para la aplicación solicitada. El efecto diferenciador es que en el primer caso cerrar la aplicación finaliza también el gestor del nodo, mientras que en la segunda el gestor es capaz de sobrevivir a la aplicación que ha lanzado.

Para lanzar un gestorNodo se debe ejecutar icaro-d con la opción *-nodo*, indiferente a mayúsculas o minúsculas.

4.4. Implementación de las trazas

Para nuestro proyecto hemos diseñado una nueva visualización de las trazas que creemos que mejora la anterior versión. Primero comentaremos su implementación.

4.4.1. Detalles de implementacion

La anterior versión de las trazas estaba implementada en su totalidad en el paquete imp de icaro/infraestructura/recursosOrganizacion/recursoTrazas. En este paquete se incluyen la claseGeneradora, la gui y todo lo necesario para el buen funcionamiento del componente.

La nueva versión de las trazas se encuentran en el paquete imp2 en la misma ruta anteriormente dada.

Como se ve en la imagen, el paquete consta de Tabs para implementar cada uno de los paneles del JTabbedPane que más tarde comentaremos, componentes para implementar la tabla de las trazas a mostrar, el recursoTrazasImp2 que hereda de la claseGeneradora, y la ventanaTrazas que contiene la implementación de la visualización y usa tanto Tabs como componentes.

Nota: la claseGeneradora es la misma que en la versión anterior; es decir, para usarla habrá que importarla del paquete imp:

La localizacion de la version previa del recurso de trazas es la siguiente:

```
import icaro.infraestructura.recursosOrganizacion.recursoTrazas
.imp.ClaseGeneradoraRecursoTrazas;
```

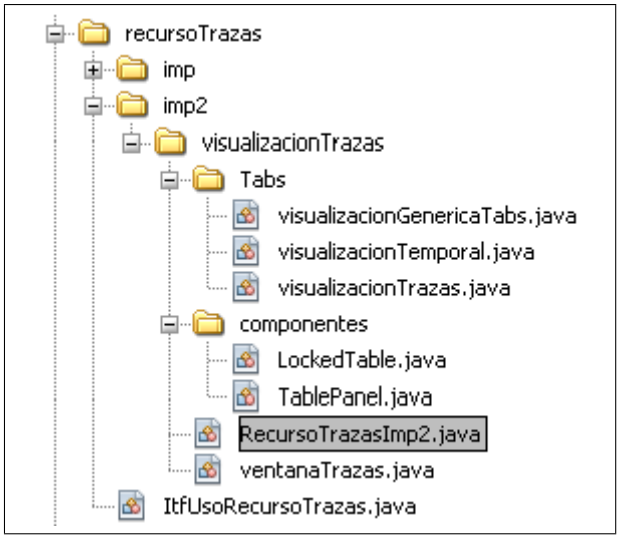


Figura 21: Jerarquia del paquete de trazas

4.4.2. Uso de la nueva implementacion

Como primer paso, hay que señalar que el recurso de las trazas se crea en las clases generadoras de organización, ArranqueSistemaConCrtlGestorO y ArranqueSistemaSinAsistente. En estas dos clases, se hace la llamada a la clase generadora del recurso de trazas de la siguiente forma:

```
ItfUsoRecursoTrazas recursoTrazas  =
ClaseGeneradoraRecursoTrazas.instance();
```

Este método devuelve una instancia de la visualización que queremos usar de la siguiente manera:

```
instance = new RecursoTrazasImp2(
NombresPredefinidos.RECURSO_TRAZAS);
```

Es en el punto recién descrito donde el usuario podrá cambiar la versión de la visualización que quiera usar.
NOTA: en el código fuente se puede observar que la creación de RecursoTrazasImp está comentado.

4.4.3. Nueva visualizacion

En este apartado describiremos el uso de las nuevas trazas mediante capturas de pantalla. La principal diferencia entre estas trazas y las anteriores reside en que nosotros hemos separado las trazas de agentes, recursos y gestores, en pestañas distintas. En la actualidad, la pestaña Diagrama Temporal está sin implementar.

Además, mostramos en todo momento en cada una de ellas en un panel lateral una lista de los componentes de cada clase dependiendo de la pestaña en la que nos encontremos.

Figura 22: Cabecera de la ventana de trazas

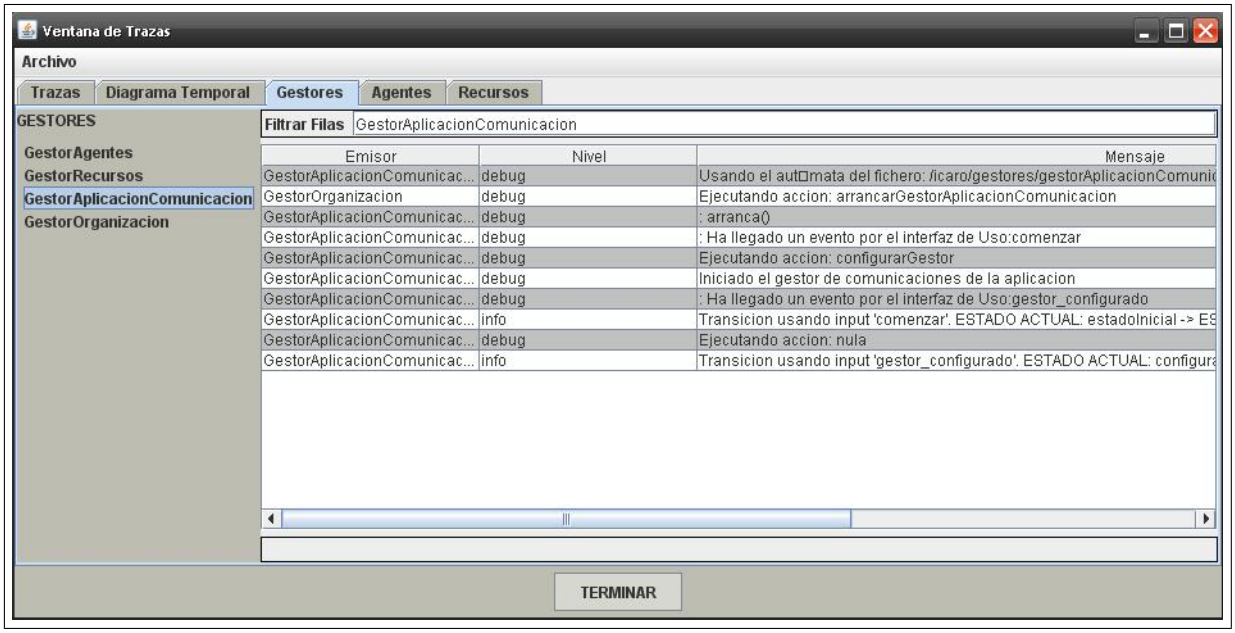


Figura 23: Ventana de trazas

Si seleccionamos uno de los componentes del panel lateral, se mostrarán sólo las trazas de dicho componente, usando de forma automática el filtro "Filtrar Filas" que hemos incorporado a la visualización como una herramienta adicional. Este filtro podrá ser usado por el usuario para buscar lo que quiera en las trazas de manera rápida y sencilla.

Si el usuario quiere visualizar las trazas de un sólo componente en una ventana auxiliar sólo tendrá que hacer doble click sobre uno de los componentes del panel lateral, y aparecerá una ventana como esta:

Ahora, la información que aparece en cada traza es la siguiente: Emisor (el componente que envía la traza) , Nivel (debug,info,error) , Mensaje (el contenido de la traza) y Recepción (momento exacto de llegada de la traza).

Nota: Como se ve en la figura las columnas están más compactas para que cupiera la imagen en el documento. Esta opción es muy útil ya que permite al usuario ver las trazas a su antojo ajustando el tamaño de las trazas al mensaje mostrado. Por esta razón, aconsejamos no excederse en el tamaño del mensaje de la traza, ya que si se llega al tamaño máximo no se podrá mostrar.

Otro detalle que hemos añadido a la visualización es que al seleccionar una traza, aparecerá en la parte inferior del panel la información del mensaje seleccionado. Esto

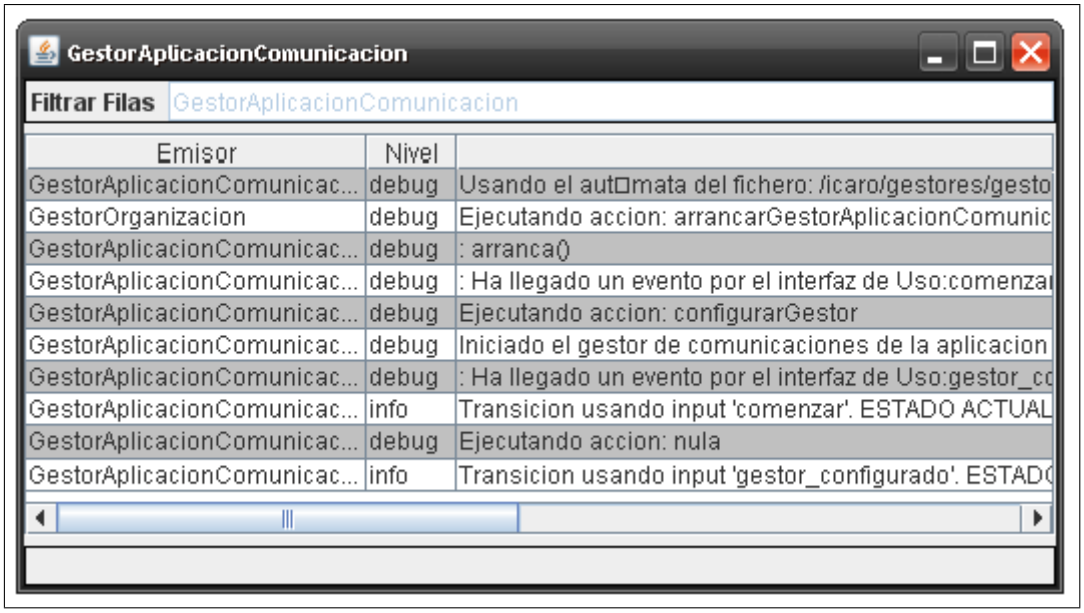


Figura 24: Visualizacion de las trazas de un componente en una ventana a parte

Emisor	Nivel	Mensaje	Recepcion
GestorAplicacionCo...	debug	Usando el aut□mata ...	2010.05.03.15.46.16.406
GestorOrganizacion	debug	Ejecutando accion: ar...	2010.05.03.15.46.21.671
GestorAplicacionCo...	debug	: arranca()	2010.05.03.15.46.21.812
GestorAplicacionCo...	debug	: Ha llegado un event...	2010.05.03.15.46.22.109
GestorAplicacionCo...	debug	Ejecutando accion: co...	2010.05.03.15.46.22.125
GestorAplicacionCo...	debug	Iniciado el gestor de c...	2010.05.03.15.46.22.140
GestorAplicacionCo...	debug	: Ha llegado un event...	2010.05.03.15.46.22.187
GestorAplicacionCo...	info	Transicion usando in...	2010.05.03.15.46.22.203
GestorAplicacionCo...	debug	Ejecutando accion: n...	2010.05.03.15.46.22.265
GestorAplicacionCo...	info	Transicion usando in...	2010.05.03.15.46.22.265

Figura 25: Columnas de la ventana de trazas

puede ser útil si el tamaño de la traza es excesivamente grande:

Hemos añadido también la utilidad de guardar las trazas en un fichero de texto en el menú Arrchivo en la parte superior izquierda del panel. Se puede seleccionar guardar todas las trazas o lo de un sólo componente a elegir.

En cuanto al botón “Terminar” en la parte inferior del panel, explicaremos a continuación lo que hace y cómo lo hace. Al hacer click sobre este botón, se cerrarán TODAS la ventanas y aplicaciones abiertas con Icaro. Esto ocurre porque en la implementación del botón envía una petición de terminación total al gestor:

```
gestorOrganizacion.aceptaEvento(
new EventoRecAgte("peticion_terminar_todo_usuario",...);
```

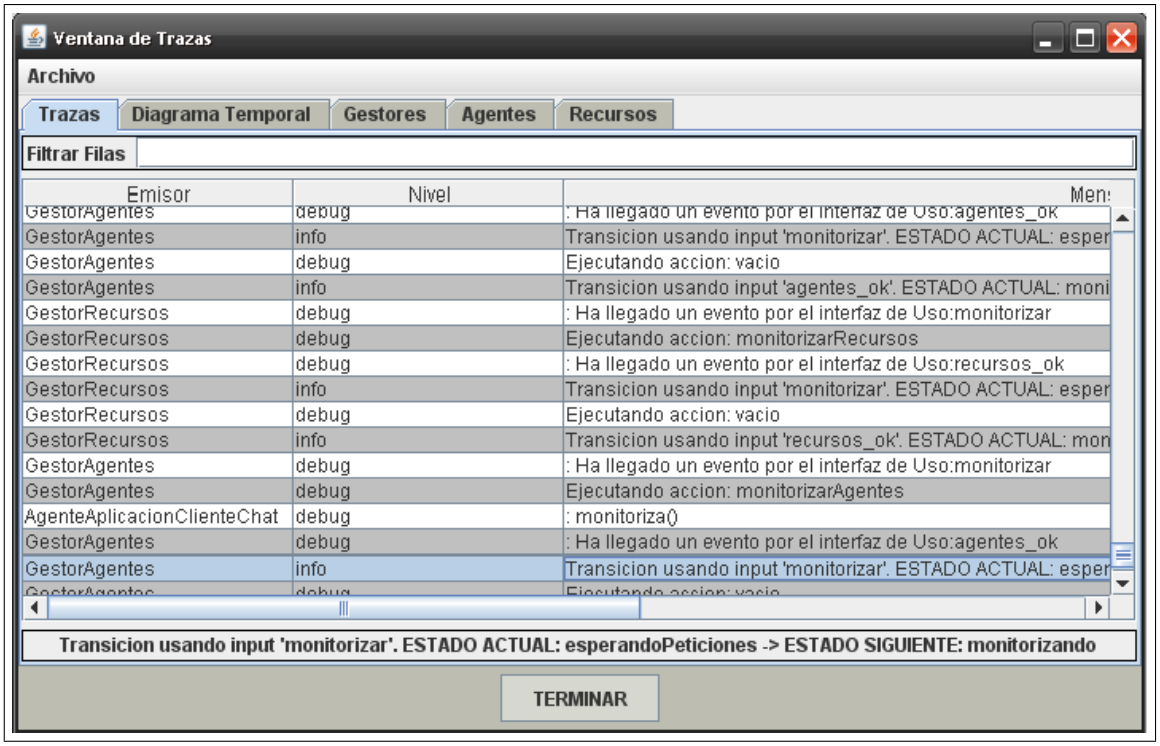


Figura 26: Texto resaltado de la ventana de trazas

Figura 27: Menu de la ventana de trazas

4.5. Aplicaciones distribuidas

4.5.1. Guía de ejemplo

Introduccion

En este pequeño manual se describen los pasos y las pautas a seguir para diseñar e implementar una aplicación con icarod, en primera instancia local, para posteriormente hacer de su distribución una tarea trivial, por medio de un ejemplo práctico sencillo.

El texto consta de las siguientes secciones: funcionalidad del ejemplo, detalles previos de diseño, implementación, descripción de aplicación, y distribución.

Funcionalidad del ejemplo

La aplicación ejemplo consiste en un sencillo mini chat de envío de mensajes au-

tomáticos. Primero, el usuario tendrá que conectarse al resto de usuarios (agentes). Esto se hace principalmente para comprobar si están arrancados o no y evitar excepciones remotas en el caso distribuido. Una vez conectado, los usuarios podrán mandar un mensaje ("MENSAJE A ESCRIBIR") al resto de usuarios.

En este caso, hay dos usuarios, pero podrían cuantos se quisieran. Puede verse un diagrama de secuencia que ilustra la funcionalidad del ejemplo en la figura 28. Si bien es cierto que como aplicación local tiene poco interés este ejemplo, es muy ilustrativo de la técnica de implementación y diseño de aplicaciones distribuidas que se aconseja.

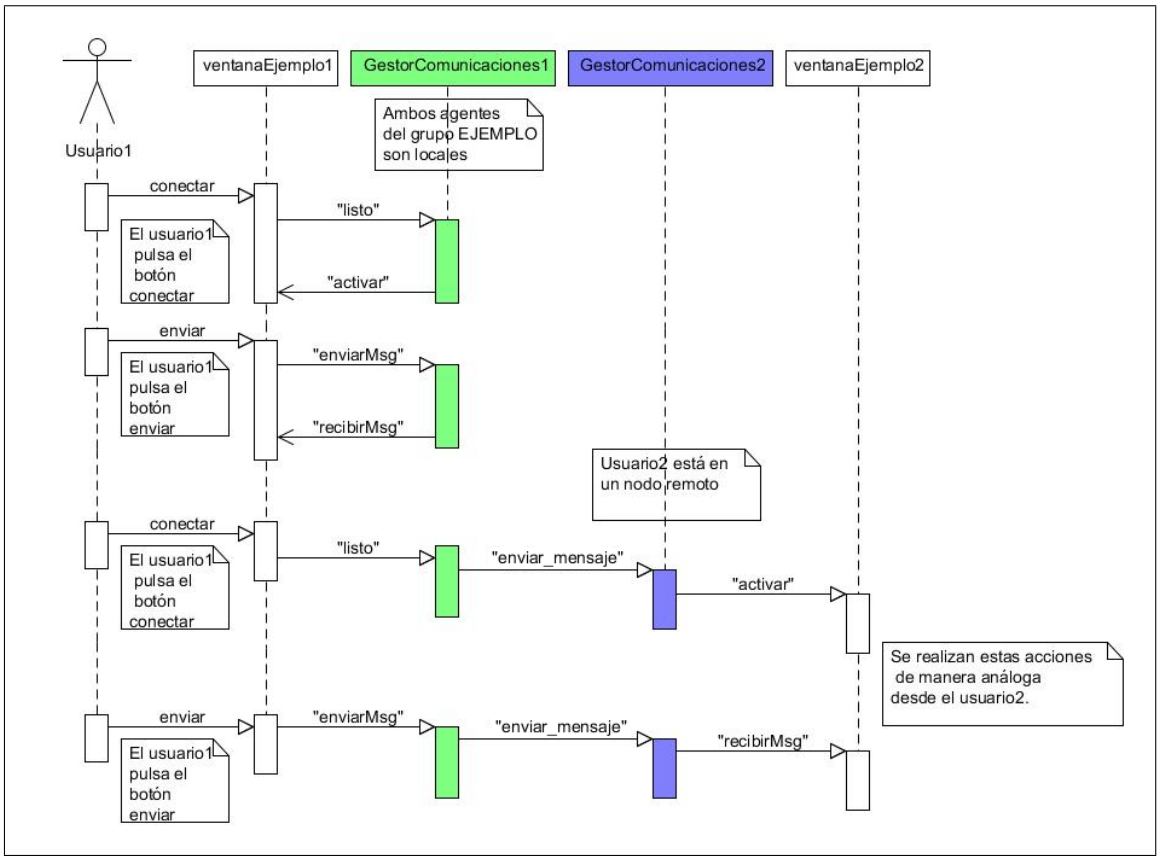


Figura 28: Secuencia de acciones del chat

El título de la ventana es nombreAgente / nombreUsuario. Como se puede obsservar, este recurso es un Jframe que contiene un JtextArea y dos botones, conectar y enviar. Como el usuario todavía no se ha conectado, el botón enviar está deshabilitado. Se incluye un ActionListener para ambos botones que se implementa en las acciones semánticas del ejemplo. Al pulsar sobre el botón "conectar", el agente establecerá una conexión con el otro agente y si ocurriera un error se comunica a través de la traza y de la propia ventana. Además se habilita el botón enviar. Las ventanas de ambos usuarios después de pulsar el botón conectar pueden verse en la figura 30. Al pulsar sobre el botón "enviar", se manda el mensaje "MENSAJE A ENVIAR : nombreUsuario" al otro agente, y éste lo muestra a través de su ventana como puede verse en la figura 31

Detalles previos del diseño

Es importante hacer hincapié en el diseño de la aplicación a desarrollar antes de la implementación. El autómata representa el comportamiento de las instancias de

Figura 29: Visualizacion del cliente

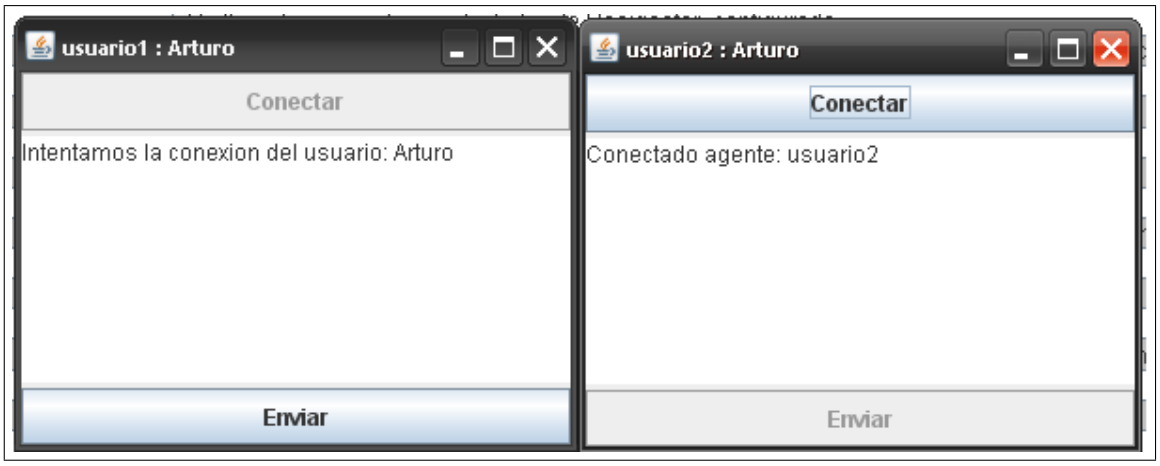


Figura 30: Conexión con exito

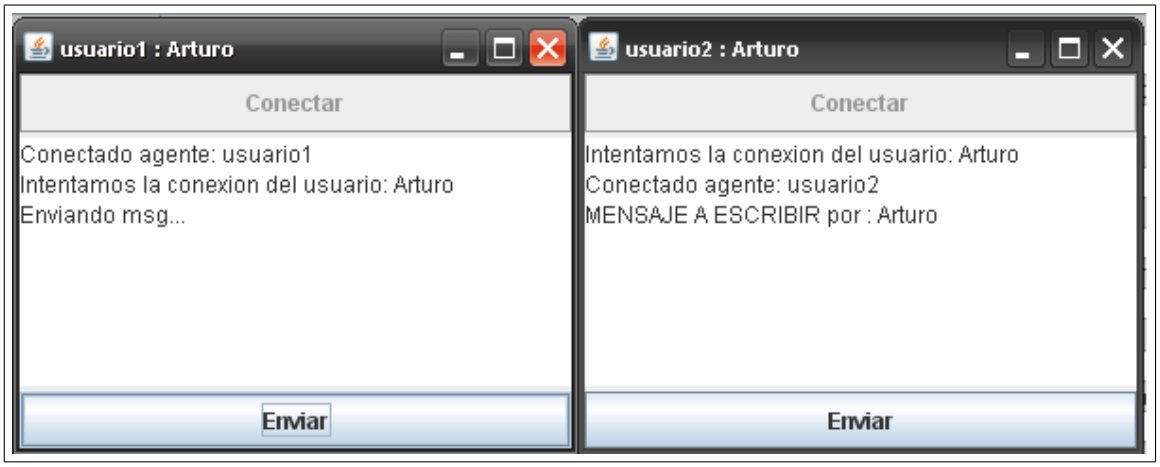


Figura 31: Mensaje enviado con exito

agente. Si bien es cierto que para el desarrollo de esta tarea en icaro hay que ceñirse a unas reglas bastante estrictas en cuanto al diseño de los agentes (arranque,gestion de errores, terminacion), hay cierto grado de libertad al plantear el autómata de dicho agente. El autómata de las dos instancias de agentes del ejemplo puede verse en la

figura: 32

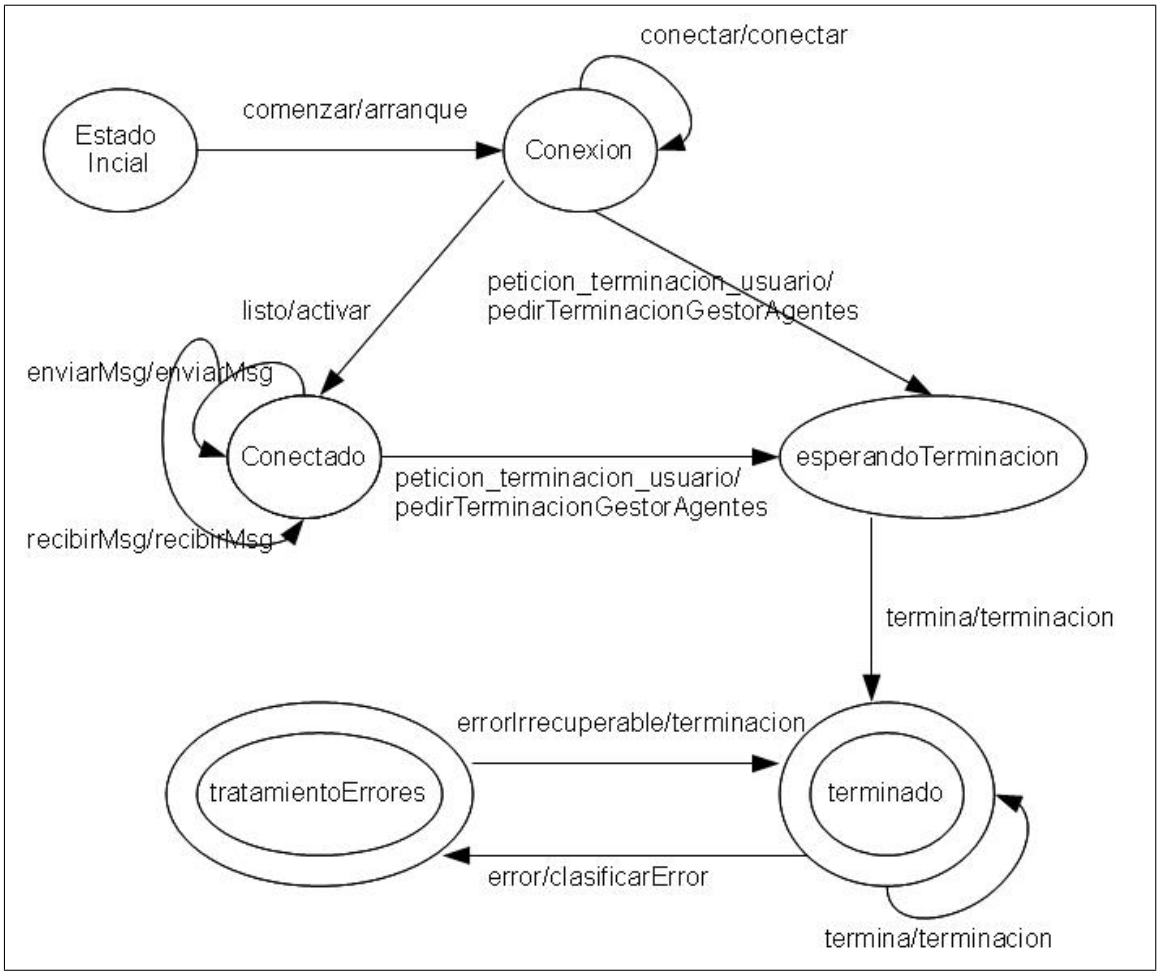


Figura 32: Maquina de estados de cada agente

El formato de la transición es "input" /accion. Hay estados que deben ser comunes en todos los autómatas de icaro: estadoIncial, esperandoTerminacion, tratamientoErrores y terminado. Estos estados controlan arranque, gestion de errores y terminación del arranque. Todas las acciones presentes en el autómata deben ser implementadas en las acciones semánticas del agente en cuestión, archivo que a continuación se describirá.

Implementación: Acciones Semanticas y Recurso de Visualizacion

Esta clase se encarga de implementar toda la funcionalidad del agente y, en este de ejemplo, de relacionar el recurso asociado al agente con la funcionalidad mencionada. El recurso es la visualización explicada al principio del texto. A continuación, se explica la implementación de esa funcionalidad mediante la clase AccionesSemánticas-AgenteAplicacionEjemploAprendizaje. El diagrama de clases que muestra los clases que conforman agente y recurso del ejemplo se pueden ver en la figura 33

El esquema del fichero es el siguiente:

```
public class AccionesSemanticasAgenteAplicacionEjemploAprendizaje
 extends AccionesSemanticasAgenteReactivo {

 /* Atributos */
```

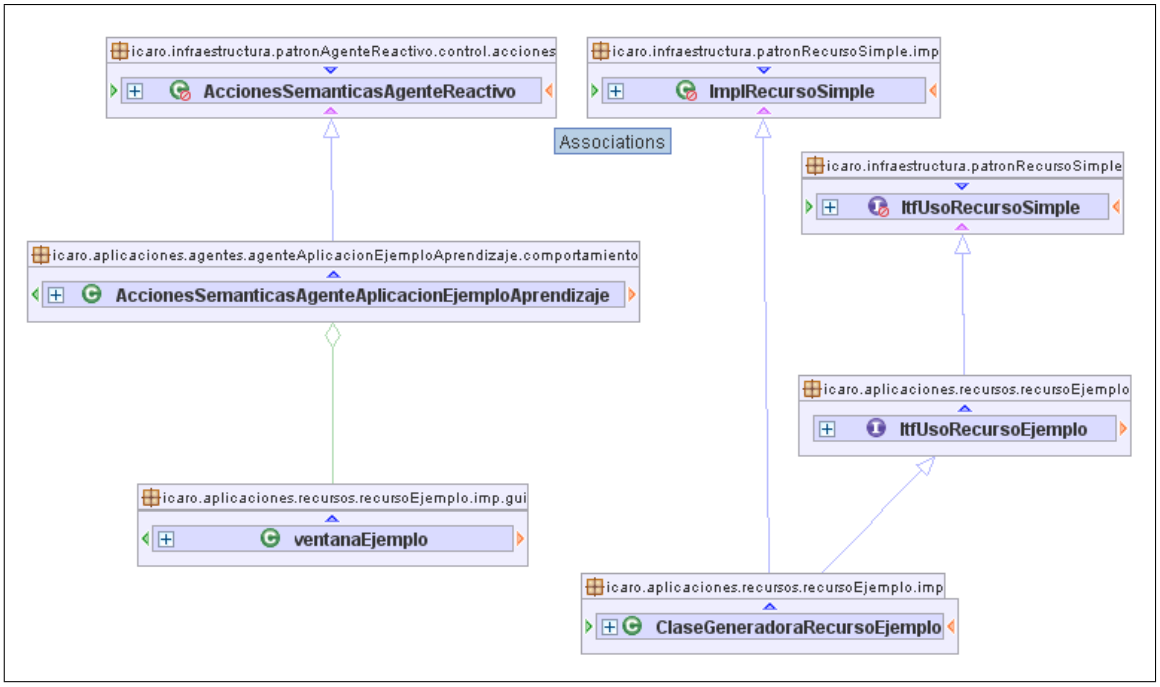


Figura 33: Diagrama de clases del ejemplo

```
  /*Arranque */

  /* Metodos propios : en este caso, segun el automata anteriormente
  descrito, conectar, activar, enviarMsg, recibirMsg. */

  /* Metodos para gestion de errores y terminacion del agente */
}
```

Como se puede observar, esta clase hereda los métodos y atributos de AccionesSemánticasAgenteReactivo, acción que se ha de realizar para todos los agentes que el usuario desee crear. Los métodos para gestión de errores y terminación son siempre iguales para todos los agentes, por lo que se aconseja copiarlos del ejemplo de manera exacta si no se quiere añadir ninguna nueva funcionalidad a esas tareas. En el arranque se deben crear e inicializar los atributos y componentes a usar del agente en cuestión. Así, sólo queda detallar los métodos propios del agente del ejemplo, especificados en su autómata.

Antes de comentar estos métodos se debe realizar una aclaración sobre el método de envío de mensajes entre agentes. Se aconseja fervientemente usar los componentes ComunicacionAgentes, Mensaje y GestorAplicacionComunicacion, ya que como observará el usuario a continuación reducen el trabajo del usuario notoriamente.

Un método descriptivo de la funcionalidad es conectar, que realiza la tarea de establecer la comunicación entre los agentes, usando los componentes anteriormente mencionados, ComunicacionAgentes, Mensaje y GestorAplicacionComunicacion.

```
/*Se crea un mensaje para un grupo de agentes, declarado en la
    descripcion de la aplicacion, con los parametros: origen,
    grupoAgentes, input (en este caso listo, ver el automata), sin
    parametros */
Mensaje p = Mensaje.crearInputParaGrupo(nombreAgente, EJEMPLO, listo);
```

```
/*Se envia el mensaje al gestor de comunicaciones con los parametros:
   input  ("enviar_mensaje", el paquete anteriormente creado, origen y
   destino (en este caso el gestor de comunicaciones, quien se
encargara de hacerlo llegar al destino deseado sea local o remoto).*/
ComunicacionAgentes.enviarInput("enviar_mensaje", p, nombreAgente,
   NombresPredefinidos.NOMBRE_GESTOR_APLICACION_COMUNICACION);
```

Esto es todo lo que el usuario tiene que hacer para establecer la comunicación con el grupo de agentes "EJEMPLO" que en este caso es otro agente. Es decir, los paso a seguir son:

- Crear el mensaje a enviar, con el origen , destino (en este caso un grupo de agentes), el input, y sin tener parámetros. Se podría haber mandado el mensaje al otro agente de forma directa creando un input básico para el mensaje (véase documentación sobre la guía de distribución), pero de esta forma se deja abierta la ampliación de la aplicación con más agentes dentro del grupo *EJEMPLO*.
- Enviar el mensaje al GestorAplicacionComunicacion del nodo local, mediante el input `"enviar_mensaje"` a través del CommunicacionAgentes, obteniendo el nombre exacto del Gestor de Comunicaciones local de los NombresPredefinidos. Es el propio gestor del nodo quien se encarga de hacer llegar el mensaje a su destino sea local o remoto.

En el método enviarMsg, se realiza una tarea parecida, pero esta vez creando el mensaje con parámetros:

```
Mensaje p = Mensaje.crearInputParaGrupo(nombreAgente, "EJEMPLO", "
   recibirMsg","MENSAJE A ESCRIBIR por : " + nombreUsuario);
ComunicacionAgentes.enviarInput("enviar_mensaje",p, nombreAgente,
   NombresPredefinidos.NOMBRE_GESTOR_APLICACION_COMUNICACION);
```

Esta vez, al crear el mensaje se incluyen los parámetros, que en este caso es el mensaje a mandar al resto de agentes. Al fijarse en el autómata, el usuario observará que el input "recibirMsg" para el estado "conectado" realiza la acción "recibirMsg" que es un método implementado en las acciones semánticas del agente y que recibe como argumento un String, que es el mensaje que se introduce en el código anterior como parámetro del mensaje.

Descripción de la aplicación. Fichero XML

Se aconseja que el usuario al comenzar a desarrollar una aplicación tanto distribuida como local con icaro utilice el fichero de descripción de aplicación que se incluye en la carpeta

`icaro-a3\config\icaro\aplicaciones\descripcionOrganizaciones\descripcionLimpiaParaDistribuido.xml`

En él se detalla el esquema a seguir y los datos necesarios para generar una descripción de aplicación correcta. El fichero de descripción del ejemplo es

descripciónAplicacionEjemploAprendizaje.xml.

Primero se explica la descripción para un nodo local. En las propiedades globales el usuario puede declarar el nombre usuario del nodo local:
`<icaro:propiedad atributo="nombreUsuario" valor="Arturo"/>`
A continuación, se declaran los gestores, agentes y recursos de manera análoga a como se hace usualmente, con la salvedad de que en los gestores hay que declarar también el nuevo gestor de comunicaciones:

```
<icaro:DescComportamientoGestores>
 <icaro:DescComportamientoAgente nombreComportamiento="
    GestorOrganizacion" rol="Gestor" tipo="Reactivo" />
 <icaro:DescComportamientoAgente nombreComportamiento="GestorAgentes"
    rol="Gestor" tipo="Reactivo" />
 <icaro:DescComportamientoAgente nombreComportamiento="GestorRecursos"
    rol="Gestor" tipo="Reactivo" />
 <icaro:DescComportamientoAgente nombreComportamiento="
    GestorAplicacionComunicacion" rol="Gestor" tipo="Reactivo"/>
</icaro:DescComportamientoGestores>
```

Este nuevo gestor también se debe añadir en las instancias de los gestores. Como norma general se aconseja incluir el recurso local en los componentes gestionados de este gestor.

Una vez declarados los agentes y recursos de la aplicación, en el caso del ejemplo son “usuario1” y “usuario2”, y “vis1”, en las instancias de dichos agentes el usuario debe especificar el nodo y el grupo al que pertenecen, si es que pertenecen a alguno. En el ejemplo:

```
<!-- Agente usuario1 -->
<icaro:Instancia id="usuario1"
refDescripcion="AgenteAplicacionEjemploAprendizaje">
 <icaro:listaPropiedades>
  <icaro:propiedad atributo="nodo" valor="nodoLocal" />
  <icaro:propiedad atributo="grupo" valor="EJEMPLO" />
 </icaro:listaPropiedades>
</icaro:Instancia>

<!-- Agente usuario2 -->
<icaro:Instancia id="usuario2"
refDescripcion="AgenteAplicacionEjemploAprendizaje">
 <icaro:listaPropiedades>
  <icaro:propiedad atributo="nodo" valor="nodoLocal" />
  <icaro:propiedad atributo="grupo" valor="EJEMPLO" />
 </icaro:listaPropiedades>
</icaro:Instancia>

<!-- Recurso vis1 -->
<icaro:Instancia id="vis1" refDescripcion="RecursoEjemplo" xsi:type="
   icaro:Instancia">
 <icaro:listaPropiedades>
  <icaro:propiedad atributo="nodo" valor="nodoLocal"/>
 </icaro:listaPropiedades>
</icaro:Instancia>
```

De esta forma tenemos que tanto el usuario1 como el usuario2 como el recurso se

ejecutan en el nodoLocal, es decir, no hay distribución alguna. Es aquí donde el usuario tendrá que realizar los cambios necesarios para distribuir su aplicación, tarea que se comentará más adelante.

Como aclaración al desarrollador hay que comentar que para obtener cierta información de este xml si hubiera necesidad, se cuenta con los métodos adecuados en el componente Directorio, esta clase agrupa métodos de uso frecuente que sin embargo se encuentran repartidos en varias clases, proporcionando un punto común para acceder a detalles como por ejemplo al repositorio o a la configuración, haciendo un poco más transparente al desarrollador su utilización. Dentro de la aplicación ejemplo, se obtiene el nombre del usuario especificado en el xml de la siguiente forma:

```
nombreUsuario = Directorio.getValorPropiedadGlobal("nombreUsuario");
```

Si bien es cierto que toda la información del xml se obtiene de manera transparente al usuario, con este componente se dota de la herramienta adecuada para manejar toda la información de la descripción a su antojo. Así, si el usuario quiere añadir una propiedad global a su descripción que sea por ejemplo "apellidos", sólo tiene que incluir dicha información en las propiedades globales de la siguiente manera:
`<icaro:propiedad atributo="apellidos" valor="Skywalker"/>`
Para posteriormente acceder a ella así:
`Directorio.getValorPropiedadGlobal("apellidos");`

Paso de local a distribuido

Si el usuario ha seguido las pautas anteriormente descritas, esta tarea es trivial. Al hacer pasar todas las comunicaciones por el GestorAplicacionComunicacion, sólo hay que declarar los nodos y especificar el nodo en el que se encuentra cada agente. En este ejemplo se arrancará cada nodo manualmente de forma independiente, sin emplear el gestor de nodo.

Como primer paso, hay que declarar los nodos a utilizar en las propiedades globales de la siguiente forma, donde atributo es el nombre de cada nodo, y valor su correspondiente ip:

```
<icaro:propiedad atributo="nodo1" valor="147.96.123.151"/>
<icaro:propiedad atributo="nodo2" valor="147.96.124.157"/>
```

A continuación se especifica en las instancias de agentes y recursos, el nodo al que pertenece cada uno de ellos. En el ejemplo:

```
<!-- Agente usuario1 -->
<icaro:Instancia id="usuario1"
refDescripcion="AgenteAplicacionEjemploAprendizaje">
 <icaro:listaPropiedades>
  <icaro:propiedad atributo="nodo" valor="nodo1" />
  <icaro:propiedad atributo="grupo" valor="EJEMPLO" />
 </icaro:listaPropiedades>
</icaro:Instancia>

<!-- Agente usuario2 -->
<icaro:Instancia id="usuario2"
 refDescripcion="AgenteAplicacionEjemploAprendizaje">
```

```
  <icaro:listaPropiedades>
   <icaro:propiedad atributo="nodo" valor="nodo2" />
   <icaro:propiedad atributo="grupo" valor="EJEMPLO" />
  </icaro:listaPropiedades>
 </icaro:Instancia>

 <!-- Recurso vis1 -->
 <icaro:Instancia id="vis1" refDescripcion="RecursoEjemplo" xsi:type="
    icaro:Instancia">
  <icaro:listaPropiedades>
   <icaro:propiedad atributo="nodo" valor="nodo1"/>
  </icaro:listaPropiedades>
 </icaro:Instancia>
```

En este caso, la ip especificada para el nodo1 en las propiedades globales resulta ser la ip local, de manera que el agente "usuario1" se ejecutará localmente y el agente "usuario2" en el nodo2 con la ip "147.96.124.157". Obviamente el usuario puede añadir todos los agentes y nodos que quiera en la descripción, desplegando la aplicación a su antojo.

Si se han seguido todos los pasos y se ha utilizado el método descrito, la distribución de una aplicación local es tan simple como esto.

El código fuente tanto del ejemplo como del resto del proyecto se encuentra en la dirección:

`https://svn.kenai.com/svn/icaro-a3~repositorio`

4.6. Detalles de Implementación

Comentaremos a continuación otros cambios que se han implementado, a menudo cambios pequeños pero que tienen un impacto apreciable sobre el funcionamiento de la infraestructura. Además se incluirá en este apartado las instrucciones acerca de cómo puede lanzarse una aplicación distribuida utilizando Icaro-D

4.6.1. Modificaciones de la Infraestructura

Aqui mostraremos los cambios que se han realizado sobre la infraestructura, bien para ampliar funcionalidades, o para modificar su comportamiento.

- Arranque: Se han hecho los cambios necesarios para reconocer las opciones que se explican con detalle en el apartado siguiente, para lanzar gestores de nodo y desplegar aplicaciones. Además, dado que se ha añadido un nuevo gestor, se ha modificado el autómata que controla las acciones del gestor de organización para arrancar el gestor de comunicaciones antes que los demás, esto evita fallos en caso de que se intente emplear este gestor para alguna comunicación durante el arranque del gestor de agentes, o de alguno de los agentes que éste último gestiona.

- Descripción de aplicación: Durante la implementación se ha intentado modificar lo menos posible la interpretación de las descripciones, de modo que las descripciones de aplicaciones anteriores a esta versión de Icaro fueran compatibles, o por lo menos que los cambios necesarios fueran mínimos. Finalmente se ha conseguido que el único cambio en la estructura sea la adición de un nuevo tipo de agente, llamado ReactivoRemoto, para que el gestor sepa, durante el arranque, que agentes deben ser agregados al registro RMI, y por lo tanto quedar visibles al exterior. En caso de utilizar el gestor de comunicaciones para el envio de mensajes, no es necesario que los agentes pertenezcan a este tipo, únicamente el propio gestor, que si tendrá como tipo ReactivoRemoto para permitir el envío de mensajes. Sin embargo se ha dejado esta opción por si los desarrolladores prefieren gestionar las comunicaciones de forma manual.

- Cambios en los patrones de agente: El patrón agente ha sido modificado para permitir la ejecución de métodos desde un cliente remoto, como se indica en la descripción de RMI,concretamente los metodos usados para pedir el nombre del agente y para aceptar un evento, el resto de métodos se considera que no deben ser accesibles desde fuera. Además se ha incorporado el método *enviarMensaje(...)*, que simplifica el envio de mensajes a través del gestor. También se ha añadido el método *trazar(...)*, que se encarga de localizar, si lo hay, el recurso de trazas y hacerle llegar la traza con el formato apropiado.

4.6.2. Arranque de aplicaciones

Aunque más adelante en la aplicación de ejemplo se indicará como lanzar una aplicación distribuida comentaremos aqui los distintos métodos existentes para arrancar una aplicación, usando las modificaciones implementadas en este proyecto.

- Arranque de una aplicación centralizada: Para lanzar una aplicación basada en Icaro-D sin partir de un entorno de desarrollo como Netbeans, se debe tener en cuenta los requisitos de la libreria, concretamente hay que copiar los directorios "config" y "schemas", siguiendo la misma estructura que tienen en el proyecto. El directorio "config" contiene las descripciones de aplicación para lanzar, mientras que el directorio "schemas" contiene los archivos necesarios para validar las descripciones. Obviamente debe incluirse el directorio "lib" con todas las librerias externas requeridas.

  Un ejemplo de como puede lanzarse una aplicación es con el comando:
  `c:>java -jar icaro-a3.jar descripcionAplicacionChat`
  Notese que no se usa la extensión de fichero xml para el nombre de la descripción, ya que la infraestructura lo añade por si misma. Para un entorno windows se recomienda crear un fichero batch e incluir el comando para lanzar la aplicación. Para esto basta con crear un nuevo fichero de texto, copiar el comando en el y cambiarle la extensión por .bat

- Arranque de una aplicación distribuida En el caso básico una aplicación distribuida se lanza de forma idéntica a una centralizada, los cambios se reflejan únicamente en el fichero de descripción, de modo que se ejecuta como se ha indicado más arriba. Para este tipo de arranque debe ejecutarse manualmente en

cada equipo la aplicación con su descripción de arranque, normalmente la misma para todos, aunque no es imprescindible, por tanto debe prepararse la aplicación para casos en los que un nodo no este listo todavía.

- Arranque de un nodo: Si lo que necesitamos es arrancar un nodo para que se quede una infraestructura básica en espera de lanzar alguna aplicación, solo tenemos que añadir la opción "-nodo", con lo que indicaremos al arranque que no deseamos lanzar ninguna aplicación, de momento.

- Desplegar aplicación: Si tenemos nodos en espera, y queremos lanzar una aplicación que los utilize, usaremos la opción "-desplegar". Este es el caso en que necesitaremos usar el gestor de nodo. El arranque hara lo siguiente: buscará en la descripción xml proporcionada que nodos se han declarado, y los intentará lanzar en orden. En el caso del nodo local su proceso será identico al que habria tenido de ser una aplicación centralizada, en otro caso intentará comunicar al gestor de nodo del otro extremo que desea lanzar una aplicación, descrita en el fichero xml. Nótese que lo único que se envia al gestor de nodo es el nombre de la descripción, por lo que el nodo deberá contener todo lo necesario para lanzar dicha aplicación, o el arranque de la aplicación fallará. Esto incluye tanto la infraestructura Icaro, como las fuentes de los agentes, recursos y dominio, y tambien el fichero con la descripción xml. El fichero xml deberá ser el mismo para todos los nodos.

5. Validacion

Para la validación del trabajo realizado se han implementado dos aplicaciones distribuidas con IcaroD. La primera consiste en un chat con una sala a la que se conectan una serie de clientes. La segunda es un modelo de toma de decisiones. A continuación se explicará en detalle cada una de ellas.

5.1. Aplicaciones ejemplo

5.1.1. Chat

Como primer ejemplo de desarrollo de una aplicación distribuida con icarod se hizo un pequeño chat a partir de cero que a continuación se explicará en detalle. Este chat consiste en una sala y una serie de clientes que se conectan a dicha sala, e intercambian mensajes entre ellos, siendo estos visibles a todos los clientes y a la sala. Es una aplicación sencilla pero que explota al máximo toda la funcionalidad de icarod, ya que se han utilizado la mayoría de los métodos y componentes desarrollados en el proyecto.

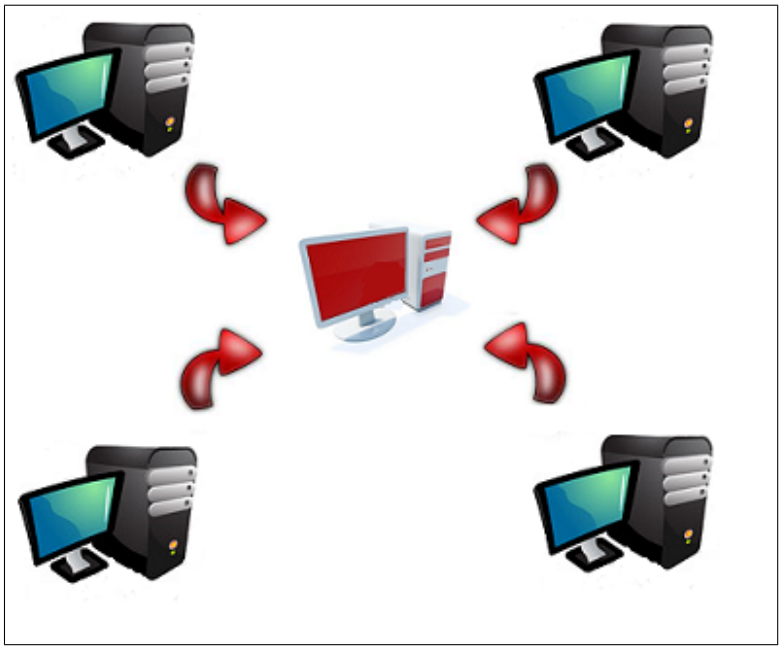


Figura 34: Cuatro clientes conectados entre si a traves de un servidor (Sala de chat)

Diseño

En esta aplicación hay dos tipos de agentes, la sala y el cliente. Cada uno de ellos tiene su comportamiento separado uno del otro pero totalmente complementarios. La comunicación entre sala y clientes se muestra en el diagrama de secuencia de la figura 35

Secuencia de la ejecución:

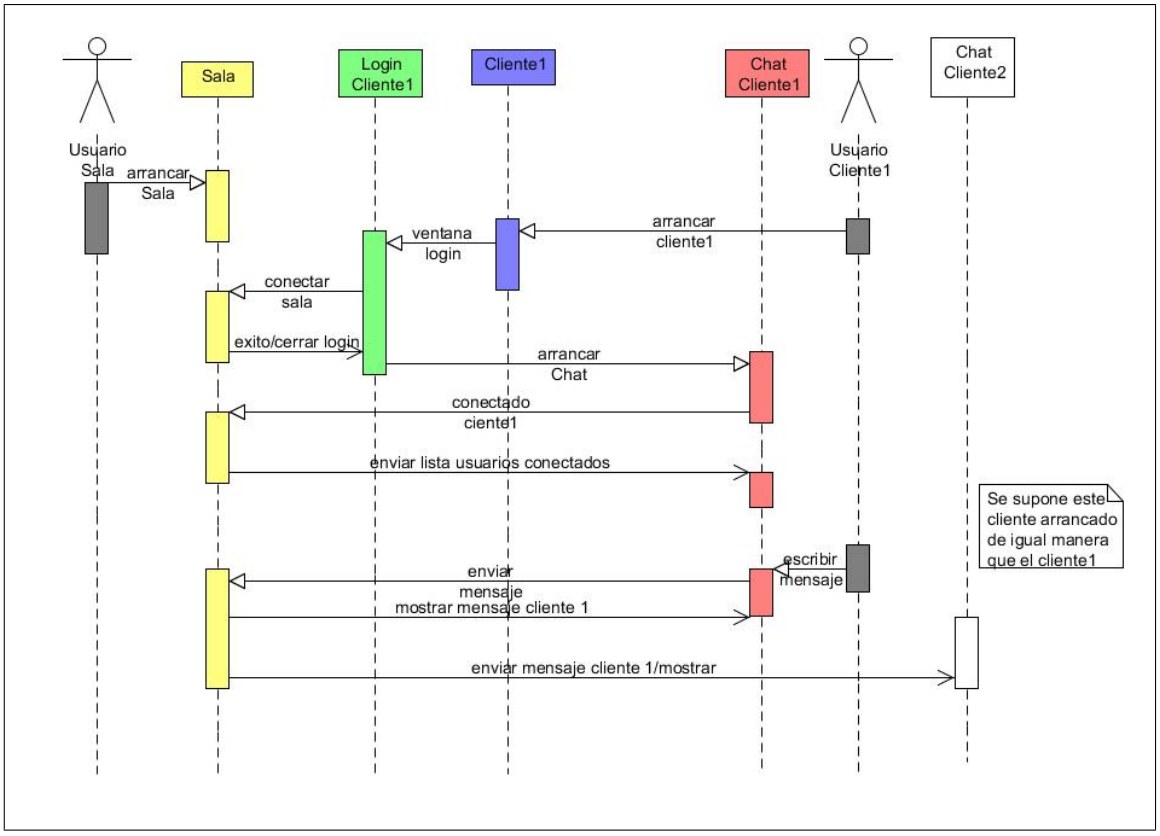
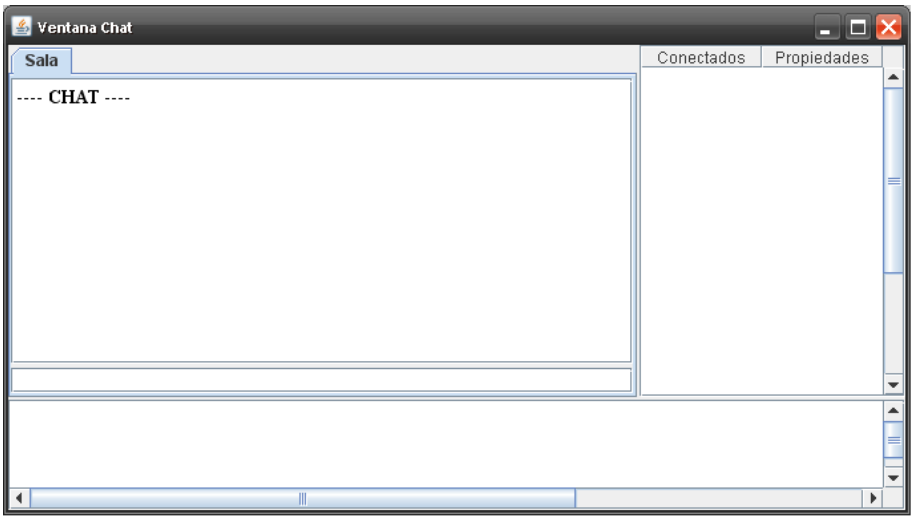


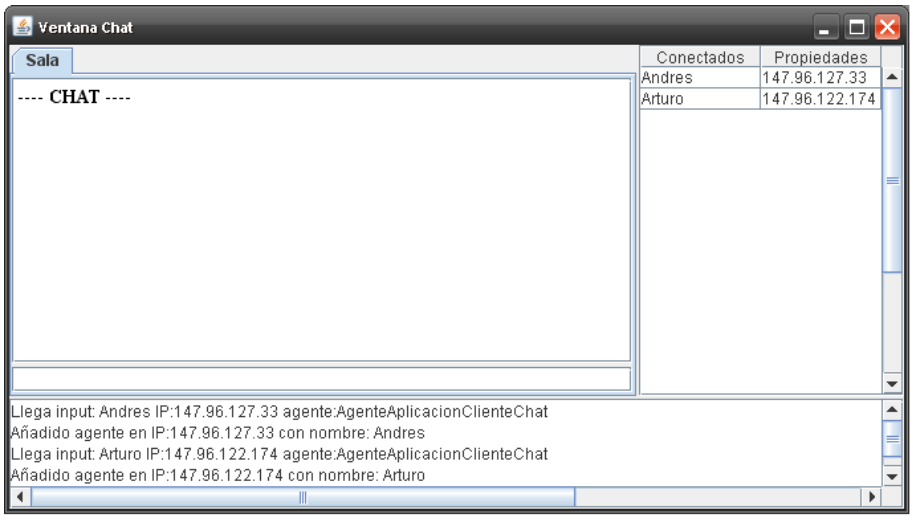
Figura 35: Diagrama de secuencia de la aplicacion chat

- Se arranca la sala

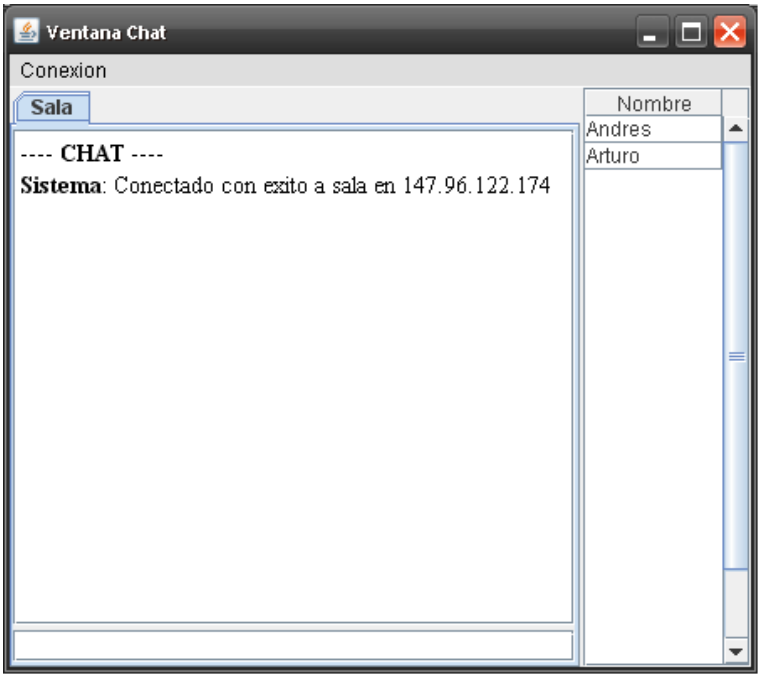


- Se arrancan los clientes y aparece una ventana de login para conectarse a la sala en la dirección ip adecuada y con el nombre de usuario deseado.

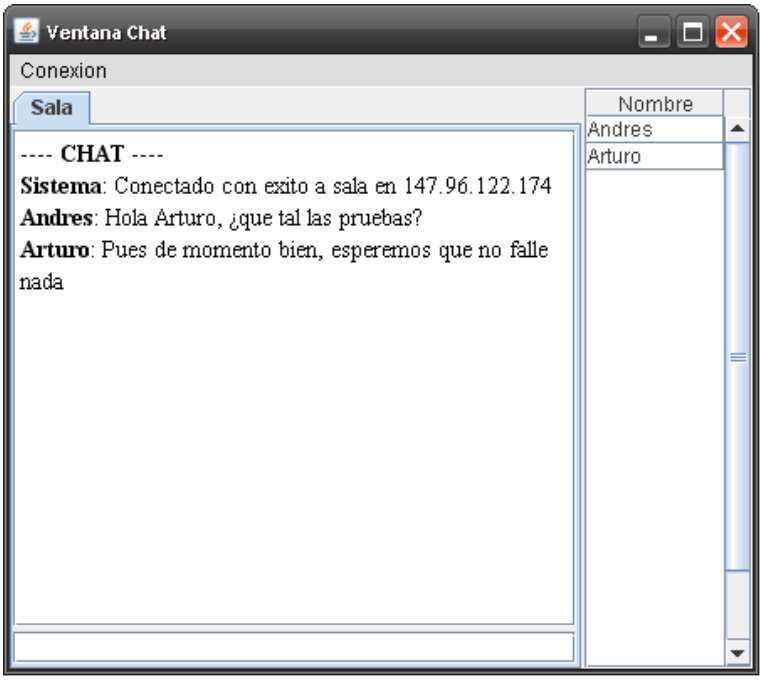
- Al logearse, el cliente se intenta conectar a la sala en cuestión. Si ocurriera un error se notifica con la causa, y si se produce la conexión con éxito se registra en la sala, se muestra la ventana del chat (cliente) y se añade el nombre de usuario de este cliente en la lista de conectados de esa sala. Esta lista de usuarios conectados se mandará a todos los clientes que se vayan conectando.



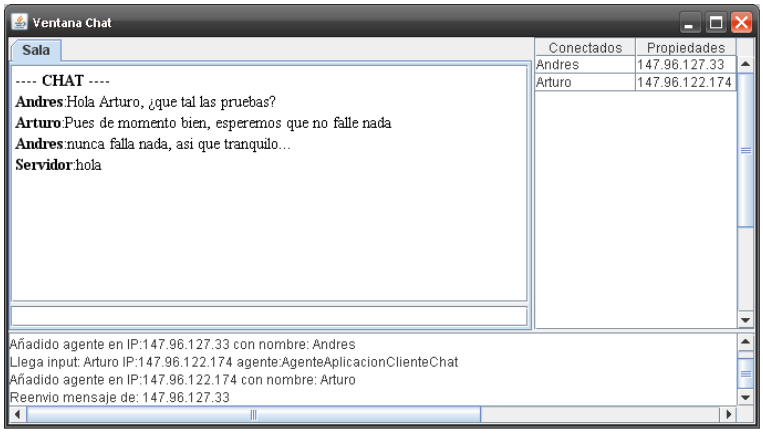
- Una vez conectado, la sala habilita el chat del usuario y le notifica el exito de la conexión y los usuarios que hay en la sala



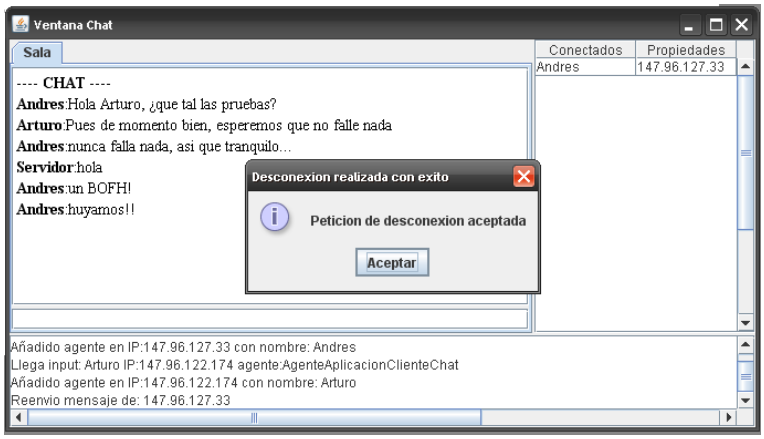
- Una vez conectados a la sala, los clientes pueden ya intercambiar mensajes a través de la sala. Esto es, cuando un cliente escribe un mensaje, éste se envía a la sala, quien lo reenvía a todos los clientes conectados a ella.



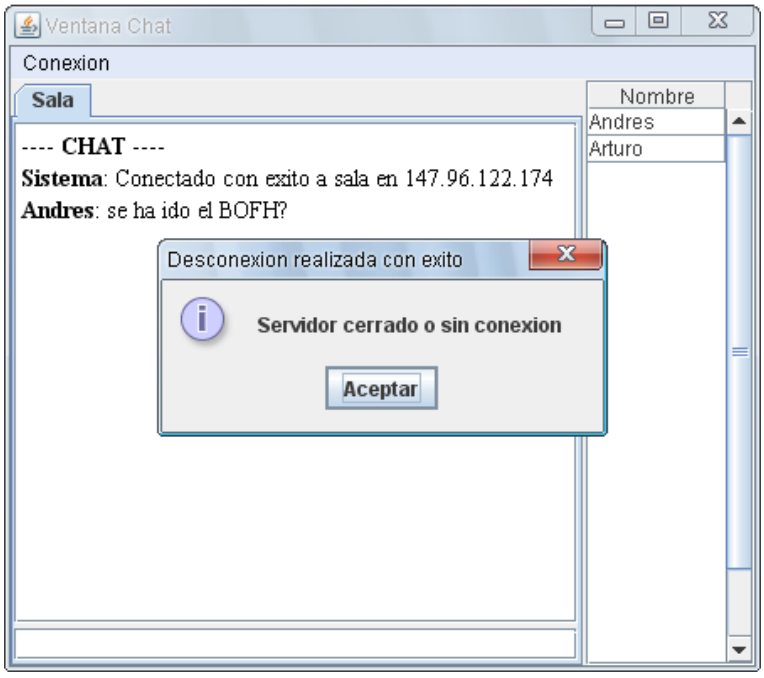
- En la sala se registran todos los eventos del chat tales como los mensajes escritos y las conexiones y desconexiones de clientes. Desde la sala también se pueden enviar mensajes a los clientes.



- Cualquier fallo en la conexión se notifica tanto a los clientes como a la sala. De esta manera, si se cierra la sala habiendo clientes arrancados y conectados a ella, los clientes reciben una notificación y se cierra la aplicación. Si un cliente cierra su ventana, vuelve a aparecer la ventana de acceso y se notifica a la sala su desconexión.



Y el mensaje que le llega al cliente:



Este comportamiento se describen por medio de los autómatas de sala y cliente, que se pueden ver en la figura 36 (sala), y en la figura 37 (cliente).

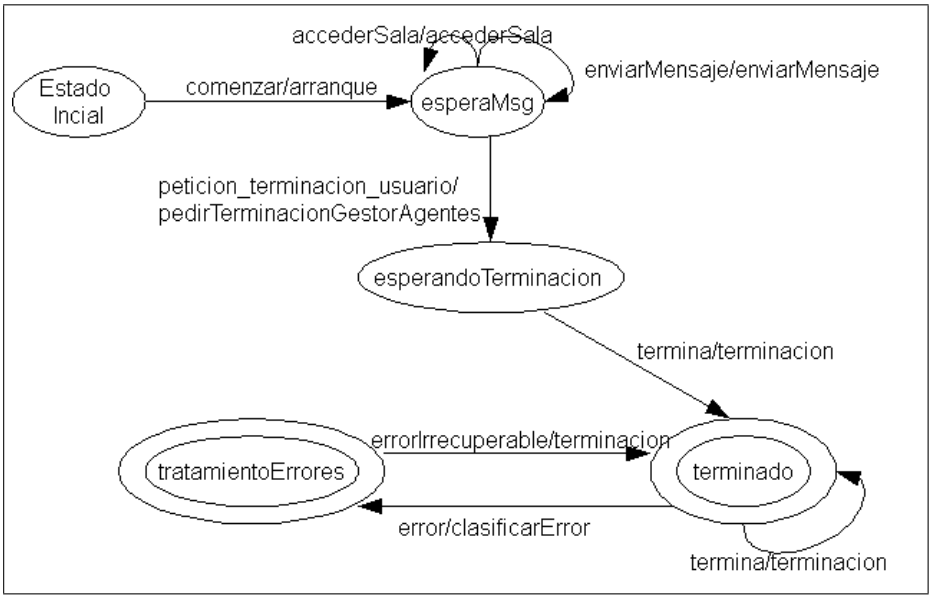


Figura 36: Automata de la sala de chats

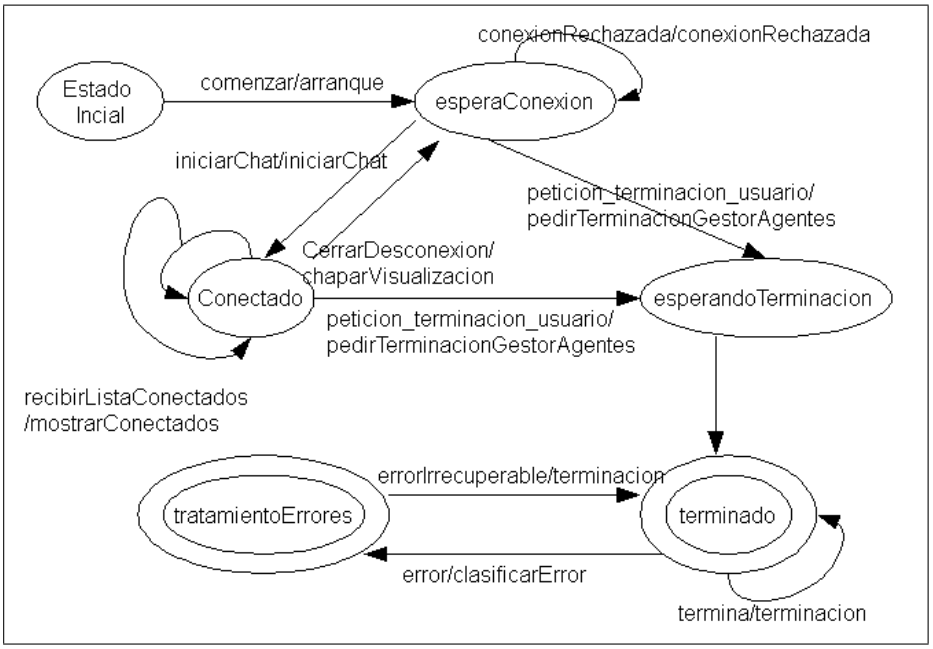


Figura 37: Automata del cliente chat

Como norma general para esta aplicación suponemos que hay un nodo que arranca la sala y un cliente de forma local, y el resto de nodos arrancan cada uno un cliente que se conectará a la sala en cuestión. Esta tarea se realiza en el fichero xml de descripción de aplicación de cada nodo. Para este ejemplo, el cliente sólo necesita saber donde está la sala, y ésta no necesita conocer la ubicación de los clientes. Esto último es debido a que en el momento en que un cliente se intenta conectar a la sala, le manda su ubicación en el proceso, y es en ese momento en el que el agente de la sala guarda dicha información para su posterior uso.

De esta manera, el fichero de descripción es el prácticamente igual tanto para la sala como para el cliente. Sólo se diferencian en la declaración de los nodos y en el tipo de componente que se quiera desplegar en cada nodo. Estos son los puntos clave de la descripción:

Propiedades globales

Para el nodo que ejecute la sala hay que declarar el nodoSala como nodoLocal y se podrá obviar el nombreUsuario ya que aunque aparezca en la descripción, la aplicación lo ignorará al estar ejecutando la sala:

```
<icaro:PropiedadesGlobales>
 <icaro:intervaloMonitorizacionGestores>15000</icaro:
    intervaloMonitorizacionGestores>
 <icaro:activarPanelTrazasDebug>false</icaro:activarPanelTrazasDebug>
<icaro:listaPropiedades>
 <!-- Chat nuevo -->
 <icaro:propiedad atributo="activarGestorComunicaciones" valor="true"/>
 <icaro:propiedad atributo="puertoRMI" valor="1099"/>
 <icaro:propiedad atributo="mostrarTrazas" valor="true"/>
 <icaro:propiedad atributo="nodoSala" valor="nodoLocal"/>
</icaro:listaPropiedades>
</icaro:PropiedadesGlobales>
```

Sin embargo para el nodo que ejecute el cliente, se tiene que declarar el nodo en el que está la sala y el nombre del usuario que se usará por defecto en la ventana de login:

```
<icaro:PropiedadesGlobales>
 <icaro:intervaloMonitorizacionGestores>15000</icaro:
    intervaloMonitorizacionGestores>
 <icaro:activarPanelTrazasDebug>false</icaro:activarPanelTrazasDebug>
<icaro:listaPropiedades>
 <!-- Chat nuevo -->
 <icaro:propiedad atributo="activarGestorComunicaciones" valor="true"/>
 <icaro:propiedad atributo="puertoRMI" valor="1099"/>
 <icaro:propiedad atributo="mostrarTrazas" valor="true"/>
 <icaro:propiedad atributo="nodoSala" valor="147.96.127.290"/>
 <icaro:propiedad atributo="nombreUsuario" valor="Arturo"/>
</icaro:listaPropiedades>
</icaro:PropiedadesGlobales>
```

Instancias de los agentes

En este punto sólo hay que tener en cuenta que hay que adecuar los nodos de cada instancia dependiendo del despliegue deseado. Por ejemplo, el cliente tiene las siguientes instancias de la siguiente manera:

```
<icaro:AgentesAplicacion>
<icaro:Instancia id="AgenteAplicacionClienteChat" refDescripcion="
   AgenteAplicacionClienteChat">
 <icaro:listaPropiedades>
  <icaro:propiedad atributo="nodo" valor="nodoLocal"/>
 </icaro:listaPropiedades>
</icaro:Instancia>
```

```
<icaro:Instancia id="AgenteAplicacionServidorChat" refDescripcion="
    AgenteAplicacionServidorChat">
 <icaro:listaPropiedades>
  <icaro:propiedad atributo="nodo" valor="nodoSala"/>
 </icaro:listaPropiedades>
</icaro:Instancia>
</icaro:AgentesAplicacion>
```

Como se puede observar, el agente cliente está en el nodo local y el agente sala en el nodo sala declarado en las propiedades globales.

Visualizacion

La declaración del recurso de visualización del chat es común a cliente y sala, ya que como se explicará posteriormente este recurso ejecuta una visualización distinta según sea el nodo sala o un nodo cliente.

```
<icaro:RecursosAplicacion>
<icaro:Instancia id="VisualizacionAgenteChat1" refDescripcion="
    VisualizacionAgenteChat" xsi:type="icaro:Instancia">
 <icaro:listaPropiedades>
  <icaro:propiedad atributo="nodo" valor="nodoLocal"/>
 </icaro:listaPropiedades>
</icaro:Instancia>
</icaro:RecursosAplicacion>
```

Implementación

Para la implementación de esta aplicación se han utilizado los componentes que se muestran en el diagrama de clases de la figura 38

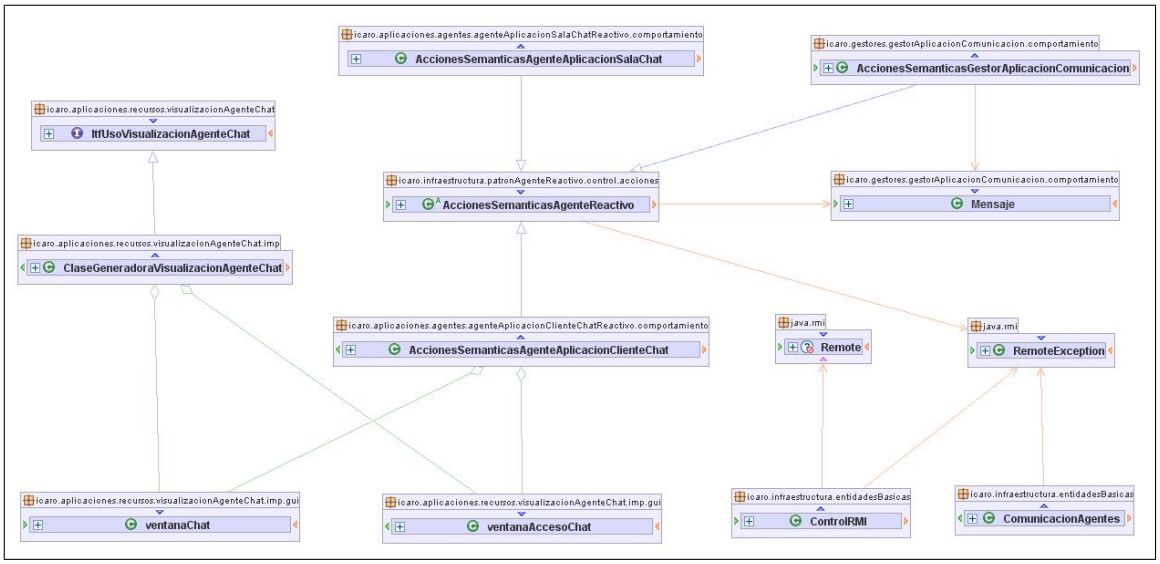


Figura 38: Diagrama de clases de la aplicación chat

Como se puede observar se usan las clases típicas de icarod: ComunicacionAgentes, ControlRMI, Mensaje y GestorComunicaciones. Se incluyen en el diagrama también las clases encargadas de los agentes (AccionesSemánticasClienteChat y AccionesSemánticasSalaChat) y el recurso (ventanaChat, ventanaAccesoChat,etc). En esta aplicación

se engloban todos los posibles métodos de comunicación distribuida desarrollados en icarod. Es decir, este chat es un ejemplo de validación de la mayor parte de la funcionalidad del proyecto. A continuación se adjunta un ejemplo de uso dentro del chat de las distintas formas de comunicación desarrolladas:

- Desde el patrón del agente se envía directamente un mensaje simple a través de controlRMI sin pasar por el gestor de comunicaciones: en el método enviarMensaje del agente sala.

```
public void enviarMensaje (String id, String mensaje) {
        /* Creo objeto a enviar */
        Object [] msg = {id,mensaje};
        /* Envio el mensaje a todos excepto el origen */
for (int i = 0; i < registroClientes.size(); i++) {
 Cliente c = registroClientes.elementAt(i);
 if (!c.id.equals(id)) {
  try {
                 sendRemoteInput("recibirMsg", msg, nombreAgente,
                    c.agente.getNombreAgente());
             } catch (RemoteException ex) { /* Omitido por
                claridad */ }
 }
}
    }
```

- Envío de un mensaje simple por medio del gestor de comunicaciones: en el método finalizarChat del agente cliente.

```
public void finalizarChat(){
        Object [] params = {nombreCliente, miIP, nombreAgente};
        // Input para el servidor del chat
        Object [] input = {"peticionDesconexion", params,
           nombreAgente, agenteServidor, sala, miIP};
        trazar ("Enviada peticion desconexion: [Nombre:"+
           nombreCliente+" | ipLocal:"+miIP + " | Agente:"+
           nombreAgente+"]");
        // Envio input empaquetado a mi gestor de comunicaciones
           para que lo trate
        ComunicacionAgentes.enviarInput("
           enviar_msg_a_destinatario_conocido", input,
           nombreAgente,gestorComunicaciones);
    }
```

- Envío de mensajes por medio del gestor de comunicaciones (método recomendado) : en el método conectar del agente cliente.

```
public void conectar () {
        trazar ("Recibida peticion conectar");
         // Saco los datos del login
         nombreCliente = login.getNombre();
         sala = login.getSala();
            // Averiguo mi propia IP
            try {
                 trazar ("Intento conexion");
```

```
                    miIP = InetAddress.getLocalHost().getHostAddress
                        ();
                    // Esta vez voy a usar un mensaje, que es la
                        forma mas sencilla de mandar inputs
                    Mensaje p = Mensaje.crearInputBasico(nombreAgente
                        , agenteServidor, "peticionConexion",
                        nombreCliente, miIP, nombreAgente);
                    // Le pido a comunicacion agentes que envie el
                        mensaje a mi gestor
                    ComunicacionAgentes.enviarInput("enviar_mensaje",
                         p, nombreAgente, gestorComunicaciones);
                    trazar ("Enviada peticion conexion: [Nombre:"+
                        nombreCliente+" | ipLocal:"+miIP + " | Agente
                        :"+nombreAgente+"]");
                }
                catch (Exception imposibleConectar) {
                    JOptionPane.showMessageDialog(null, "clienteChat:
                         imposible conectar (Excepcion en conectar)");
                    trazar ("Excepcion en conectar");
                }
            }
```

5.1.2. MasterIA

Esta aplicación se ha usado de ejemplo ya que a diferencia del chat, que fue desarrollada por los integrantes del proyecto, se distribuyó con IcaroD a partir de su versión local con IcaroMini. Para su explicación se va a usar gran parte del trabajo realizado por Daniel Garijo Verdejo en su trabajo "Agentes Inteligentes y Sistemas Multiagentes".

Descripción del problema

El problema radica en modelar el comportamiento de varios robots (o agentes) del mismo tipo para que lleguen a un acuerdo entre sí ante una situación específica. En concreto, cuando se están dirigiendo a un objetivo y se les envía la orden de que un miembro del grupo se dirija a un objetivo distinto del original.

En [1] se propone un protocolo de comunicación para resolver esta situación, que es el que se ha utilizado a la hora de realizar la implementación, (simplificado). Dicho comportamiento se puede observar en la figura 39

En la figura 39 podemos ver cómo el agente está realizando una tarea inicial (1) y le llega la tarea de ir a un nuevo objetivo (2). Basándose en sus parámetros internos (energía, distancia al objetivo, riesgo, etc.) (3), calcula su aptitud para atender al nuevo objetivo mediante una función de evaluación (4). Tanto si no puede ir al objetivo como si puede, envía la información a los robots cercanos e intercambia con ellos los resultados obtenidos por la función de evaluación (5). Una vez han llegado a un consenso, todos saben quién es el más apto, y el robot adecuado se dirigirá al nuevo objetivo (6).

En caso de empatar, basta con dar más peso a determinados parámetros frente a otros en la función de evaluación. Los robots que hayan empatado mantendrán una discusión tal y como se muestra en el siguiente diagrama de secuencia, (en figura 39),

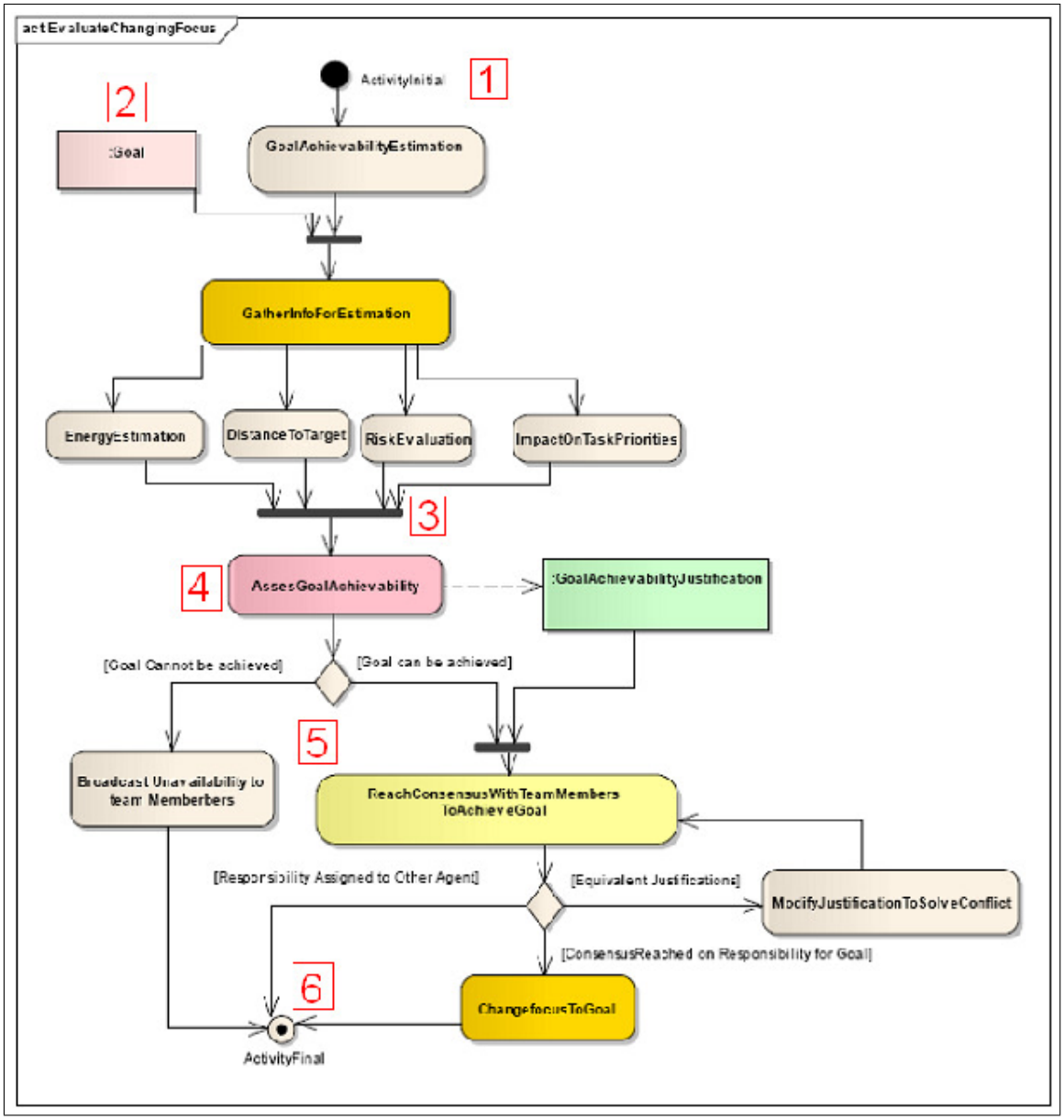


Figura 39: Diagrama de actividad MasterIA

en términos de las acciones de comunicación de tipo Inform:

La solución que se ha utilizado en la implementación está simplificada. No se ha compuesto una función de evaluación basada en batería, distancia, dificultad de llegar al objetivo, etc.; sino que se le ha asignado a cada robot un valor. Se ha elegido así porque el problema que se quiere resolver es cómo se ponen de acuerdo los robots variando la función de evaluación, no cómo se realiza la construcción de la misma.

Resolución

El equipo que debe de tomar las decisiones está formado por cuatro agentes (o robots). Todos ellos poseen el mismo comportamiento, y están modelados por una máquina de estados, por lo que se trata de agentes reactivos. Gracias al uso de la plataforma Icaro, podemos especificar de manera simple la máquina de estados del agente y sus acciones semánticas asociadas, por lo que encontrar la solución al prob-

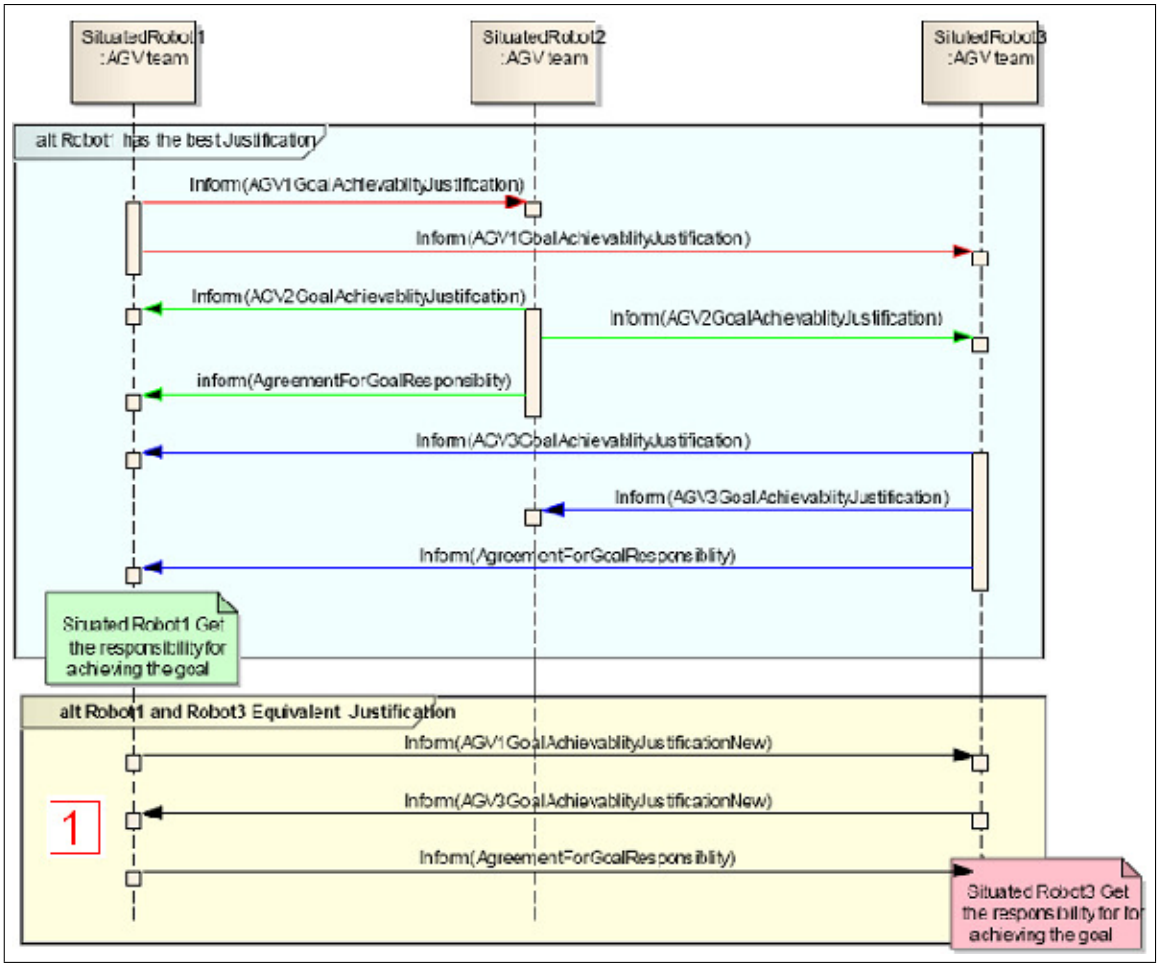


Figura 40: Diagrama de secuencia MasterIA

lema radica en modelar la máquina de estados de manera adecuada para que sigan el protocolo de [1] y lleguen a una decisión correcta. En la figura 41 se explica con detalle los estados y transiciones mediante los que se rige el agente:

Los estados I e I2 son estados iniciales. I viene definido en la plataforma por defecto, (arranca el agente), e I2 simula que llega la orden de tomar la decisión de ir al objetivo nuevo o no, por lo que desencadena que se envíe la evaluación del agente al resto del equipo. Una vez enviadas, se pasa a esperar las respuestas de todos los agentes que forman el equipo, (estado E).

En el estado E pueden ocurrir varias alternativas:

- El agente recibe una respuesta: se comprueba si el agente tiene todas las que estaba esperando. En caso de tenerlas todas se notifica al propio agente y se sigue en el mismo estado, (se transitará cuando llegue la notificación, más adelante).
- El agente recibe todas las respuestas del resto del equipo, y no tiene la mejor evaluación: manda una confirmación al que considera mejor agente y pasa al estado final O1, que se corresponde a ir al objetivo original.
- El agente recibe todas las respuestas del resto del equipo, y tiene la mejor evaluación: se transita al estado E1, en el que se queda esperando las confirmaciones

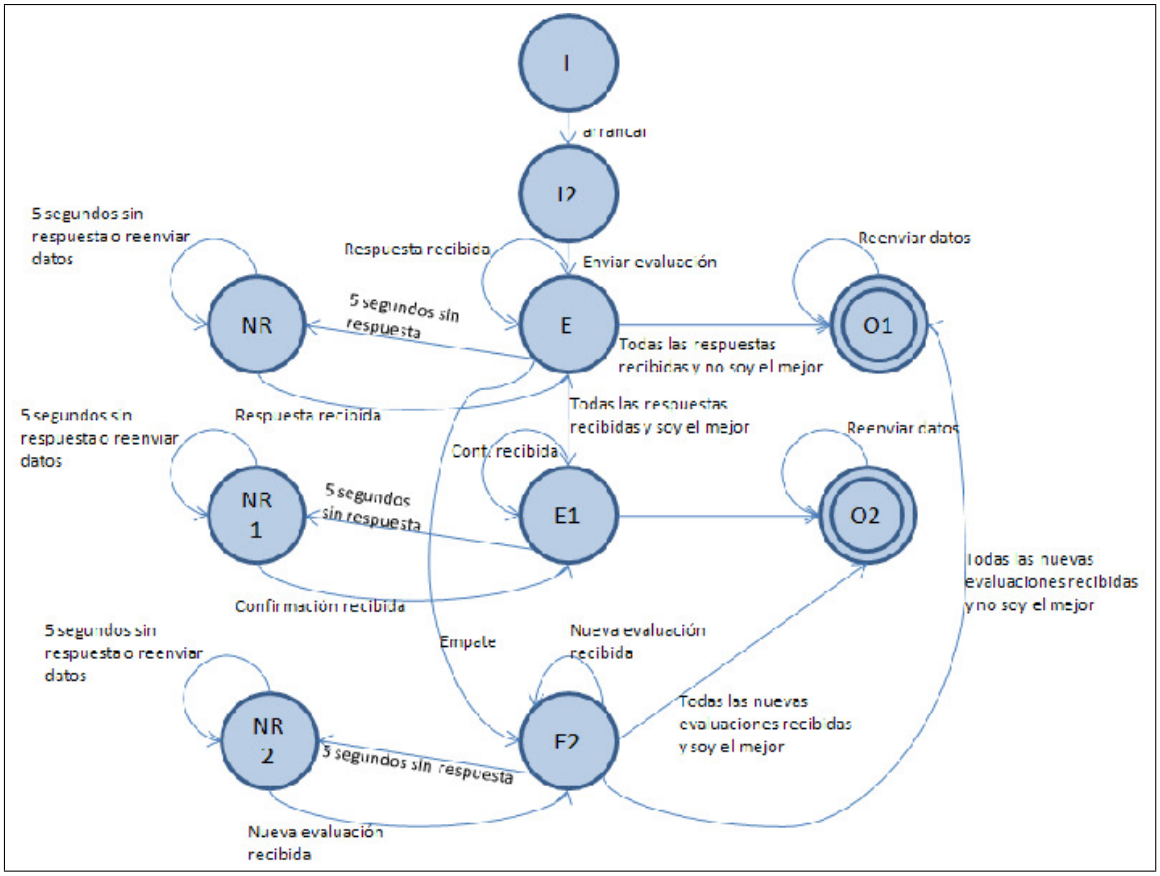


Figura 41: Diagrama de estados MasterIA

por parte del resto del equipo.

- El agente recibe todas las respuestas del resto del equipo, y tiene la mejor evaluación empatando con uno o varios miembros: se transita al estado E2, en el que se pasan a esperar las nuevas evaluaciones de los agentes del equipo. Según [1], la nueva evaluación se corresponde a un reajuste de parámetros en la función de evaluación, (dando más importancia a algunos atributos frente a otros). En nuestro caso, se ha escogido como nueva evaluación el id del agente, siendo el menor id la mejor evaluación.

- El agente lleva esperando 5 segundos y no ha recibido respuesta: pasa al estado NR, en el que pide al resto de los agentes de los que no ha recibido respuesta que se la reenvíen. Si recibe una respuesta, vuelve al estado E. Los estados NR, NR1 y NR2 tienen el mismo fin: indican que el agente lleva 5 segundos esperando su evaluación, confirmación o nueva evaluación y proceden a pedírsela al agente implicado directamente. E1 y E2 son muy parecidos a E. E1 espera las confirmaciones, y cuando las recibe todas, se dirige al estado O2, que se corresponde a ir al objetivo nuevo. En cambio en E2 se resuelven los empates: cada agente envía y recibe las nuevas evaluaciones y toma la decisión de dirigirse al objetivo original o al nuevo en función de las mismas. En ambos estados, si no se recibe una respuesta en 5 segundos, se transita a los estados NR1 y NR2 respectivamente. Aunque no se ha indicado en todos los estados para no complicar el diagrama, la mayoría de los estados pueden recibir una petición de reenvío de datos, (tanto de

evaluaciones, confirmaciones o nuevas evaluaciones). Dicha petición no implica un cambio de estado, tan sólo el reenvío del dato indicado al agente que lo pide.

A la hora de implementar el comportamiento en la plataforma Ícaro, se ha supuesto que un agente conoce previamente a los que componen su equipo. El desarrollo se ha realizado incrementalmente para comprobar la corrección, en tres fases. En una primera fase se desarrolló el intercambio de evaluaciones, (estados I, I1, E, NR), para después incorporar las confirmaciones, (estados NR1, E1, O1, O2). En último lugar se desarrollaron los empates, (estados NR2, E2).

Distribucion del ejemplo

Para la distribución de esta aplicación local se han seguido los siguientes pasos.

- Primero se ha modificado el fichero de descripción xml para adecuarlo a las necesidades de IcaroD. Así, según se ha visto en apartados anteriores, se han declarado los nodos en que se desplegará la aplicación (en este caso dos, con dos robots cada uno), se ha añadido el gestor de comunicaciones a la declaración e instancias de los gestores, y en cada una de las 4 instancia del agente se ha añadido el nodo y grupo al que pertenece. En este caso, los 4 robots pertenecen al mismo equipo, por lo que el grupo será el mismo para las 4 instancias, "ROBOTS".
- Segundo, se han realizado los cambios necesarios en las Acciones Semánticas del agente de manera que todas las comunicaciones pasen por el Comunicador de agentes y el Gestor de Comunicaciones. Como en esta aplicación se envían mensajes a un grupo de agentes, se ha usado el envío de mensajes a grupos por medio del Gestor de Comunicaciones descrito en los apartados de diseño e implementación.

Este el método de enviar a un sólo agente:

```
public void mandaMensajeAAgenteId(String input,Object infoComplementaria
    , String IdenAgenteReceptor){
try {
 Mensaje p;
 if (infoComplementaria != null) {
 p = Mensaje.crearInputBasico(nombreAgente, IdenAgenteReceptor, input,
     infoComplementaria);
} else {
 p = Mensaje.crearInputBasico(nombreAgente,IdenAgenteReceptor, input);
 }
ComunicacionAgentes.enviarInput("enviar_mensaje",p, nombreAgente,
NombresPredefinidos.NOMBRE_GESTOR_APLICACION_COMUNICACION);
 } catch (Exception e) { }
 }
```

Y este el método que envía un mensaje a todos los agentes del equipo:

```
public void mandaMensajeATodos(String mensaje, Integer evaluacion){
try {
 ArrayList mensaje = new ArrayList();
 mensaje.add(evaluacion.toString());
```

```
  mensaje.add(this.nombreAgente);
  Mensaje p = Mensaje.crearInputParaGrupo(nombreAgente,"ROBOTS", mensaje,
      mensaje);
            ComunicacionAgentes.enviarInput("enviar_mensaje",p,
               nombreAgente ,NombresPredefinidos.
               NOMBRE_GESTOR_APLICACION_COMUNICACION);
         } catch (Exception e) {}
     }
```

Como se puede observar, en ambos métodos se crea el mensaje a enviar con los parámetros requeridos y se ordena al Comunicación Agentes que se lo mande al Gestor de Comunicaciones del nodo, y que él se busque la vida. Por eso, ahora para todas las comunicaciones entre robots se usa el Gestor de Comunicaciones. Es éste el que se encarga de primero buscar localmente y después remotamente. Si el destino es local, se realiza el envío del mensaje por medio del Comunicación Agentes. Si por el contrario es remoto, se envía el mensaje a través del ControlRMI al Gestor de Comunicaciones del nodo destino, quien buscará localmente en su nodo al destino y si lo encuentra, le enviará el mensaje. En el diagrama de secuencia de la figura 42 se muestra el proceso de enviar la evaluación a todos los robots.

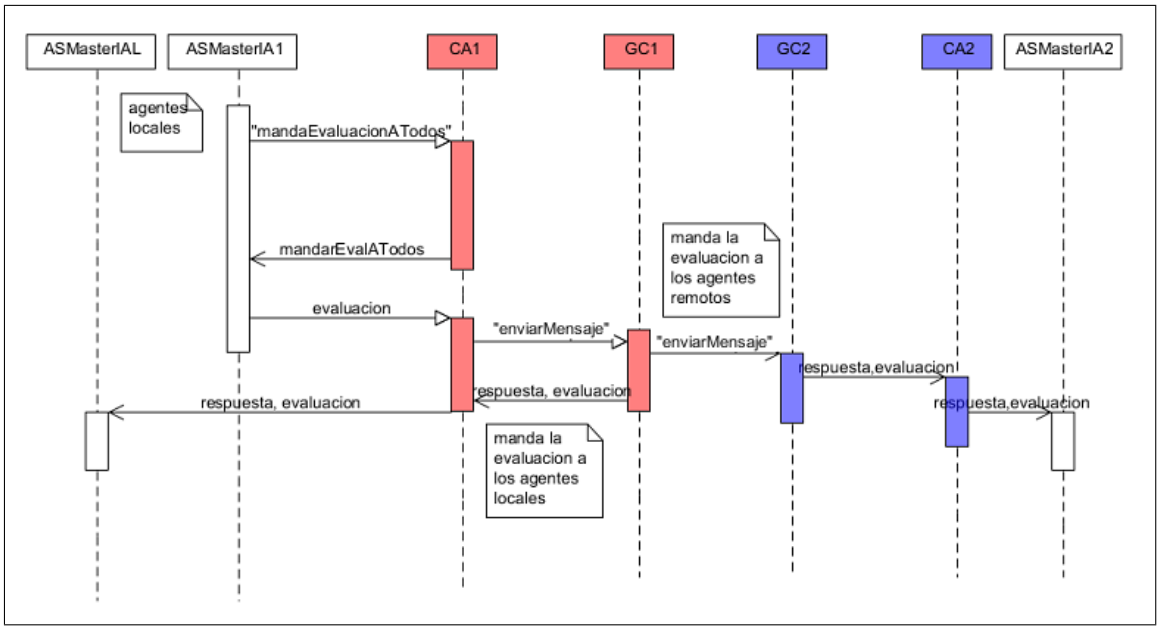


Figura 42: Diagrama de secuencia para la evaluacion MasterIA

En la figura anterior, ASMasterIA es AccionesSemanticasMasterIA, CA es ComunicacionAgentes y GC es GestorComunicaciones. Sólo se representa una tarea de las muchas que realiza la aplicación ya explicadas. Este es sólo un ejemplo del funcionamiento distribuido del MasterIA. El resto de tareas siguen la misma secuencia de acciones. Puede verse un diagrama de clases ilustra los componentes usados en la aplicación en la figura 43

Una vez hecho esto, la aplicación ya está distribuida. Se puede observar que el autómata de la aplicación no se ha modificado, esto es debido a que la comunicación distribuida es totalmente transparente al usuario. Como tarea opcional se podría haber añadido más estados al autómata del agente para control de errores remotos en la

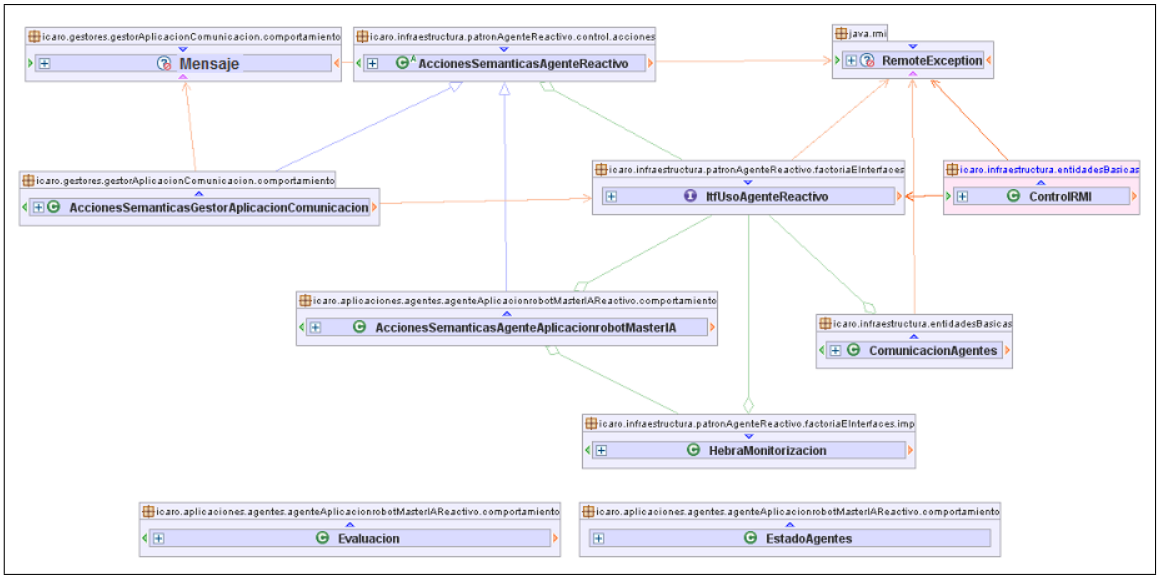


Figura 43: Diagrama de clases MasterIA

aplicación. Pero al ser este un ejemplo de cómo pasar de local a distribuido y al estar diseñada e implementada dicha tarea en el chat, se ha decidido obviar esa parte en esta aplicación. Las trazas resultantes pueden verse en las figuras 44 y 45.

Ejemplos de ejecución

- *Agentes 2,3 y 4 empatan*
  En este caso se ha dado dado la mejor evaluación a tres agentes distintos: el agente 1 tiene 107, y los agentes 2, 3 y 4 109.

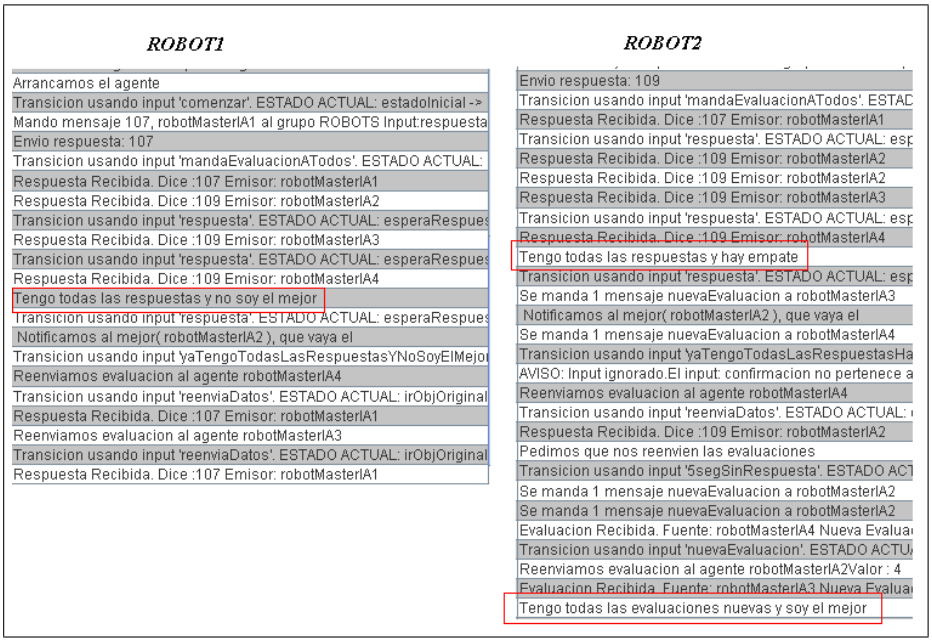


Figura 44: MasterIA, Trazas robots 1 y 2 para el caso de empate

El resultado del triple empate es que el agente de menor id, el agente 2, se dirige al objetivo, mientras que los agentes 3 y 4 se dirigen al original. Los agentes

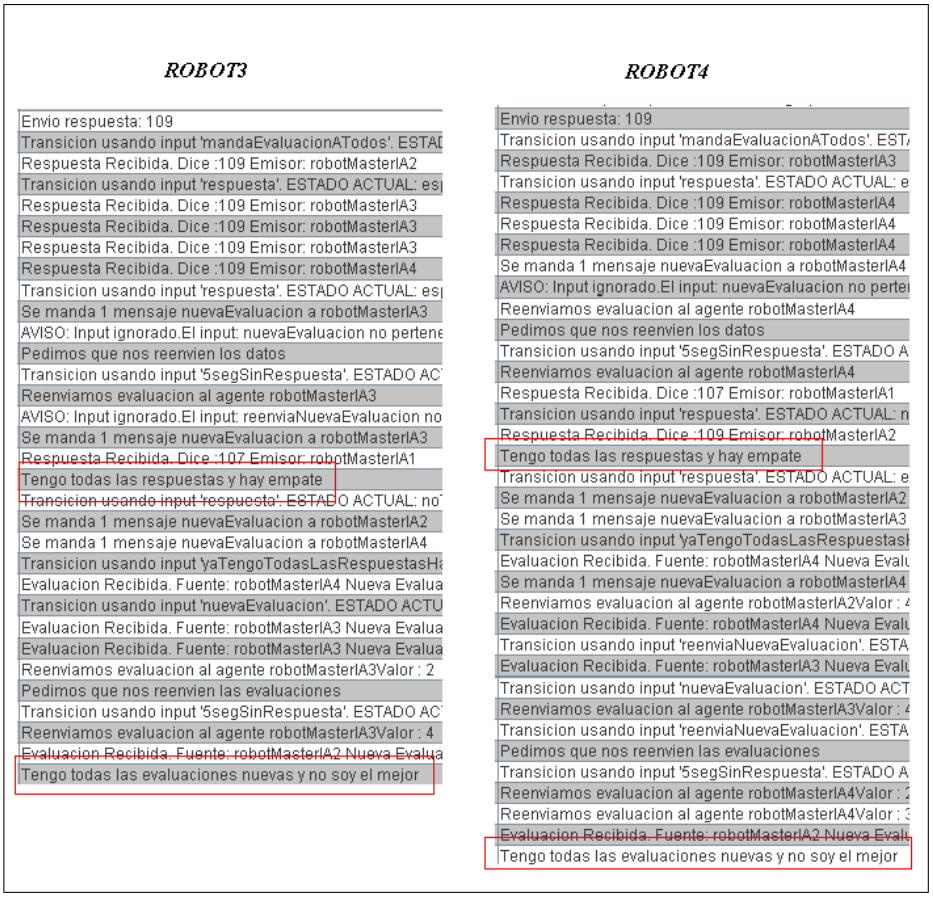


Figura 45: MasterIA,Trazas robots 3 y 4 para el caso de empate

envían las evaluaciones tal y como en el ejemplo 1, pero al recibir todas las respuestas ven que los tres tienen la mejor evaluación. Proceden a enviarse la nueva evaluación respectivamente, y cuando las reciben todas se dirigen al objetivo que les corresponde. No se espera confirmación del resto de los agentes, (en este caso, del agente 1).

- *El agente 4 tiene la mejor evaluación*
  layout Para la realización de esta ejecución, se han elegido los valores 107 para el agente 1, 108 para el 2, 107 para el 3 y 109 para el 4. En la siguiente imagen se puede ver el resultado de la ejecución en los agentes 1 y 4, (no se han incluido las trazas de los agentes 2 y 3 porque son casi idénticas a las de 1).

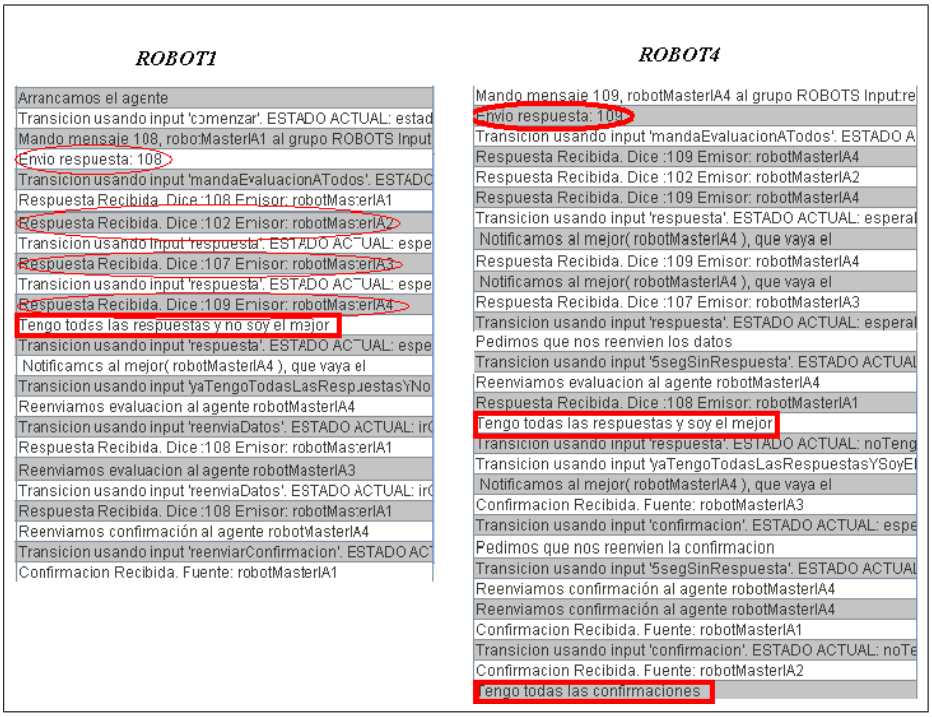


Figura 46: Trazas de la comunicación entre robots del ejemplo MasterIA sin empates

6. Conclusiones y trabajo futuro

6.1. Conclusiones

Se ha integrado con éxito la distribución de componentes de Icaro nodos de una red de computadores mediante el nuevo gestor de comunicaciones. Además se han implementado las mejoras y modificaciones a Icaro que permiten a este componente cumplir con su función. Se ha integrado un nuevo sistema de visualización para las trazas, capaz de ordenar y filtrar las trazas generadas con la infraestructura, lo cual se ha comprobado durante el desarrollo del proyecto, simplifica la depuración de las aplicaciones desarrolladas sobre esta plataforma.

Este trabajo ofrece a los usuarios de esta plataforma la ventaja de poder crear aplicaciones distribuidas, ocultando gran parte del trabajo inherente a este tipo de desarrollo. Y ofreciendo una herramienta, la ventana de trazas, que alivia la tarea de depurar el trabajo realizado. Finalmente las mejoras del código fuente simplifican el uso de la plataforma, centralizando el acceso a las funcionalidaddes ofrecidas por la plataforma, como son el recurso de trazas, el repositorio de componentes o la configuración de Icaro.

Se ha logrado cumplir con el objetivo principal del proyecto, que era la integración en la plataforma Icaro de la capacidad de distribuir y comunicar sus componentes entre los nodos de una red. Únicamente han quedado sin cumplir dos objetivos: la representación gráfica de los componentes y la edición de los protocolos de comunicaciones. Sin embargo el primero se considera un objetivo secundario, y el segundo fue descartado durante el desarrollo por considerarlo innecesario a la vista de las necesidades reales de la implementación. Además la aplicación de simulación de redes se consideró demasiado extensa para tratarse de un simple ejemplo de validación, siendo reemplazada por la aplicación de chats, considerada más didactica.

Nuestra contribución a la infraestructura Icaro supone una notable mejora sobre la versión de la que partimos originalmente, proporcionando una plataforma más completa. Además, la experiencia de los autores durante la fase de validación, en la que han sido desarrolladas dos aplicaciones de software distribuido, ha sido que se ha cumplido con el objetivo de permitir realizar aplicaciones distribuidas de forma sencilla, conseguiendo que la mayoría del trabajo de comunicaciones sea transparente a los usuarios.

6.2. Trabajo futuro

Tras finalizar el proyecto encontramos algunos puntos que creemos que podrían desarrollarse más en profundidad, asi como otros que hemos identificado como posibles mejoras de la infraestructura Icaro, a continuación vamos a indicar cuales han sido:

6.2.1. Tratamiento de Trazas

Durante el desarrollo de una aplicación con Icaro se generan una enorme cantidad de trazas, la actual visualización permite hacer filtrados de las mismas y ordenarlas

según varios criterios, sin embargo, en ocasiones querremos guardar las trazas en un fichero para consultarlas a continuación, cosa que aunque esta implementada, tiene un serio defecto, y es la escasa legibilidad del fichero, es dificil encontrar las cosas, y no hablemos ya de establecer comparaciones o buscar patrones de ejecución.

Por tanto, la mejora propuesta consiste en guardar las trazas en formato xml, y permitir abrirlas en una visualización, similar a la ventana de trazas, pero sin necesidad de encontrarnos en un entorno de ejecución de la propia aplicación. Las mejoras sugeridas serian las siguientes:

- Guardar las trazas en xml: Utilizando xml se podría definir un formato que simplificara recorrer las trazas generadas, además, este estándar permitiría desarrollar aplicaciones externas que puedan tratar las trazas de diferentes maneras, pudiendo seguir la estructura original con la que fueron creadas.
- Separar las trazas de su visualización: Si las trazas se guardan de forma independiente, se simplificaría la creación de visualizaciones más sofisticadas sin necesidad de tocar los datos. Esto está parcialmente contemplado, ya que el componente JTable de Java, utilizado para mostrar las trazas, ya tiene separado el modelo de datos (en nuestro caso un simple *defaultTableModel*) de la visualización (el propio JTable). Esto puede aprovecharse simplemente extendiendo el componente JTable de Java, para admitir distintas visualizaciones, por ejemplo, una visualización simple con los datos más importantes, y otra más detallada.
- Lanzar la visualización de trazas indendientemente del resto de la infraestructura: Con esto se permitiría cargar un fichero de trazas xml sin necesidad de cargar el equipo de objetos innecesarios, es decir, implementar el recurso de trazas como un componente independiente del resto, que no obstante es utilizado desde la implementación de Icaro como un recurso.
- Implementar cronograma: Esta era una de las ideas originales que estaban planificadas, sin embargo, dado que se trataba de un trabajo separado de la línea principal de proyecto, finalmente no ha habido tiempo para poder implementar un cronograma realmente útil. La idea sin embargo consistía en disponer de un gráfico, actualizable en tiempo real, donde se muestren los cambios producidos en los estados de los agentes trazados, asi como los mensajes que van recibiendo, lo que permitiria comprobar de forma sencilla fallos en la recepción de mensajes o en la transición de estados.

6.2.2. Gestión de fallos en nodos

Actualmente la gestión de fallos se limita a indicar qué ha ocurrido, y parar la ejecución. Dado que las aplicaciones distribuidas son candidatas especialmente sensibles a sufrir fallos impredecibles, como cortes en la línea, debería preparse la infraestructura para reaccionar de forma adecuada, cambiando el actual mecanismo simple de reintentos por otros más inteligentes capaces de determinar el fallo y actuar en consecuencia.Por ejemplo, redistribuyendo la aplicación, reiniciando el nodo caído en otro disponible y haciendo reconverger la red para que los mensajes pendientes se redirijan correctamente.

6.2.3. Control de la carga de comunicaciones

Dado que es probable que muchas de las comunicaciones realizadas sean a través de redes lentas, por ejemplo internet, sería positivo disponer de un mecanismo que permita realizar las comunicaciones de forma eficiente, buscando las mejores rutas para los mensajes. Por ejemplo puede que dados tres nodos, sea mas rápido que A envie los mensajes a C través de B en lugar de enviarlos directamente a C. Esto permitiria implementar la gestión de fallos en nodos comentada anteriormente, ya que podria permitir calcular las rutas más cortas, asi como determinar, caida una ruta, si existe una alternativa viable, de forma parecida, tal vez, al protocolo de redes OSPF (*Open Shortest Path First*).

6.2.4. Arranque parcial

Actualmente un fallo durante el arranque de algún componente supone un fallo de todo el sistema, y esto no es deseable. Una posible mejora del arranque sería poder indicarle qué componentes son imprescindibles para darle una funcionalidad mínima, cuáles son necesarios para completar sus funciones, e incluso cuáles están disponibles de forma redundante. Esto permitiría, por ejemplo, probar nuevas versiones de componentes existentes, marcando la versión antigua como un *back-up*, o seguro, y en caso de fallo del nuevo componente, la versión original pueda tomar el control de sus tareas y sustituirle.

6.2.5. Ventana de arranque

Normalmente, durante el arranque se puede ver lo que está ocurriendo en la ventana de trazas, sin embargo, a la hora de empaquetar una aplicación creada con Icaro para su distribución a los usuarios finales, dicha ventana de trazas suele desactivarse. Esto deja a los usuarios sin información acerca del resultado de la ejecución, no se informa de ningún tipo de fallo ni se muestra el estado de la carga de la aplicación. En caso de aplicaciones con tiempo de carga largos esto puede resultar algo confuso.Para evitar ese problema se debería incorporar una ventana de lanzamiento de la aplicación, donde pueda verse lo que se está haciendo el momento, así como de cualquier fallo que se detecte.

6.2.6. Mejorar empaquetado

Actualmente, para distribuir una aplicación con Icaro es necesario incluir varios ficheros adicionales al archivo .jar que contiene la mayoria del código. En algunos casos incluso está contenido dentro de varios niveles de directorios sin sentido aparente para el usuario que instale la aplicación. Debería reorganizarse la infraestructura para reducir el código a un unico fichero jar, que pueda ser importado como una libreria, y que pueda empezar a utilizarse sin necesidad de ir copiando los ficheros necesarios.

6.2.7. Creación dinámica de agentes

Actualmente la creación de agentes se hace de forma directa,esto es, se instancian los agentes que hayan sido declarados en la descripción de la aplicación. Esto es poco flexible, para empezar por los motivos comentados en el apartado de arranque parcial ¿es necesario que todos los agentes funcionen?. ¿Pueden ignorarse los problemas en el arranque de algunos componentes? Además, existe la posibilidad de que no concozcamos el número de agentes que se van a necesitar. Por ejemplo, imaginemos que, en el ejemplo del chat mostrado en el capítulo de validación se necesite crear varias salas por demanda, asignando un nuevo agente cada vez que un cliente solicita una nueva sala. No sabemos cuántos van a ser necesarios, y resultaría de lo más ineficiente crear un número elevado de agentes y tenerlos esperando, probablemente no realizando ninguna acción durante mucho tiempo.

7. Referencias y enlaces

7.1. Referencias

- http://icaro.morfeo-project.org/documentacion/documentacion-de-ingenieria/lng/es
- http://es.wikipedia.org/wiki/Java_remote_method_invocation
- http://java.sun.com/javase/technologies/core/basic/rmi/index.jsp
- http://es.wikipedia.org/wiki/GNU_General_Public_License
- http://chuwiki.chuidiang.org/index.php?title=Serialización_de_objetos_en_java
- http://es.wikipedia.org/wiki/Sistema_multi_agente

7.2. Descargando el proyecto

La copia de trabajo del proyecto implementado se puede conseguir en la siguente dirección:

- http://kenai.com/projects/icaro-a3/sources/subversion/show

Además, se subirá a la página de la forja morfeo

- https://forge.morfeo-project.org/frs/?group_id=77

Índice de figuras