

PROYECTO DE FINAL DE CARRERA
Ingeniería Técnica en Informática de Gestión

PLATAFORMA DISTRIBUIDA BASADA EN AGENTES PARA
LA EJECUCIÓN PARALELA DE TRABAJOS

Memoria

Tarragona, marzo del 2006

Autor:

Angel Rodríguez Rozas

Directores de proyecto:

Dr. Antonio Moreno Ribas

Sr. David Isern Alarcón

Sr. David Sánchez Ruenes

INDICE

1. Introducción	5
1.1 Objetivos principales del proyecto	9
1.1 Esquema general del documento	11
2. Tecnologías	13
2.1 Agentes y Sistemas Multi-Agente (SMA).....	13
2.2 JADE.....	15
2.3 Movilidad de los agentes en JADE (agent mobility)	19
2.3.1 Casos de estudio	23
2.4 Tecnologías Web	25
3. Descripción general de la arquitectura del sistema dinámico de ejecución distribuida basada en agentes	29
3.1 Descripción básica de la arquitectura	31
3.2 El cliente	32
3.2.1 Los clientes como nodos de la plataforma	33
3.2.2 Información sobre los recursos de los nodos	33
3.3 El servidor	36
3.3.1 Diseño genérico de un trabajo (Job)	38
3.3.2 Los agentes responsables de los trabajos (AgentWork)	39
3.3.3 El servidor de nombres (namespace)	40
3.3.4 Los nodos	41
3.3.5 Diseño genérico del <i>scheduler</i> (planificador)	43
3.3.6 El agente gestor de eventos (AgentPlatformEventManager)	44
3.3.7 Creación dinámica de trabajos. El servidor de peticiones (ServerRequest).....	47
3.3.8 El agente gestor de la plataforma (AgentGridManager).....	48
3.4 Robustez del sistema. Recuperación dinámica ante caídas de Nodos. "Trabajo que empieza, trabajo que termina"	50
3.5 Optimización al enviar ficheros por la red y facilidad para su manejo. Creación de la clase "NetFile"	52
3.6 Resumen de las características que ofrece el sistema al desarrollador	54

4. Aplicación del sistema a un problema concreto: construcción semiautomática de Ontologías a partir de la Web	55
4.1 Adaptación de los componentes genéricos a la aplicación	55
4.1.1 Los trabajos de la aplicación: JobSearch. Uso del constructor de taxonomías (taxoBuilder)	58
4.1.2 Diseño del scheduler para esta aplicación	63
4.2 Empleo de las tecnologías Web para enriquecer la aplicación	66
4.2.1 Descripción básica de la arquitectura global	68
4.2.2 El servidor de peticiones (ServerRequest). Diseño del protocolo	71
4.2.3 Aplicación Web en J2EE basada en el patrón Modelo-Vista-Controlador (MVC)	76
4.2.4 La persistencia de los usuarios y de sus proyectos. Aplicación del patrón DAO para acceso a datos.....	79
4.2.5 AJAX para una mejor interacción	82
4.2.6 Edición de los proyectos mediante un applet	85
4.2.7 Seguridad en el lado del cliente. Firma del applet	88
5. Evaluación/Resultados	91
5.1 Pruebas de funcionamiento	91
5.2 Pruebas de rendimiento	109
5.3 Pruebas de robustez	114
6. Conclusiones y valoración	117
7. Trabajos futuros	119
8. Manuales	121
8.1 Instalación por parte de los nodos del sistema	121
8.2 Instalación del servidor	121
8.3 Requisitos en el cliente final	123
9. Referencias bibliográficas y enlaces de interés	125

1. Introducción

Los continuos avances científicos hacen surgir proyectos cuyos requerimientos computacionales son cada vez más exigentes, debido a que la complejidad de los mismos es cada vez mayor. La capacidad de computación resulta clave para que su aplicación sea posible. Incluso en otras áreas no científicas, la demanda de computación para resolver diferentes tipos de problemas resulta tremendamente elevada.

Algunos ejemplos en la actualidad con estos requerimientos pueden ser estudios del genoma humano, la predicción de la meteorología, terrenos como la simulación aeronáutica (y otros muchos tipos de simulaciones), la Física de Altas Energías, la Física de Partículas (como se estudia en el CERN¹ [CERN06]), la astronomía, la energía nuclear, la bioquímica, el diseño de semiconductores, etc., y como decía anteriormente, otros tipos de problemas de carácter no científicos, como por ejemplo el renderizado de una película animada realizada por ordenador, en la que cada fotograma de la misma puede obtenerse en tiempos muy elevados, como bien reconoce la empresa Pixia, creadora de productos de este tipo, quien ha afirmado que el coste en tiempo de renderizado de una sola imagen (fotograma) oscila en su caso entre 6 y 19 horas (en otros casos el tiempo es aun mayor). Sabiendo que en un segundo se necesitan varias imágenes consecutivas (depende del estándar se adoptan diferentes cantidades) para dar sensación de movimiento, se calcula que una película de 90 minutos de duración podría tardar, con una sola unidad de proceso, cerca de un centenar de años en ser completamente renderizada.

Para satisfacer dichas necesidades, además de otros requisitos como el ancho de banda, la capacidad de almacenamiento o la capacidad de memoria, está claro que limitarse al empleo de una única unidad de procesamiento con las características tecnológicas actuales es insuficiente. En la actualidad, con la tecnología del silicio, estamos viendo como la famosa "Ley de Moore", la cual predice que la capacidad de los microprocesadores se duplica cada 18 meses, se está viendo comprometida. Prueba de ello es la tendencia de las grandes compañías que fabrican estos chips - como Intel [Intel06], AMD [AMD06], Sun Microsystems [Sun06], entre otros- a reorientar sus arquitecturas hacia la replicación de hardware, sacando máquinas con dos o más unidades de procesamiento. Si bien estas nuevas arquitecturas siguen mejorando la capacidad de procesamiento de una máquina, está claro que por sí solas siguen sin poder abordar problemas del tipo anteriormente mencionados. También quería mencionar la futura aparición de los computadores cuánticos, los cuales están en fase de investigación y desarrollo, que puede llegar a tener un gran impacto en el mundo de la computación. Esta revolución debe aparecer debido a la dificultad de rebasar un cierto umbral (al cual nos estamos aproximando) de miniaturización de los componentes electrónicos en los que se basan los chips actuales. Pero este tema no es objeto de estudio de este proyecto, con el que se persigue buscar soluciones conscientes de la tecnología que existe en la actualidad y que existirá en los próximos años.

¹ El **CERN** es la "**Organisation Européenne pour la Recherche Nucléaire**". Las siglas CERN vienen de su antiguo nombre **Consejo Europeo para la Investigación Nuclear** (*Conseil Européen pour la Recherche Nucléaire*, en francés). Se trata de un laboratorio de investigación en Física de partículas.

Estamos hablando de la necesidad de utilizar más de un procesador simultáneamente para poder tener una mayor potencia de cálculo, o en otros casos (según la finalidad que se persigue) podemos hablar de un mejor aprovechamiento de una cantidad mayor de recursos (como ciclos de cpu, ancho de banda, capacidad de almacenamiento o capacidad de memoria), con la finalidad de hacer que el tiempo necesario para resolver un problema se vea altamente reducido y/o pueda ser tratado computacionalmente. Se trata de descartar la adopción de una única solución que sea muy buena (que además en poco tiempo se verá superada por otra nueva), y pensar en agrupar muchas soluciones para que, uniendo sus esfuerzos, consigamos un mejor rendimiento final.

En los últimos años han surgido diferentes arquitecturas que intentan cumplir con este objetivo de construir un sistema que supere las limitaciones de capacidad de cómputo, o de forma más general, de recursos. Una de las características que distinguen las unas de las otras es la manera en que se distribuyen y gestionan sus principales recursos, que son las unidades de proceso y elementos de comunicación. Ellos deben de ser gestionados por un software, otro elemento principal. De todas ellas hay dos que han sido muy aceptadas, como son los clusters y los más recientes denominados "grids" que a continuación describo.

- Clusters: por un lado, el cluster (véase la *figura 1*) es una arquitectura que reúne una serie de recursos dedicados, distribuidos entre diferentes máquinas, habitualmente de similares características, las cuales están conectadas entre sí mediante una red de alta velocidad (idealmente). Están bajo el control de un determinado software que gestiona dichos recursos, haciendo su correspondiente balanceo de la carga (load balancing). Como podemos observar, se trata de una arquitectura rígida, donde los componentes son fijos, están dedicados y son homogéneos. Su perímetro está marcado por una red de área local (LAN) y sus recursos son estáticos. Por tanto, se conoce perfectamente de antemano el hardware disponible y en consecuencia podemos hablar de que la programación es muy consciente de los recursos existentes. Esta arquitectura presenta muchas ventajas. El hecho de saber con bastante exactitud la capacidad de cálculo de la que disponemos, y si resulta que el problema que queremos resolver es determinista en cuanto al tiempo necesario a partir de la capacidad de los recursos disponibles, y además es necesario resolverlo en una cantidad determinada de tiempo, esta arquitectura encaja muy bien. Por ejemplo, con la predicción de la meteorología. También estamos ante una arquitectura eficiente, ya que cuenta con una conexión de red de alta velocidad y de baja latencia. La distancia entre nodos no es grande y los saltos necesarios para alcanzarse entre sí tampoco. Los clusters también se pueden diferenciar entre ellos según los objetivos con las que han sido creados. Pueden requerir alta disponibilidad (recursos redundantes, de forma que si uno falla haya otro disponible), alto rendimiento (requerir una determinada capacidad de cálculo, normalmente medida en flops, que mide las operaciones en punto flotante por segundo). Pero también presentan una serie de problemas. Como se puede observar, estos tipos de supercomputadores requieren de una fuerte inversión al alcance de pocas entidades. También requieren de un software específicamente diseñado. Además puede resultar que la tecnología quede obsoleta en poco tiempo y la inversión no sea rentable. Digamos que es difícil

renovar la tecnología después de haber realizado una gran inversión. El incremento del coste es proporcional al número de máquinas que queramos adquirir.



Figura 1. Recinto que alberga un cluster

- Grids: como alternativa a los clusters surgieron los grids (Grid Computing), desarrollado inicialmente en ámbitos científicos a principios de los 90. Un grid es una arquitectura capaz de soportar sistemas heterogéneos (es decir, sistemas que pueden tener características muy diferentes) para compartir sus recursos a gran escala de forma coordinada, que están conectados entre sí mediante una red área extensa (como puede ser Internet u otras redes). La idea que persigue es diferente, ya que se basa en el acceso de forma remota a recursos computacionales de naturaleza que puede ser diferente, de una manera transparente para el usuario mediante la integración de dichos recursos para que puedan interoperar, permitiendo una gran capacidad de cálculo. Estamos ante un reto más amplio ya que se trata de desarrollar una arquitectura distribuida en la cual se pierde la uniformidad de los recursos. Si bien estos pueden estar geográficamente dispersos, son gestionados y utilizados por una organización en concreto. Como se puede observar los grids deben ser altamente dinámicos, ya que no disponemos de recursos rígidos, ni podemos saber en ningún momento de cuantos recursos disponemos. Este dinamismo requiere que la arquitectura sea capaz de adaptarse a los constantes cambios inmediatos e inesperados de los recursos disponibles (una organización o particular que haya decidido compartir sus recursos puede eliminarlos o puede aportar más recursos en un momento dado). Se puede observar como una gestión eficiente del mismo es una tarea ardua. Una de las características que se pide a un grid es una eficiente planificación (scheduling) de los recursos disponibles, para que el balanceo de la carga sea lo más equitativo posible de forma que se aprovechen lo mejor posible. También se le pide que sea capaz, de alguna forma, de descubrir nuevos recursos de manera que pasen a formar parte del grid. Además, es obvio que debe de ser capaz de

reconocer y poder clasificar a los mismos, característica fundamental para el scheduling. También debe proporcionar una manera de acceder a los datos remotos y de una transferencia lo más eficiente posible de los mismos, ya que de entrada no es posible determinar las características de las redes por las cuales circularán dichos datos. Otro importante reto es el de la tolerancia a los fallos, de forma que si un nodo del grid cae por el motivo que sea, los trabajos ejecutados en los mismos no se vean comprometidos. Esto debe de ser transparente al usuario final, el cual debe de tener la garantía de que un trabajo solicitado acabe realizándose exitosamente. Por tanto, esto requiere una gestión de recuperación ante fallos, reasignando las tareas a otros recursos para que puedan llevarlas a cabo. Para conseguirlo se necesita un mecanismo para monitorizar los nodos del grid.

Como vemos las ventajas de los grids son muchas, y entre ellas destacan la gran reducción de costes respecto a los clusters. Además estamos delante de un sistema más flexible y dinámico cuya capacidad de proceso puede ser tan elevada o más que la de los clusters (puede ser un sistema en el que participen computadores de cualquier parte del mundo). Además, la renovación de los recursos tecnológicos se va haciendo constantemente y no resulta tan costoso como en el caso de los clusters.

Por mencionar un ejemplo, hace poco tiempo se desarrolló un proyecto que utiliza una arquitectura de tipo grid, denominado **SETI@home** [SETI06], que es un experimento científico que utiliza computadores conectados a Internet en la Búsqueda de Inteligencia Extra-Terrestre (SETI en inglés). Un usuario puede participar instalando un salva pantallas gratuito en su escritorio que descarga, y que en el momento que se activa, aprovechando un estado ocioso de la capacidad de cómputo de su máquina, analiza señales procedentes del espacio, que son captadas mediante una antena muy potente (véase la *figura 2*). Como vemos, este tipo de arquitecturas son menos eficientes que los clusters, pero aprovechan el hecho de que en la actualidad tenemos una red barata como Internet, que conecta a la mayoría de ordenadores del mundo, reuniendo un hardware muy heterogéneo y en su mayoría barato, pero que con el paso del tiempo es cada vez más potente, para disponer de unos recursos computacionales de forma flexible y dinámica, de manera que obtienen económicamente un entorno de ejecución paralelo y distribuido que nos ofrece un rendimiento muy alto. Es decir, es un enfoque distinto que se adapta mejor a la tecnología actual. Permite que instituciones que no disponen de muchos recursos puedan acceder a un potencial de cómputo considerable de forma remota a través de la red mundial, gracias a la reunión de sus esfuerzos junto a otras instituciones (y también junto a usuarios particulares). Además se puede observar que un grid no sólo sirve para aplicar problemas de naturaleza paralela o divisible, sino que también sirven para explotar los recursos computacionales que a lo largo del mundo están siendo desaprovechados en todo momento (ciclos ociosos). Por tanto, se trata de una tecnología que nos puede ofrecer una alta capacidad computacional con costes reducidos.

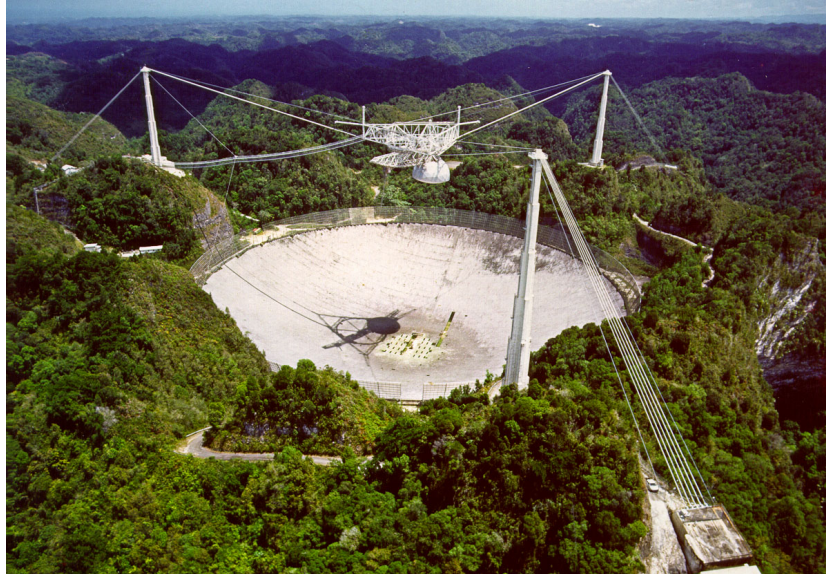


Figura 2. Arecibo, la antena más grande del mundo, de 304 metros de diámetro. Recibe constantemente señales de radio del exterior para posteriormente ser analizadas mediante los computadores de usuarios de Internet para buscar un patrón inmerso en el ruido de fondo espacial generado por la radiación universal.

1.1 Objetivos principales del proyecto

En este proyecto he intentando desarrollar un sistema dinámico de ejecución distribuida, mediante el cual se pueda realizar trabajo en paralelo, con una arquitectura parecida más bien a lo que he descrito anteriormente como grid. Es decir, se trata de un sistema en el cual diferentes computadores localizables a través de la red Internet ofrecen sus recursos, de manera dinámica, para que mediante una política de planificación (scheduling) adaptada al tipo de problema que se pretenda aplicar, se pueda resolver de la manera más eficiente posible. Quiero dejar claro que, en esta primera parte, he diseñado una arquitectura genérica, la cual podrá ser adaptada de la forma más sencilla posible a las características requeridas por el problema particular que se pretenda resolver en cada caso. Obviamente, la naturaleza del trabajo que se quiera aplicar tiene que ser paralelizable para poder sacar provecho de la capacidad de cálculo en paralelo del sistema y así lograr la resolución del mismo en un tiempo más reducido.

Para lograrlo he utilizado una alternativa que hasta ahora no ha sido explotada para estos fines, pero que como veremos se adapta muy bien a la arquitectura de los grids. Se trata del paradigma de agentes y sistemas multi-agente. Éste paradigma de desarrollo de software ha surgido hace poco años, en la década de los 90. Un sistema multi-agente es un sistema compuesto de diversos componentes software distribuidos que interactúan y cooperan para resolver problemas complejos que no pueden ser resueltos individualmente. Si bien es muy utilizado en el campo de la inteligencia

artificial, no sólo se restringe a éste dominio sino que se puede aplicar en diversos aspectos de la informática relacionados con resolución de problemas complejos y de carácter más bien distribuido, que puede proveer y proveerá un mejor acceso al conocimiento y recursos de computación.

Es por este motivo por el cual pienso que los agentes nos ofrecen una muy buena alternativa de alto nivel para diseñar sistemas paralelos complejos, como son las arquitecturas de tipo grid, las cuales requieren de características que son soportadas de manera flexible por el paradigma de los agentes, como una gran capacidad de control dinámico de la ejecución, movilidad de código, acceso remoto a recursos computacionales, comunicaciones complejas entre diversos nodos, cooperación y a la vez capacidad de independencia, robustez, etc.

No hay duda que para lograr los objetivos de crear una arquitectura tan flexible, dinámica, heterogénea y a la vez transparente, y genérica, es necesario utilizar una tecnología de desarrollo que sea capaz de satisfacer todos estos requisitos. Y es este uno de los principales argumentos para justificar el empleo de la tecnología Java y el paradigma orientado a objetos, junto con el paradigma de los agentes y sistemas multiagente (mediante un framework de desarrollo de agentes, llamado JADE [JADE06]).

En resumen, de modo general, los principales objetivos de mi proyecto han sido los siguientes:

- Estudiar las posibilidades de creación y gestión de Sistemas Multi-Agente distribuidos en JADE, haciendo hincapié en los aspectos de movilidad de agentes.
- Crear un entorno de ejecución paralelo y distribuido de carácter genérico y dinámico de características similares a las de una arquitectura de tipo grid, basado en el lenguaje de programación JAVA [JAVA06] y el framework de JADE [JADE06], para que se puedan aplicar diferentes tipos de problemas computacionales de naturaleza divisible.
- Adaptar un problema basado en la construcción semiautomática de Ontologías² a partir de la Web, de forma que opere sobre el sistema diseñado, consiguiendo reducir la granularidad de las tareas a ejecutar y el tiempo de respuesta.
- Construir una interfaz de acceso al sistema para los usuarios finales, que permita la gestión de trabajos paralelos de forma transparente.
- Evaluar el rendimiento de todo el sistema una vez implementado para diferentes condiciones de ejecución (nivel de concurrencia, política de balanceo de la carga, manejo de recursos y grado de robustez).

² Según [Weigand97], una ontología es una estructura de datos que describe los conceptos en el mundo o algún dominio, algunas de sus propiedades y cómo los conceptos se relacionan con los demás.

1.2 Esquema general del documento

Antes de describir las dos partes principales del proyecto (capítulos 3 y 4 respectivamente), en el capítulo 2 resaltaré las características de las diferentes tecnologías más novedosas que he empleado dentro del grupo de trabajo al cual pertenece mi proyecto, llamado GruSMA ("Grup de Sistemes Multi-Agent") [GruSMA06].

En el capítulo 3 se explica toda la arquitectura del sistema distribuido creado, dando detalle de los elementos más importantes y destacando las decisiones de diseño tomadas.

En el capítulo 4 se ve como se adapta un problema real al sistema construido. En esta parte se verá un ejemplo práctico donde se destacará la capacidad de adaptación de los componentes genéricos del sistema distribuido a los requerimientos específicos de dicho problema y para finalmente valorar los resultados obtenidos.

Después de estos apartados, en el capítulo 5, se mostrarán una series de pruebas, tanto de funcionamiento global de la aplicación, así como de rendimiento y otras de robustez valorando los resultados obtenidos. En el capítulo 6 llegarán mis conclusiones y valoración sobre este proyecto, así como las posibles líneas de trabajos futuros que pueden aparecer tras este trabajo.

Para acabar, en el capítulo 7 se mostrará el código fuente del sistema distribuido. En el capítulo 8 se explican mediante unas manuales como instalar las diversas partes del código producido. Finalmente, en el capítulo 9 se muestran las referencias bibliográficas y enlaces de interés relacionados con el proyecto.

2. Tecnologías empleadas

En este capítulo describiré las distintas tecnologías que he empleado para desarrollar mi proyecto, pero sólo lo haré de aquellas que resulten más novedosas respecto a las que se han empleado dentro del grupo al cual pertenece este proyecto, llamado GruSMA ("Grup de Sistemes Multi Agent") [GruSMA06], ya que la descripción de las restantes se pueden encontrar en la documentación de otros proyectos que otros compañeros han desarrollado.

- Java:

El presente proyecto, como remarco en la introducción, se ha desarrollado completamente en Java. Este lenguaje orientado a objetos y multiplataforma, encaja perfectamente para cumplir con los requerimientos marcados desde un inicio. Además, el proyecto está construido en base a un framework para sistemas multiagente escrito en Java, denominado JADE, en su versión 3.3, que es el que se utiliza en los proyectos de GruSMA.

- Agentes y Sistemas Multi-Agente:

Uno de los objetivos marcados en este proyecto es estudiar las capacidades que nos ofrece JADE para la gestión de sistemas paralelos. Concretamente, investigar sobre la movilidad de agentes como aproximación al diseños de estos sistemas. Y finalmente, aplicar el sistema construido sobre un problema concreto.

Así pues, después de estudiar la movilidad de los agentes y de ver las posibilidades que tenía, se tomó la decisión de crear una plataforma genérica para soportar la ejecución de trabajos en paralelo.

- Tecnologías de la web:

Por otro lado, para lograr gestionar la aplicación paralela, se ha querido diseñar una interfaz que permita llevar el control de los trabajos, de la concurrencia y que soporte múltiples usuarios. En está aplicación se pretende crear un sistema para la construcción semiautomática de Ontologías a partir de la Web. Para ello se ha considerado oportuno utilizar las tecnologías Web como alternativa para interactuar con el cliente. De esta forma podremos mantener de una forma elegante, una aplicación flexible, en la cual la parte de visualización se delega a los clientes (mediante un applet de Java) y la parte de la lógica se desarrolla en el servidor.

2.1 Agentes y Sistemas Multi-Agente (SMA)

La relación que existe entre los conceptos de agente y un Sistema Multi-Agente es que el primero es el componente básico sobre el que se construye el segundo. Es decir, un

Sistema MultiAgente está compuesto por diversos agentes. Por tanto, para entender mejor el concepto más amplio de los dos, el de Sistema Multi-Agente, antes hay que comprender que es un agente. Según Jennings y Wooldridge [Jenn&Wool98], un agente "es un sistema computacional que está situado en un ambiente y que es capaz de acciones autónomas en este medio para alcanzar sus objetivos de diseño".

Las principales propiedades que puede tener un agente son:

- Autonomía: es la capacidad de operar sin la intervención directa de los humanos, y de tener algún tipo de control sobre las propias acciones y el estado interno.
- Sociabilidad/Cooperación: los agentes han de ser capaces de interactuar con otros agentes a través de algún tipo de lenguaje de comunicación.
- Reactividad: los agentes perciben su entorno y responden en un tiempo razonable a los cambios detectados.
- Pro-actividad o iniciativa: deben ser capaces de mostrar que pueden tomar la iniciativa en ciertos momentos.
- Movilidad: posibilidad de moverse a otros entornos a través de una red electrónica.
- Veracidad: un agente no comunicará información falsa premeditadamente.
- Benevolencia: es la propiedad que indica que un agente no tendrá objetivos conflictivos, y que cada agente intentará hacer lo que se le pide.
- Racionalidad: el agente ha de actuar para conseguir su objetivo.
- Aprendizaje: mejoran su comportamiento con el tiempo.
- Inteligencia: usan técnicas de IA para resolver los problemas y conseguir sus objetivos.

Un Sistema Multi-Agente es un entorno distribuido y poco acoplado en el que un conjunto de agentes cooperan para conseguir un objetivo común de manera que individualmente no sería posible.

Las principales ventajas de la utilización de un sistema Multi-Agente son:

- Modularidad: se reduce la complejidad de la programación al trabajar con unidades más pequeñas, que permiten una programación más estructurada
- Eficiencia: la programación distribuida permite repartir las tareas entre los agentes, consiguiendo paralelismo (agentes trabajando en diferentes máquinas).

- Fiabilidad: el hecho de que un elemento del sistema deje de funcionar no tiene que significar que el resto también lo hagan; además, se puede conseguir más seguridad replicando servicios críticos y así conseguir redundancia.
- Flexibilidad: se pueden añadir y eliminar agentes dinámicamente.

Notemos como las características de movilidad, modularidad, y flexibilidad de los agentes Sistemas Multi-Agente nos pueden ofrecer una seria alternativa para la construcción de una arquitectura de tipo grid para la ejecución paralela, distribuida y dinámica de tareas, como las que se pretenden en este proyecto.

2.2 JADE

Como framework de desarrollo de agentes se ha utilizado JADE (*Java Agent Development Framework*) [JADE06]. A continuación describiré de modo resumido los componentes básicos que participan en la arquitectura de la plataforma JADE, en su versión 3.3 [JADE, PG33], la cual está basada en las especificaciones de FIPA [FIPA, Spec06], un comité que elaboró un estándar para la construcción de arquitecturas para Sistemas Multi-Agente. Describiré los elementos más importantes para luego poder comprender bien como está desarrollada la arquitectura propuesta en este proyecto, sustentada en las capacidades de ejecución distribuida de los Sistemas Multi-Agente.

JADE es considerado por sus creadores en [JADE, PG33], como “un middleware que facilita el desarrollo de sistemas multi-agente” que incluye un entorno de ejecución donde los agentes podrán ser ejecutados, una librería de clases para dar soporte al desarrollo de agentes y una suite de herramientas gráficas que permite gestionar y monitorizar la actividad de la plataforma.

Cada instancia que se ejecuta del run-time de JADE crea uno o más componentes llamados *Container* (contenedor), en el sentido de que contienen agentes. Un contenedor pertenece a una *Plataforma* (véase la *figura 3*). De todos los contenedores de una única plataforma, uno especial es el llamado *Main-Container*, el primero que se crea. Los demás contenedores deberán ser registrados en la plataforma como “no-principales”, indicando el nombre de la máquina y el puerto de la máquina donde reside el Main-Container. Otra característica a tener en cuenta es que los identificadores de los agentes (AID) deben de ser únicos dentro de una plataforma. Como vemos, con JADE el programador se desentiende de las dependencias del hardware y del software y solo reconoce contenedores y agentes, independientemente de su localización física.

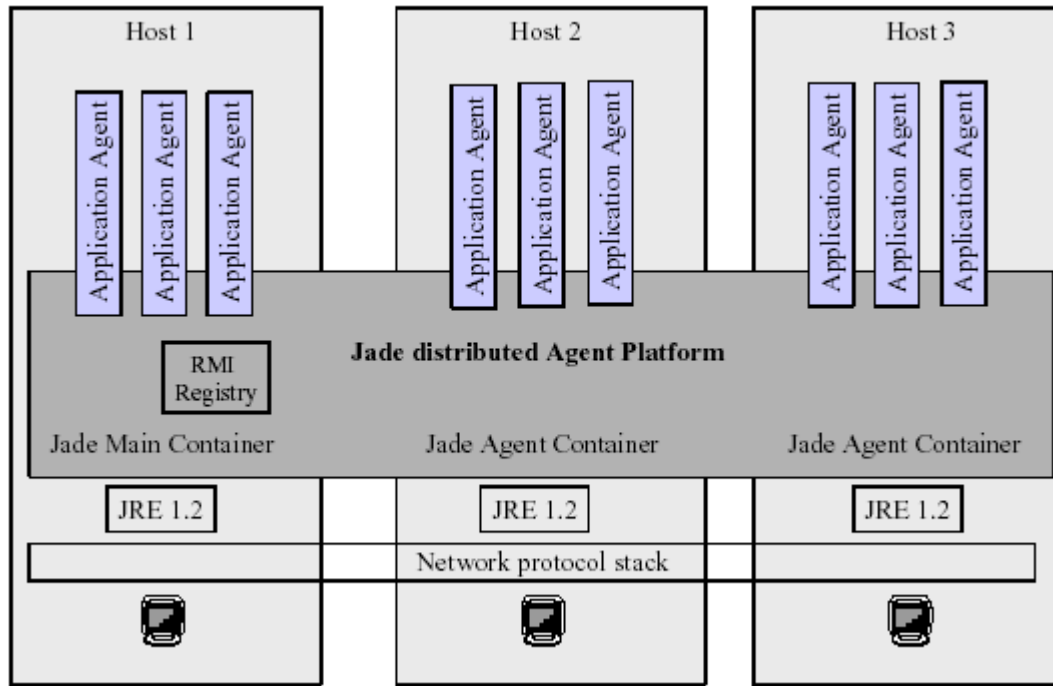


Figura 3. Aspecto general de la plataforma propuesta por JADE [JADE, PG33]

Como se puede ver en la *figura 4*, en una plataforma JADE residen dos agentes especiales que se crean automáticamente, llamados AMS (*Agent Management System*) y DF (*Directory Facilitator*), que residen en el contenedor principal (*Main-Container*).

- AMS (*Agent Management System*): es un agente que provee el servicio de nombres (únicos) y representa la autoridad en la plataforma (se le puede hacer peticiones para mover a un agente, crear agentes, matar agentes, etc.).
- DF (*Directory Facilitator*): provee un servicio de "Páginas Amarillas", en el cual los agentes registran sus servicios y así otros agentes puedan descubrirlos preguntándole posteriormente por los agentes que los sirven, y así poder ser solicitados.

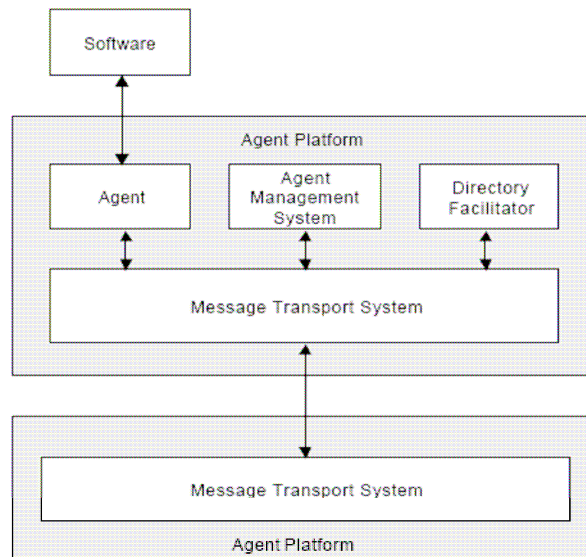


Figura 4. Componentes principales de la plataforma JADE

Los creadores de JADE comentan en [JADE06] que la arquitectura permite a cada agente descubrir dinámicamente otros agentes y comunicarse entre ellos según el paradigma del peer-to-peer. JADE proporciona los servicios básicos necesarios para las aplicaciones peer-to-peer distribuidas, pero su arquitectura, al menos hasta la versión 3.3, está basada claramente en el paradigma cliente servidor.

JADE también nos ofrece una herramienta gráfica con la cual podemos monitorizar y controlar de forma visual la plataforma en todo momento (véase la *figura 5*), lo cual viene muy bien para el administrador. Se trata de un nuevo agente cuyo nombre es *Remote Monitoring Agent (RMA)*.

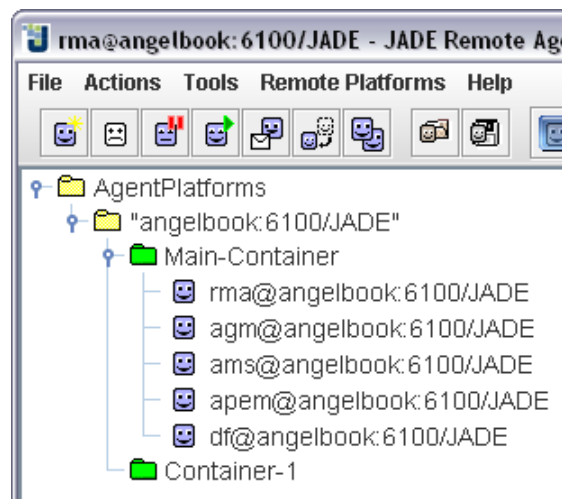


Figura 5. El agente RMA nos muestra una interfaz gráfica con la información del estado de la plataforma actualizada en todo momento.

Por último, destacar la existencia del *Message Transport System* (véase la figura 4), a veces llamado *Agent Communication Channel* (ACC), que es el componente que controla todos los intercambios de mensajes tanto intra-plataforma como inter-plataformas. Este componente utiliza internamente RMI³ (*Remote Method Invocation*) [RMI06], cuyo registro se crea junto al Main-Container. El uso del mismo es interno y por tanto transparente al programador.

Los agentes se comunican mediante un mecanismo de paso de mensajes. Un mensaje es considerado una acción comunicativa.

Estos mensajes tienen las siguientes características:

- Están escritos en un lenguaje de comunicación de agentes (normalmente FIPA ACL [FIPA, ACL06]).
- El contenido está expresado con un lenguaje de contenidos (normalmente FIPA-SL [FIPA, SL06]).
- El lenguaje de contenidos puede hacer referencia a una Ontología.
- Incluyen los nombres de emisor y receptor.

Para facilitar el desarrollo aun más, se definen una serie de protocolos de interacción de alto nivel. Las conversaciones entre agentes suelen seguir unos ciertos patrones, secuencia típicas de mensajes, cuya frecuencia de uso ha provocado la creación de estos protocolos de interacción. FIPA ha estandarizado algunos de ellos [FIPA, IPs06], los cuales se pueden emplear en la mayoría de ocasiones dado la diversidad de la que disponemos. Todo y así, mediante JADE un desarrollador puede crear sus propios protocolos, adaptándolos a sus necesidades en caso que no encuentre uno adecuado de los propuestos como estándares.

Como he remarcado anteriormente, sobre el tema de agentes y Sistemas Multi-Agente hay mucha documentación en proyectos de otros compañeros que se puede consultar. En este apartado he tratado de dar una pincelada general repasando aquellos conceptos que he considerado oportunos para poder comprender el diseño de mi arquitectura. Ahora sí que entraré en detalle en el siguiente apartado en el tema de la movilidad de los agentes, ya que es un tema novedoso dentro del grupo de trabajo GruSMA.

³ RMI (*Remote Method Invocation*) es una tecnología que sirve básicamente para invocar procedimientos remotos de forma más o menos transparente para el desarrollador, facilitando así la interoperabilidad entre aplicaciones JAVA distribuidas.

2.3 Movilidad de los agentes en JADE (agent mobility)

Uno de los puntos más interesantes de la tecnología de agentes y sistemas multiagente para los objetivos de este proyecto, además de la capacidad de ejecución distribuida, es la posibilidad de conseguir movilidad de agentes de forma que sean capaces de trasladarse de una localización a otra, ya sea intra-plataforma o inter-plataformas. Como ya he comentado, éste era uno de los retos principales de mi proyecto, ya que los anteriores proyectos de GruSMA no habían explotado esta posibilidad tan interesante.

La movilidad de los agentes en el caso de JADE, como se puede intuir, se trata de la posibilidad de que un agente que reside en un contenedor en un momento determinado, sea capaz de migrar hacia otro contenedor (otra localización). En la versión 3.3 de JADE [JADE, PG33] sólo se permite la posibilidad de movilidad intra-plataforma. Mientras que desarrollaba este proyecto ha salido una nueva versión de JADE, la 3.4 (hace algo más de un mes en el momento que escribo estas líneas), con la cual se permite la movilidad también inter-plataformas.

Según la documentación de JADE (versión 3.3) [JADE, PG33] se ofrecen dos operaciones básicas de movilidad: una de ellas es la de moverse y otra es la de clonarse (un agente es clonado en otro contenedor). La ejecución de cualquiera de las dos operaciones sobre un agente provocan que su estado pase a transitorio (véase la *figura 6*), estado que pertenece al conjunto de estados del ciclo de vida de un agente. Para aplicar cualquiera de las dos operaciones sobre un agente, se puede hacer de dos maneras distintas: puede comenzarse por el propio agente o bien vía solicitud (mensaje) al AMS.

A continuación se detallaran los procesos. Pero antes quiero hacer una aclaración. Siempre que pretendemos mover o clonar un agente, utilizando cualquier posibilidad para hacerlo, JADE nos exige que le proporcionemos un destino. El destino final es un contenedor, y para especificarlo JADE nos pide un objeto del tipo `jade.core.Location`, que como podemos ver, no es un objeto de tipo contenedor (`AgentContainer`). Se trata de una interfaz abstracta que un agente no puede instanciar (digamos que es una forma de darle más robustez). Cada container tiene asociado un objeto de tipo `Location`. Se puede obtener la localización del contenedor en el cual está “viviendo” un agente mediante la ejecución del método `here()` de la clase `Agent`. Y como veremos, también podemos solicitar al AMS un listado de las localizaciones actuales de las que dispone la plataforma. También podemos solicitar al AMS en qué localización (dónde “viven”) otros agentes.

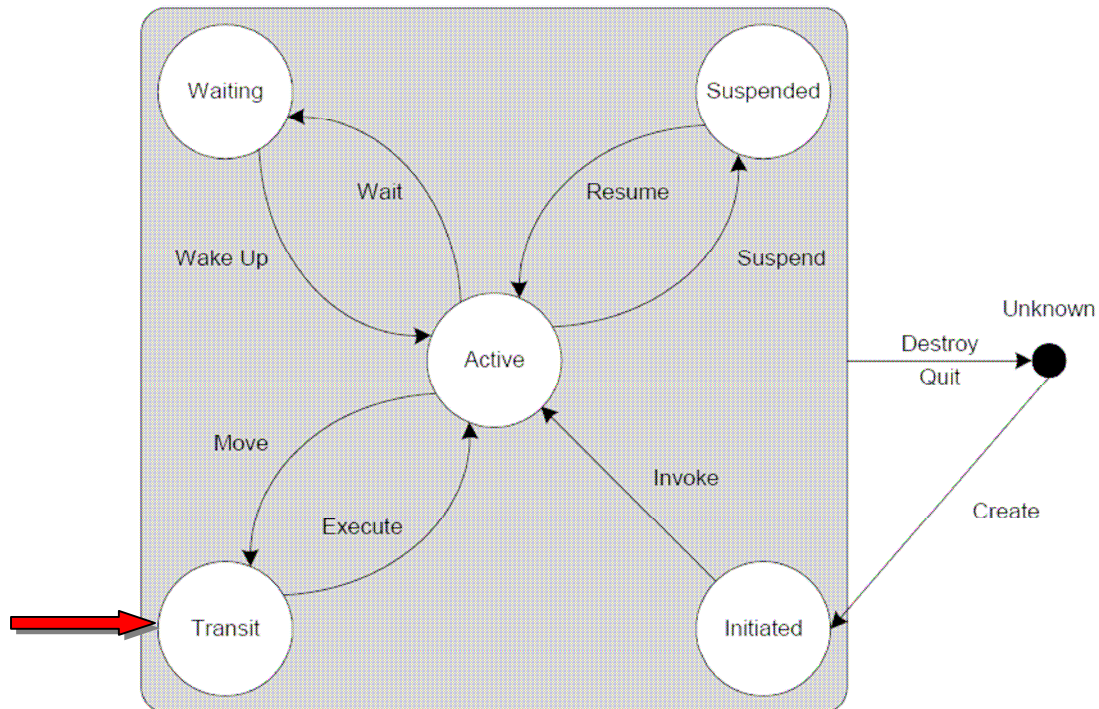


Figura 6. Ciclo de vida de un agente, definido por FIPA

Un aspecto importante es conocer que la movilidad de un agente implica la movilidad tanto del código como del estado a través de la red (si es que el destino resulta estar en otra máquina virtual), que se hace de manera implícita para el desarrollador que utiliza JADE cuando solicita la operación de mover o la de clonar a un agente a otra localización. Esto lo consigue JADE de forma interna mediante el potente mecanismo de serialización y deserialización que soporta el lenguaje JAVA. Esta característica se presenta muy interesante para mi sistema distribuido, ya que la introducción de la movilidad del código y del estado, es una novedad importante respecto a otras aproximaciones de arquitecturas de este tipo.

En el momento en que migra un agente, el método llamado `beforeMove()` de la clase `Agent` es invocado sobre el propio agente en la localización de origen, cuando la operación de migración ha sido completada con éxito. En ese momento, la instancia del agente en el contenedor de destino se activa y la instancia en el origen acaba. Esto puede suponer serios problemas a los desarrolladores si no comprenden bien el mecanismo de funcionamiento exacto. Como consecuencia inmediata, aquellos datos del agente que sean modificados o aquellos que sean creados en el ámbito global del agente dentro de éste método, no se verán efectuados en el destino como podría pensarse a primera vista. Así pues, éste método será empleado para liberar posibles recursos que el agente tenga asignados en ese instante en la localización de partida, como por ejemplo cerrando un fichero abierto. El método `afterMove()` es ejecutado en la localización destino, en el momento que haya llegado y su identidad sea reconocida.

Paralelamente, existen los métodos `beforeClone()` y `afterClone()` que siguen una lógica parecida, pero para el caso de clonación de un agente. La única diferencia está en que en este caso, la instancia del agente sigue viva en la localización de origen. Los cuatro métodos pertenecen a la clase `Agent` y son de tipo `protected`.

Por un lado, la forma más sencilla de conseguir tanto la movilidad como la clonación es mediante el empleo de dos métodos públicos que la clase `Agent` proporciona:

- `public void doMove(Location destino):` el agente sobre el que se ejecuta el método migra a la localización (container) especificada por el parámetro destino.
- `public void doClone(Location destino, String nombre):` se hace un clon del agente sobre el que se ejecuta el método, en la localización(container) especificada por el parámetro destino, que tendrá el nombre indicado en el segundo parámetro de tipo `String`.

Por otro lado, la segunda forma de conseguirlo es mediante la Ontología que nos proporciona JADE en su versión 3.3 [JADE, PG33], `jade-mobility-ontology`. Esta Ontología nos ofrece todos los conceptos y acciones necesarias para soportar la movilidad de los agentes. Dicha ontología se encuentra mapeada en una clase de JADE llamada `jade.domain.mobility.MobilityOntology`, que extiende otra llamada `JADEManagementOntology`, y está creada empleando el patrón de diseño `Singleton`, que básicamente tiene como objetivo asegurarnos la existencia de una única instancia de una clase dentro de un entorno de ejecución. Esta instancia se obtiene mediante el método `getInstance()`. Este método, como es lógico, es estático. No hay que perder de vista que el objetivo final del empleo de esta ontología es conseguir que un agente se comunique con el AMS, para poder solicitarle una acción relacionada con la movilidad (mover o clonar un agente a un destino) con unas determinadas exigencias, de forma que ambos agentes se entiendan perfectamente (hablando el mismo lenguaje).

A continuación describiré las posibilidades que nos ofrece esta ontología, para poder explotarlas convenientemente en el momento de su empleo. La ontología está compuesta de cinco conceptos y dos acciones, que tienen asociados cada uno una clase particular dentro del paquete `jade.domain.mobility`:

- `Mobile-agent-description`: describe algunas características asociadas a un agente que va a moverse. La clase asociada es `MobileAgentDescription` y tiene 5 slots:
 - o `Name`, de tipo `AID`, que es obligatorio especificarlo.
 - o `destination`, de tipo `Location`. El destino, que es obligatorio.
 - o `agent-profile`, es un concepto de tipo `mobile-agent-profile`, y es opcional especificarlo o no.
 - o `agent-version`, de tipo `String`, opcional.
 - o `signature`, de tipo `String`, opcional.
- `mobile-agent-profile`: describe el entorno de computación necesario para el agente móvil. La clase asociada es `MobileAgentProfile` y tiene 3 conceptos:
 - o `system`, es un concepto de tipo `mobile-agent-system`, opcional.

- o `language`, es un concepto de tipo `mobile-agent-language`, opcional.
 - o `os`, es un concepto de tipo `Mobile-agent-os`, obligatorio
- `mobile-agent-system`: describe el sistema en tiempo de ejecución (versión de JADE) usado por el agente móvil. La clase asociada es `MobileAgentSystem` y tiene 4 slots:
 - o `name`, de tipo `String`, obligatorio.
 - o `major-version`, de tipo `Integer`, obligatorio
 - o `minor-version`, de tipo `Integer`, opcional
 - o `dependencies`, de tipo `String`, opcional
- `mobile-agent-language`: describe el lenguaje de programación usado por el agente móvil. La clase asociada es `MobileAgentLanguage` y tiene 4 conceptos:
 - o `name`, de tipo `String`, obligatorio.
 - o `major-version`, de tipo `Integer`, obligatorio
 - o `minor-version`, de tipo `Integer`, opcional
 - o `dependencies`, de tipo `String`, opcional
- `mobile-agent-os`: describe el sistema operativo requerido por el agente móvil. La clase asociada es `MobileAgentOS` y tiene 4 conceptos:
 - o `name`, de tipo `String`, obligatorio.
 - o `major-version`, de tipo `Integer`, obligatorio
 - o `minor-version`, de tipo `Integer`, opcional
 - o `dependencies`, de tipo `String`, opcional
- `move-agent`: se trata de una acción para solicitar mover un agente desde una localización a otra. Está representada por la clase `MoveAction`. Esta acción tiene un solo slot como argumento de tipo `mobile-agent-description` (concepto descrito anteriormente), que es obligatorio.
- `clone-agent`: se trata de una acción para solicitar la clonación de un agente, que posiblemente tenga una localización distinta a la del agente original. Está representada por la clase `CloneAction`. Esta acción tiene dos slots como argumentos, el primero de tipo `mobile-agent-description` (concepto descrito anteriormente) y el segundo de tipo `String` (el nombre del nuevo agente), ambos obligatorios.

Hay que remarcar que en la documentación oficial de JADE [JADE, PG33] se indica que esta Ontología no forma parte del estándar especificado por la FIPA [FIPA, Spec06] y que están a la espera de su pronunciación para actualizar la ontología a los requerimientos que especifique. De esta manera es de esperar que se vea modificada en las futuras versiones de JADE.

En resumen, utilizando el AMS para solicitar una operación de migración, un agente debe de instanciar un objeto de tipo `MoveAction`, cuyo argumento es un objeto de la clase `MobileAgentDescription` con el cual se especifican los parámetros necesarios en cada caso.

También, como he comentado anteriormente, esta ontología extiende otra, denominada `JADEManagementOntology`, la cual soporta dos acciones más, también

interesantes. Una es `where-is-agent`, cuyo parámetro es un identificador de agente (AID), que sirve para localizar (devuelve un objeto de tipo `Location`) a un agente cuyo identificador es conocido. Otra acción es `query-platform-locations`, que no tiene argumentos, y que nos devuelve una lista de todas las localizaciones de la plataforma disponibles en ese momento.

2.3.1 Casos de estudio

Hasta aquí he explicado la teoría para comprender las opciones que tenemos con JADE. Pero todos sabemos que lo mejor para conocer las cosas es ponerlas en práctica y experimentar con ellas. Ahora se plantean preguntas como qué pasa si tenemos un agente que queremos mover con datos instanciados (objetos). ¿Se mantienen una vez en el destino?, ¿es necesario que el código fuente exista en el destino?, ¿qué pasa si tenemos una clase de la cual aún no se ha instanciado ningún objeto, y ésta no está en el destino?, ¿todas las clases que utiliza un agente móvil deben de ser serializables?, ¿se mantiene el estado de los atributos del agente en el destino?

A continuación iré resolviendo todas estas preguntas. Comentaré una serie de pruebas en las que un agente que se está ejecutando en una máquina, al crearse, decide moverse a otra. En el momento de su creación se comunica con el AMS, preguntándole por las localizaciones que tiene. A continuación, el agente recibe la respuesta con el listado, y decide moverse a un contenedor (localización) remoto. En el momento de llegar saluda por la salida estándar (mediante `System.out.println`).

- **Caso número 1:** el agente no tiene datos y su código fuente no está en el destino (la clase padre del agente, `Agent`, sí que está en el destino, pero lo que no está es el código fuente de la clase heredada, `AgentMobile`).

Resultado: en éste caso el agente se mueve perfectamente, visualizando su saludo y permaneciendo en el destino. Vemos como conseguimos la movilidad del código en éste caso.

- **Caso número 2:** el código fuente del agente está también en el destino, en el mismo paquete y el mismo nombre de la clase, pero lo he modificado, haciendo que el método saludar saque por la salida un mensaje diferente.

Resultado: en este caso el agente se ha movido, pero el código del agente ha sido substituido por el código del agente de la versión del destino. Podemos tener problemas en el momento de desarrollo de los agentes si no tenemos en cuenta este factor. El código resulta que es substituido en el destino. Es decir, no migra a través de la red, sino que se carga en memoria los bytecodes (el código que es interpretado por la JVM) de la versión local. Puede parecer curioso este comportamiento, pero por otro lado sería poco eficiente si en la serialización se comprobara si el código es realmente el mismo o no. Así que podemos pensar que lo que se hace al serializar una clase para enviarse por la red (el código), es comprobar si ésta no se encuentra en el destino, verificando la ruta de la

clase dentro del classpath de la aplicación en ejecución. Si no se encuentra, se envía, y si se encuentra se carga en memoria la versión local en caso de haberse cargado anteriormente (Java incorpora un mecanismo para cargar los bytecodes de las clases en el momento oportuno, mediante los *Cargadores de Clases*).

- **Caso número 3:** el código fuente del agente no está en el destino (será código móvil). Pero esta vez lleva una objeto(instancia) de tipo `Integer` con valor igual a 1.

Resultado: en este caso el agente ha saludado en la llegada y ha sacado por la salida estándar el valor 1 del `Integer` instanciado. Ha funcionado como esperábamos, llevándose consigo el código más el estado. Hay que notar que la clase `Integer` es serializable.

- **Caso número 4:** el código del agente no está en el destino (será móvil). Pero esta vez lleva una instancia de un objeto de tipo `File`, es decir, un fichero que se encuentra en el origen.

Resultado: si no utilizamos para nada la instancia de tipo `File`, el agente se ejecuta con normalidad. Pero si introducimos un código en la llegada del agente que haga uso del `File` para leerlo, montando un objeto de la clase `BufferedReader`, el agente se produce un excepción ya que no encuentra el fichero en el espacio del destino. Es decir, no se lleva un fichero abierto, como es lógico, porque entre otras cosas no sabría dónde ubicarlo en el destino.

- **Caso número 5:** el código del agente no está en el destino (será móvil). Esta vez, al llegar al destino, instancia un objeto de una clase que he creado, es decir, un objeto de una clase que no pertenece al API de j2se (*java 2 standard edition*) de Java, y que tampoco está en el destino (deberá ser móvil).

Resultado: funciona correctamente, es decir, se lleva el código de la clase por completo, que se carga (los bytecodes) en memoria la primera vez que se usa en el destino, crea varias instancias y se pueden utilizar con normalidad.

- **Caso número 6:** el código del agente no está en el destino (será móvil). Esta vez, a diferencia del caso anterior, lleva una instancia de un objeto de la clase que he creado (la misma que la del caso anterior) que no está en el destino (deberá ser móvil) y por lo tanto deberá llevarse su estado.

Resultado: ha fallado. El agente no ha llegado a migrar, lanzando una excepción en el origen, de tipo `java.io.NotSerializableException`. Aquí lo que ocurre es que el código de la clase sí que se puede transportar por la red (código móvil) pero no puede recuperar su estado porque la clase no es serializable y por tanto no se pueden binarizar sus valores para ser llevados al destino.

- **Caso número 7:** idéntico caso que el anterior, pero modificando la clase creada implementando la interface `Serializable` del paquete `java.io`. Esta vez, a diferencia del caso anterior, lleva una instancia de un objeto de la clase

que he creado (la misma que la del caso anterior) que no está en el destino (deberá ser móvil) y por lo tanto deberá llevarse su estado.

Resultado: ha funcionado. El agente ha migrado y se ha movido tanto el código como el estado. Es decir, el código de la clase ha viajado por la red (código móvil), como en el caso anterior, y además ha mantenido los valores que tenía previa acción de movimiento.

Por tanto ya hemos hecho un estudio profundo sobre cómo funciona la movilidad de los agentes en la versión 3.3 de JADE [JADE, PG33] y versión 1.4 y 1.5 de Java. En resumidas cuentas, podemos destacar que el **código** de las clases es siempre móvil, y que el **estado** de los objetos (instancias) de las clases (es decir, el conjunto de valores para sus atributos, en un cierto momento) se mantienen siempre y cuando la clase a la que pertenece el objeto, y de manera recursiva, las clases de todos sus atributos, implementen la interface `java.io.Serializable` (y por tanto son potencialmente serializables).

Por último, destacar que un agente, una vez movido a otra localización, puede acceder a todos los recursos (disco duro, red, etc.) de la máquina sobre la que se está ejecutando que estén permitidos. Es decir, tiene los mismos privilegios que el resto de elementos que participan en su entorno de ejecución local. El ámbito o contexto de ejecución (workspace) a nivel de directorios es el mismo que el del programa de destino. Los programas ejecutados en una máquina virtual de java tienen como ruta de directorio (path) base, el directorio desde el cual se ha lanzado su ejecución.

2.4 Tecnologías Web

Para la segunda parte del proyecto, que describiré con más detalle en un capítulo posterior, he empleado tecnologías relacionadas con la Web, debido a las ventajas que éstas ofrecen, para interactuar con el usuario final de la aplicación. Las aplicaciones basadas en la tecnología Web, son programas que corren en servidores Web y utilizan páginas Web como interfaz de usuario, accediendo a través de programas clientes (navegadores) disponibles en cualquier tipo de computador. De cara al usuario final, este tipo de software será más sencillo, económico, independiente de la plataforma, más móvil, más confiable, y a menudo más poderoso que el software de escritorio.

Si bien hasta hace poco el software de escritorio era considerablemente más potente, esta diferencia se está viendo reducida constantemente gracias en parte a la cada vez mayor velocidad de las conexiones a Internet, y también gracias al desarrollo de nuevas tecnologías abiertas que están mejorando aspectos importantes tanto para los usuarios finales como para los desarrolladores. Entre estas ventajas se pueden valorar las siguientes:

- **Compatibilidad multiplataforma.** El hecho de que tecnología Web está basada en estándares abiertos, hace que las aplicaciones Web logren alcanzar

una compatibilidad multiplataforma más fácilmente que las aplicaciones de software descargables. Varias tecnologías incluyendo Java (J2EE [J2EE06]), PHP [PHP06], Flash [Flash06], ASP.net [ASP06] y AJAX (*Asynchronous Javascript And Xml*), entre otras, permiten un desarrollo efectivo de programas soportando todos los sistemas operativos principales.

- **Actualización.** De cara al usuario final, las aplicaciones basadas en web tienen la ventaja de estar siempre actualizadas sin requerir que el usuario deba preocuparse por ello y de forma transparente. Además, en entornos cooperativos donde hay muchos clientes, resulta una tarea ardua el mantenimiento de software de escritorio, necesitando incluso asistencia técnica especializada para este tipo de labores.
- **Inmediatez de acceso.** No es necesario llevar procesos largos de instalación y configuración en el lugar de los clientes. Con las aplicaciones web, basta la creación de una simple cuenta de usuario para acceder de forma inmediata a la aplicación, que esta lista para trabajar sin importar cuál es su configuración o su hardware.
- **Facilidad de prueba.** Las fases de prueba para valorar las aplicaciones en tanto a su funcionalidad práctica como a su robustez, se ven reducidas tanto en tiempo como en dinero, gracias a un proceso más ágil.
- **Clientes más ligeros.** Las aplicaciones basadas en web consumen muchos menos recursos computacionales (a excepción de la conexión), tanto en memoria RAM como en procesamiento, ya que los programas residen y corren en su mayor parte en servidores del proveedor, quien dispone de hardware especializado para soportar cargas de trabajo mayores, que el cliente no debe adquirir.
- **Precio.** El despliegue que deben de realizar las empresas desarrolladoras de software para vender su producto es mucho menor en el caso de las que utilizan las tecnologías de la web (por ejemplo, la distribución del producto es menos costosa), traduciéndose directamente a una reducción del precio final del producto.
- **Los datos también van online.** Los datos están almacenados en el servidor, lo cual implica que el usuario final no necesite adquirir potentes servidores para almacenarlos (datacenters). A su vez, como las empresas disponen de especialistas, los datos serán almacenados y distribuidos de una forma más robusta y fiable (en el sentido que disponen de herramientas especializadas en copias de seguridad, recuperación ante fallos, etc.).
- **Múltiples usuarios concurrentes.** Las tecnologías basadas en web están pensadas para soportar a múltiples usuarios accediendo concurrentemente, y resulta más sencillo cooperar entre los distintos usuarios de un entorno.

Una vez vistas las numerosas ventajas que nos ofrecen las tecnologías web, hay que decidir por cuales se optan. Las tecnologías que he seleccionado han sido las siguientes:

- Para la generación de páginas dinámicas del lado del servidor:

- **J2EE (Java 2 Enterprise Edition)**

Cabe decir que en cuanto a la generación de páginas dinámicas del lado del servidor hay tres grandes tecnologías que se han impuesto en los últimos años: PHP, ASP.NET y J2EE [J2EE06] (JSPs y servlets, que en el fondo un JSP acaba siendo un servlet).

Las 3 opciones nos ofrecen las características necesarias para este proyecto, si bien cabe destacar que PHP es una tecnología libre, quizás la más productiva de todas para aplicaciones de talla no demasiado grande. Pero quizás a día de hoy soporte menos la carga en aplicaciones que requieren soportar muchos usuarios, o no sea la más idónea en entornos complejos que manejen muchas transacciones. En estos casos, las alternativas que ofrecen tanto ASP como JSP son más robustas y escalables, por lo menos en la actualidad.

Y puestos a elegir entre ASP y J2EE (*Java 2 Enterprise Edition*: JSP y servlets), se puede valorar que la implantación de ASP está limitada para arquitecturas basadas en tecnología Microsoft. J2EE se beneficia del desarrollo de APIs por parte de la comunidad Java, y además los servidores pueden correr en cualquier arquitectura hardware y sistema operativo que soporte la tecnología Java. Tenemos contenedores de aplicaciones (servidores) J2EE libres como Apache Tomcat, que nos permiten correr las aplicaciones tanto en entornos UNIX [UNIX06] (Linux [LINUX06], Solaris [SOLARIS06], xBSD [BSD06], etc.) como en entornos de Microsoft [MICROSOFT06]. Por tanto se optado por **J2EE**.

- Para la persistencia de datos (almacenamiento de datos):

- **MySQL**

En cuanto a la persistencia, había que elegir una base de datos para almacenar datos relacionados con los usuarios y sus trabajos. Como no se requiere mucha carga, el motor de bases de datos de licencia libre MySQL [MySQL06] es más que suficiente para responder a las necesidades de este proyecto.

- En el lado del cliente:

- **JavaScript**

Por otro lado, en el lado del cliente se ha utilizado la ya clásica tecnología JavaScript [JS, Spec] para hacer más rica la interactividad con el entorno gráfico por parte de los usuarios finales.

- **Ajax (Asynchronous Javascript And XML)**

Si bien destacaba en el inicio de este apartado que las tecnologías web son cada vez más ricas y se están aproximando a las capacidades de las de escritorio, se puede justificar con la aceptación de una reciente tecnología que

está de moda en la actualidad conocida con el nombre de AJAX, acrónimo de Asynchronous Javascript And XML (JavaScript y XML asíncronos). Como su acrónimo indica, no se trata de una tecnología nueva, sino de la reunión de un conjunto de tecnologías maduras que se combinan entre sí para crear aplicaciones más interactivas. Gigantes del software como Google, Microsoft y Yahoo, por mencionar algunos, están apostando fuerte por esta técnica para enriquecer sus productos orientados a la web.

Básicamente AJAX se basa en la posibilidad de realizar llamadas asíncronas desde el cliente con un lenguaje de Script (JavaScript) al servidor, manteniendo una comunicación intercambiando datos en formato XML sobre HTTP, donde el servidor responde finalmente con una respuesta que genera código HTML, el cual puede ser incorporado de manera dinámica al contenido del documento que el usuario está visualizando, sin recargar la página, gracias a que desde el propio JavaScript se puede acceder a los elementos HTML que estructuran la visualización del documento, por medio de una estructura de datos que representa el contenido de la página, conocido con el nombre de DOM (*Document Object Model*). Éste es el principal cometido de AJAX, permitir que las aplicaciones web sean más interactivas permitiendo una mejor experiencia para el usuario final.

- **applets de Java**

Se han utilizado applets de Java en el cliente para obtener una mayor potencia a la hora de construir algunos elementos de la interficie gráfica de usuario.

Más adelante se verá en qué puntos del proyecto estás tecnologías han tenido su aparición y se comprobará en qué lugares han encajado adecuadamente.

3. Descripción general de la arquitectura del sistema dinámico de ejecución distribuida basada en agentes

Recordemos que unos de los principales objetivos de este proyecto es la creación de un sistema dinámico para poder ejecutar código de forma distribuida y paralela, que presenta la novedad de estar basado en un sistema multi-agente, el cual nos proporciona una serie de características que encajan muy bien para cumplir con los requisitos que dicho sistema exige, como pueden ser el hecho de tener una gran capacidad de control dinámico de la ejecución, movilidad de código, acceso remoto a recursos computacionales, comunicaciones complejas entre elementos software distribuidos, cooperación y a la vez capacidad de independencia, robustez, etc.

Aquellos problemas que por su naturaleza puedan ser divisibles en “trozos” más pequeños e independientes, serán candidatos ideales para ser ejecutados en un entorno como el que se ha creado, ya que podrán obtener un mejor rendimiento. Además, se ha pretendido que el sistema sea capaz de adaptarse de forma sencilla a nuevos problemas, gracias a que el diseño está basado en un enfoque genérico, habiendo desarrollado componentes capaces de ser adaptados a diferentes necesidades para poder optimizar el rendimiento de la ejecución en cada caso.

Por tanto, desde un inicio, el diseño del sistema ha estado marcado por la búsqueda de cumplir con los siguientes objetivos:

- Coordinar la ejecución distribuida de tareas independientes de forma diligente.
- Gestionar eficientemente los recursos (la potencia de cálculo, la memoria disponible, el ancho de banda de las conexiones, etc.) que nos ofrecen los computadores que formarán parte del sistema.
- Conseguir robustez ante la aparición de fallos inesperados. Básicamente, que sea capaz de recuperarse ante la caída (o fallada) de computadores que forman parte del sistema.
- Desarrollar componentes genéricos capaces de adaptarse de manera sencilla a los diversos problemas, para poder optimizar tanto la gestión de los recursos como el rendimiento de la ejecución en cada caso.
- Conseguir un sistema escalable, manteniendo un rendimiento aceptable ante un aumento considerable de la carga.

El framework de JADE está desarrollado en Java, y por tanto es cierto que desde un inicio era necesario desarrollar el sistema en Java. Pero a decir verdad, un lenguaje de programación orientado a objetos como Java nos ofrece unas características ideales para conseguir desarrollar un sistema de este tipo (con las características comentadas). Así pues, gracias al conocimiento y el uso del paradigma de la programación orientada a objetos, el diseño se ha construido con la idea clara de producir un **código modular, genérico, extensible y reutilizable**. De ésta forma se

consigue cumplir con el objetivo general que persigue la Ingeniería del Software, que es obtener un software de calidad, es decir, un software que se adecua de la mejor manera posible a los requisitos exigidos. A continuación describo brevemente en que consisten las características mencionadas en éste párrafo sobre el código desarrollado:

- **Modular:** los componentes están claramente separados, de forma que cada uno se especializa en una función determinada, garantizando un acoplamiento mínimo.
- **Genérico:** el código está escrito pensando en obtener un alto grado de abstracción, de forma que nos permita adaptarlo en última instancia a perfiles particulares que se adapten mejor a un conjunto homogéneo de problemas. En la programación orientada a objetos, esto se puede conseguir haciendo un uso adecuado de interfaces(establecimiento de contratos), clases abstractas, relaciones de herencia entre clases, polimorfismo de inclusión (y por ende, ligadura dinámica).
- **Extensible:** un código se considera que es extensible si se adapta fácilmente a los cambios en los requisitos, de tal manera que la inclusión de nuevos componentes que se prevén incorporar en un futuro, no supone modificar de forma significativa la estructura general del código. Además, como he destacado, debe ser relativamente fácil adaptarse a un cambio (coherente) en las especificaciones. Ésta características se pueden conseguir con las anteriores que se han comentado en este apartado.
- **Reutilizable:** hablamos de software reutilizable cuando tiene la bondad de ser incorporado (completamente o alguna/s parte/s) en nuevas aplicaciones sin ser modificado.

La arquitectura está diseñada siguiendo el paradigma cliente-servidor (véase la *figura 7*). A continuación se describe de forma general su diseño, para luego explicar con más detalle sus principales componentes.

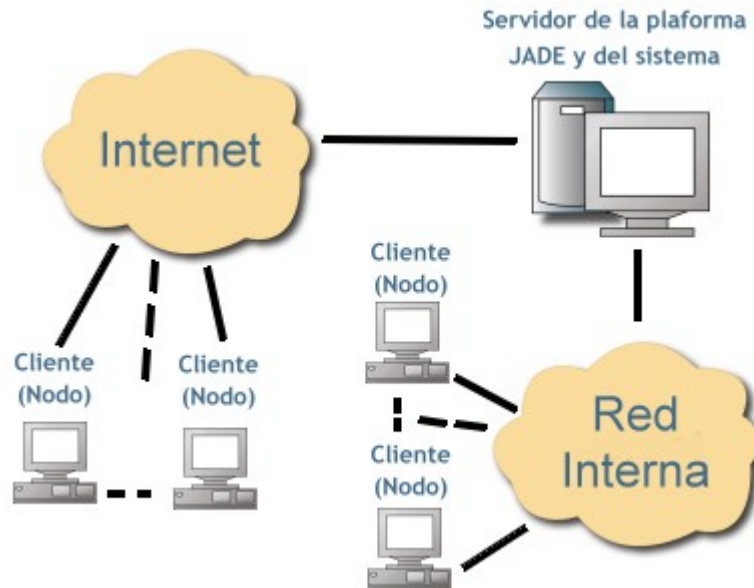


Figura 7. Esquema gráfico de la organización física de los computadores que forman el sistema, construido sobre la plataforma JADE

3.1 Descripción básica de la arquitectura

Por un lado, el **servidor** es la máquina central del sistema que gestiona y utiliza todos los recursos coordinando las tareas necesarias.

El resto de computadores de la arquitectura serán llamados **nodos**, que serán los que entreguen sus recursos al sistema para que puedan ser utilizados para llevar a cabo las tareas que se requieran en un momento determinado. Para ellos, se ha implementado un software (programa cliente) que deben tener instalado en su computador, y que deben de ejecutar para formar parte del sistema. Más adelante explicaré como funciona.

Para que los nodos se puedan comunicar con el servidor, como se puede observar en la *figura 7*, basta con que estén conectados a la red de Internet y no haya ningún componente de red que impida dicha conexión (básicamente, *firewalls*), o bien se puedan comunicar a través de una red interna (por ejemplo una LAN).

Es necesario establecer un mecanismo por el cual los nodos puedan ser inscritos en el sistema para formar parte de él y poder compartir así sus recursos. Este mecanismo puede ser estático (menos conveniente) y dinámico. Como es lógico, en un arquitectura de este tipo, es mejor, pero a la vez más complejo, implementar un **mecanismo dinámico** que soporte la **inscripción** al sistema por parte de los nodos en cualquier momento. También debe ser **robusto ante fallos** de los nodos, ya que estos pueden dejar de participar en el sistema en cualquier momento por muchas causas (ya sea porque el usuario que utiliza el computador que hace de nodo decida

cerrar el programa cliente, o bien porque el computador se apague inesperadamente, etc.).

Pero antes de que un nodo pueda darse de alta en el sistema, el servidor debe estar operativo. Como se puede intuir, para llevar a cabo el control de alta de los nodos al sistema, en el servidor debe de existir un servicio adecuado para ello. Además, en el servidor se tienen que realizar otras muchas tareas de coordinación y control.

Es importante diferenciar en este punto que la plataforma de JADE tiene su propia organización (contenedores, agentes, etc.) y mi objetivo es aprovechar esta organización para colocar encima una capa lógica propia (nodos, recursos, trabajos, etc.) para construir el sistema. Para aclarar las ideas, pongo como ejemplo a los nodos del sistema. El sistema manejará una estructura de datos llamada **Node**, que tendrá una serie de atributos asociados a un nodo del sistema, entre los cuales, uno será la localización(Location) del contenedor que tiene asociado. Notemos aquí como JADE nos permite alcanzar un grado más elevado de abstracción, permitiendo al desarrollador despreocuparse de la capa de red (sockets, direcciones ip, puertos, etc.), identificando a un nodo del sistema por la localización del contenedor que tiene asociado. Para enviar un agente a ese nodo desde el servidor, bastará con conocer dicha localización, la cual estará almacenada en la estructura de datos que antes he mencionado (Node).

3.2 El cliente

De los clientes, vistos como nodos, se pretende que ofrezcan sus recursos para que sean utilizados por el sistema. Por tanto, cuanto mejor sean sus recursos (ya sea en base a una mayor potencia o una mayor capacidad) más interesante resultará. El hecho de que el software del cliente también esté desarrollado en Java presentaba un problema inicial en base al tema de los recursos. Resulta que todo programa Java se ejecuta sobre una instancia de la máquina virtual (JVM, *Java Virtual Machine*). La ejecución del mismo acaba siendo un proceso más para el sistema operativo, pero lo que hace la máquina virtual en cada instancia es reservar de antemano un espacio de memoria limitado para la ejecución. Por defecto, el tamaño de es de 64 megabytes de RAM. Pero esto no interesa, ya que lo ideal es que el sistema aproveche la mayor cantidad posible de los recursos disponibles, y como no, la memoria es uno de los más importantes. Para solucionar éste inconveniente, en el momento de ejecutar el software cliente se le puede indicar unos parámetros a la máquina virtual donde se nos permite establecer la cantidad de memoria RAM que queremos utilizar (debe de ser inferior o igual al máximo disponible físicamente). Como cada computador tendrá una cantidad propia, he realizado un programa de instalación (también en java) que comprueba la cantidad de memoria física disponible en el computador, y posteriormente genera un fichero .bat (para Microsoft) y .sh (para UNIX) que serán los que lancen el software cliente, en los cuales se indica a la instancia de la máquina virtual que utilice dicha cantidad. De esta manera, podemos utilizar toda la memoria física disponible de cada computador.

3.2.1 Los clientes como nodos de la plataforma

Como ya he comentado, los clientes del sistema se llamarán nodos. Una vez ejecutado el programa cliente, se inscriben en la plataforma JADE, creándose un contenedor de agentes único asociado durante toda su ejecución, y donde estarán los agentes que requieran ser ejecutados en el computador sobre el que corre dicho programa. Hasta ese momento solo la plataforma JADE conoce su existencia. Por tanto, es necesario que el cliente sea reconocido por el sistema para que acabe siendo un nodo. Para ello, en el servidor hay un servicio de registro en el sistema, el cual debe de ser solicitado en este momento. Tal servicio será ofrecido por un agente que permanece en el servidor. Así pues, en el cliente se debe de crear un agente que sea capaz de solicitar tal servicio, para llevar a cabo el registro completo. Este agente se llama **AgentRegistry (AR)**. En este acto comunicativo, el agente AR informa al agente del servidor que ofrece el servicio de registro, que se ha dado de alta en la plataforma JADE, y posteriormente, le manda la información de los recursos que pone a disposición el computador cliente. Una vez completado el registro, el cliente ya es reconocido como nodo del sistema, poniendo a disposición del sistema los recursos que ha ofrecido en el momento del registro.

3.2.2 Información sobre los recursos de los nodos

Es necesario conocer que recursos nos ofrece un nodo, para saber que capacidad tendrá para ejecutar tareas. Es decir, cada tipo de tarea necesita disponer de una serie de recursos para su ejecución, y por tanto resulta más que conveniente conocer con detalle los recursos que ofrece cada nodo, para poder hacer una gestión y utilización óptima de los mismos en base a los requerimientos de dichas tareas, buscando un correcto balanceo de la carga. Para ello, el software del cliente debe de proporcionar un mecanismo que sea capaz de obtener los recursos disponibles, con la dificultad añadida de que se va a ejecutar sobre diversos tipos de arquitecturas hardware y sistemas operativos diferentes. Extraer toda la información que pretendía sobre los recursos de los nodos, hasta la versión 1.4 de java, no era viable por medio de su API. Entonces, en un primer momento consideré la opción de conseguir comunicarme con un programa nativo al computador cliente para conseguir acceder a llamadas del sistema que consiguieran obtener una información de bajo nivel. En un primer momento, hice un programa en c sobre Windows que accedía al API de Microsoft Win32, mediante sus librerías de enlace dinámico (dll, *dynamic link library*) y conseguí obtener la información de la ram disponible. Algo más costoso fue conseguir, desde el mismo programa en c, obtener información sobre la frecuencia del microprocesador (a través del registro de Windows). Finalmente, una vez conseguido todo lo anterior, tenía la nueva dificultad de comunicar un programa en java (el programa cliente) con el programa de c para obtener su información. Esto lo conseguí ejecutando el programa de c desde el programa de java, comunicando canales de entrada y salida de ambos programas (entrada del programa de java con salida del programa de c y

viceversa) para establecer una conversación entre ellos. Como se puede ver, a pesar del logro, este programa solo sirve para Windows. Tendría que desarrollar otro distinto para Linux, otro para Solaris (seguramente podría servir la mayor parte del código fuente del de Linux), otro para MacOS, etc., siendo una tarea engorrosa que ofrece poca extensibilidad (no se adaptaría a nuevos sistemas operativos).

Para superar esta barrera, decidí en un primer momento implementar una interfaz (llamada `Monit`) en java (un especificación abstracta) para monitorizar los recursos que tiene el computador, la cual utilizaría en mi programa, de manera que me independizara de la implementación de éste componente. Así, utilizando esta interface creada, el código del cliente no se verá modificado sea como sea la manera de su implementación, en el sentido que si cambia la manera de obtener dicha información de bajo nivel, no se vea afectado en nada más que implementar una nueva clase que siga soportando la interface.

Después de crear la interfaz, por suerte, me enteré que con el API de java 1.5 habían incorporado nuevas clases de monitorización que te informaban de los recursos más esenciales. Entonces, solo hacía falta hacer una implementación de una clase nueva que soportara la interface `Monit`, llamada `MonitImpl` que utilizará el API de java 1.5 para tal propósito. Y decía por suerte, porque el hecho que esté diseñado todo en java, hace que el código, ahora sí, sea extensible (eso sí, los ingenieros de Sun son los que habrán tenido que desarrollar el código nativo necesario para hacer las clases de su API compatibles con todos los sistemas operativos que tengan soporte para java 1.5). Visto las posibilidades que nos ofrece java 1.5 en éste apartado, he ampliado la interfaz añadiendo nuevos métodos que aportar más información que si bien no será requerida para éste proyecto, se ofrece por si en algún momento resulta necesaria. Por curiosidad, por lo que he visto deduzco que han los desarrolladores de Sun han aplicado, de forma acertada, el patrón de diseño *Abstract Factory* para éste propósito, y conseguir así esta independencia con los diferentes sistemas operativos. Ahora mostraré las cabeceras de los métodos de la interfaz `Monit`:

```
public InformationMagnitude getInformationMagnitude();

public void setInformationMagnitude(InformationMagnitude mag);
```

InformationMagnitude es un clase de tipo Enum, que sirve para poder trabajar con las diferentes magnitudes que existen en el mundo de la informática para cuantificar cantidades de memoria. Así pues, se deberá indicar con qué magnitud queremos trabajar, ya que cuando pedimos, por ejemplo, la cantidad de memoria física del sistema, debemos de saber en que unidad de medida está la respuesta numérica (por ejemplo en megabytes).

```
public String getArchitecture();
```

Devuelve un String que identifica al arquitectura del computador

```
public float getJVMTotalPhyscalMemorySize() throws
NonSpecifiedMagnitudeException;
```

Devuelve la cantidad de memoria física disponible para la instancia de la JVM

```
public int getNumberOfProcessors();
```

Devuelve el número de procesadores que tiene el computador

```
public String getOperatingSystem();
```

Devuelve un String que identifica al sistema operativo del computador

```
public String getOperatingSystemVersion();
```

Devuelve la versión del sistema operativo del computador

```
public float getCommittedVirtualMemorySize() throws  
NonSpecifiedMagnitudeException;
```

Devuelve la cantidad de memoria virtual garantizada disponible para ejecutar los procesos

```
public float getFreePhysicalMemorySize() throws  
NonSpecifiedMagnitudeException;
```

Devuelve la cantidad de memoria física libre del computador

```
public float getFreeSwapSpaceSize() throws  
NonSpecifiedMagnitudeException;
```

Devuelve la cantidad de espacio swap libre del computador

```
public float getTotalPhysicalMemorySize() throws  
NonSpecifiedMagnitudeException;
```

Devuelve la cantidad de memoria física del computador

```
public float getTotalSwapSpaceSize() throws  
NonSpecifiedMagnitudeException;
```

Devuelve la cantidad de espacio swap del computador

Utilizando esta clase de monitorización, el programa cliente puede ahora conocer los recursos de su máquina. Ahora, lo que el agente AGM le pide al cliente, tras comunicarle que se ha inscrito en la plataforma JADE, es una estructura de datos cuyos atributos contienen toda la información que el sistema requiere para gestionar y utilizar posteriormente sus recursos. Para ello se ha construido dos nuevas clases:

- **Computer:** contiene información relacionada con el computador. Tiene 4 atributos:

- `TotalPhysMem`: memoria física total
- `FreePhysMem`: memoria física disponible
- `Freq`: frecuencia del microprocesador
- `Bandwide`: ancho de banda de la conexión a Internet

- **NodeInformation**: contiene información relacionada con el nodo. Tiene 4 atributos de los siguientes tipos:

- `Location location`: la localización de su contenedor asociado
- `String containerName`: el nombre del contenedor
- `AID agentNodeManagement`: el identificador del agente que gestiona el nodo. (En un primer momento parecía ser necesario, pero más tarde se verá que no es necesario. Se mantiene por precaución)
- `Computer c`: objeto de la clase descrita anteriormente.

Como se ve puede observar, la clase `NodeInformation` contiene tanto información relativa a la plataforma JADE como la propia del computador.

Por tanto, en el momento del registro al sistema, el cliente da a conocer información relativa sobre él, tanto de gestión (por ejemplo, su localización) como de recursos, por medio de la clase `NodeInformation`. Estos recursos quedan a disposición del sistema, quien a partir del registro reconoce al cliente como un nodo.

3.3 El servidor

Ahora que se ha descrito como funciona la parte del cliente, el siguiente paso es describir el otro extremo, el servidor del sistema. La "inteligencia" del sistema, en cuanto a la gestión y la coordinación de su funcionamiento, ha sido delegada un agente especial, que recibe el nombre de **AGM** (*AgentGridManager*).

Este agente, participa en prácticamente todas las acciones que realiza el sistema, pero como es lógico, necesita de otros componentes especializados en ciertas funciones, para darle el soporte requerido y tener un diseño correctamente modularizado.

En los siguientes apartados se conocerán con detalle los agentes y los componentes que se encuentran en el servidor, pero ahora se muestra una lista con una breve descripción de todos ellos, para tener una idea global de su estructura:

- **agente AGM (AgentGridManager)**: como se acaba de comentar, este agente será el que aporte la "inteligencia" en el sistema. Interviene prácticamente en todas las acciones, y se apoya en los diferentes componentes para llevar a cabo el funcionamiento. Permanece en el servidor durante toda la ejecución del sistema. A grandes rasgos, ofrece un servicio de registro al sistema para los nodos, además de tratar las peticiones que llegan al sistema de nuevas tareas, y de recibir y almacenar los resultados que estas producen. También interviene en el mecanismo de recuperación del sistema ante fallos de los nodos. Para

todas realizar todas estas responsabilidades, se apoya de manera adecuada en los demás componentes que se mencionan en este punto.

- **agente APEM (AgentPlatformEventsManager):** agente encargado de escuchar y tratar los diferentes eventos que se producen en la plataforma de JADE. Por tanto, su ciclo de vida comenzará cuando el servidor se ponga en funcionamiento y acabará cuando el servidor termine su ejecución, igual que en el caso de agente AGM. De todos los eventos que existen, habrán unos cuantos que interesan ser tratados de cara al sistema. Como se verá más adelante, este agente interviene de manera clave para conseguir que el sistema sea robusto ante fallos (caída de nodos).
- **agente AW (AgentWork):** en este caso, se trata de un tipo de agente que se crea con la finalidad de hacerle responsable de la ejecución completa de un trabajo. Se crearán tantos agentes de este tipo como trabajos se vayan creando dinámicamente. Básicamente, se encargará de llevarse un trabajo creado en el servidor hacia un nodo, para que lo ejecute una vez allí y posteriormente devuelva los resultados producidos al agente AGM. Una vez entregue estos resultados, su ciclo de vida termina.
- **NodeManager:** se trata de un componente que se encargará de gestionar los nodos y recursos del sistema. Por un lado, deberá almacenar los nodos de forma óptima. Por otro, y ligado con lo anterior, deberá adoptar una política de scheduling (planificación) adecuada, para que en el momento preciso seleccione al mejor nodo para enviarle una nueva tarea, para conseguir un correcto balanceo de la carga.
- **ServerIDManager:** es un servidor concurrente, que se encarga de gestionar un espacio de nombres (namespace), que no es más que un ámbito en el cual un identificador debe ser único. Los nombres de los agentes que forman parte de una plataforma JADE, deben ser únicos. Y esto es lo que precisamente nos ofrece este componente.
- **ServerRequest:** también es un servidor concurrente, pero su misión es la de "dialogar" con el exterior con el objetivo de crear una puerta de entrada al sistema, para que se le pueda solicitar tanto la ejecución de nuevas tareas, como la obtención de los resultados que producen las mismas.
- **Node:** abstracción software que representa a un nodo del sistema.
- **Job:** abstracción software que representa a una tarea que será ejecutada por un nodo del sistema.

3.3.1 Diseño genérico de un trabajo (JOB)

El objetivo principal del sistema construido es poder ejecutar tareas de forma paralela, utilizando de manera eficiente los diversos nodos. De hecho, es necesario que un problema pueda descomponerse en diferentes tareas que sean independientes entre sí. Así pues, tiene sentido pensar en una abstracción software que encapsule métodos y atributos relacionados con las tareas. Si además se pretende que el sistema pueda en un futuro ejecutar diversos tipos de tareas diferentes, el enfoque debe estar orientado a la construcción de un componente genérico. Resulta interesante que éste sistema distribuido pueda resolver diversos problemas que por su naturaleza se adaptan bien a un entorno de tipo grid.

Para ello, he construido un clase abstracta llamada `Job`, de forma que en el sistema se trabaje con éste tipo de manera genérica y así conseguimos la extensibilidad deseada. Sus atributos, de modificador protegido (`protected`) son los siguientes:

- `String jobId`: identificador del trabajo. Útil para su gestión.
- `JobState state`: indica el estado en que se encuentra el trabajo. `JobState` es un tipo enumerado que contiene los diversos estados del ciclo de vida de un trabajo (`CREATE`, `SENDING`, `PROCESSING`, `RECEIVING`, `EXTRACTING_RESULTS`, `KILL`).
- `Location destination`: indica la localización destino dónde se ejecutará realmente dicho trabajo (como veremos, la localización se escogerá mediante una política de scheduling apropiada para un correcto balanceo de la carga, en tiempo de ejecución).
- `String[] args`: array de argumentos de tipo `String` que se le pasaran en el momento de su creación. Opcionales para cada tipo de tarea específica.
- `List<NetFile> inNetFiles`: lista de ficheros de entrada necesarios para el ejecutar el trabajo (si requiere). Como podemos observar, efectivamente, no se trata de la clase `File` del paquete `java.io` del API estándar. Se trata de una nueva clase que he desarrollado para que los ficheros sean fácilmente y eficientemente transportados por la red. Sus detalles de implementación se describen en próximo apartado. De momento, la idea es que cada instancia servirá para manipular un fichero.
- `List<NetFile> outNetFiles`: lista de ficheros de salida, producidos por la ejecución del trabajo. Evidentemente, su uso será opcional.
- `List<Serializable> outObjects`: lista de objetos de salida producidos por la ejecución del trabajo, que soporten la serialización, ya que los resultados producidos deben de ser enviados por la red desde un nodo hacia el servidor.

Por otro lado, su constructor y sus métodos son los siguientes:

- `public Job(String jobId, String[] args, List<NetFile> inNetFiles):` el constructor
- `public boolean saveJob(String relativePath):` guarda por serialización el Job sobre el que se ejecuta el método en el disco duro, en el path relativo donde se ejecuta. Así pues, todos los atributos de un Job específico deben de ser serializables, tanto para viajar por la red como para tener la opción de ser guardados.
- `public static Job readJob(String relativePath, String jobId):` recupera un Job guardado anteriormente por medio del empleo del método anterior. Evidentemente, éste método es estático, ya que no tendría nada de sentido realizar una instancia de un objeto de tipo Job para recuperar otro Job.
- `public abstract boolean execute():` éste método es el más relevante. Una vez que un trabajo (Job) llegue a un nodo, como se verá próximamente, será ejecutado por un agente. La ejecución pasa por invocar éste método. Como se puede observar, es un método abstracto que cada tipo de Job diferente deberá de implementar adecuadamente. Vemos cómo con éste diseño de la clase Job conseguimos un código genérico, extensible y reutilizable.
- `public int compareTo(Object o):` éste método se implementa cumpliendo con la interfaz implementada Comparable. La comparación se basa en la comparación de identificadores.

Los demás métodos son “getters y setters” para obtener y establecer respectivamente los diferentes atributos de la clase.

Así pues, ya disponemos de un tipo de trabajo genérico gracias al cual se podrá tratar de manera uniforme a todos los diferentes tipos trabajos. Evidentemente, por cada tipo de trabajo diferente que queremos crear, deberemos desarrollar una clase nueva que herede de Job y que implemente el método `execute`, además de tener la posibilidad de añadir nuevos métodos y atributos si es necesario.

Los trabajos serán generados en el propio servidor, que mediante una política de planificación (scheduling) apropiada, decidirá a que nodo enviarlo para su ejecución. Para que se realice su transporte y correcta ejecución, así como para el envío del resultado final, es necesario que otro elemento se encargue de dicha responsabilidad. Éste elemento es un nuevo agente, denominado AgentWork (AW, agente de trabajo).

3.3.2 Los agentes responsables de los trabajos (AgentWork)

El agente AgentWork(AW) interviene en el momento en que se haya generado un nuevo trabajo en el servidor y se haya decidido en que localización deberá ejecutarse mediante la política de scheduling. El servidor crea un agente de este tipo por cada trabajo nuevo, delegándole la responsabilidad de llevárselo consigo al nodo de destino,

y una vez allí ejecutar el trabajo. Una vez acabado, deberá de comunicarse con el servidor para el envío de los resultados.

Como que un agente va a trabajar con la estructura de datos Job, no será necesario crear diferentes tipos de agente AW especializados. Esto se ha conseguido gracias al hecho que el componente Job sea genérico, y por tanto sea tratado de manera uniforme ante una cierta variedad de tipos (tipos de trabajos). Llegados ha este punto, se observa como el **código móvil** producido con la **movilidad de los agentes** juega un papel importante en la arquitectura del sistema dinámico. Vemos como el software del cliente no necesita albergar todos los diferentes tipos de trabajos, de hecho no contiene ni si quiera la clase abstracta Job. De esta manera, la inclusión de diferentes tipos de trabajos en el futuro, no requerirá de actualizaciones del software cliente, objetivo fundamental para conseguir la extensibilidad que se pretendía desde un principio. Esta es una característica que ningún sistema grid actual permite, y por tanto es una de las grandes novedades que este sistema aporta.

3.3.3 El servidor de nombres (namespace)

Anteriormente comenté que en la plataforma JADE todos los agentes contienen un identificador (AID, *Agent Identifier*) único en todo su ciclo de vida. El identificador consta de varios campos, de los cuales uno es el que se deja elegir para el desarrollador, conocido como nickname (véase la *figura 8*). El resto de campos viene determinado por la nombre de la máquina que hace de servidor de la plataforma, y el puerto donde se está ejecutando, que como se ve, serán iguales para todos los agentes. El hecho que el nombre lo decida el desarrollador en el momento de crear un agente, complica relativamente la manera en conseguir identificadores únicos, dado que estamos ante un sistema dinámico distribuido. En nuestro caso, hay agentes que se crean en el servidor y otros que se crean en el cliente (el agente AR, *AgentRegistry*). Tal nivel de concurrencia y paralelismo en el dominio del tiempo para la creación dinámica de agentes, hace que sea arriesgado optar por el uso tradicional en Java de la ejecución del método *System.currentTimeMillis()*, que devuelve un número de tipo `long` indicando el tiempo transcurrido en milisegundos desde el 1 de enero de 1970, para uso de identificadores para los agentes. Aunque la probabilidad de coincidencia pueda ser relativamente baja, es arriesgado asumir esta solución. Así pues, como alternativa se ha optado por crear un espacio de nombres (namespace), que garantice la generación de identificadores únicos en el momento que se le solicite uno. Un espacio de nombres es un ámbito en el que un identificador debe ser único. Éste servidor tendrá una variable de tipo entero, que era incrementando en una unidad tras cada solicitud, y que ejecutará en exclusión mutua la parte crítica del incremento de ésta variable, ya que estamos hablando de un servidor concurrente (atiende las peticiones concurrentemente). De esta manera, cada vez que en la plataforma se cree un nuevo agente, sea desde un nodo cliente o desde el servidor, le solicitará al servidor de nombre un identificador, que será utilizado como parte del nickname del agente. Y digo como parte porque además he decidido que en el nickname de un agente esté también una cadena que haga notar de que tipo de agente se trata, que viene bien sobretodo para hacer *debug* en la ejecución.

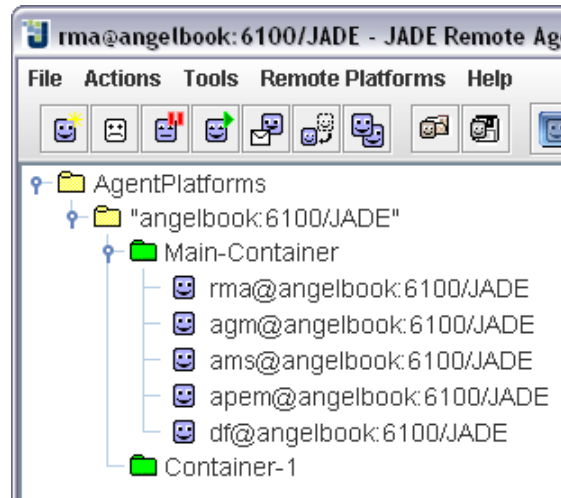


Figura 8. En la imagen tenemos una representación en forma de árbol de los agentes y contenedores pertenecientes a la ejecución de una plataforma. Como podemos observar, los AIDs de los agentes (al lado de los iconos de caras sonrientes) sólo se diferencian unos de otros por la parte que precede al símbolo '@', que ese es el nickname

3.3.4 Los nodos

Es conveniente desarrollar una estructura de datos que permita representar a los nodos, de forma que a partir de ella, se puedan conocer los recursos de los que dispone de una manera sencilla, y así poder gestionarlos adecuadamente. También resulta interesante que mantenga información relacionada con los trabajos que tenga asignados un nodo en cada momento, para poder realizar tareas de recuperación ante una posible caída del mismo. Por tanto, dicha estructura de datos, encapsulará métodos y atributos relacionados con los nodos de la plataforma, y será creada basándose en la información que proporciona el agente AR en el momento de darse de alta en el sistema (a través de un objeto de tipo `NodeInformation`).

Como había comentado, en el momento que un cliente se da de alta en el sistema, a través del servicio específico que el agente AGM ofrece para tal fin, dicho agente recibe un objeto de la clase `NodeInformation` (descrita en un apartado anterior), que es utilizado para que el agente AGM cree una instancia de la clase `Node`. Objetos de tipo `Node` serán los que definitivamente maneje el planificador como se verá en el siguiente apartado.

Un término importante a introducir en éste momento es el de *slot*. Por slot, en el contexto de éste sistema, se entiende como una unidad de medida que denota la capacidad que tiene un nodo determinado, para soportar la ejecución de trabajos concurrentemente con un buen rendimiento. De forma que si decimos que un nodo

dispone de 3 slots, indica que está capacitado para soportar la ejecución de 3 trabajos concurrentemente respondiendo de manera notable en términos de rendimiento. El hecho de reconocer que un nodo tiene capacidad de un slot, está relacionado con los recursos que tenga un computador y las características del tipo de trabajo que se vaya ejecutar. Así que éste será un criterio que se debe de adecuar a cada tipo de trabajo según los recursos que se crear necesarios para su óptima ejecución.

La estructura Node se compone de diversos atributos, de los cuales, lo más relevantes son los siguientes:

- `int key`: identificador del nodo. Útil para su gestión.
- `String containerName`: es el nombre del contenedor que tiene asociado el nodo.
- `Location location`: es la localización del contenedor del nodo.
- `Computer computer`: información relativa a los recursos del computador. Ésta estructura fue descrita con más detalle en un apartado anterior.
- `int slots`: indica la cantidad de slots que tiene un nodo. En el momento de la creación de un nodo será calculado en función de los recursos que tenga el computador y del criterio adquirido según el tipo de trabajo con el que se trate.
- `int freeSlots`: indica la cantidad de slots libres que tiene un nodo. Desde el momento en que se asigna un trabajo nuevo a un nodo, hasta que finalmente se obtienen sus resultados, se considera que dicho nodo está consumiendo un slot. Por tanto, ésta variable servirá para controlar de una manera uniforme, la cantidad de recursos libres que dispone un nodo en cada instante de tiempo.
- `boolean kill`: es un boolean que sirve para marcar un nodo como "muerto" en un caso que se describirá próximamente.
- `List<Job> currentJobs`: lista de objetos tipo Job, que indica los trabajos que tiene asignados un nodo en un momento determinado. Servirá básicamente para ofrecer la robustez necesaria en un sistema distribuido y dinámico, ante la posible caída de nodos.

Y sus métodos ha destacar son:

- `public Node(NodeInformation ni)`: el constructor, que recibe como parámetro un objeto de tipo `NodeInformation`. En él se calcula el número de slots que dispone el computador, llamando a `calculateSlots()`.
- `private int calculateSlots()`: método que calcula el número de slots que dispone el nodo a partir de la información dada el parámetro que recibe el constructor de tipo `NodeInformation`. Se ha de basar en los recursos de los que dispone el computador y de los requerimientos de los trabajos que se van a ejecutar.

- `public void addJob(Job job):` se añade un nuevo Job a la lista de Jobs del Nodo que actualmente están a su cargo. Se invoca en el momento que el planificador decide a que nodo debe destinarse cada trabajo nuevo.
- `public boolean removeJob(Job job):` éste método es complementario al anterior. Elimina un Job de la lista de jobs que el nodo tiene a su cargo. Su invocación se produce en el momento que el agente AGM recibe los resultados del trabajo una vez finalizado.
- `public List<Job> getCurrentJobs():` devuelve la lista de trabajos que el nodo tiene a cargo en el momento de su invocación.

Los demás métodos son de tipo "getters" y "setters" que sirven para leer y establecer respectivamente los atributos de la clase.

La necesidad de implementar los 3 últimos métodos relacionados con trabajos, viene dada por el hecho de poder recuperar los trabajos que se realizan en los nodos ante una posible caída de los mismos. Más adelante se verá como se realiza esta acción.

3.3.5 Diseño genérico del *scheduler* (planificador)

El gestor de los nodos es un componente que juega un papel muy importante dentro de la plataforma. Es el encargado de gestionar los nodos del sistema, almacenándolos internamente y eliminándolos cuando sean dados de baja. Además, será el que determine el criterio de planificación (scheduling), que dependerá de cada problema, siendo el responsable del correcto balanceo de la carga.

Siguiendo la idea de construir un entorno genérico adaptable a cada tipo de problema particular, era conveniente ofrecer una manera flexible de incorporar una política de planificación que se ajuste adecuadamente al tipo de trabajo que se pretenda realizar. Para ello se ha diseñado un componente abstracto, llamado `NodeManager`, relativamente complejo, cuyas responsabilidades se pueden resumir en:

- Gestionar los nodos de la plataforma: debe de ser capaz de almacenar los nodos que se van dando de alta, así como eliminarlos en el momento que sean dados de baja por el motivo que sea.
- Ofrecer una política de scheduling (planificación) adecuada, dando la posibilidad de seleccionar al mejor nodo disponible para asignarle un trabajo en un momento determinado.

Su interfaz contiene los siguientes métodos abstractos:

- `public boolean addNode(Node node):` añade un nodo a la estructura de datos interna que los almacene.
- `public Node removeNode(K id):` elimina un nodo de los almacenados (si existe) a partir de su identificador

- `public Node getNode(K id):` obtiene un nodo por su identificador (si existe)
- `public Node getBestNode():` devuelve aquel nodo que, a partir de la política de scheduling particular, considere que es el más adecuado (en el momento de su invocación) para mandarle el siguiente trabajo.
- `public void addAFreeSlot(Node n):` resta un slot libre al nodo dado por argumento. Este método debe ser invocado en el momento que se asigna un nuevo trabajo a un nodo.
- `public void removeAFreeSlot(Node n):` sirve para liberar un slot del nodo dado por argumento. Se invocará en el momento que un nodo haya finalizado un trabajo.
- `public Node getNodeFreeSlots(int freeSlots):` devuelve un nodo que contenga tantos slots libres como los indicados por el argumento de entrada, en el momento de su invocación
- `public SlotsBean getInfoSlots():` devuelve un objeto de tipo `SlotsBean` que contiene 2 atributos: el número total de slots (la suma de los slots de todos los nodos), y el número total de slots libres (la suma de los libres) que contiene la plataforma.

Después se deberá de desarrollar una clase que implemente esta interfaz. Esta implementación determinará la manera en que se almacenan los nodos (y por consiguiente, determinará los costes temporales de las operaciones relativas a la gestión de los nodos) y también la política de planificación (scheduling). Vemos como de esta manera se sigue cumpliendo con el objetivo de construir una plataforma genérica.

3.3.6 El agente gestor de eventos (AgentPlatformEventManager)

Ya he hablado sobre la gestión de los nodos, y he comentado en varias ocasiones el hecho de que un nodo pueda dejar de formar parte de la plataforma por diferentes motivos. Como sabemos, ante una arquitectura distribuida como la desarrollada en este proyecto, es muy probable que sucedan caídas de nodos en cualquier momento. Un usuario de Internet que decida ofrecerse para formar parte de la plataforma, tiene a su disposición la posibilidad de cerrar el programa cliente en cualquier momento de forma voluntaria. Pero también pueden darse casos de lo más variado, en los cuales se cierre de forma brusca el programa cliente. Por ejemplo, se puede dar una caída del suministro eléctrico del cual depende un computador dado, por diversas causas: un fallo de la subestación eléctrica, daños en una línea, una sobrecarga, una acción adversa por un ser humano, etc.

Ante estas causas el sistema debe ser robusto, en el sentido de que debe de responder y poder recuperarse adecuadamente, incluso haciéndolo de manera transparente al usuario final que utilice sus servicios. El peor de los casos es la caída de un nodo en un instante de tiempo determinado en el que tenga en ejecución uno o varios trabajos.

Como ya he comentado, la clase Node la diseñé pensando en esta circunstancia. En ella se mantiene una lista de los trabajos que en un instante de tiempo están bajo su responsabilidad.

Llegados a este punto, queda por saber cómo el sistema controla y cómo se entera de los sucesos relacionados con la caída de nodos. Una primera aproximación sobre la manera de implementarlo consistía en ir haciendo un control periódico a base de lanzar *pings*⁴ entre agentes situados en los diversos nodos. Podría ser realizado por parte del servidor hacia los nodos, o bien realizar una cadena de dependencias de responsabilidades, en la cual un nodo sería responsable de otro nodo, que a su vez tenga otro nodo bajo su responsabilidad. Ésta última aproximación es mas compleja que la primera, y requiere que el servidor mantenga una estructura interna de las relaciones de responsabilidad para poder recuperarse de forma correcta ante todos los casos.

Pero como sabemos, la plataforma está construida en base al framework de JADE, y mirando su funcionamiento uno puede observar que internamente ya hace control de monitorización sobre los contenedores que forman parte de la plataforma (que se ejecutan sobre un computador determinado). En mi caso, me di cuenta tras ver como el agente RMA nos muestra información actualizada relativa a los contenedores “vivos” que tiene la plataforma. Si a esta observación, le añadimos que éste agente se ejecuta de manera optativa en una plataforma -o sea, que es un elemento independiente de la estructura básica del sistema- llegué a la conclusión que de alguna forma, debe de existir un mecanismo de suscripción a eventos en la plataforma JADE, a través del cual se nos informe de los mismos.

Dado que en la documentación oficial de JADE no encontramos información relativa a los eventos de la plataforma, tuve que mirar el código fuente para ver como el agente RMA realiza su suscripción a eventos y como los recibe.

En primer lugar, hay que mandar un mensaje al agente AMS para suscribirnos a los eventos de la plataforma. Para ello, el contenido del mensaje debe ser un `String`, el cual está definido como constante en la clase `AMSSuscriber` que está en el paquete `jade.domain.introspection`. Dicha constante tiene el nombre de `PLATFORM_EVENTS`. Consecuentemente, a partir de ese instante el AMS notificará al agente que haya realizado tal suscripción, de los eventos que se producen a base de envíos de mensajes. Podíamos implementar un comportamiento cíclico que lea estos mensajes de tipo evento (haciendo un *matching* adecuado a los mensajes de tipo evento) y poder tratarlos. Pero aún se facilitan más las cosas si hacemos uso de un comportamiento abstracto de tipo cíclico que nos proporciona JADE, implementado en

⁴ Por ping se entiende un mecanismo cuya finalidad es la de comprobar la existencia de conexión entre dispositivos de una red por medio del envío de un paquete de datos especial por parte del origen que solicita una respuesta de tipo eco al destino. En las redes TCP/IP se hace mediante un protocolo llamado ICMP

la clase que anteriormente he mencionado, `AMSSuscriber`. Esta clase nos ofrece una método abstracto que tiene la siguiente cabecera:

```
protected abstract void installHandlers(Map handlersTable);
```

Como vemos, se nos pasa por parámetro una tabla de hash. El objetivo es incorporar un comportamiento en el agente, que herede del comportamiento `AMSSuscriber`, implementando dicho método abstracto, rellenando la tabla de hash pasada por parámetro con elementos cuya clave (key) sea identificadores de eventos (de tipo `String`, que podemos encontrarlos como constantes en la clase `IntrospectionVocabulary`) y como valor un objeto de tipo `EventHandler`, que es una interface con un único método a implementar, cuya signatura es `"handle(Event ev)";` en este método es donde deberemos incorporar el código que queramos que se ejecute después de producirse el evento. El parámetro de tipo `Event` recibido, contendrá información relacionada con el tipo de evento particular que se haya producido.

Entonces, cuando el AMS manda un mensaje de evento, lo manda a todos los agentes suscritos a los eventos. El comportamiento cíclico incorporado en el agente para tal fin, intercepta este tipo de mensajes y extrae el tipo de evento indicado en el propio mensaje. A continuación utiliza el nombre del evento como clave para indexar la tabla de hash que el agente ha rellenado, para obtener un objeto de tipo `EventHandler` (asociado a dicha clave). Si el agente ha decidido añadir a la tabla de hash un manejador de eventos para el tipo de evento que se haya producido en un momento determinado, se invocará su método `handle`. En otro caso, el comportamiento obtiene una referencia nula (`null`), con lo cual no hará nada (el agente no ha querido que le avise de tal evento). Y como el comportamiento es cíclico, éste se vuelve a bloquear en espera de recibir el siguiente mensaje de evento que el AMS le mande, cuando se produzca uno nuevo.

A continuación describo algunas de las constantes definidas por JADE que identifican a los eventos, definidas en la ontología `IntrospectionVocabulary`:

- Eventos relacionados con los contenedores y su ciclo de vida:
 - `ADDEDCONTAINER`: evento que informa sobre la creación de un nuevo contenedor en la plataforma.
 - `REMOVEDCONTAINER`: evento que informa sobre la eliminación de un contenedor en la plataforma. Éste método es el más importante para nuestro sistema, ya que es el que nos indica cuando un nodo ha caído. Esto es debido a que cada computador tiene asociado un contenedor único. JADE sólo nos informa del nombre del contenedor en cuestión, y entonces tenemos que ser capaces de reconocer que nodo es el que está asociado a dicho contenedor. Así que la estructura de almacenamiento de nodos conviene que tenga presente este aspecto.
- Eventos relacionados con los agentes y su ciclo de vida:

- BORNAGENT: informa del “nacimiento” de un nuevo agente en la plataforma.
- DEADAGENT: relacionado con el anterior. Informa de la “muerte” de un agente de la plataforma.
- SUSPENDEDAGENT: informa sobre la suspensión de un agente.
- RESUMEDAGENT: complementario del anterior. Informa que un agente previamente suspendido se acaba de “despertar”
- MOVEDAGENT: informa sobre el movimiento de un agente.

Estos sólo son un subgrupo de todos los eventos que se producen en la plataforma de JADE y a los cuales nos podemos suscribir. Resaltar que el total de eventos es muchísimo mayor al número de los que aquí he descrito. Para poder consultarlos podemos mirar la documentación del API de JADE o bien mirar el código fuente de la ontología `IntrospectionVocabulary`.

El agente de mi plataforma que se suscribe a los eventos lo he llamado `AgentPlatformEventManager(APEM)`, agente manejador de los eventos de la plataforma). Éste agente se crea al iniciarse el sistema, e interactúa con el agente AGM para la recuperación en el caso de la caída de nodos.

3.3.7 Creación dinámica de trabajos. El servidor de peticiones (ServerRequest)

Ya hemos visto que la plataforma es capaz de realizar trabajos (JOBS), los cuales son asignados a un nodo para su ejecución, haciendo responsable a un agente AW (AgentWork) para transportarlo, ejecutarlo y devolver sus resultados. También hemos visto que tenemos a otro agente APEM (AgentPlatformEventManager) que se suscribe a los eventos de la plataforma (ya explicaremos el mecanismo de recuperación de caídas de nodos en su apartado correspondiente).

Todos estos componentes, junto al NodeManager (el gestor de los nodos y scheduler de la plataforma), interactúan con el agente AGM (AgentGridManager) para coordinar todo correctamente. Ahora hace falta explicar como nos llegan desde el exterior la solicitud de trabajos a realizar. No hay que olvidar que es el servicio final que se ofrecen de manera implícita a los usuarios del sistema.

Dado que queremos hacer independientes, en cierto modo, a la parte cliente que utilice el sistema, de la parte del servidor, es necesario establece un nexo que una el “mundo del sistema multi-agente” con el exterior (clientes). Éste componente es un servidor concurrente que atenderá en todo momento la solicitud de trabajos. Éste servidor hace de puente de dichas solicitudes, para finalmente pedirle al agente AGM

que sirva dicha solicitud. Dicho servidor recibe el nombre de `ServerRequest` (servidor de peticiones).

Así pues, el servidor de peticiones escucha en un determinado puerto las solicitudes de los clientes. Dichas solicitudes acaban llegando al agente AGM, quien las transforma en trabajos (Jobs), para que puedan ser ejecutados en los nodos de la plataforma. Una vez acabados los trabajos, sus resultados son enviados por el agente AW, al agente AGM, quien almacena los resultados según cada caso particular de trabajo, para luego, como veremos en el capítulo 4, ser recuperados por el usuario final correspondiente.

Dado que el sistema puede no tener los recursos suficientes para atender una nueva solicitud, en tal caso el agente AGM notifica al servidor que no puede atender dicho trabajo. En ese caso, el servidor notificará al programa cliente (no al usuario final) que de momento el servidor está colapsado, y que lo intente en un futuro (indicándole el tiempo) nuevamente.

Para lograr que el cliente pueda solicitar al servidor la ejecución de trabajos, así como negociar el reintento de solicitudes a causa de falta de recursos, como establecer un diálogo para la obtención de los resultados finales de los trabajos, se ha diseñado un protocolo específico para tal fin. Dicho protocolo será descrito con detalle en un apartado posterior. Ahora la idea que quiero transmitir es que el servidor recibe peticiones de los clientes, y acaban transformándose en la creación de nuevos trabajos que se asignan dinámicamente a los nodos según la política de scheduling adoptada en cada caso particular.

3.3.8 El agente gestor de la plataforma (AGM)

Ya he mencionado a lo largo del texto qué función desempeña este agente dentro del sistema. Se trata del "cerebro" del sistema, el cual se apoya en los diferentes componentes para realizar su tarea de coordinación y gestión adecuada. En el momento de su creación, pone en marcha los servidores `SearchRequest` y `NameSpaces`, y pone a disposición sus servicios.

A continuación se describe los servicios que ofrece:

- alta de un nodo al sistema: este servicio ya ha sido comentado anteriormente con detalle. Básicamente ofrece la posibilidad de dar de alta a un nodo para formar parte del sistema.
- Asignación de trabajos (Jobs): este servicio es utilizado por el servidor `SearchRequest`. Se trata de un método que realiza lo siguiente:
 - Solicita al scheduler (que pertenece al `NodeManager`) el mejor nodo disponible en el momento de su ejecución (debe tener al menos un slot libre).

- Si el NodeManager le indica que no hay nodos disponibles, entonces devuelve un valor null indicando este hecho al servidor. En caso contrario continúa.
 - Crea un Job a partir de la información de solicitud realizada por el servidor. En el Job se indica el contenedor asociado al nodo seleccionado previamente (véase el apartado que describe a la clase Job para recordar su estructura).
 - Se elimina un slot libre de los que dispone el nodo seleccionado a través del NodeManager.
 - Se añade el job creado a la lista de trabajos que el nodo tiene almacenada en su estructura de datos.
 - Crea un agente AW pasándole como argumento el trabajo a realizar y el identificador del propio agente AGM (para saber a quien enviarle los resultados obtenidos tras la ejecución del trabajo en el nodo).
 - Crea un nuevo comportamiento (el propio agente AGM) para atender posteriormente la entrega de los resultados finales tras la ejecución del job.
 - Inicializa el agente AW que se a creado previamente. Ahora el agente AW, recoge en su inicio los argumentos que se la han pasado y comienza su labor: se mueve (junto con su job) al destino; una vez que llega ejecuta el job; finalmente, se comunica con el agente AGM para entregarle los resultados obtenidos.
- Reasignación de trabajos a nodos: este servicio será utilizado por el agente APEM cuando reciba un evento relacionado con la caída de un nodo.

Como vemos, el agente AGM puede tener muchas “conversaciones” a la vez (concurrentes). Para que cada comportamiento del agente AGM lleve a cabo correctamente la conversación, necesita distinguir adecuadamente entre los diversos mensajes que le llegan al propio agente. Para tal propósito se hace uso de una técnica de emparejamientos de atributos de los mensajes (utilizando la clase que nos ofrece JADE llamada `MessageTemplates`). Dentro de un comportamiento, cuando se pretende esperar la llegada de un mensaje concreto bloqueándose hasta su recepción, se debe de hacer uso de esta técnica y de esta manera poder despertarse solamente tras recibir el mensaje esperado.

En el caso de los comportamientos del agente AGM que esperan los resultados de los trabajos, precisamente se hace matching (emparejamiento) con el identificador del trabajo. Así pues, el agente AW cuando manda los resultados, rellena el atributo `ConversationId` del mensaje con el identificador del job. De esta manera, el comportamiento del agente apropiado se encuentra bloqueado en la espera de recibir los resultados empleando una plantilla `MessageTemplates`, haciendo matching mediante el atributo `ConversationId` y con el mismo identificador del job.

Podíamos pensar que para almacenar simplemente los resultados de un trabajo, da igual qué comportamiento atienda la recepción. Pero el caso es que cada

comportamiento que espera la recepción de los resultados, sabe que nodo está realizando el trabajo, de manera que cuando recibe el resultado, además de almacenarlo adecuadamente, ha de liberar el slot del nodo que se estaba consumiendo por la ejecución del trabajo. Por ese motivo es necesario asociar de manera precisa un comportamiento con el trabajo correspondiente. También será interesante realizar dicha distinción para la recuperación del sistema ante la caída de nodos, mecanismo que se explica a continuación.

3.4 Robustez del sistema. Recuperación dinámica ante caídas de Nodos. "Trabajo que empieza, trabajo que termina"

Llega el momento de explicar el funcionamiento de la recuperación ante el caso adverso en el que un nodo del sistema se cae por motivos cualesquiera.

A lo largo del este texto he ido justificando en muchas ocasiones decisiones de diseño relacionadas con este asunto. Como se ha visto, los objetos de la clase Node mantienen una lista con los trabajos que se están ejecutando en un instante dado. También sabemos que por cada trabajo en ejecución, el agente AGM tiene un comportamiento específico para la recepción de los resultados. En cada comportamiento se tiene una referencia al nodo responsable del propio trabajo, necesaria para indicar posteriormente al componente NodeManager, tras la recepción de los resultados, que incremente en una unidad la cantidad de slots libres que posee dicho nodo. Por otro lado, tenemos al agente APEM, pendiente de recibir los eventos de que se producen en la plataforma.

Entonces, hay que pensar qué pasa si en un momento dado, cae un nodo con trabajos en marcha y cómo tratar tal situación. Para ello se describe la secuencia de pasos que el sistema realiza:

- Primeramente, hay que poder enterarse del suceso: el agente APEM será el responsable. Como se trata de un evento de la plataforma de JADE, el agente APEM se informará del contenedor que ha sido eliminado.
- Después es necesario identificar al nodo implicado: necesitamos hacer un mapping (o mapeo) de lo que es un contenedor de JADE, a lo que viene a ser su nodo asociado, para saber de quién se trata (que nodo ha caído). Los nombres de los contenedores en JADE siguen el patrón "Container-X" donde 'X' es un número que se va incrementando en cada nuevo nombramiento, empezando desde 1. Dado que cada nodo tiene asociado un único contenedor, el identificador utilizado para ellos es precisamente el número 'X'. Acto seguido, cuando el agente APEM comprueba de qué contenedor se trata, extrae su parte numérica y la utiliza para pedir al NodeManagement la instancia relativa del nodo correspondiente, que tiene almacenado internamente.

- Una vez que obtiene dicha instancia, se obtiene la lista de trabajos que tenía en ejecución el nodo.
- A continuación, se itera sobre la lista. Por cada job, se crea un hilo (thread) que se encarga de solicitar al agente AGM una reasignación de nodo. El agente AGM pide al NodeManagement el mejor nodo disponible. Si no hay nodo disponible, se intenta más tarde, haciendo que el hilo se ponga a dormir para que no consuma ciclos de cpu.
- Acto seguido, se reduce en una unidad los slots libres del nuevo nodo seleccionado, y se añade a la lista de trabajos del nodo, el trabajo que estamos recuperando.
- El agente APEM envía un mensaje al agente AGM con el identificador de conversación del trabajo a recuperar, y pasándole como contenido del mismo el nuevo nodo seleccionado. Aquí hay que observar que el agente AGM dispone de un comportamiento que espera la recepción de los resultados de un trabajo, el cual contiene una referencia al nodo que lo realiza, para que en el momento de recibir los resultados le libere un slot. Pues bien, el objetivo del envío del mensaje del agente APEM hacia el AGM, que será tratado por este comportamiento, no es más que la actualización en dicho comportamiento de la referencia del nodo caído, hacia el nuevo nodo que se encargará de realizar nuevamente el trabajo. Así, cuando reciba en última instancia los resultados, solicite al NodeManagement que libere un slot del nodo que el sistema ha elegido en la reasignación del trabajo.
- Ya sólo queda crear, desde el hilo que se ha creado en el agente APEM, un nuevo agente AW, que se encargue de ejecutar el trabajo nuevamente en el nodo seleccionado. Pero antes de iniciar la ejecución del agente AW, por motivos de sincronización, debe de esperar la confirmación del cambio de nodo por parte del comportamiento del agente AGM que espera los resultados (mediante el envío de un mensaje de confirmación). El caso es que si iniciamos la ejecución al agente AW sin hacer el sincronismo adecuado, podría ocurrir que éste realizase el trabajo y enviara los resultados antes de que realmente se haya producido un cambio de referencia hacia el nuevo nodo en el comportamiento, dejando el sistema inconsistente. Notemos como se necesita hacer matching con sumo cuidado para que los mensajes sean tratados por quien deseamos que realmente los reciba.
- Por último, una vez realizado el sincronismo, el agente AW inicia su ejecución. Cuando envíe los resultados, tras finalizar el trabajo, el comportamiento del agente AGM los almacenará y liberará correctamente el slot del nodo que finalmente ha realizado el trabajo.

Cabe decir que es posible que el nuevo nodo al que se le ha asignado el trabajo pueda caer nuevamente. Pero esto no es un inconveniente ya que el sistema se volvería a comportar igual, siguiendo los mismo pasos que se acaban de describir.

De esta manera, siempre y cuando el servidor no sea quien caiga, un trabajo que sea solicitado al sistema siempre acabará siendo ejecutado y de manera transparente al usuario final.

3.5 Optimización al enviar ficheros por la red y facilidad para su manejo. Creación de la clase "NetFile"

En el diseño de la clase Job, se ha visto como éste puede manejar ficheros. Puede ser que un trabajo necesite ficheros para llevar a cabo su ejecución. También puede ocurrir que los resultados producidos tras la ejecución del trabajo, una parte o su totalidad sean almacenados en ficheros.

Si se trabaja en local no hay problema alguno, y utilizando el API de java para la entrada y salida de ficheros es suficiente. Pero como sabemos, los trabajos que se crean en el sistema, viajan junto los agentes desde el servidor hasta los nodos, y los resultados son enviados desde el nodo hacia el servidor. Como vemos, existe la posibilidad de que hayan ficheros que necesiten ser enviados por la red. Utilizando el API de java se puede utilizar las clases del paquete de red (`java.net`) y de entrada/salida (`java.io`) para que mediante sockets y flujos, los agentes puedan enviar el contenido de los ficheros. Pero esta aproximación se aleja un poco de la flexibilidad que estábamos destacando del sistema relativa a los trabajos y la movilidad de los agentes. Y aquí es donde surgió la idea de crear una nueva clase que permitiera superar esta barrera.

La clase creada recibe el nombre de `NetFile`, destacando el hecho de que es útil para enviar ficheros por la red. La idea se basa en leer el contenido de un fichero, comprimiéndolo para ser almacenado en un array de bytes. La clase implementa la interface serializable, y como el contenido del fichero está comprimido en un array de bytes, entonces está en condiciones de ser transportado por la red por un agente sin problemas (recordemos que todos las instancias que se lleva un agente, si queremos mantener su estado, deben de ser serializables).

Lógicamente, la clase también debe de proporcionar métodos para recuperar el contenido del fichero original. En concreto, proporciona un método para leer directamente el contenido del fichero en un `String`. Y otro método al cual se le pasa por parámetro una ruta del disco duro para que sea almacenado en él.

El constructor tiene la siguiente signatura:

```
public NetFile(File f)
```

Como vemos, recibe por parámetro un objeto de tipo `File`, que no es más que una representación abstracta de una ruta de un fichero o directorio. Lógicamente, en este caso interesa que sea una ruta a un fichero.

A continuación se describen los atributos (privados) de la clase:

- `transient File fileIn`: referencia al fichero pasado en el constructor. Se ha utilizado el modificador `transient` para indicar al mecanismo de serialización que éste atributo no debe de ser serializado (en el momento que se realice la serialización). El motivo es que no tiene sentido una referencia a un fichero local en otro computador destino.
- `String fileName`: es el nombre del fichero.
- `byte[] compressedData`: array de bytes que contiene los datos del fichero comprimidos.

Y sus métodos son los siguientes:

- `void compressFile()`: comprime el contenido del fichero dejando el resultado en el array de bytes `compressedData`.
- `String contentToString()`: descomprime el array de bytes `compressedData`, obteniendo el contenido original del fichero, que es devuelto dentro de un objeto de tipo `String`.
- `boolean contentToFile(String path, boolean defaultName)`: descomprime el array de bytes `compressedData`, obteniendo el contenido original del fichero, almacenándolo en un nuevo fichero del disco duro en la ruta indicada por el argumento de entrada `path`.

Así pues, los pasos para que un trabajo (job) se lleve un fichero del cliente al servidor serían los siguientes: supongamos que el fichero se llama "datos.txt" y que se encuentra en el path base del contexto de ejecución local. Entonces crearía una instancia de la clase `NetFile` del siguiente modo:

```
...
File f = new File("datos.txt");
NetFile nf = new NetFile(f);
...
```

O bien, de forma más directa:

```
...
NetFile nf = new NetFile(new File("datos.txt"));
...
```

Cuando el agente se mueva a otra localización y comience la ejecución del trabajo en el nodo, el contenido del mismo se puede recuperar de la siguiente manera:

```
...
String content = nf.contentToString();
...
```

O bien, almacenándolo en un directorio del computador (por ejemplo, en un directorio llamado `\"./inputFiles/\"`):

```
...
nf.contentToFile(\"./inputFiles/");
...
```

Y posteriormente ser leído como cualquier otro fichero:

```
...
File f = new File(\"./inputFiles/datos.txt\");
FileInputStream fis = new FileInputStream(f);
...
```

Evidentemente, esta clase también se puede utilizar en local, pero cobra más sentido cuando se necesite serializar un fichero para ser enviado por la red ofreciendo sencillez y eficiencia (ya que el contenido está comprimido).

Para la compresión del contenido se hace uso de las clases `java.util.zip.Deflater` y `java.util.zip.Inflater`.

3.6 Resumen de las características que ofrece el sistema al desarrollador

Hasta aquí ha llegado la explicación de la parte del sistema dinámico. Como vemos, el sistema construido pone a disposición del desarrollador un entorno capaz de ejecutar, de forma paralela, aquellos tipos de trabajos que puedan ser divididos en tareas independientes. Además, su diseño extensible permite mucha flexibilidad a la hora de crear nuevos tipos de tareas, implementando nuevas clases que extiendan de `Job`, permitiendo mediante la característica del código móvil, que los clientes no necesiten tener instalado un software nuevo o actualizado cada vez, ya que cambiando de forma sencilla el código relativo a las tareas en el servidor, disponemos instantáneamente de la última versión del código en todos los nodos. Además, también se permite adaptar la manera de gestionar los nodos y la política de planificación (`scheduling`) para que el balanceo de la carga sea lo más óptimo posible en cada caso (definiendo cuantos recursos requiere un slot, cambiando la manera de almacenar los nodos en base a una política de `scheduling` más acorde con cada tipo de problema, etc.). Además, el sistema ofrece de manera transparente al desarrollador, un nivel de robustez que garantiza la recuperación de los trabajos que se están realizando ante casos adversos como la caída de nodos, que suelen ser habituales en sistemas de tipo grid.

4. Aplicación del sistema a un caso real: construcción semiautomática de Ontologías a partir de la Web

Una vez explicado como funciona el sistema, llega el momento de ver cómo se ha podido integrar una aplicación concreta de búsqueda de información en la Web. Este capítulo está dividido en dos grandes apartados:

- En el apartado 4.1 se describe la manera en cómo se ha podido integrar esta aplicación particular al sistema, identificando qué componentes se han tenido que desarrollar y destacando las ventajas que el diseño genérico ofrece para que se pueda adaptar óptimamente.
- Por otro lado, en el apartado 4.2 se explican las mejoras aportadas a la aplicación para que sea más funcional y flexible de cara al usuario final, creando una interfaz gráfica para la interacción con la aplicación y el sistema, que sirve para gestionar el proceso de búsqueda. Se verá como el uso de las tecnologías Web cobran protagonismo para lograr estos objetivos.

4.1 Adaptación de los componentes genéricos a la aplicación

Se trata de integrar en el sistema una aplicación que intenta extraer conocimiento a partir de la información disponible en la Web. Esta información se encuentra completamente desestructurada y hace que para un usuario que intenta buscar información sobre algún dominio en ella, sea una tarea complicada y con mucha pérdida de tiempo navegando por páginas que en su mayoría no suelen estar relacionadas con lo que busca.

El corazón de esta aplicación es un componente software creado por David Sánchez Ruenes (profesor de la Universidad Rovira i Virgili y codirector de este proyecto), llamado **TaxoBuilder**. Se trata de un componente parametrizable que, a grandes rasgos, permite construir una estructura de datos de relaciones taxonómicas (conceptos de un dominio clasificados jerárquicamente en forma de "árbol", desde los más generales a los más específicos) y no taxonómicas (conceptos de un dominio que carecen de una relación jerárquica, es decir, relaciones entre conceptos de "árboles" distintos) a partir de un concepto y unos parámetros introducidos, basándose en la información que hay en la Web, haciendo un análisis inteligente de su contenido. Estos conceptos tienen asociados enlaces a páginas Web para que puedan ser visitadas, con la garantía de estar realmente relacionadas con lo que se pretende buscar. Como se ha dicho, este componente es parametrizable, en el sentido que nos permite indicarle ciertas opciones para que se adapte mejor a las necesidades de búsqueda por parte de quien lo utiliza.

Esta herramienta puede tener diversos usos, y en particular es realmente útil para que un usuario pueda construir un Ontología que recoja conocimiento sobre un dominio en concreto, de forma que permita navegar por la Web de una manera más inteligente y precisa sobre aquello que realmente quiere buscar. Y se afirma que el proceso de navegación es más inteligente y preciso, debido a que permite que un usuario final pueda realizar búsquedas por la Web de una manera más efectiva que si lo hiciese utilizando un buscador popular como puede ser Google, MSNSearch, Yahoo, entre otros, lo cuales son más generalistas y nos devuelven muchísimos resultados que en la mayoría de casos hacen que el éxito en la búsqueda sea muy costoso o en ocasiones un fracaso. Y es ésta una de las muchas aplicaciones que encajan en el sistema que se ha construido en este proyecto.

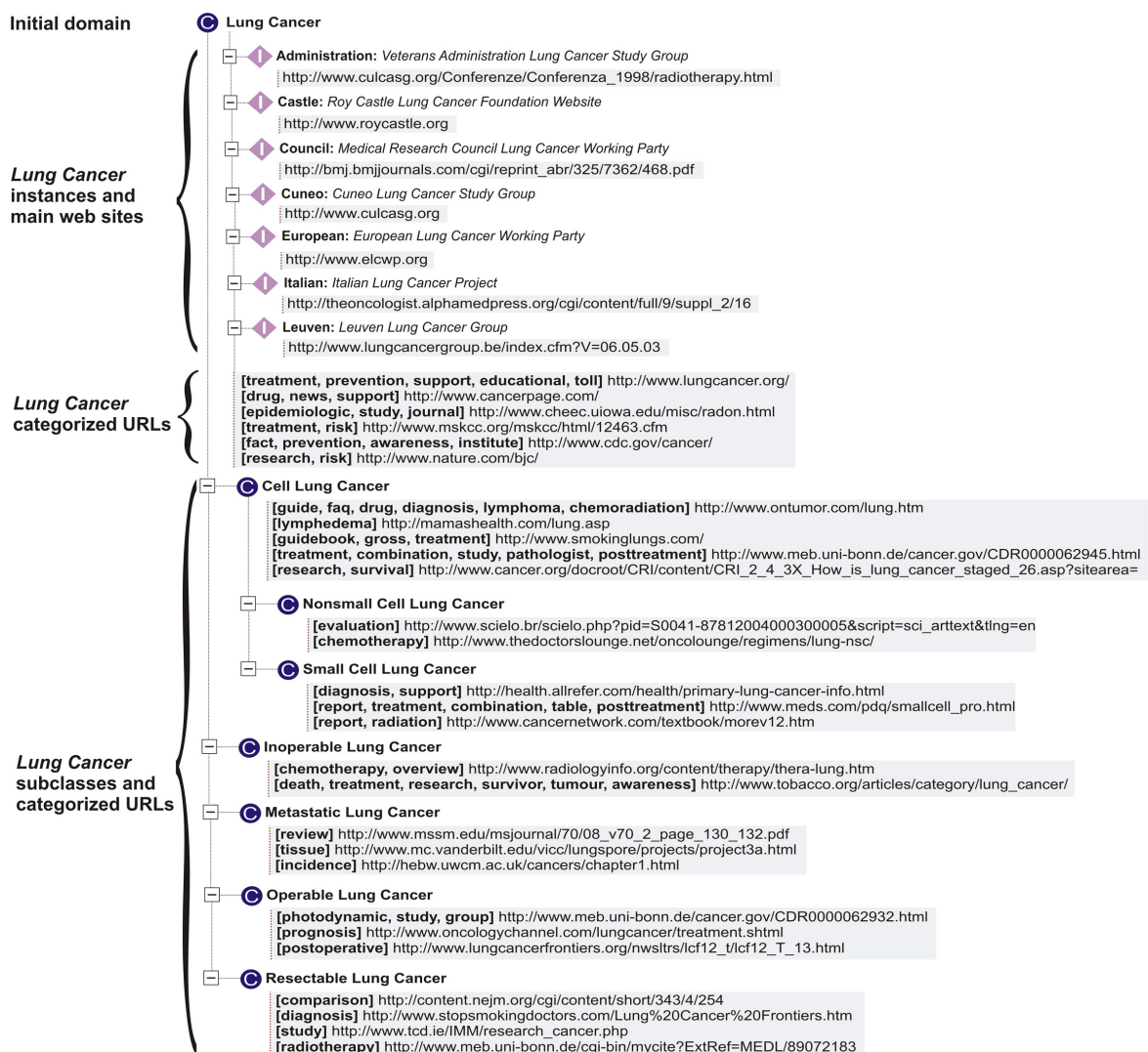


Figura 9. Ejemplo de resultados que devuelve la herramienta TaxoBuilder

Cabe destacar que cada búsqueda que realiza el componente TaxoBuilder es independiente la una de la otra en términos computacionales. Es decir, si un usuario decide buscar 4 conceptos a la vez, pues se crean 4 instancias independientes del constructor de taxonomías. Una vez que acaba una búsqueda, la aplicación muestra

los resultados en un nuevo espacio separado pero dentro de un marco de trabajo integrado.

El tiempo que tarda una búsqueda en realizarse con este componente software es indeterminado y depende de varios factores:

- El primer factor hace referencia a los parámetros que el usuario de la aplicación ajuste (nivel de precisión, nivel de aprendizaje, etc).
- El segundo factor depende del funcionamiento de la red de Internet, en base al nivel de carga que pueda tener.

Normalmente éste tiempo puede ser muy elevado, ya que hemos de pensar que el constructor de taxonomías analiza detalladamente el contenido de páginas Web completas, en el cual encontramos información que estorba (aquella relacionada con etiquetas del lenguaje html, que sirven para maquetar el contenido de un documento), y además debe de realizar análisis sintácticos y semánticos para determinar la validez de una determinada información (basándose en técnicas de estadística).

En resumen, éstas son las características que presenta este problema particular:

- Podemos tener muchas instancias de un mismo componente (TaxoBuilder).
- Las instancias son independientes computacionalmente entre ellas y por tanto estamos ante un problema divisible (véase la *figura 10*).
- Cada instancia consume mucho tiempo para obtener resultados.
- Además necesitan unos 120MB de memoria RAM cada una (con la versión actual del TaxoBuilder, la cual se encuentra en actualmente en continua evolución).

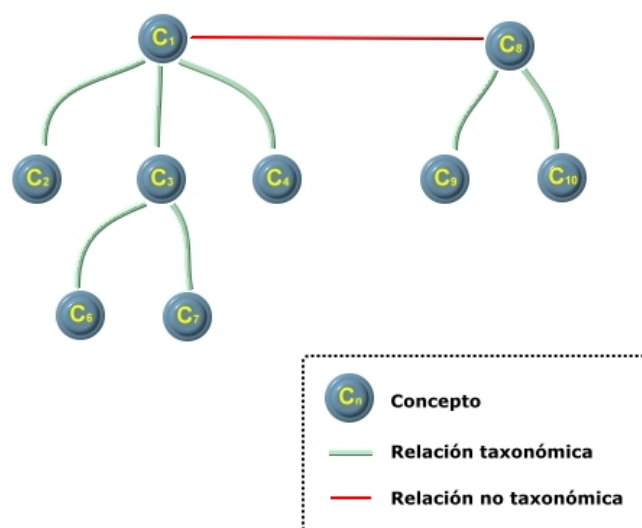


Figura 10. La Ontología se va elaborando con TaxoBuilder, mediante la unión de resultados como los de la imagen, que se van obteniendo de manera independiente.

Las características de este tipo de problema encaja perfectamente en un sistema como el construido en este proyecto. No precisamos de tiempos críticos, tenemos un problema costoso que consume muchos recursos (memoria RAM y ancho de banda) y que a la vez es divisible en partes independientes computacionalmente. Un entorno de ejecución como el que ofrece el sistema construido es ideal para que esta aplicación funcione con un mejor rendimiento que si se ejecutara en un único computador. Además, el sistema permitirá que diversos usuarios puedan acceder a la aplicación concurrentemente. Lógicamente, cuantos más recursos disponga el sistema (depende de los nodos), más usuarios podrá aguantar de manera óptima y mejor rendimiento ofrecerá a cada uno en particular.

Otro proyecto anterior en GruSMA [LM&FR05] construyó un tipo de aplicación similar mediante un sistema multi-agente, pero dado que sus objetivos que perseguían eran otros no relacionados con la capacidad de cómputo, ésta se ejecutaba sobre una única máquina, limitándose el uso de la aplicación a un usuario local a la máquina y con pocas instancias del constructor de taxonomías. Por tanto, la arquitectura de esta aplicación es totalmente diferente ya que tiene otro enfoque. Pero lo que si se ha podido aprovechar para este caso es la tarea de renderización de un fichero OWL⁵ [OWL06] en un componente gráfico swing para poder visualizar en forma de árbol las relaciones taxonómicas.

En el siguiente apartado se describe como los trabajos se han adaptado al entorno del sistema para que puedan ser ejecutados. Posteriormente, se explica como el componente NodeManager ha sido desarrollado con el objetivo de conseguir adaptar una política de planificación eficiente para este problema. Y es aquí donde se pondrá de manifiesto la capacidad del sistema creado, para conseguir adaptarse de manera óptima y flexible a un tipo de problema particular.

4.1.1 Los trabajos de la aplicación: JobSearch. Uso del constructor de taxonomías (TaxoBuilder)

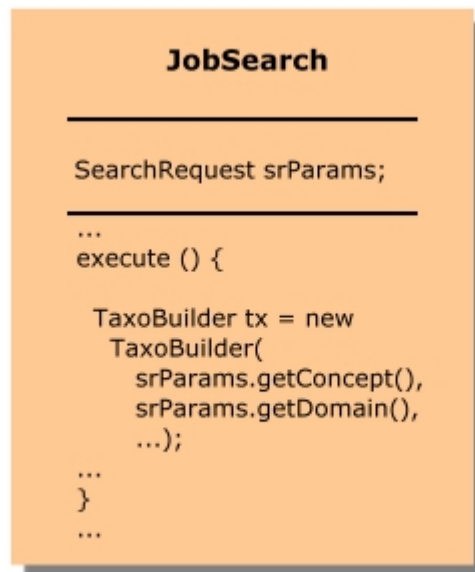
Es el momento de explicar que tipo de trabajos realizará el sistema para esta aplicación. Tenemos la clase abstracta Job, que nos proporciona una manera flexible de crear nuevas factorías (clases) de trabajos heredando de ella. En este caso, como se trata de realizar una búsqueda por la Web, la nueva clase se ha llamado JobSearch (trabajo de búsqueda).

Este trabajo se basa en utilizar el componente TaxoBuilder, pasándole unos parámetros a criterio del usuario de la aplicación, para que realice una búsqueda de la manera que ya hemos comentado. Para hacerlo más compacto, se ha creado un clase nueva, llamada SearchRequest, para que mediante una instancia se pueda almacenar dichos parámetros (todos) en una sola localización. Esta instancia llegará a través de

⁵ **OWL** es el acrónimo del inglés Web Ontology Language, un lenguaje de marcado para publicar y compartir datos usando ontologías en la WWW. OWL tiene como objetivo facilitar un modelo de marcado construido sobre RDF y codificado en XML.

una petición de búsqueda al `ServerRequest`, que a su vez lo notifica al agente AGM. Si éste considera oportuno realizar el trabajo en ese instante de tiempo, se crea un objeto de tipo `JobSearch`. `JobSearch` incorpora un nuevo constructor al cual se le pasa por parámetro la instancia del objeto `SearchRequest`. También implementa, de forma obligada por la herencia de la clase `Job`, el método `execute()`. En dicho método se crea una instancia del constructor de taxonomías (ver la *figura 11*) pasándole los parámetros que solicita, sirviéndoselos desde el objeto `SearchRequest` que incorpora el propio trabajo. El trabajo creado en el servidor (ver *figura 12*, instante (t1)) será entregado a un agente AW (el cual no se ha modificado para nada, ya que opera con la clase `Job`), quien se encargará de transportarlo (ver *figura 12*, instante (t3)) al nodo que el planificador haya creído oportuno. Una vez el agente aterrice en la nueva localización, invocará al método `execute` de la clase `Job` que, por ligadura dinámica, acabará ejecutando realmente la versión del método `execute` de la clase `JobSearch`. Cuando acabe el constructor de taxonomías de hacer su trabajo, el método `execute` prosigue, y lo que hace a continuación es almacenar los resultados obtenidos (dos tablas de hash y dos ficheros de texto) en los atributos que la clase `Job` proporciona para tal fin (ver *figura 12*, instante (t4)):

- Las dos tablas de hash son añadidas a la lista de objetos serializables de resultados, llamada `outObjects`.
- Los dos ficheros de texto (en formato OWL) son añadidos a la lista de objetos de tipo `NetFiles` de resultados, llamada `outNetFiles`.



```

JobSearch

SearchRequest srParams;

...
execute () {

    TaxoBuilder tx = new
    TaxoBuilder(
        srParams.getConcept(),
        srParams.getDomain(),
        ...);

    ...
}
...

```

Figura 11. Detalles de la implementación de la clase `JobSearch`

Evidentemente, los objetos de tipo `NetFiles` son también serializables (ver capítulo 3.5) y por tanto podrían ir dentro de la lista de resultados `outObjects`. Pero se ha preferido distinguir para notar que efectivamente se trata de ficheros. Finalmente, estos resultados son enviados al agente AGM (ver *figura 12*, instante (t5)).

Se puede observar (ver *figura 11*) que el TaxoBuilder es utilizado en el método `execute` de la clase `JobSearch`. Esta clase está en el servidor, y de hecho sus instancias se crean en él para ser enviadas a ejecutar a un nodo (código móvil). El problema está en que el

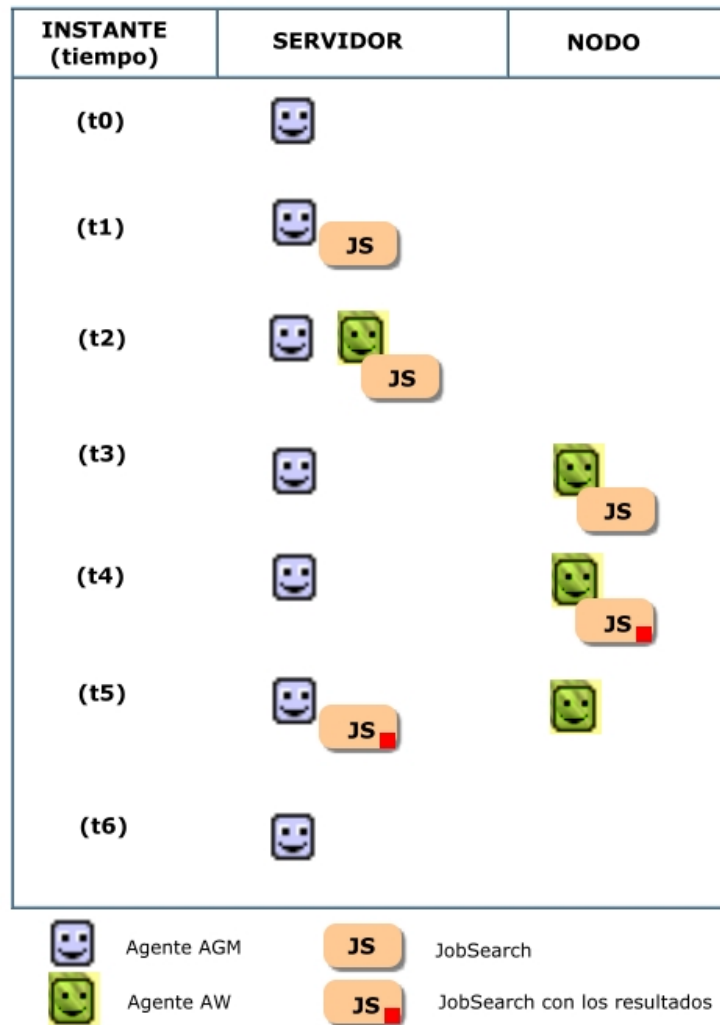


Figura 12. Ciclo de vida de un trabajo

TaxoBuilder utiliza un gran número de librerías y su espacio total en disco es de unos 200 Mbytes, lo cual hace que sea inviable, en este caso particular, hacer que el código de este componente sea móvil. Por tanto, en este caso nos interesa que el software del TaxoBuilder con todas sus librerías estén instalados en los nodos clientes. Notemos que el código de la clase `JobSearch` sí que es móvil, lo único que haremos es que el componente TaxoBuilder, por sus 200 Mbytes que ocupa, esté instalado en los nodos que utilizaremos para ejecutar esta aplicación. Así pues, cuando un agente AW se lleve un trabajo de tipo `JobSearch` a un nodo, se llevará consigo todo el código de la clase a excepción de lo que ya se encuentre instalado en el cliente, que en este caso será el TaxoBuilder. Además, como el trabajo no va a ser ejecutado en el servidor y el TaxoBuilder no va a ser enviado por la red, es una pérdida de espacio (200 Mbytes)

tener a éste en el propio servidor únicamente para lograr que el programa se pueda compilar. Para solventar este obstáculo, se ha creado una interfaz llamada `AbstractTaxoBuilder`, que tiene como objetivo desacoplar al componente `TaxoBuilder` de la siguiente manera: la interfaz contiene el mismo método que pone a trabajar al `TaxoBuilder` y además dos métodos más para adaptar sus dos constructores (después de la adaptación solamente tendrá un único constructor vacío, necesario para crear una instancia de una clase cargada en tiempo de ejecución, mediante el empleo del método `Class.forName()`). Entonces, en el método `execute` se hace referencia únicamente a la interfaz, para no necesitar el componente `TaxoBuilder` en el servidor con el único objetivo de poder compilar. Y luego, para instanciar realmente el `TaxoBuilder` en el nodo donde se ejecuta realmente, lo que se hace es cargar el código en tiempo de ejecución para tenerlo disponible, haciendo uso del método `Class.forName()`. Mediante esta técnica, se necesita tener un constructor vacío en la clase. Evidentemente, el `taxoBuilder` ha requerido ser modificado mínimamente para soportar dicha interfaz.

Nótese una vez más la flexibilidad que ofrece el sistema para el desarrollador. Si bien hemos destacado siempre la potencia que nos ofrece la movilidad del código, vemos como no se nos priva de poder adaptar un problema particular que precisamente requería tener una parte estática de código. Y repito que, tanto el código del agente AW como el de la clase `JobSearch` (a excepción del `TaxoBuilder`), sigue siendo código móvil que viajará de forma dinámica a los nodos del sistema. Así pues, todo el código relativo al agente AW y su trabajo asignado, que realiza la instancia del `TaxoBuilder`, la petición de búsqueda, y la entrega de los resultados al agente AGM, no se encuentra instalado en los nodos clientes porque es móvil.

Para que se vea más claro, a continuación se muestran las partes clave del código que ayudan a aclarar las ideas.

- La interfaz **AbstractTaxoBuilder**:

```
public interface AbstractTaxoBuilder extends Serializable {

    public abstract void abstractConstructorTaxoBuilder(params);

    /** donde 'params' son todos los parámetros que tenía el constructor del
    taxoBuilder antes de ser modificado */

    public abstract void abstractConstructorTaxoBuilder (params2);

    /** donde 'params2' son todos los parámetros que tenía el 2ndo
    constructor del taxoBuilder antes de ser modificado */

    public abstract Object[] search();

    /** es la misma signatura del método que tiene el taxoBuilder para
    comenzar la búsqueda. Está para que pueda ser invocada la acción de
    búsqueda a través de la interfaz */

}
```

- Ahora se muestra como queda el **taxoBuilder**:

```
public class taxoBuilder implements AbstractTaxoBuilder {
```

```
public TaxoBuilder() {
}

/** ahora sólo hay un único constructor pero sin parámetros */

public abstract void abstractConstructorTaxoBuilder(params) {
    ...
}

/** donde 'params' son todos los parámetros que tenía el constructor del
taxoBuilder antes de ser modificado. En este método lo que se hace es lo
mismo que se hacía en su constructor */

public abstract void abstractConstructorTaxoBuilder (params2) {
    ...
}

/** donde 'params2' son todos los parámetros que tenía el 2do
constructor del taxoBuilder antes de ser modificado. Aquí se hace lo
mismo que se hacía en el 2do constructor */

public abstract Object[] search() {
    ...
}

/** este método permanece igual que antes. Es donde se realiza el
proceso de búsqueda */
```

- De esta manera, el método **execute** de la clase JobSearch queda de esta forma:

```
public boolean execute() {
    try {
        Class TaxoBuilder = Class.forName("taxo.taxoBuilder");
        AbstractTaxoBuilder atb =
            (AbstractTaxoBuilder) TaxoBuilder.newInstance();

        atb.abstractConstructorTaxoBuilder(
            searchRequest.getKeyword(),
            searchRequest.getDom(),
            searchRequest.getNumWords(),
            searchRequest.getNumLevels(),
            searchRequest.getSelection(),
            searchRequest.getLearning(),
            searchRequest.isBuscarInst(),
            searchRequest.isBuscarNoTaxo(),
            searchRequest.getMainEngine(),
            searchRequest.getSecondaryEngine(),
            searchRequest.getWebDom()
        );

        Object[] res = atb.search();
        ...
    }
}
```

Como se puede observar a partir del código, en el método `execute` de la clase JobSearch no se hace referencia alguna al componente `taxoBuilder`. Solamente en el momento de ejecutar el método se carga el código en tiempo de ejecución, pero como el método solamente será ejecutado en nodos, no hace falta tener el dicho componente en el servidor para hacer la compilación del código fuente a bytecodes.

4.1.2 Diseño del scheduler para esta aplicación

En el capítulo 3 de este documento, se analizó un componente del sistema distribuido llamado `NodeManagement`. Éste se encarga de gestionar los nodos de la plataforma y de adoptar una política de scheduling para seleccionar el mejor nodo de entre los que dispone, en el momento que el agente AGM le solicite uno.

De este componente solamente presentamos una interfaz abstracta, con la que el sistema trabaja. La idea era permitir que podamos desarrollar diversas implementaciones que soporten esta interfaz, de forma que cada una se adapte de la mejor manera posible a las necesidades de cada tipo de aplicación diferente. Y esto es lo que se ha hecho para esta aplicación. Como sabemos, cada trabajo realiza una instancia del `taxoBuilder`, componente que durante su ejecución ocupa unos 120 Mbytes de memoria RAM, y que su consumo de ciclos de cpu no resulta demasiado intensivo en general, debido a que tiene muchas operaciones de entrada y salida hacia la red Internet (tiene “picos” de consumo de cpu). Por lo tanto, lo que más afecta al rendimiento de su ejecución es el hecho que haya o no haya espacio de memoria RAM disponible (120 Mbytes por instancia del `taxoBuilder`). Si resulta que ejecutamos muchas instancias del mismo, agotaremos la memoria física disponible, y por tanto el mecanismo de swapping del sistema operativo (actualmente la mayoría lo incorporan) intervendrá excesivamente, realizando muchos intercambios de páginas de memoria hacia el disco duro y viceversa, de forma que el tiempo de respuesta de la ejecución se verá empeorado.

Por tanto, a partir de estas observaciones, y considerando que todos los ordenadores de nuestra universidad (URV) que se iban a utilizar para ejecutar la aplicación tienen un ancho de banda grande, y una cpu con una capacidad “aceptable” (todos con una frecuencia superior o igual a 1,6 GHz) se tomaron las siguientes decisiones para esta aplicación:

- a) Un slot necesita de los siguientes recursos:
 - . 128 Mbytes de memoria RAM. En este caso es necesario que el nodo disponga de esta cantidad de memoria RAM para una tarea, de forma que pueda ejecutarse de manera óptima.
 - . Consumo de cpu por “picos”. Con la potencia de los ordenadores que disponemos no ha resultado necesario poner restricciones.
 - . Ancho de banda relativamente aceptable. El ancho de banda que disponemos en la URV es más que suficiente para abastecer a todos los nodos, de manera que tampoco se ponen restricciones.

- b) La política de planificación (scheduling), que se encarga de seleccionar al mejor nodo en el momento de asignar un nuevo trabajo para balancear correctamente la carga, se basa en el siguiente criterio:
- . Se considera que un nodo que esté ocioso, es decir, que no tiene asignado ninguna tarea para ejecutar, es mejor candidato que otro que esté realizando alguna tarea.
 - . Por otro lado, entre los nodos que estén realizando alguna tarea, es mejor aquel que más slots libres tenga.

Se recuerda que las operaciones que requiere el componente NodeManager relativa a la gestión del almacenamiento de los nodos y la política de planificación son las siguientes:

```
public boolean addNode(Node node);  
  
public Node removeNode(K id);  
  
public Node getNode(K id);  
  
public Node getBestNode();  
  
public void addAFreeSlot(Node n);  
  
public void removeAFreeSlot(Node n);
```

Como se ve, se trata de operaciones para poder añadir un nuevo nodo a la estructura de datos, eliminar un nodo por su identificador, obtener el mejor nodo en base al criterio que se adopte en cada caso, y por último, consumir un slot de un nodo y como operación inversa, poder liberar un slot de un nodo.

Por tanto, en base a los criterios anteriores adoptados para esta aplicación, había que decidir la manera de implementar una versión del NodeManager que fuese lo más eficiente posible. Esto es, mirar que tipo de estructura de datos se puede aplicar para que las operaciones relacionadas con la gestión de los nodos, tengan un coste temporal que sea lo menor posible.

En un principio se pensó la alternativa de tener una única lista, ordenada por el número de slots libres que tenían los nodos. Pero esta aproximación no encaja bien para seleccionar los nodos ociosos como más prioritarios, debido a que la operación de seleccionar al mejor tendría un coste lineal ($O(n)$). Además, como la lista es ordenada, la operación de inserción y de eliminación de un nodo tendrían también coste lineal. También las operaciones de añadir/eliminar un slot libre en el caso de una implementación óptima de la cola ordenada tendría un coste lineal.

Después, se pensó en separar los nodos en dos colas diferentes: una de ociosos y otra de ocupados. De esta forma, para obtener al mejor nodo, bastaría con seleccionar al primero de la lista de ociosos, que si está vacía, entonces se seleccionaría a un nodo de la lista de ocupados. El problema de esta solución era que la lista de ocupados debe de estar ordenada, y los costes de la mayoría de las operaciones continuaba siendo

lineal. El hecho de que las operaciones tengan un coste lineal ($O(n)$), provoca que a medida que crezca el número de nodos que maneja el sistema, el rendimiento se irá deteriorado. En cambio, si se consigue que estas operaciones tengan un coste temporal constante ($O(1)$), hará que el rendimiento de las mismas esté optimizado al máximo y sea independiente del número de nodos.

En este momento, se observó un nuevo detalle: resulta que a partir de un cierto umbral de slots libres, el rendimiento de la ejecución de un trabajo de este tipo no se veía mejorado. Haciendo pruebas comprobamos que una máquina con 5 slots libres, tenía un rendimiento muy similar que otra máquina con las mismas características pero con la única diferencia de que disponía de 6 slots libres.

Finalmente, teniendo en cuenta todo lo anterior, se optó por diseñar una estructura de datos híbrida, en la que intervienen diferentes tipos de colas para almacenar los nodos. Se trata de tener una cola de ociosos, otra de los que tienen 1 sólo slot libre, otra de 2, de 3, de 4, de 5, y finalmente, otra de más de 5. Esta última cola no necesita estar ordenada, ya que como hemos dicho, a partir de este umbral el rendimiento no mejoraba. Así pues, en base a los slots libres y ocupados que dispongan los nodos, éstos se moverán de una cola a otra.

Ya tenemos la operación de obtener al mejor con un coste temporal constante $O(1)$, ya que es cuestión de mirar el primer nodo de la cola de ociosos, que si está vacía, se mira en la de "5 o más libres", si también esta vacía, se mira en la de 4, y así sucesivamente hasta llega a la cola de los que tienen un slot. En todo caso, se trata de mirar el cabeza de la lista. También la operación de añadir un nodo es constante $O(1)$ ya que se trata de mirar en qué cola va, en función de los slots libres y totales. Ahora quedaban con coste lineal $O(n)$ la operación de eliminar un nodo (se tiene que recorrer todas las colas ya que no se sabe dónde estará). También tienen un coste lineal $O(n)$ las operaciones de consumir un slot y la de liberar un slot a un nodo, debido a que provocan que se mueva de una cola a otra. El problema no está en la inserción en la nueva cola, sino en el momento de buscar el nodo en su cola actual para extraerlo (en el peor de los casos todos los nodos estarán en la misma cola, por tanto el coste es $O(n)$).

Pero como las operaciones de consumir un slot y liberar un slot son muy habituales, seguramente las que más, era interesante seguir pensando en mejorar los costes de las operaciones para intentar que sean también constantes $O(1)$. Esto se ha conseguido de la siguiente manera:

- Sin cambiar la estructura básica, es decir, sin modificar el hecho de tener las diferentes colas clasificadas, se trata de modificar la forma en que se almacenan los nodos dentro de cada cola. Es decir, lo ideal es poder acceder directamente a un nodo por su identificador, para poder sacarlo de la cola (que se produce tanto en el momento de eliminar un nodo, como para consumir o liberar un slot). Por tanto, la solución pasa por convertir las estructuras de tipo cola, por una estructura híbrida que sea una cola y a la vez una tabla de hash. Dicha estructura se encuentra implementada en el API estándar de Java, y su clase se llama `LinkedHashMap`. Esta estructura sigue siendo una cola, ya que garantiza el orden de inserción. Por tanto, respecto a la solución anterior, aporta la mejora de poder acceder por una clave a un valor, que en el caso de

los nodos sería por su campo de identificación. Dicha estructura se puede utilizar como una tabla de hash normal, pero tiene el añadido de que internamente mantiene una lista doblemente encadenada, manteniendo el orden de inserción, y por tanto, nos ofrece operaciones para poder ser tratada como una cola de tipo FIFO (*First Input First Output*).

Ahora por fin, ya tenemos absolutamente todas las operaciones con coste constante $O(1)$. Está claro que para ello las colas deben de ser doblemente enlazadas, pero mirando el código fuente de la implementación de la clase `LinkedHashMap` del API de Java pude comprobar que efectivamente hacían uso de una lista doble enlazada.

Por tanto, gracias a este diseño relativamente complejo, el rendimiento en base a la sobrecarga que produce el trabajo que realiza el componente `NodeManager` en el sistema, está optimizado al máximo para esta aplicación.

Aquí se destaca una vez más, que esto es posible gracias al diseño genérico que se ha realizado para desarrollar el sistema, dando la posibilidad de adaptarse de manera óptima a cada caso.

4.2 Empleo de las tecnologías Web para enriquecer la aplicación

Se podía utilizar el componente `TaxoBuilder` de manera más autónoma, pero para esta aplicación se ha decidido que un usuario pueda interactuar con ella, de modo que pueda guiar el proceso de búsqueda. El usuario final interactuará con una interfaz gráfica para realizar peticiones al sistema y obtener conocimiento sobre un dominio que le pueda interesar. Para ello, comienza introduciendo un concepto a buscar (por ejemplo, "sensor") y la aplicación utilizaría el `taxoBuilder` para extraer conocimiento de la Web a partir de este concepto. El resultado vendría dado en un fichero con formato OWL, en el cual se construye una estructura de datos de conceptos con relaciones taxonómicas y no taxonómicas entorno al dominio "sensor". Además, como hemos comentado, los conceptos tienen asociados direcciones a páginas Web, donde podemos encontrar información relacionadas sobre dicho concepto. Con la aplicación se pretende que el usuario pueda seleccionar entre todos los conceptos devueltos por el componente `taxoBuilder`, aquellos que le interesen, con el fin de ampliar su búsqueda de conocimiento sobre ese dominio. Por cada concepto seleccionado para buscar, la aplicación utiliza una nueva instancia del componente `taxoBuilder` para que le devuelva los resultados esperados. Así se va construyendo poco a poco la Ontología deseada sobre ese dominio (en el ejemplo, "sensor"). También el usuario tiene la opción mediante la aplicación de guardar el resultado global en un único fichero de tipo OWL de forma que pueda posteriormente visualizarlo en una aplicación especializada en ficheros con este formato, para poder manipularlos y/o visualizarlos de manera independiente a la aplicación con la que ha sido creado.

Así pues, cada vez que un usuario le pide a la aplicación que busque sobre un concepto (en el caso inicial solicita una búsqueda para un único concepto, y posteriormente puede seleccionar varios a la vez), acaba traduciéndose en un trabajo para el sistema, en el cual se creará una instancia del *taxoBuilder*. Como vemos, el proceso de construcción de la Ontología final se desarrolla de manera semiautomática, en el sentido que el usuario va guiando la búsqueda por los caminos que le interesa, pero es la aplicación (mediante el *taxoBuilder*) la que internamente realiza el trabajo construyendo los resultados parcialmente.

Para poder gestionar la ejecución de las tareas basadas en el *taxoBuilder* con el objetivo de darle el sentido a la aplicación que se ha descrito en el párrafo anterior, ésta tiene que aportar una capa de interactividad rica, aportando la flexibilidad que todo usuario final espera de una aplicación. Para ello se elegida las tecnologías Web como una alternativa que cumple bien con tales requisitos.

La aplicación cliente, como ya hemos avanzado, está basada en Web. En concreto está basada en la tecnología J2EE (*Java 2 Enterprise Edition*) para la generación de páginas dinámicas y el acceso a la base de datos⁶.

De modo general, el funcionamiento de la aplicación Web es el siguiente: primero, permitirá el registro de usuarios. Una vez registrados, podrán autenticarse en el sistema y tendrán un espacio personal. En dicho espacio podrán modificar sus datos, así como darse de baja en el sistema si lo desean. También disponen de un apartado para la creación de proyectos personales. Un proyecto personal constará de una búsqueda de conocimiento sobre un dominio sobre el cual el usuario está interesado, construyendo de manera semiautomática una Ontología.

Como había comentado antes, un búsqueda comienza con un concepto, generando como resultado una serie de conceptos relacionados con el inicial, que tienen asociados enlaces a páginas Web. La búsqueda puede continuar si el usuario así lo desea, seleccionando alguno o algunos de los conceptos recibidos en la búsqueda anterior, para profundizar en ellos. Todos los resultados que se van obteniendo se van mostrando en un applet de java, haciendo uso de un componente gráfico basado en pestañas para separar los resultados parciales de cada concepto buscado, para una óptima visualización.

Toda la Ontología que se va generando forma parte de un proyecto. Si el usuario desea realizar otra búsqueda sobre algún otro dominio podrá crear otro proyecto a parte. A cada proyecto, el usuario le podrá poner un título y una descripción, para que tenga mejor organizado e identificado la construcción de sus Ontologías personales. Para que el sistema sea flexible de cara a los usuarios finales, todas las búsquedas que van realizando dentro de un proyecto particular, quedan almacenadas de forma transparente, para que cuando el usuario salga de la sesión de trabajo con el proyecto o bien cierre el navegador de manera brusca, y más tarde entre de nuevo en su proyecto, se encuentre todo tal y como estaba. Es más, la flexibilidad del sistema

⁶ Para ejecutar la aplicación el usuario necesita un navegador Web que incorpore como plug-in la máquina virtual de Java (JVM), necesaria para ejecutar en un momento determinado un applet.

permite que un usuario que realice una búsqueda, como ésta puede tardar bastante tiempo en devolver los resultados, pueda cerrar la sesión (o bien su navegador o apagar su propio ordenador) y pueda entrar más tarde, de manera que si el sistema ya había encontrado los resultados, se le muestre de igual forma que si hubiera mantenido la sesión viva durante todo el tiempo desde que hizo la petición. Y en caso de no haber acabado todavía la búsqueda, en cuanto ésta termine también se le mostrará sin problemas. También cabe destacar la ventaja que supone este modo de funcionar, el cual permite que un usuario pueda cambiar tanto de navegador Web, como de computador, así como de localización geográfica, sin que esto afecte al estado de sus proyectos, los cuales son independiente de los factores anteriores. Siempre podrá conectarse a través de cualquier navegador desde cualquier parte donde haya acceso a la red Internet, validarse en el sistema, y obtener de forma inmediata acceso a sus datos y sus proyectos personales.

4.2.1 Descripción básica de la arquitectura global

La arquitectura global (de la aplicación junto con el sistema) está compuesta por el servidor del sistema distribuido, los nodos, y los usuarios finales de la aplicación.

Igual que pasaba con los nodos en el caso del sistema distribuido, para que los usuarios de la aplicación puedan conectarse al sistema físicamente, necesitan poder “ver” al servidor. Para ello, como se muestra en la *figura 13*, el cliente debe de tener una conexión a Internet o bien estar conectado a la red interna donde se encuentre el servidor. Como vemos, tanto los nodos como los computadores de los usuarios que usan la aplicación pueden pertenecer físicamente a las mismas redes. La diferencia entre la parte del sistema distribuido y la parte de la aplicación de los usuarios es puramente lógica. Es más, incluso un computador puede ser a la vez parte del sistema distribuido y ser utilizado por un usuario de la aplicación (no se ha mostrado este caso gráficamente para que la imagen aclare mejor los conceptos).

El nexo que une al sistema con la aplicación cliente que lo utiliza, es el servidor ServerRequest. Para la interacción entre ellos, se ha creado un protocolo que especifica cuando y de que manera se intercambia la información, necesario para que logren entenderse. El protocolo se explicará con más detalle en un apartado posterior.

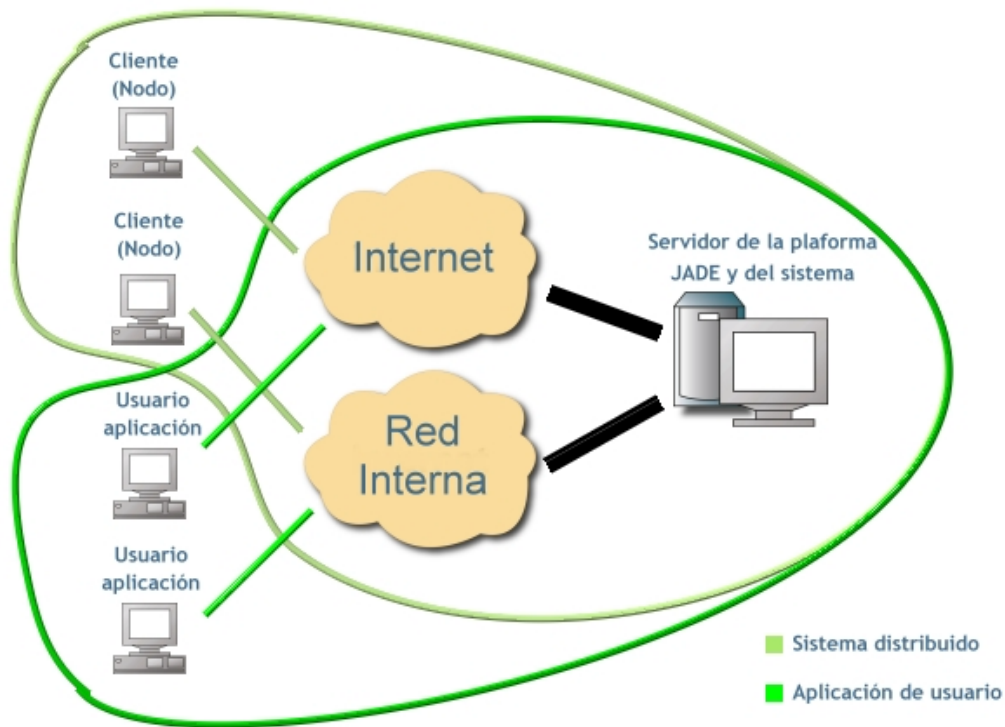


Figura 13. Arquitectura física del sistema distribuido y los clientes de la aplicación

Existe la posibilidad que un usuario decida que su computador forme parte del sistema distribuido y a la vez lo utilice como cliente para utilizar la aplicación, ya que ambos software se pueden ejecutar de manera independiente, debido a que están separados lógicamente, pero conectados por medio del servidor. Se puede pensar que el servidor tiene 2 caras hacia el exterior. Una para los nodos, y otra para la aplicación.

Como ya he comentado, la aplicación de usuario se conecta a través de un navegador Web al servidor, y este a su vez enlaza con el sistema distribuido. La parte del servidor que está de cara a la aplicación del usuario, consta de 2 partes diferenciadas tal y como se puede ver en la *figura 14*:

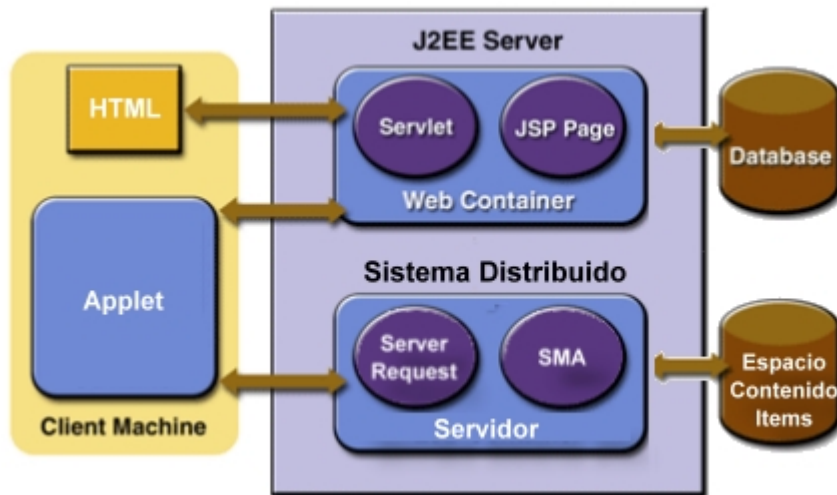


Figura 14. Diseño lógico de los componentes que interactúan entre el servidor y el cliente de la aplicación.

- a) Por un lado, tenemos la aplicación Web del usuario de la aplicación. En el servidor se encuentra un servidor Web Apache Tomcat de la versión 5.5 (en la *Figura 14* se trata del componente "J2EE Server"), escuchando por el puerto 8080, en el cual se encuentra desplegada la aplicación que genera páginas dinámicas con las que el usuario interactúa. A través de dichas páginas, un usuario se puede dar de alta en la aplicación. Una vez dado de alta, puede acceder a su espacio personal. En el puede crear sus proyectos (dándole un título y una descripción) para posteriormente poder editarlos. Toda la información relativa a los usuarios (datos personales más proyectos), es almacenada en una base de datos MySQL (véase la *figura 14*, el componente "database"), la cual es accedida por la propia aplicación J2EE de forma exclusiva.

Cuando un usuario decide editar un proyecto (trabajar con él), entra en una nueva ventana del navegador, descargándose un applet (su código) desde el propio servidor J2EE. Este applet, internamente, interactúa con la aplicación J2EE para que almacene la información relacionada con ese proyecto (en la base de datos). Pero no almacena toda la información. Como he comentado, un usuario de la aplicación, dentro de un proyecto, empieza buscando un concepto. Tras los resultados, se le muestra una serie de conceptos que están relacionados con el buscado, los cuales pueden ser candidatos a ser buscados de nuevo, generando más conceptos. Es decir, cada búsqueda de un concepto ha generado unos resultados. A la búsqueda de un concepto le he llamado *ítem*. Un ítem identifica un concepto buscado en un proyecto determinado, que a su vez pertenece a un usuario concreto.

Tanto la información referente a los usuarios, como a los proyectos y sus ítems son almacenados en la base de datos por la aplicación J2EE. De ahí que el applet deba conectándose a ella (como se muestra en la *figura 14*), para indicarle que registre o elimine ítems del proyecto sobre el cual se está trabajando.

- b) Por otro lado, el mismo applet, trabajando sobre un proyecto concreto, se conecta con el servidor `ServerRequest`, que atiende las peticiones de búsqueda para que se acaben realizándose por parte del sistema distribuido. Este servidor ha sido creado desde cero, y el applet se conecta a él por medio de un socket. Ambos utilizan un protocolo para comunicarse, que se describirá en un apartado posterior. Los resultados que genera una búsqueda (listado de conceptos relacionados taxonómicos y no taxonómicos, así como ficheros de extensión OWL), se almacenan en el disco duro (véase la *figura 14*, el componente llamado "Espacio para Contenidos de Ítems") por medio de la técnica de serialización, no en la base de datos. Es decir, en la base de datos almacenamos los ítems que tiene un proyecto (su identificador y a que usuario y proyecto pertenecen), y en el disco duro almacenamos los resultados que el `taxoBuilder` nos proporciona (contenido asociado a un ítem). Posteriormente, el applet recibe los resultados por medio del servidor `ServerRequest` y los muestra al usuario representándolos gráficamente. En resumen, el applet del cliente se conecta tanto a la aplicación J2EE (para interactuar con la base de datos), como al sistema distribuido para realizar peticiones y recibir los resultados relacionados con los ítems.

De esta forma, cuando queramos reeditar un proyecto (volver a trabajar con él), lo que hace el applet para recuperar su estado es lo siguiente:

- Se conecta a la aplicación J2EE para que le diga que ítems tiene su proyecto actual, consultando a la base de datos.
- Tras recibir el listado de ítems, el applet se conecta al servidor `ServerRequest` para solicitarle los resultados guardados relativos a cada ítem, uno por uno. Conforme va recibiendo los resultados los va visualizando al usuario. Estos resultados los recibe el applet en forma de objetos de Java, por medio del mecanismo de serialización y a través de un socket.

Cada resultado referente a un ítem, se muestra dentro del applet mediante un componente gráfico (del API de gráficos swing de Sun) llamado `JtabbedPane`, que es un elemento que puede contener diversas pestañas, en la que cada una muestra un panel independiente compuesto por otros elementos gráficos (botones, árboles, listas, etc.).

Con esta arquitectura conseguimos desacoplar lo que es la aplicación Web de lo que es el sistema distribuido, quedando un diseño más limpio.

4.2.2 El servidor de peticiones (ServerRequest). Diseño del protocolo

La interacción entre el servidor de peticiones (`ServerRequest`) y la aplicación del cliente, en concreto, el applet, se realiza para llevar a cabo dos tipos de acciones:

- a) Realizar una petición de búsqueda.
- b) Solicitar los resultados de una búsqueda solicitada.

Debido a que una búsqueda puede tardar mucho tiempo en generar los resultados, no tiene sentido que la petición quede “congelada” en el servidor hasta que se consigan los resultados. El problema está en que estamos haciendo utilizando un socket que tiene asociado un puerto en el servidor, limitando la disponibilidad del servicio considerablemente.

Un problema que tiene la arquitectura TCP/IP [TCPIP, Spec06] para las aplicaciones de tipo cliente-servidor es el hecho de no poder desarrollar aplicaciones que sean capaces de mantener un mecanismo de suscripción de clientes, los cuales necesitan ser avisados por el servidor de ciertos eventos esperados, en el momento en que se producen. Esto es debido a la existencia de computadores clientes con direcciones IP privadas para uso dentro de una red interna, que hace que un computador externo a dicha red no pueda establecer una conexión de forma directa con él. Para solucionar esta barrera, en la mayoría de casos se suele adoptar un mecanismo de “polling”, en el cual son los clientes quienes preguntan al servidor cada cierto intervalo de tiempo si se han producido nuevos eventos⁷. Dicho mecanismo es menos eficiente que el primero que he comentado. En el primero, la comunicación entre cliente y servidor sólo se produce en el momento en que se producen los eventos y por tanto la información que se transmite es siempre necesaria. En cambio, mediante “polling”, en muchas ocasiones un cliente preguntará al servidor si hay novedades en los eventos que espera, respondiéndole de forma negativa, y por tanto, hay un exceso de tráfico que produce ineficiencia. Pero como no queda alternativa ha día de hoy, es un mecanismo comúnmente aceptado.

Además, como es el cliente quien esporádicamente hace peticiones al servidor, en cada acto comunicativo, debe notificarle al servidor de qué proyecto se trata, y a qué usuario pertenece, ya que para todas las acciones es necesario saberlo.

En el caso de solicitar una petición de búsqueda, el cliente debe notificarle al servidor los criterios para esa búsqueda, a través de los parámetros ha introducido el usuario mediante la interfaz gráfica. Dichos parámetros se encapsulan dentro de una clase creada con tal fin, que finalmente es enviada por serialización al servidor. Es decir, no solo se trata de un protocolo orientado a texto, sino que también se aprovecha del mecanismo de serialización para enviar de forma más eficiente los parámetros. Para el caso de solicitar los resultados de una búsqueda, en caso de éxito (los resultados están disponibles) el servidor envía al cliente, también por serialización, un par de ficheros de texto (haciendo uso de dos objetos de la clase `NetFile`), y dos tablas de hash que contienen listados de conceptos taxonómicos y no taxonómicos respectivamente. En la *figura 15* se detalla gráficamente el funcionamiento del

⁷ El servidor si que es accesible desde los clientes debido a que el primero dispone de una IP pública y los segundos, los que tienen una IP privada, emplean un mecanismo llamado NAT (Network Address Translation), que básicamente consiste en utilizar un computador de la red con IP pública, como representante para hacer peticiones al exterior, el cual si que es visible para el servidor, quien tras recibir sus peticiones puede responderle. Finalmente, el representante acaba devolviendo la respuesta al computador con IP privada que había realizado realmente la solicitud. Es decir, se trata incorporar un computador de la red con IP pública que haga proxy para comunicar los computadores con IP privada con el exterior.

protocolo creado, para el caso de peticiones de búsqueda. Y en la *figura 16*, para el caso de las peticiones de los resultados.

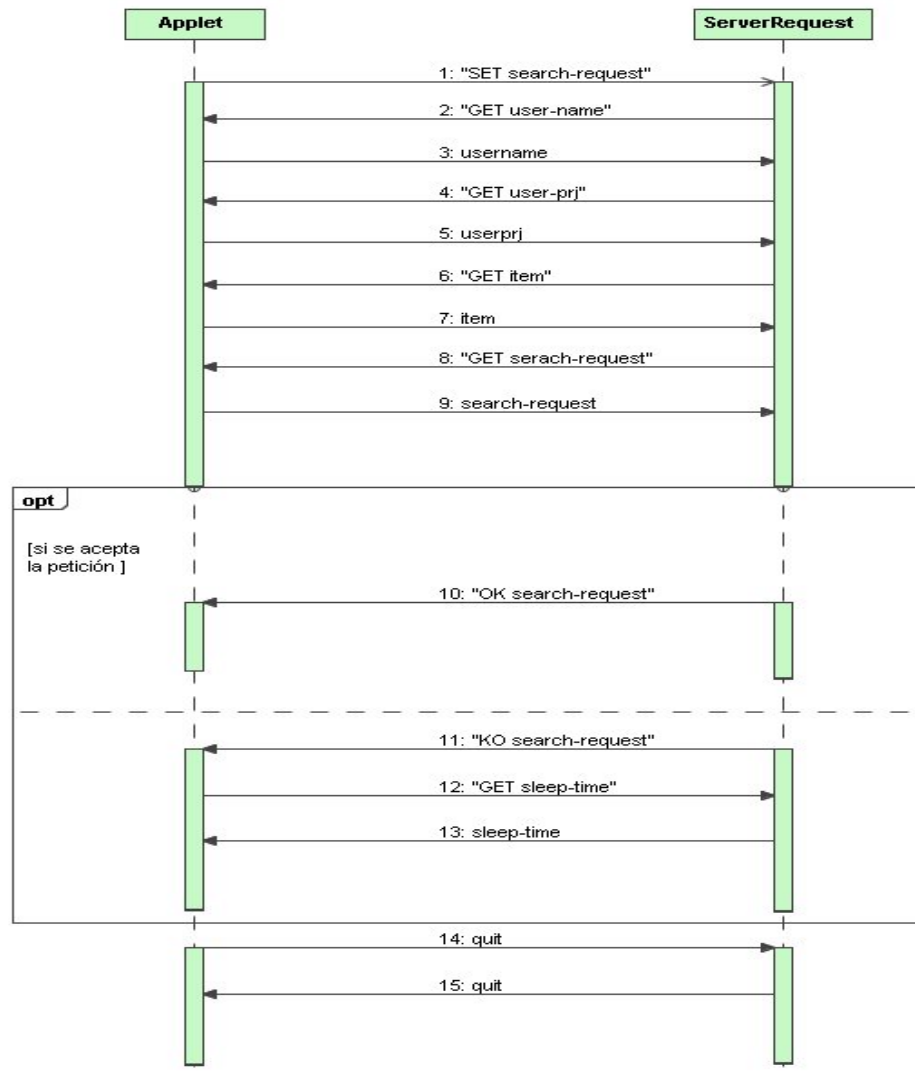


Figura 15. Diagrama de secuencias de mensajes intercambiados, definidos por el protocolo, para la petición de una búsqueda de un concepto para un proyecto perteneciente a un usuario

- Petición de búsqueda: el applet primeramente se conecta a la aplicación Web para que registre en la base de datos un nuevo ítem asociado al concepto que se quiere buscar, que a su vez pertenece a un proyecto del usuario que hace la petición. Una vez registrado en la base de datos, se comunica con el servidor ServerRequest. Le realiza una petición de búsqueda mediante el mensaje "SET search-request". Después, el servidor solicita al cliente el nombre del usuario ("GET user-name"), el identificador del proyecto ("GET user-prj") y el del ítem ("GET item-id"). Finalmente, le pide los parámetros introducidos para esta búsqueda ("GET search-request"). Cuando los recibe, el servidor interactúa con el sistema para mirar si dicha petición se puede atender o no. Si se atiende, se lo confirma al cliente indicando que el trabajo de búsqueda se ha puesto en

marcha en el sistema. En caso contrario (en la *figura 15*, sería el caso “else” del bloque de opción), el servidor le indica que no puede ser atendida, y acto seguido le indica el tiempo que tiene que transcurrir para que vuelva a intentarlo (en milisegundos). Esto se ha hecho con el objetivo de conseguir un sistema **escalable**, que soporte el mayor número de clientes simultáneamente. El criterio adoptado para el cálculo del tiempo de espera está relacionado con la carga que tiene el sistema, ya que es la única información que tenemos. Si resulta que todos los nodos tienen todos sus slots ocupados (trabajando al 100%) en ese momento, la probabilidad de que un slot de un nodo sea liberado en un breve periodo de tiempo será alta. **No podemos decir exactamente** que el tiempo de espera sea inversamente proporcional al número de slots ocupados, debido a que el tiempo que tardan en realizarse estos trabajos particulares es indeterminista. Pero como de **indeterminismo** se trata, sí que es una fórmula que se aproxima bastante bien al carácter aleatorio real.

Así pues, la fórmula para calcular el tiempo de espera, teniendo en cuenta la **escasa información** de la que se dispone, es la siguiente:

$T = 100000 \text{ ms} / \text{totalSlots}$, donde:

- T: es el tiempo de espera para solicitar de nuevo una petición de búsqueda cuando no hay slots libres.
- totalSlots: es el número total de slots que tiene el sistema en el momento del cálculo (que están todos ocupados).

Así pues, si resulta que en el sistema hay un único slot, y éste está ocupado, la aplicación deberá de esperar 100 segundos para intentarlo de nuevo.

Cabe destacar que todo este mecanismo es transparente para el usuario final, ya que se hace en *background*. De esta forma, un usuario que lanza una petición de búsqueda ve como siempre el sistema la atiende la petición. Lo que pasa es que si resulta que está muy cargado, pues normalmente tardará más tiempo en procesarse. El usuario es consciente de la carga aproximada que tiene el sistema, ya que en el interfaz gráfico del applet, habrá un apartado donde se indica el número de recursos aproximados que hay en cada momento. Esto se verá con más detalle en el apartado correspondiente a la visualización gráfica del espacio de trabajo.

- Solicitar resultados: el applet se comunica con el servidor `ServerRequest`. Le realiza una petición para obtener los resultados de una búsqueda mediante el mensaje “GET results”. Después el servidor le pide, igual que en el caso anterior, el nombre del usuario, el identificador del proyecto, y el identificador del ítem por el que se pregunta. Una vez recibidos, consulta en el espacio del disco duro donde se almacenan los resultados reales (los 2 ficheros de texto junto con las 2 tablas de hash). Si el servidor los encuentra, se los devuelve por medio de la serialización (2 objetos de tipo `NetFile` más 2 de tipo `Hashtable`), que en la *figura 13* queda reflejado en el envío del mensaje número 10. Se puede observar que los 4 objetos se envían a la vez en una sola interacción, y es cierto, ya que están metidos en un *array* de 4 posiciones. En

caso de no encontrarse disponibles los resultados (en la *figura 16*, sería el caso "else" del bloque de opción), el servidor le indica al cliente que se espere un determinado tiempo para volver a intentarlo.

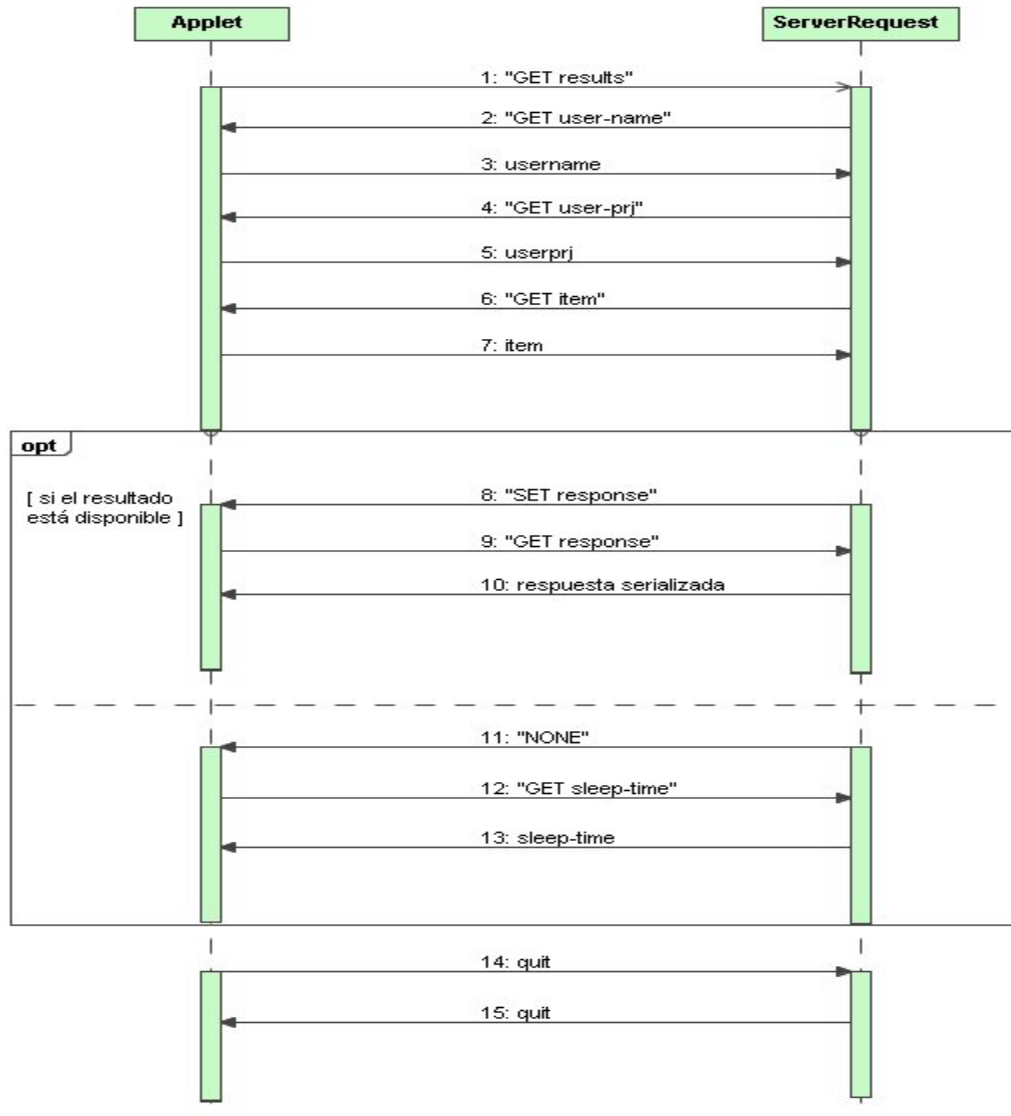


Figura 16. Diagrama de secuencias de mensajes intercambiados, definidos por el protocolo, para la petición de los resultados de una búsqueda de un concepto perteneciente a un proyecto de un usuario

Este mecanismo de tiempo de espera, igual que en el caso anterior, se emplea con el objetivo de conseguir una mayor escalabilidad, para soportar el mayor número posible de clientes concurrentes. Como el tiempo que tarda en acabar un trabajo es completamente **indeterminista**, en este caso, el criterio adoptado es el siguiente: cuantos más slots estén ocupados (realizando trabajos), mayor será el número de clientes que soliciten resultados a la vez. Por tanto, el tiempo de espera para un nuevo intento de solicitud puede considerarse, **de manera aproximada**, proporcional a la carga del sistema basado en el número de slots ocupados.

La fórmula final, teniendo en cuenta la **escasa información** de la que se dispone, queda de la siguiente manera:

$T = 10000 \text{ ms} * (\text{totalSlots} - \text{totalFreeSlots})$, donde:

T: es el tiempo de espera para un nuevo intento

(totalSlots – totalFreeSlots): es la cantidad de slots ocupados

También hay que decir de nuevo que esté mecanismo se hace en *background* y por tanto el usuario final no se entera. Las peticiones de resultados no sólo se recuperan una única vez, sino que también son realizadas cada vez que un proyecto es reeditado, el cual necesita recuperar su estado al completo nuevamente. Pero una vez que los resultados están almacenados en el servidor, la petición de los mismo siempre es exitosa.

Las dos fórmulas no se han podido ajustar mejor debido al **carácter indeterminista** que tiene este tipo de problema en particular en cuanto al tiempo de ejecución. Hay que recordar que además de depender del computador sobre el que se ejecuta (tipo de conexión a Internet, velocidad del procesador, etc.), también depende del estado de la red Internet, ya que el componente TaxoBuilder realiza muchísimas operaciones de entrada/salida sobre ella (las realiza constantemente durante toda la ejecución). Todos estos factores añaden variables aleatorias a considerar.

4.2.3 Aplicación Web en J2EE basada en el patrón Modelo-Vista-Controlador (MVC)

La aplicación Web debe permitir que la navegación para los usuarios sea sencilla, encontrando fácilmente todo lo que necesitan. A través de ella, se permitirá las siguientes acciones:

- Un usuario anónimo podrá registrarse en la aplicación rellenando un formulario de datos básicos.
- Una vez registrado, tendrá acceso a un espacio personal, desde el cual podrá gestionar sus datos, así como poder solicitar la baja de su registro.
- En ese mismo espacio personal, tendrá la opción de crear nuevos proyectos, rellenando un formulario en el que se le pide un título y una descripción.
- Una vez creado un proyecto, este aparecerá en un apartado especial donde se listan todos sus proyectos. Desde ahí podrá eliminarlos si lo considera oportuno, o bien editarlos, es decir, trabajar con ellos (sesión).

- Cuando decide editar un proyecto, entra en un nuevo apartado donde se ejecuta un applet, donde se muestra de manera gráfica todo el estado del proyecto. Todas las acciones que un usuario va realizando sobre un determinado proyecto durante sus sesiones, quedan almacenadas implícitamente por la aplicación, de forma que cada vez que lo edite, se lo encuentre tal y como estaba desde su última sesión. Además, es posible que en su última sesión dejara en proceso de búsqueda algunas peticiones al sistema, de las cuales algunas podrían tener los resultados disponibles, de forma que al volver a entrar para editarlo de nuevo, ya se visualizarían.
- Finalmente, dentro del espacio de trabajo (en el applet), tiene la opción de almacenar localmente el contenido de un resultado parcial, en un fichero en formato OWL. Este fichero lo podrá visualizar o editar mediante otra aplicación externa si le interesa.

Como ya hemos comentado, la tecnología utilizada para la creación de páginas dinámicas es J2EE. Para tener un diseño limpio y fácilmente mantenible, se ha optado por aplicar un patrón de diseño llamado *Modelo-Vista-Controlador* (MVC), el cual está basado en el patrón *Observer*. Este modelo permite tener las siguientes ventajas para el desarrollo de una aplicación Web:

- Separa claramente los 3 roles que intervienen en el desarrollo de una aplicación Web, que son el modelo, la vista y el controlador (véase la *figura 17*):
 - o El **modelo**: encapsula el estado de la aplicación, y es el responsable de la capa lógica aportando toda la funcionalidad de la aplicación. Es la única parte que accede a los datos al almacén de datos, ya sea para leerlos, modificarlos o insertarlos. Se encarga de notificar los cambios a las vistas.
 - o La **vista**: se encarga básicamente de renderizar los modelos de forma gráfica. De esta forma se consigue separar lo que es la representación (vista) de los datos, de la funcionalidad de la aplicación.
 - o El **controlador**: es el encargado de recoger todas las peticiones del exterior. A partir de ellas, decide si estas acciones del usuario tienen que pasar por los modelos (para hacer actualizaciones de los mismos) y finalmente delega a una vista la renderización de la respuesta.
- Permite superar el problema de implementar un diseño basado en el patrón *Observer* bajo un protocolo sin estado como es el HTTP, el cual está basado en acciones simples de petición-respuesta (http-request y http-response)
- Se ajusta perfectamente para las aplicaciones basadas en multi-capas, permitiendo que cada una esté separada claramente de forma lógica, haciendo que el desarrollo sea más estructurado y fácil de mantener.

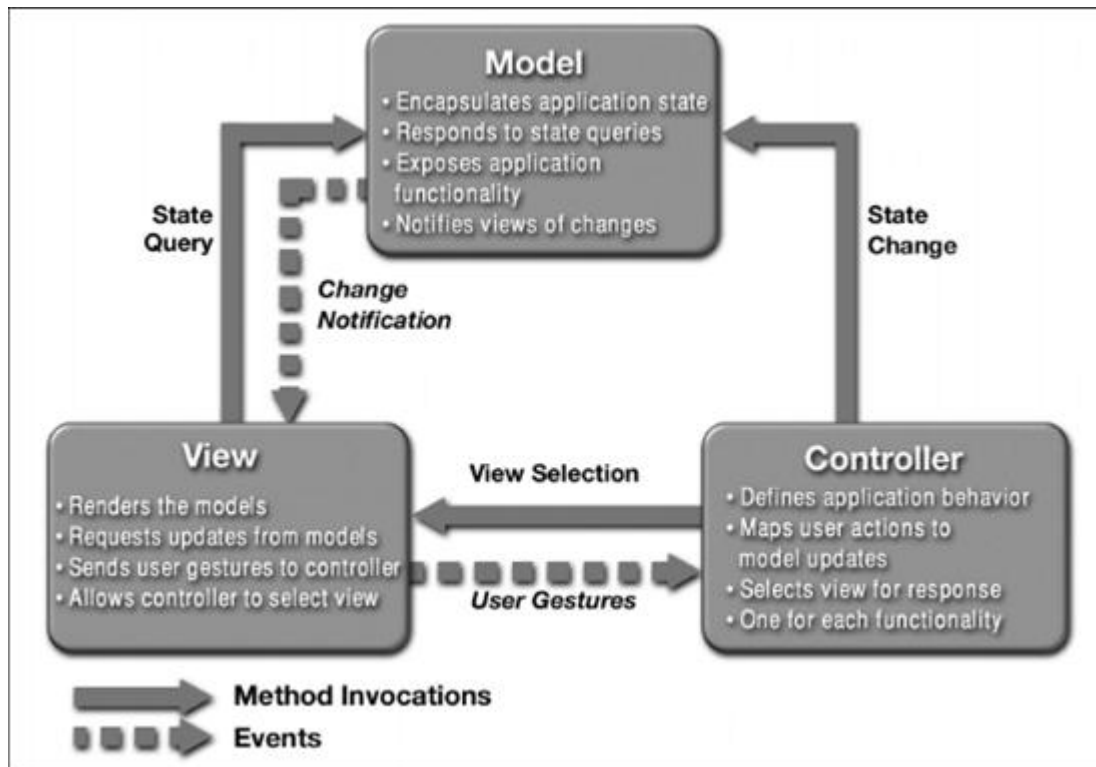


Figura 17. Esquema del patrón de diseño Modelo-Vista-Controlador (MVC).

Una secuencia típica de eventos que se dan mediante este patrón podría ser la que se muestra en la figura 18.

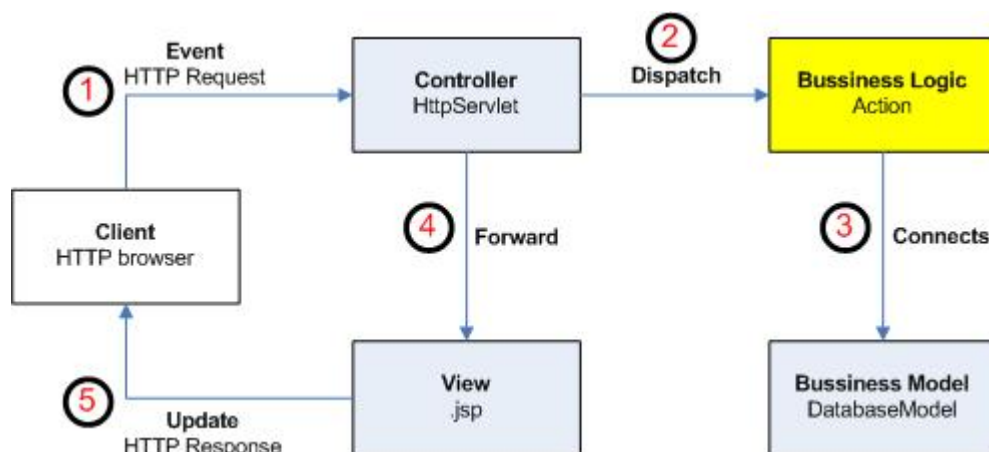


Figura 18. Secuencia de eventos típica en un diseño MVC.

A modo de ejemplo, si un usuario registrado desea conectarse al sistema, se producen los siguientes eventos:

- El usuario rellena un formulario de conexión (login y password), que los envía al servidor para intentar autenticarse. El envío se hace en forma de petición HTTP (evento núm. 1).
- Esta petición o acción del usuario pasa por el controlador, quien delega la comprobación de la validación de los datos al modelo (evento núm. 2).
- Para tal validación, necesita acceder al almacén de datos (evento núm. 3) para consultar si existe un usuario con esos datos. En caso afirmativo, marca en la sesión que el usuario está autenticado. Si no, marca que se ha producido algún tipo de error en la autenticación.
- Finalmente, el controlador toma el mando de nuevo, para delegar a una vista asociada a la acción de autenticación de usuarios (evento núm. 4), la generación de la respuesta final. La vista consultará si en la sesión actual el usuario está autenticado o no, para mostrarlo de manera gráfica generando una respuesta HTTP que viaja al cliente (evento núm. 5).

Como se puede observar, el mantenimiento de la aplicación resulta más sencillo debido a la clara estructuración de las diferentes partes implicadas.

Hasta ahora he hablado de la persistencia haciendo referencia a un almacén de datos. No he mencionado aún en este apartado el término "base de datos", porque resulta que se ha aplicado otro patrón de diseño para encapsular el acceso a datos, que se explica en el siguiente apartado.

4.2.4 La persistencia de los usuarios y de sus proyectos. Aplicación del patrón DAO para acceso a datos

La aplicación de un patrón para encapsular el acceso a datos, tiene como objetivo abstraer la manera en que se accede realmente a los datos, creando una capa por encima para que el acceso a los mismos sea uniforme. Este patrón es conocido con el nombre de DAO (Data Access Object).

De esta manera, debajo de esta capa se realiza el acceso a la fuente de datos donde realmente son almacenados. Esta fuente de datos puede ser una base de datos relacional, un conjunto de fichero serializables, o ficheros en un formato XML específico, etc. Da igual que tipo de fuente se emplee, ya que lo importante es que la aplicación siempre opera con la interfaz uniforme que se ha introducido, de manera que si se decide cambiar una fuente específica por otra en un momento dado, la estructura general del código no se vea afectada.

En la *figura 19* se detalla gráficamente el diseño del patrón DAO. El patrón se basa en tener unas clases que representan a entidades (que heredan de `IValueObject`, mapeándose directamente con las entidades que existen en un modelo relacional), asociadas a otras (que heredan de `IDao`), que implementan una serie de operaciones típicas que se dan a la hora de almacenar datos de dichas entidades.

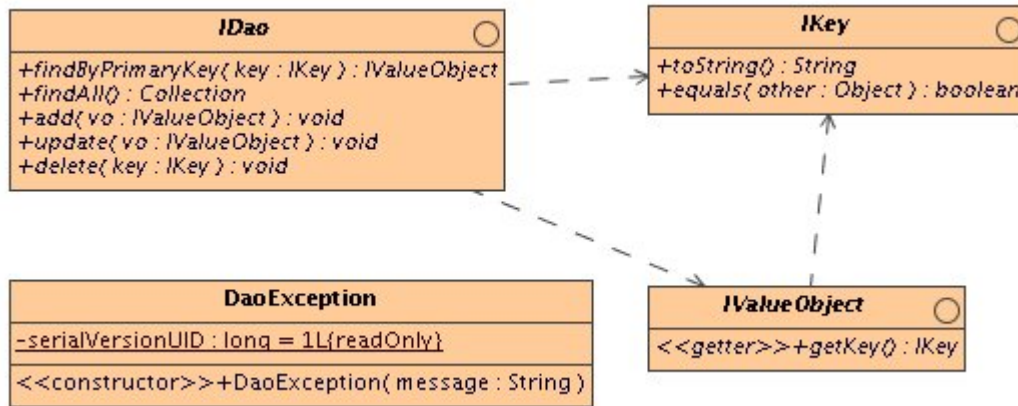


Figura 19. Esquema gráfico del diseño del patrón DAO.

Como se puede ver en la *figura 19*, existe la interfaz `IKey`, ya que se considera que toda entidad debe de tener un campo clave, a partir del cual se identifican de manera única a sus registros. Las operaciones típicas relacionadas con una entidad son las de buscar un registro a partir de su clave, leer todos los registros (para listados), y las de añadir, modificar y eliminar un registro. También es posible que para una entidad en concreto se añada de manera particular otro tipo de operación. Por ejemplo, puede interesar obtener un listado de los 10 usuarios que más proyectos tienen creados. Esto no es un problema para este diseño, ya que el patrón DAO simplemente establece unas interfaces que son comunes a todas las entidades existentes, cose que no impide añadir operaciones particulares para una entidad concreta en su implementación.

Los datos referentes a la conexión están configurados en un fichero de contexto de la aplicación con un formato basado en la especificación del servidor Apache Tomcat, llamado fichero de contexto ("context.xml"). De esta manera, no es necesario cambiar el código fuente de la aplicación por el mero hecho de cambiar de un tipo de base de datos a otra.

La base de datos elegida para la persistencia es MySQL [MySQL06], y por tanto la configuración de dicho fichero en la versión final de este proyecto está preparado para conectarse a ella.

Las entidades que tendremos vienen dadas por el diseño entidad-relación de Chain para esta aplicación (ver *figura 20*):

- Usuarios: entidad que representa a los usuarios de la aplicación
- Proyectos: entidad que representa a los proyecto de los usuarios. Un proyecto está asociado a un único usuario. Se trata de una entidad débil, que depende

de Usuarios, de forma que si un usuario es eliminado de la base de datos, automáticamente quedan eliminados todos sus proyectos.

- Items: entidad que representa a los ítems de los proyectos. Cada ítem pertenece a un solo proyecto. De esta forma, si un proyecto es eliminado de la base de datos, en consecuencia, todos sus ítems también serán eliminados.

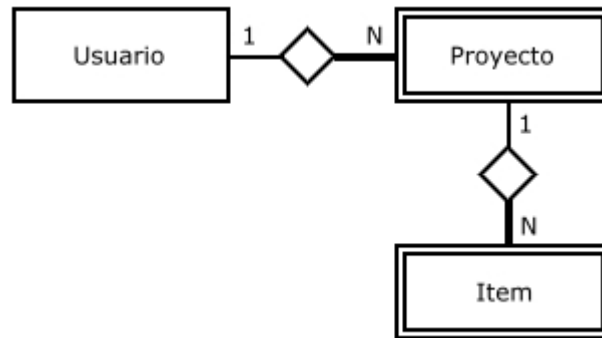


Figura 20. Modelo entidad-relación para esta aplicación.

Los campos y sus tipos de datos (en formato de MySQL [MySQL06]) de las diferentes tablas son los siguientes:

- Usuario:
 - o login VARCHAR(30) (Primary Key)
 - o password VARCHAR(30)
 - o name VARCHAR(30)
 - o last_name VARCHAR(60)
 - o rol VARCHAR(5) (puede ser admin o user)
 - o create_date DATE
 - o email VARCHAR(60)
- Proyecto:
 - o id_prj VARCHAR(30) (Primary Key)
 - o login VARCHAR(30) (Foreign Key a *login* de la tabla *Usuarios*)
 - o title VARCHAR(60)
 - o description TEXT
 - o create_date DATE
 - o path_onto VARCHAR(255)
- Item:
 - o id_item VARCHAR(30) (Primary Key)
 - o id_prj VARCHAR(30) (Foreign Key a *id_prj* de la tabla *Proyectos*)
 - o title VARCHAR(100)

En el diseño entidad-relación de la *figura 20* queda reflejado que un proyecto pertenece a un usuario. Además, *Proyecto* es una entidad débil respecto a *Usuarios* de

la misma forma que *Item* es una entidad débil respecto a *Proyecto*. Que una entidad sea débil respecto de la otra significa que existe una relación de dependencia de existencia. Un proyecto pertenece a un usuario y no tiene sentido sin su existencia (hay alguien quien lo edita). Para conseguir este tipo de relaciones en el lenguaje SQL, se hace uso de las llamadas *Foreign Key*. A continuación se explica en qué consisten y como se han empleado. Para ello, también es necesario conocer que es una *Primary Key*.

- **Primary key** define una clave principal (podría estar formado por más de un campo, pero en nuestro caso nos a bastado con un único campo para cada tabla) que sirve para identificar de manera única a un registro de una tabla.
- **Foreign Key** sirve para definir una clave foránea sobre un campo o una combinación de campos de una tabla externa. Este tipo de claves hacen referencia a otros campos (normalmente a claves principales) de otras tablas, para garantizar la integridad referencial (e.g. para que un *ítem* pertenezca a un *proyecto*, este último debe existir). Además, nos permite definir ciertas reglas de integridad interesantes, como exigir que tras la eliminación de un registro de la tabla *proyecto*, se eliminen de forma automática todos los registros de la tabla *item* que hacían referencia a dicho proyecto a través del campo referenciador *id_prj*.

4.2.5 AJAX para una mejor interacción

Es conocido en el mundo del desarrollo de aplicaciones, que las aplicaciones de escritorio son más ricas que las aplicaciones Web. Las de escritorio suelen ser más interactivas, teniendo tiempos de respuesta más rápidos. Además, en ellas no existe la necesidad de recargar por completo toda la visualización, sino que en cada interacción cambia solamente ciertos componentes gráficos afectados, dando un efecto más dinámico.

Esta barrera que separa el mundo de las aplicaciones de escritorio respecto de las aplicaciones basadas en Web, se está haciendo cada más estrecha. Siguiendo este objetivo, ha surgido una tecnología reciente conocida con las siglas de AJAX (*Asynchronous JavaScript And XML*). En realidad no se trata de una tecnología nueva, sino, como su propio nombre indica, de una agrupación de tecnologías ya maduras en el mundo de la Web. Se trata de una manera innovadora de desarrollar aplicaciones Web, que permite que estas seas más interactivas, acercándose muchos más a la riqueza de las aplicaciones de escritorio. Gigantes del software, como Microsoft (e.g. con el software perteneciente a la campaña "live"), Yahoo o Goolge (e.g. con Gmail⁸ o Google Maps⁹), entre otros, están apostando fuertemente para sus desarrollos. Básicamente consiste de lo siguiente:

- Se hace uso en el cliente Web de un lenguaje interpretado como es JavaScript

⁸ **Gmail** es el servicio de correo electrónico basado en Web (webmail) de Google. (ver figura 21)

⁹ **Google Maps** es un servicio de Google que ofrece mapas de ciudades de diversos países.

- Mediante este lenguaje, se pueden lanzar peticiones (`HttpRequest`) síncronas o asíncronas (opción por defecto y recomendable) al servidor, codificándose en XML (sobre HTTP) y de manera transparente al usuario de la aplicación. Esto se hace empleando un objeto conocido con el nombre de `XMLHttpRequest`, el cual está implementado en la mayoría de los navegadores más populares de la actualidad (para Internet Explorer la manera de instanciar el objeto es un tanto diferente, pero se puede solventar identificando de qué navegador se trata).
- Por tanto, estas peticiones se hacen en background, y esperan la respuesta por parte del servidor.
- Cuando el servidor envía la respuesta (`HttpResponse`), ésta es recuperada por medio del mismo objeto `XMLHttpRequest`, y es tratada por una función de JavaScript para tal fin (*handle response*). Dentro del propio lenguaje JavaScript, se puede acceder al contenido del documento HTML, mediante un objeto DOM¹⁰ (*Document Object Model*) [DOM, Spec06] estandarizado por el W3C (World Wide Web Consortium) [W3C06]. De esta forma, se puede modificar el contenido del documento HTML en base a la respuesta obtenida por el servidor.
- Dicho cambio se produce de forma dinámica, sin la necesidad de tener que refrescar la página al completo para ser renderizada de nuevo. De este forma, se consigue una interacción más fluida y atractiva para al usuario de la aplicación.

¹⁰ **Modelo de Objetos de Documento** (en inglés, *Document Object Model* o **DOM**) es una forma de representar los elementos de un documento estructurado (tal como una página web HTML o un documento XML) como objetos que tienen sus propios métodos y propiedades. Está estandarizado [DOM, Spec06] por el *World Wide Web Consortium* (W3C) [W3C06].

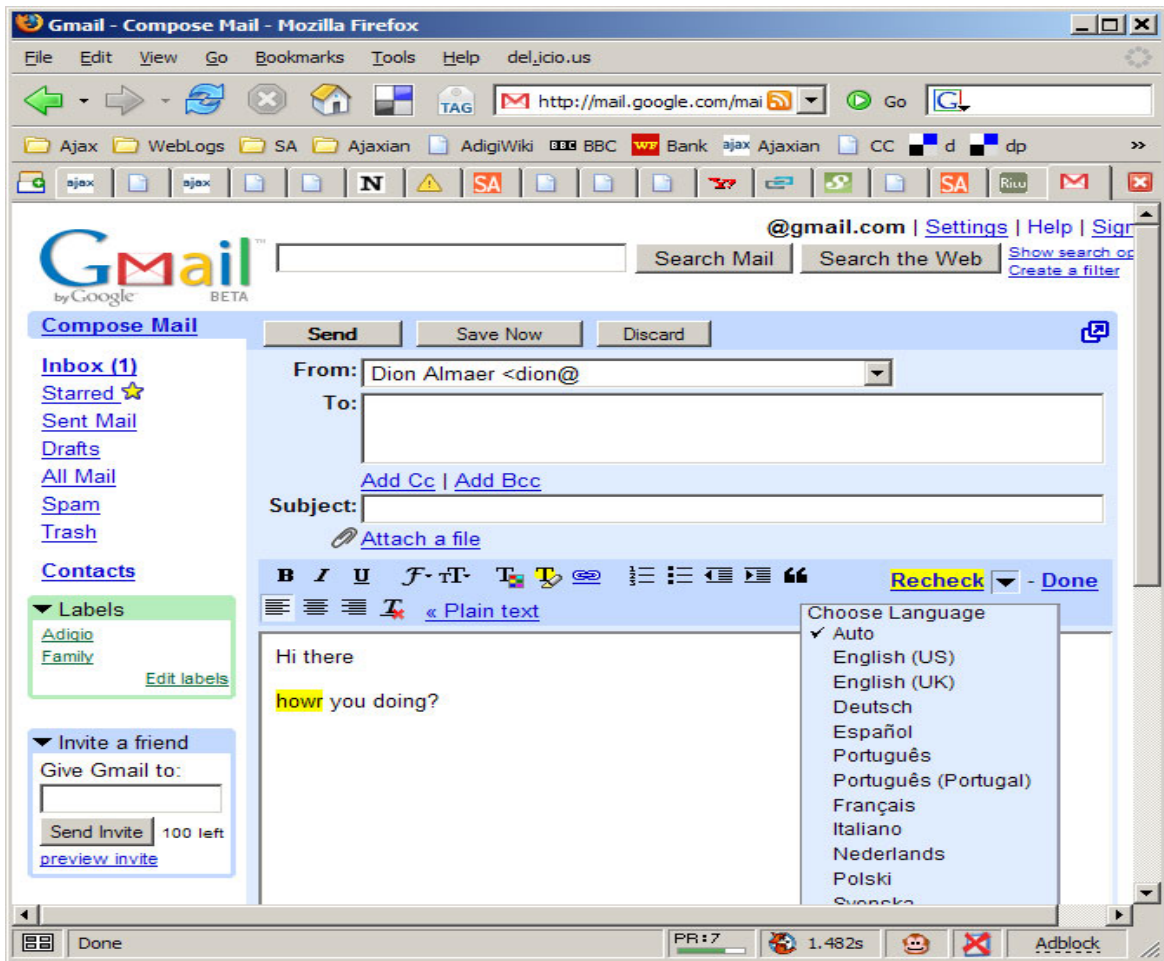


Figura 21. Gmail es un ejemplo de aplicación que hace uso de AJAX

En esta aplicación se ha empleado AJAX en algunas partes interesantes. Por ejemplo, en el momento en que un usuario entra en su espacio personal. El listado de proyectos que tiene, por defecto se encuentra oculto. Mediante un botón pueden desplegar el listado para visualizarlo, sin necesidad de recargarse la página por completo. Aquí actúa el motor AJAX, quien hace la petición al servidor para obtener el listado de proyectos del usuario. Una vez tomada la respuesta, se accede mediante el objeto DOM a un elemento HTML (una tabla vacía) por su identificador, donde se carga el contenido de la respuesta (HTML) del listado. A su vez, el listado puede ser ocultado de nuevo, dejando vacío nuevamente la tabla. Para vaciar la tabla no es necesario comunicarse con el servidor, y se accede directamente mediante el objeto DOM para hacer que su contenido sea eliminado.

En modo de resumen, se puede decir que la mayoría de aplicaciones Web que se habían desarrollado hasta la fecha, estaban basadas en un modelo de petición, espera (normalmente visualizando una página en blanco y tardando un tiempo considerable), y respuesta (renderizado de una página al completo). Da igual si el usuario a realizado una acción que solo modifica una parte pequeña de la página. La solicitud recibe como respuesta una nueva página completa que ha de viajar por la red y que se renderiza en el cliente por completo. Esto acaba traducándose por una larga espera. AJAX soluciona estos inconvenientes, recargando solamente las partes de la página que

deben de variar, de manera que se reduce el tráfico de la red. Los tiempos de espera son más reducidos, y además, se hace en segundo plano de manera transparente al usuario. Gracias a AJAX obtenemos como resultado una aplicación que resulta más interactiva y por tanto más atractiva para el usuario final.

Como desventajas que tiene, se pueden mencionar las siguientes:

- Dependencia del navegador con el lenguaje JavaScript para su funcionamiento.
- Problemas asociados con la usabilidad. Un usuario que utiliza frecuentemente los típicos botones de los navegadores de "ir atrás" y "refrescar", puede que le cueste entender que la navegación no funcione de la misma manera en la que estaba acostumbrado.

4.2.6 Edición de los proyectos mediante un applet

Los proyectos se editan mediante un applet de Java, que se ha comentado, permite crear un espacio de trabajo potente, ya que requiere de mucha interacción con el usuario, y debe de mostrar los resultados de una manera sofisticada.

La *figura 22* muestra un captura de pantalla del applet, en medio de una edición de un proyecto. A partir del gráfico, a continuación se comentan las diferentes partes de que consta:

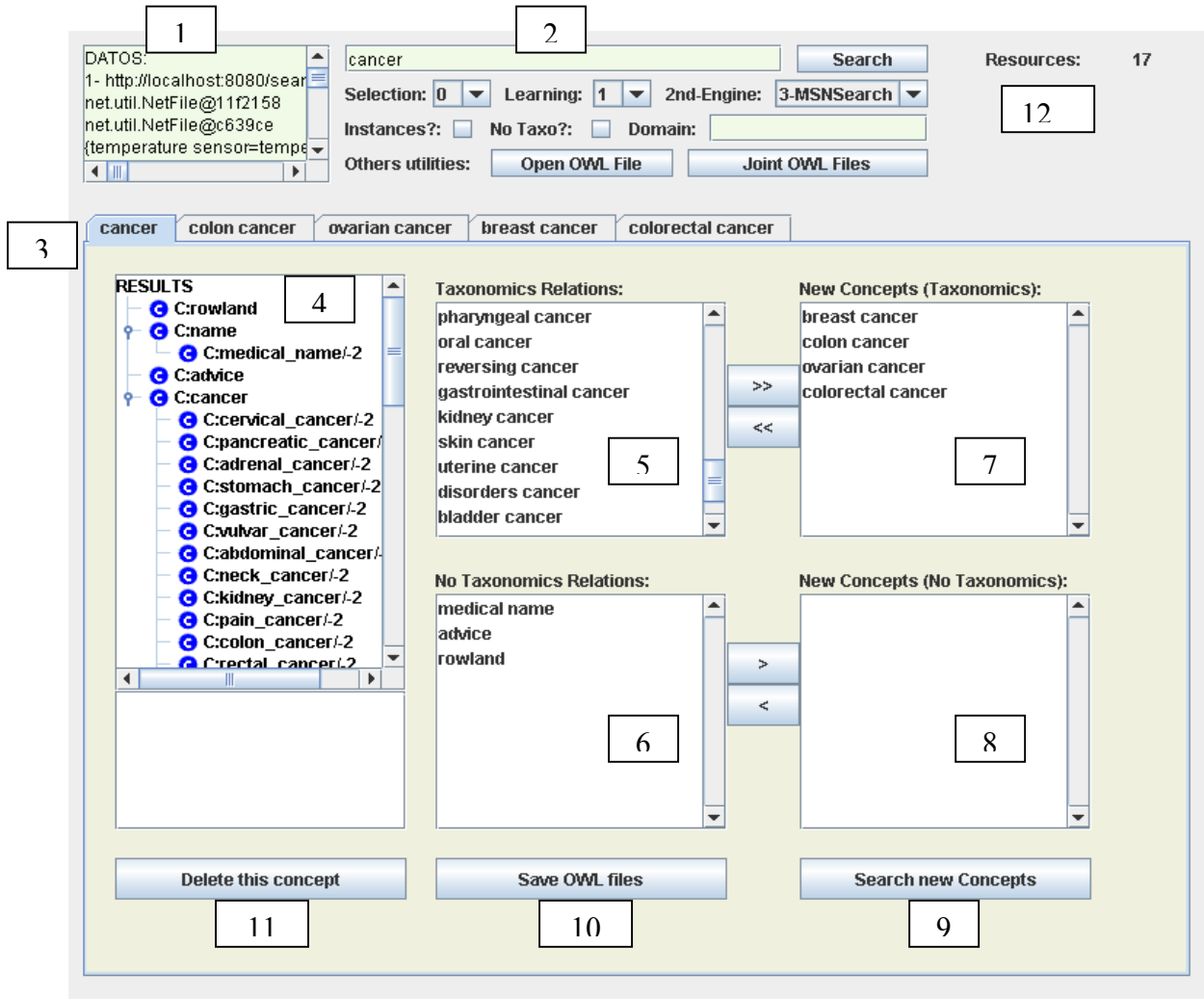


Figura 22. Aspecto que tiene el applet para la edición de un proyecto

- 1- Se trata de un área de registro de acciones que se van realizando. El objetivo es que un usuario que detecte un error en la aplicación, envíe un email de notificación, adjuntando el contenido del registro para que el desarrollador pueda interpretar mejor de dónde puede provenir el fallo.
- 2- Aquí es donde el usuario introduce los parámetros de la búsqueda (para el taxoBuilder). Las opciones son las siguientes:
 - a. La caja de texto sirve para introducir el nombre del concepto a buscar.
 - b. Selection: indica el nivel de rigidez que tendrá el buscador en el momento de decidir si un concepto encontrado es válido o no. A mayor rigidez, menor será el tamaño del resultado. El valor oscila entre 0 y 3.
 - c. Learning: indica al buscador el nivel de aprendizaje adoptará. A mayor capacidad de aprendizaje, mayor tiempo estará realizando búsquedas. El valor oscila entre 0 y 3.
 - d. Engine: permite indicar qué motor de búsqueda secundario debe de utilizar el buscador (TaxoBuilder). Las opciones son: MSNSearch, Yahoo

- y Google. En teoría, los resultados pueden variar de un buscador a otro y puede resultar interesante de cara al usuario final comparar la calidad de las búsquedas con las diferentes alternativas.
- e. Instances: indica al buscador si también se quieren obtener instancias de los conceptos encontrados (e.g. National Cancer Institute (USA) o Cancer Research (UK) dentro del dominio cáncer).
 - f. No taxo: da la opción de buscar conceptos relacionados de forma no taxonómica.
 - g. Domain: se puede especificar un dominio Web sobre el que se debe restringir la búsqueda.
- 3- Se trata de un componente gráfico basado en pestañas, para permitir visualizar todos los resultados. En cada pestaña se muestra el resultado de una búsqueda. En el ejemplo de la *figura 22*, se ha comenzado introduciendo el concepto de "cancer", cuyos resultados se encuentran en la primera pestaña, tal y como se puede observar en la imagen.
 - 4- Es un componente gráfico en forma de árbol, donde se renderiza el fichero OWL obtenido de cada resultado de búsqueda. Siguiendo el ejemplo, se trata de la representación del fichero resultante de la búsqueda de "cancer", mostrando los conceptos relacionados (taxonómicos y no taxonómicos) que se han encontrado en forma de árbol.
 - 5- Componente donde se muestra un listado de los conceptos taxonómicos de una búsqueda. En el ejemplo, se trata de los conceptos taxonómicos encontrados en la búsqueda de "cancer".
 - 6- Componente donde se muestra un listado de los conceptos no taxonómicos de una búsqueda. En el ejemplo, se trata de los conceptos no taxonómicos encontrados en la búsqueda de "cancer".
 - 7- De los conceptos taxonómicos encontrados en una búsqueda, aquellos que le interese al usuario pueden ser seleccionados, de forma que pueda profundizar realizando una nueva búsqueda para cada uno de ellos.
 - 8- Igual que en el caso anterior, pero para los conceptos no taxonómicos.
 - 9- Botón que tras ser pulsado, lanza una petición de búsqueda por cada concepto seleccionado que se encuentre en la lista 7 y 8. Cada resultado obtenido, se mostrará en una pestaña nueva. En el ejemplo, se muestra como algunos conceptos taxonómicos (e.g. "breast cancer") han sido seleccionados en el caso de "cancer", para profundizar en ellos. Se ve como los resultados han sido mostrados en nuevas pestañas.
 - 10- Botón que permite guardar el resultado que se muestra en su propia pestaña, en el computador local del usuario (en un fichero OWL).
 - 11- Botón que sirve para eliminar del proyecto el resultado de la búsqueda de un concepto. Se elimina de esta manera la pestaña asociada.

- 12- Muestra a modo de información el número de recursos que tiene el sistema. Un recurso está asociado a una petición de búsqueda. No es una información precisa, sino un modo de indicar al usuario de manera aproximada el nivel de carga que pueda tener el sistema.

4.2.7 Seguridad en el lado del cliente. Firma del applet.

Por defecto, un applet no dispone de los privilegios suficientes para emplear determinadas acciones en el computador sobre el que se ejecuta, debido a que puede comprometer la seguridad del mismo. Un código malicioso podría destrozar el sistema local por completo, o extraer información de carácter confidencial. Las restricciones de seguridad en Java son impuestas por el entorno de ejecución (JVM), por medio de un mecanismo de políticas de seguridad que se aplican a los diferentes recursos que se deben de proteger.

Hemos visto como el applet de esta aplicación se comunica con el servidor `ServerRequest`, mediante un socket. También se ha comentado la posibilidad de guardar un fichero en el computador del cliente desde el mismo applet. Para conseguir estos privilegios (acceder al disco duro y conectar a un socket) ha sido necesario utilizar un mecanismo de seguridad ampliamente utilizado. Dicho mecanismo está basando en la firma digital de un código fuente (en este caso, del applet). Un applet debe ser firmado digitalmente utilizando un certificado digital.

Una firma digital es un resumen (conocido a veces como "hash") de un mensaje mediante un función hash especial, encriptado mediante la clave privada del emisor, que se adjunta junto al texto en claro. Para comprobar la autenticación y la integridad en el destinatario, se debe, por un lado, realizar el mismo resumen del mensaje en claro mediante la misma función de hash, y por otro, desenscriptar la firma mediante la clave pública del emisor, obteniendo otro resumen. Si ambos resúmenes coinciden, resulta que el mensaje se encuentra íntegro y que ha sido firmado realmente por quien creemos que es el emisor (autenticidad y no repudio).

Ahora bien, ¿quién nos garantiza la identidad del emisor?, es decir, ¿cómo sabemos con certeza que el emisor del mensaje es realmente quien dice ser, y no es alguien que intenta hacer una suplantación de identidad?. Para superar este problema surgieron los certificados digitales, que se trata un fichero digital intransferible y no modificable, emitido por una tercera parte de confianza (AC, *Autoridad de Certificación*), que vincula de manera indisoluble a una persona o entidad con una clave pública. En los navegadores, tenemos instalado por defecto certificados de compañías como VeriSign [VeriSign06], que se trata una autoridad certificadora (AC) de prestigio mundial, en la cual tenemos que confiar plenamente. Si llega un certificado firmado por una AC raíz (de plena confianza), el navegador dará por seguro la identidad del firmante. También pueden existir ACs secundarias (intermediarias), de forma que un certificado expedido por ellas, requerirá una comprobación en cadena hasta llegar a una AC raíz. Para aclarar ideas, pondré un ejemplo hipotético: la Universitat Rovira i Virgili (URV) podría tener un certificado digital expedido por la

Generalitat de Catalunya, que hace el papel de AC secundaria, avalada por otra AC raíz como podría ser VeriSign. De esta manera, cuando recibimos un certificado de la URV en una aplicación (por ejemplo, el correo electrónico), el navegador realizaría la verificación en cadena hasta comprobar en última instancia, que el último certificado fue expedido por una AC de su confianza (AC raíz), como en este caso la AC VeriSign, dándolo por válido. En un certificado pueden estar implicadas en cadena cero o más AC intermediarias.

En un certificado basado en el estándar X.509, encontramos los siguientes campos:

- Identificación del titular del certificado: Nombre, dirección, etc.
- Clave pública del titular del certificado.
- Fecha de validez.
- Número de serie.
- Identificación del emisor del certificado.

La AC que expide un certificado, se encarga de firmarlo digitalmente con su clave privada, garantizando así la validez del mismo.

El código del applet ha sido firmado mediante una herramienta que viene con el propio j2sdk de java, llamado **jarsigner**. Y para los certificados se ha utilizado la herramienta **keytool**, que también viene junto a la herramienta jarsigner. Como no podemos comprar un certificado a una entidad certificadora de confianza mundial como VeriSign (mantener el certificado con validez para un año, actualmente puede costar 895 euros (hay de diferentes clases) excluyendo el IVA), se ha tenido que hacer a la vez el papel de entidad certificadora. Es cierto que carece de sentido el hecho de decir a un usuario que se descarga el applet, que una entidad certificadora que resulta ser yo (y que por lo tanto no es de fiar, ya que estoy haciendo el papel de AC raíz), garantiza que el que ha firmado el código es quien dice ser, siendo yo mismo. De ahí viene la advertencia que hace el navegador en el momento de recibir el applet (véase la *figura 23* y la *Figura 24*), pero esta es la única opción gratuita que existe.



Figura 23. Aviso del navegador Firefox sobre la firma digital del applet. Indica que es inválida ya que no me reconoce como entidad certificadora de confianza

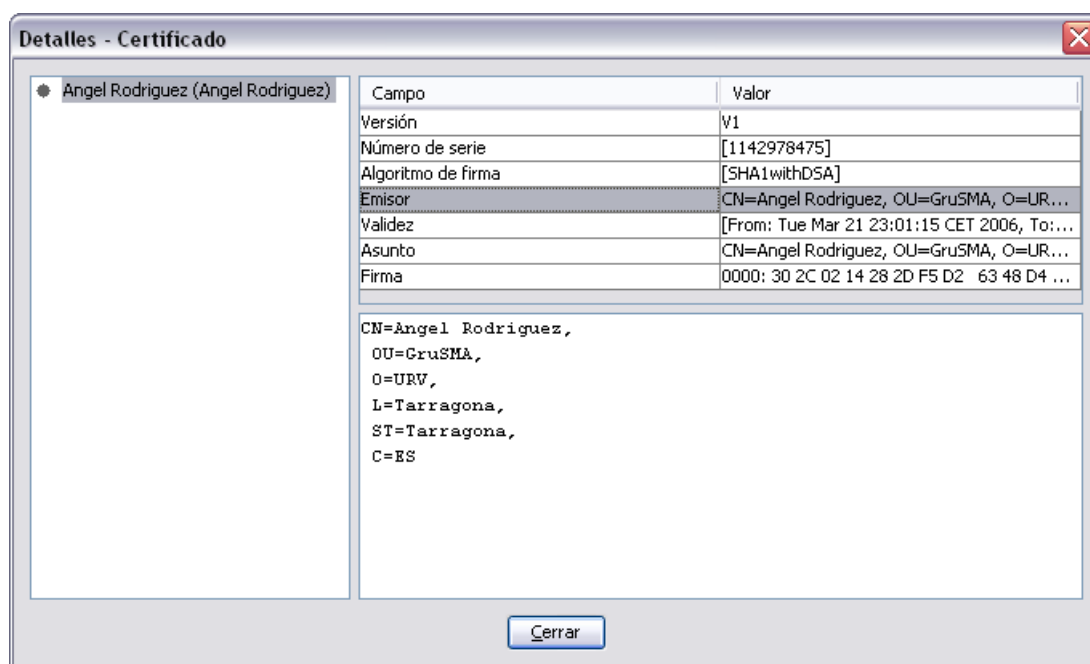


Figura 24. Podemos consultar los detalles del certificado emitido.

5. Evaluación/Resultados

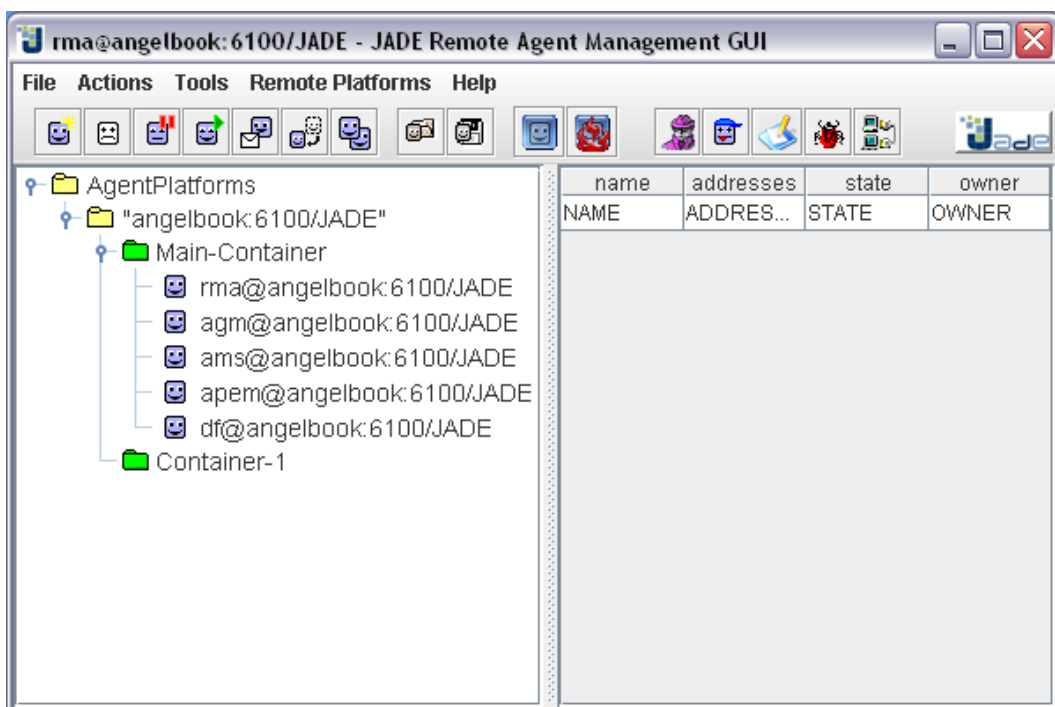
Ahora llega el capítulo de la pruebas finales. Se mostrarán 3 tipos de pruebas que evalúan diferentes aspectos. En el apartado 5.1 se muestra el funcionamiento general de la aplicación final ejecutándose sobre el sistema dinámico. En el apartado 5.2 tocará el turno de las pruebas de rendimiento, haciendo hincapié en la mejora de rendimiento que supone la disponibilidad de diversos nodos por la ejecución paralela de las tareas. Finalmente, en el apartado 5.3 se mostrará los resultados obtenidos de las pruebas de robustez del sistema, poniendo de manifiesto el grado del mismo.

5.1 Pruebas de funcionamiento

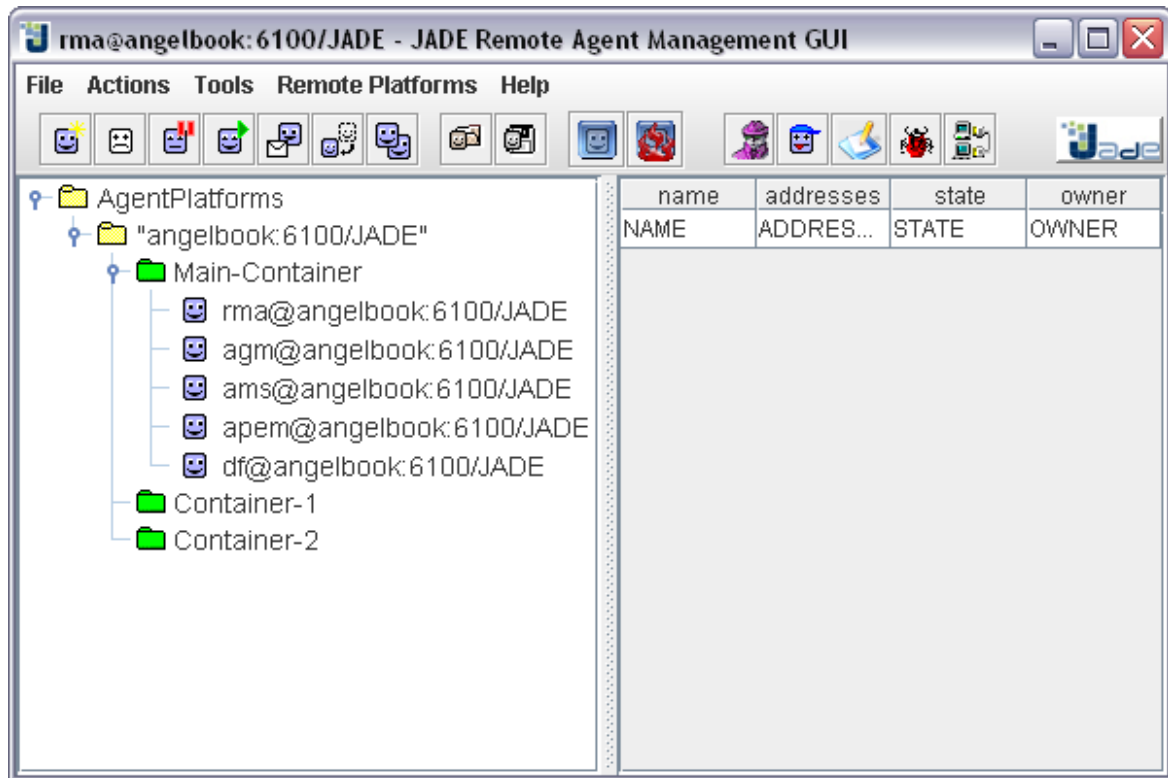
En este apartado no se pretende evaluar el rendimiento del sistema ni tampoco mostrar al detalle todas las opciones para los usuarios. Simplemente, se pretende enseñar el funcionamiento de la aplicación de forma general.

A continuación se mostrará como se enciende la plataforma, cómo se da de alta un nodo, y finalmente, como un usuario se registra en el sistema, para luego crear un proyecto y construir una Ontología.

El servidor del sistema se pone en marcha, y el agente RMA nos visualiza el estado de la plataforma de la siguiente manera:



A continuación se da de alta un nodo al sistema proporcionando 6 slots, provocando que el agente RMA nos muestre en su ventana gráfica un nuevo contenedor llamado Container-2.



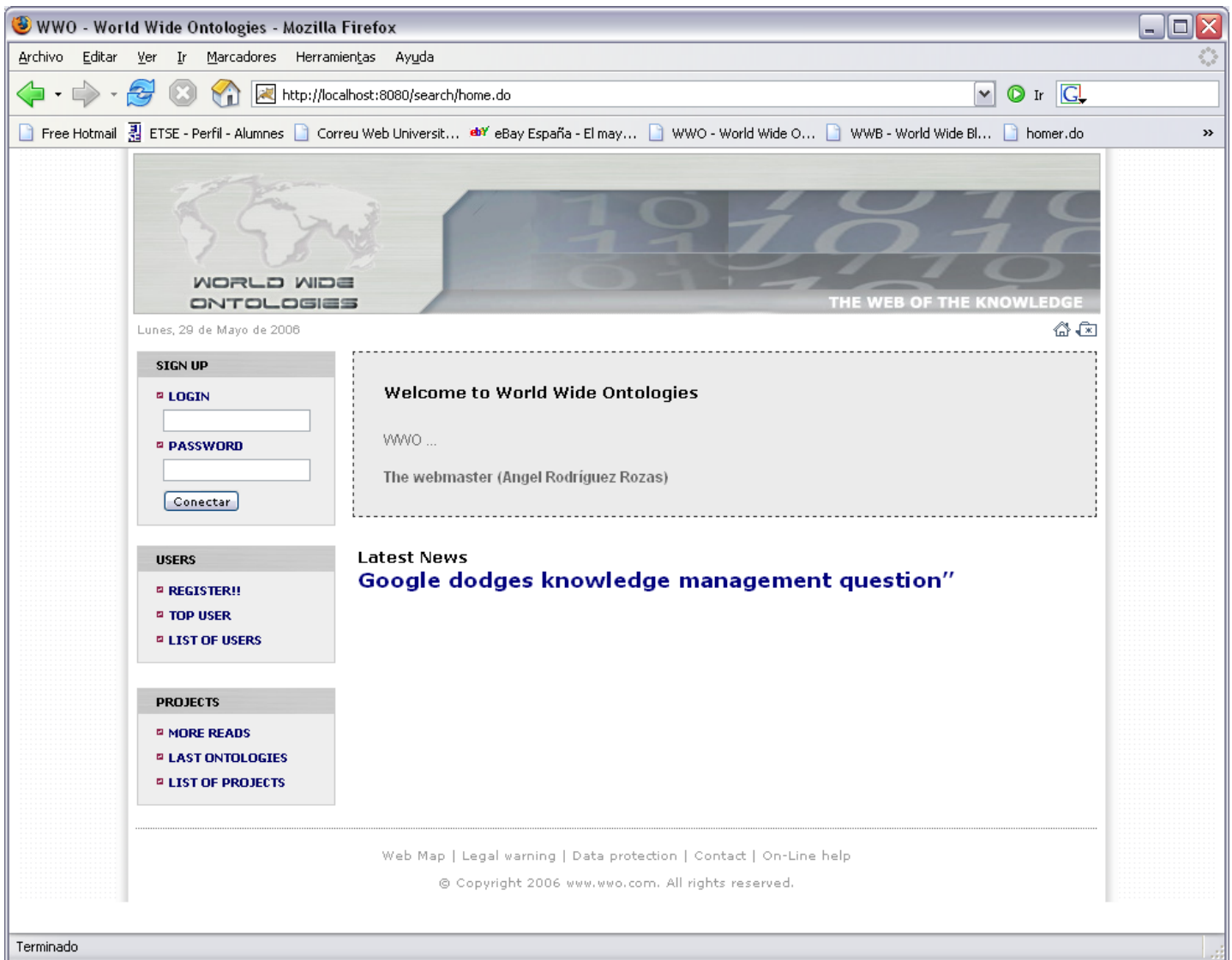
El sistema está en marcha y listo para atender solicitudes. Ahora se pone en marcha el servidor Tomcat, el cual nos muestra la siguiente información de arranque:

```
29-may-2006 10:33:15 org.apache.coyote.http11.Http11AprProtocol init
INFO: Inicializando Coyote HTTP/1.1 en puerto http-8080
29-may-2006 10:33:15 org.apache.coyote.ajp.AjpAprProtocol init
INFO: Initializing Coyote AJP/1.3 on ajp-8009
29-may-2006 10:33:15 org.apache.catalina.startup.Catalina load
INFO: Initialization processed in 1542 ms
29-may-2006 10:33:16 org.apache.catalina.core.StandardService start
INFO: Arrancando servicio Catalina
29-may-2006 10:33:16 org.apache.catalina.core.StandardEngine start
INFO: Starting Servlet Engine: Apache Tomcat/5.5.15
29-may-2006 10:33:16 org.apache.catalina.core.StandardHost start
...
INFO: Arrancando Coyote HTTP/1.1 en puerto http-8080
29-may-2006 10:33:21 org.apache.coyote.ajp.AjpAprProtocol start
INFO: Starting Coyote AJP/1.3 on ajp-8009
29-may-2006 10:33:21 org.apache.catalina.storeconfig.StoreLoader load
INFO: Find registry server-registry.xml at classpath resource
```

```
29-may-2006 10:33:21 org.apache.catalina.startup.Catalina start
INFO: Server startup in 5768 ms
```

La aplicación Web se encuentra desplegada en el espacio correspondiente del servidor Tomcat, el cual se encuentra en funcionamiento. Ya está todo el sistema al completo en funcionamiento.

Ahora interviene el usuario final. Éste abre un navegador, y escribe la URL de la página Web principal. Tendremos una vista similar a la siguiente:



Si no estamos registrados, procedemos a darnos de alta en el sistema, a partir de la opción "REGISTER" del menú "USERS". Deberemos completar un formulario como el siguiente:

WWO - World Wide Ontologies - Mozilla Firefox

Archivo Editar Ver Ir Marcadores Herramientas Ayuda

http://localhost:8080/search/register.do

Free Hotmail ETSE - Perfil - Alumnos Correu Web Universit... eBay España - El may... WWO - World Wide O... WWO - World Wide Bl... homer.do

WORLD WIDE ONTOLOGIES THE WEB OF THE KNOWLEDGE

Lunes, 29 de Mayo de 2006

SIGN UP

☐ LOGIN

☐ PASSWORD

USERS

☐ REGISTER!!

☐ TOP USER

☐ LIST OF USERS

PROJECTS

☐ MORE READS

☐ LAST ONTOLOGIES

☐ LIST OF PROJECTS

Register data

☐ Login

☐ Password

☐ Name

☐ Last name

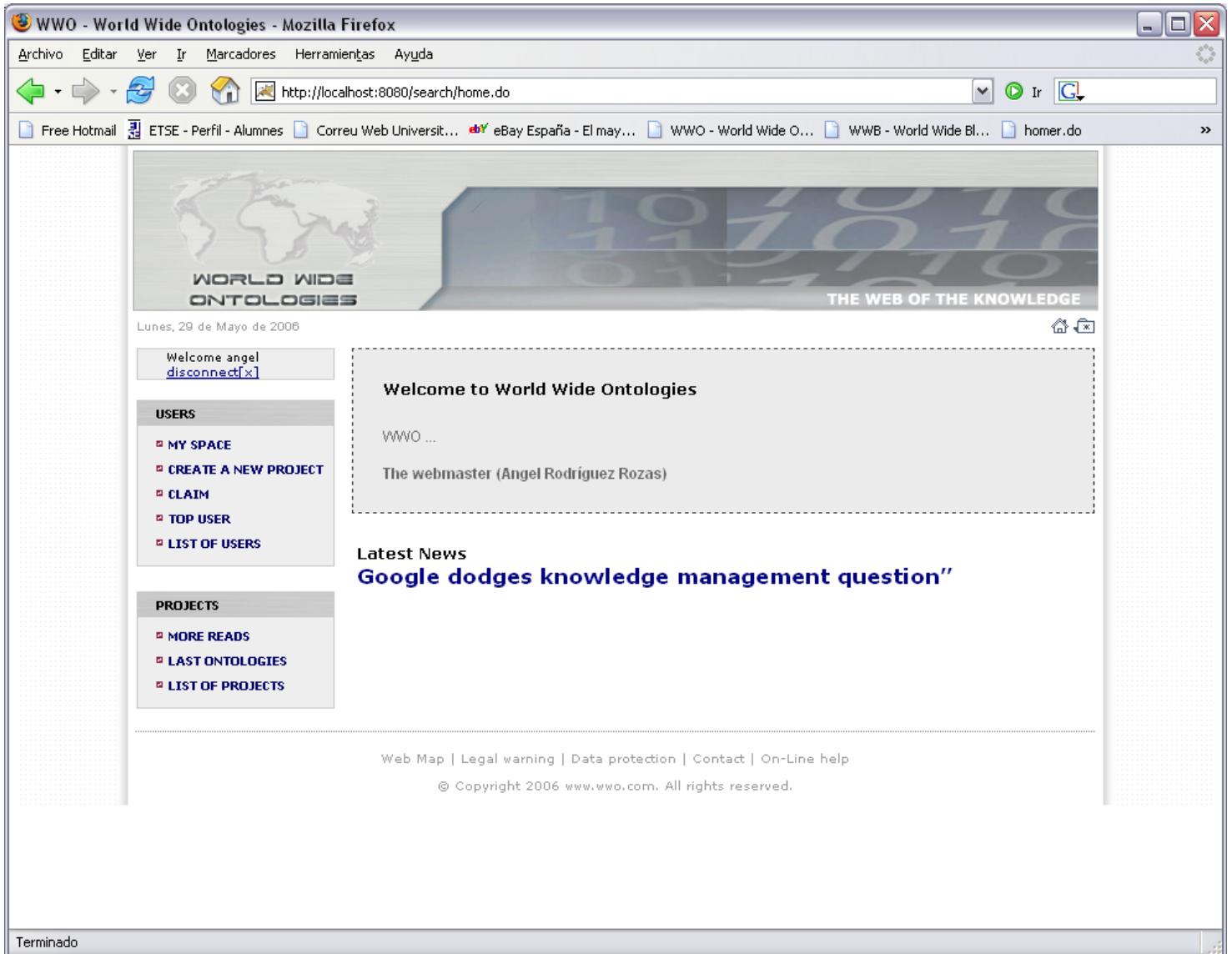
☐ Email

Protection data

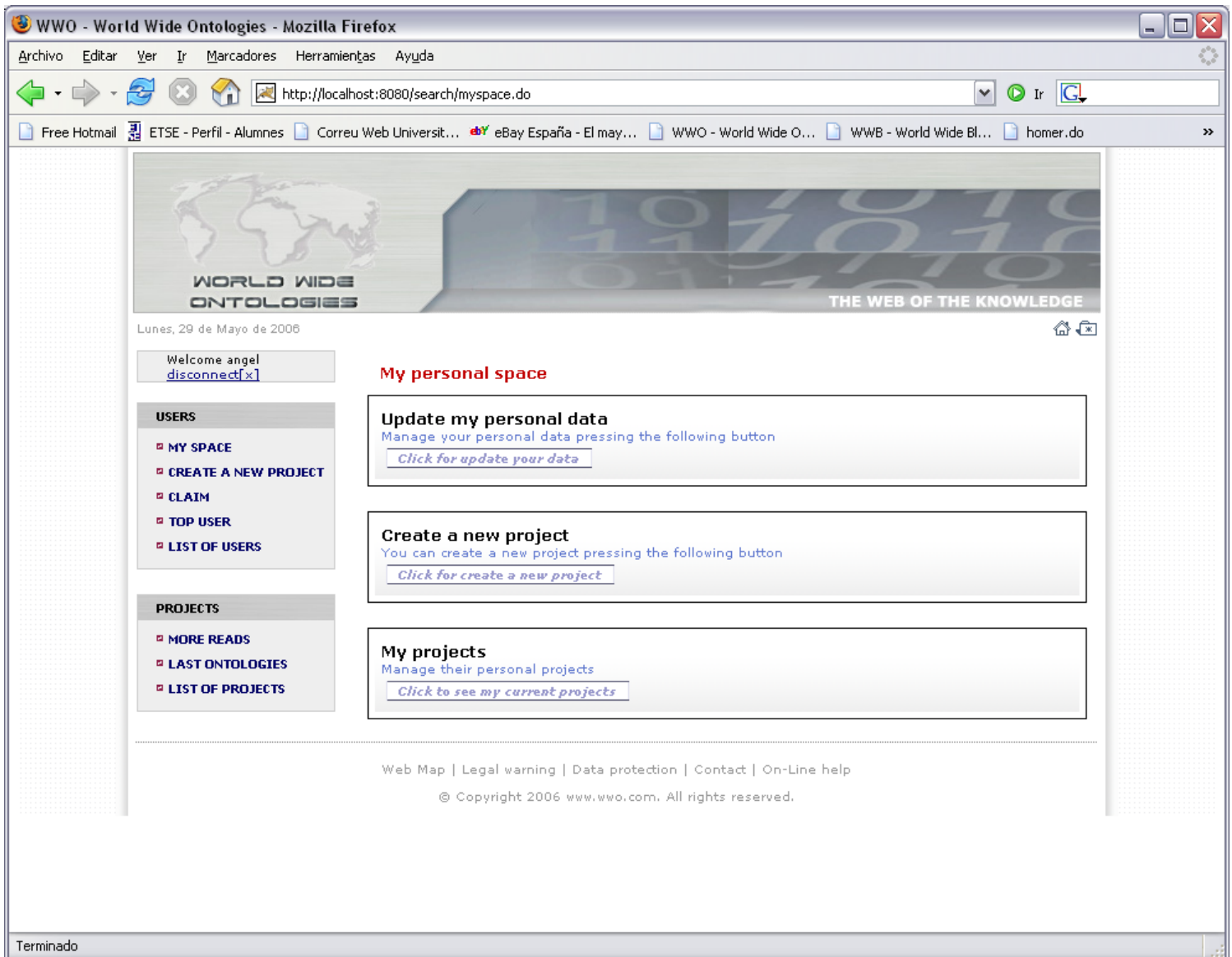
1.1.- Información al usuario de la existencia de fichero y solicitud del consentimiento para el tratamiento automatizado de los datos: En relación con los datos de carácter personal facilitados por el USUARIO en los formularios que puedan existir en el sitio web, EL PROPIETARIO cumple estrictamente la normativa vigente establecida en la Ley 15/1999 de Protección de datos de carácter personal y demás legislación que la desarrolla e informa al USUARIO que los referidos datos serán incluidos dentro de un fichero para su tratamiento automatizado prestando el USUARIO consentimiento mediante la aceptación de estas condiciones generales a dicho

Terminado

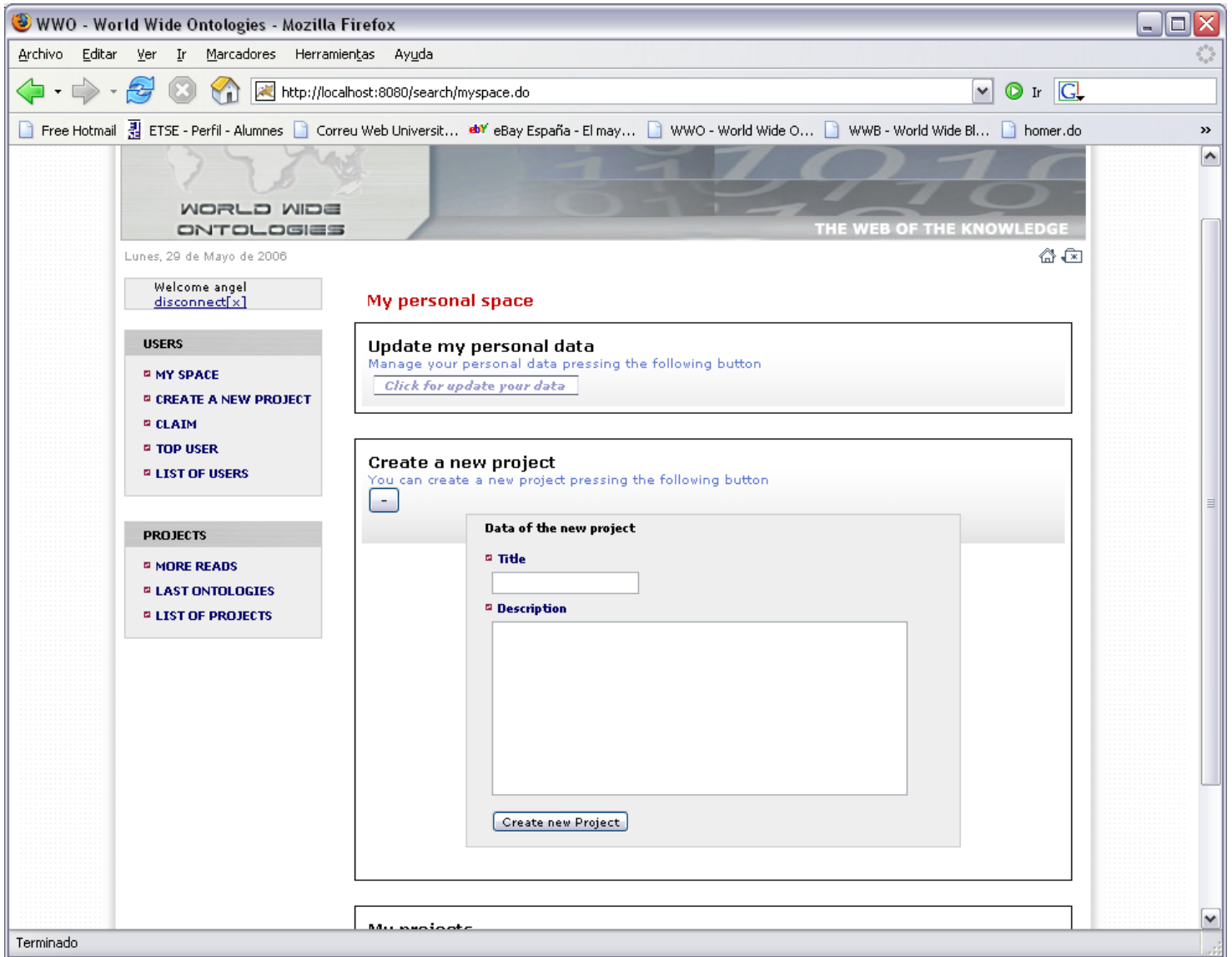
Tras confirmarnos el alta al sistema si todo está correcto, a continuación ya podemos validarnos como usuario registrado desde el formulario "SIGN UP" del menú de la izquierda, introduciendo el "LOGIN" y el "PASSWORD". Si nos validamos correctamente, el sistema nos reconoce, y el menú de usuarios presenta nuevas opciones correspondientes a los usuarios registrados:



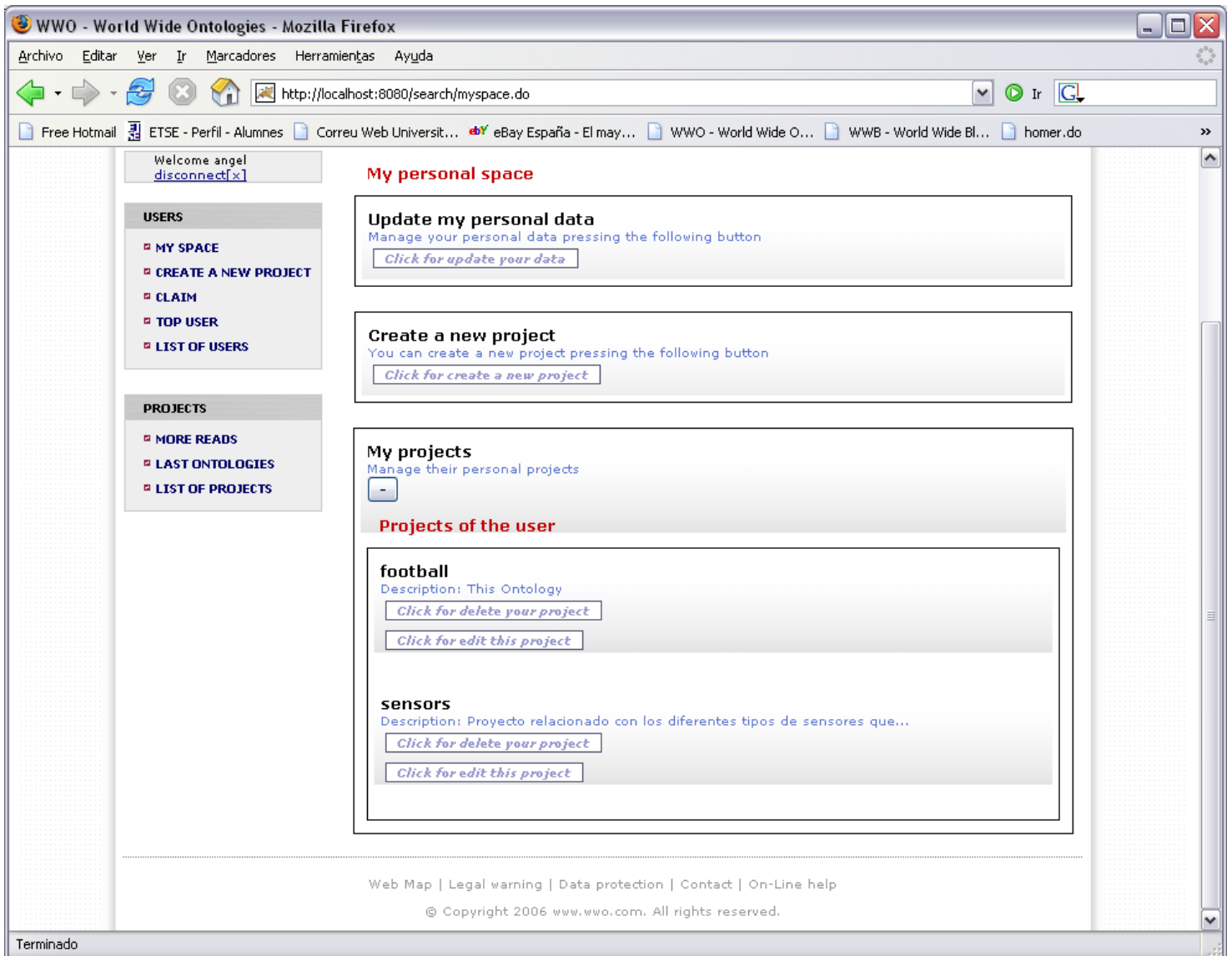
La primera opción del menú de usuarios llamada "MY SPACE" nos llevará a nuestro espacio personal, donde podremos gestionar nuestros datos de usuario, así como darnos de baja en caso necesario. También nos permite gestionar nuestros proyectos personales, de forma que podemos crear nuevos, eliminarlos, o entrar para editarlos (trabajar con ellos).



Para crear un nuevo proyecto, pulsamos sobre el botón "Click for create a new project" del apartado "Create a new project".

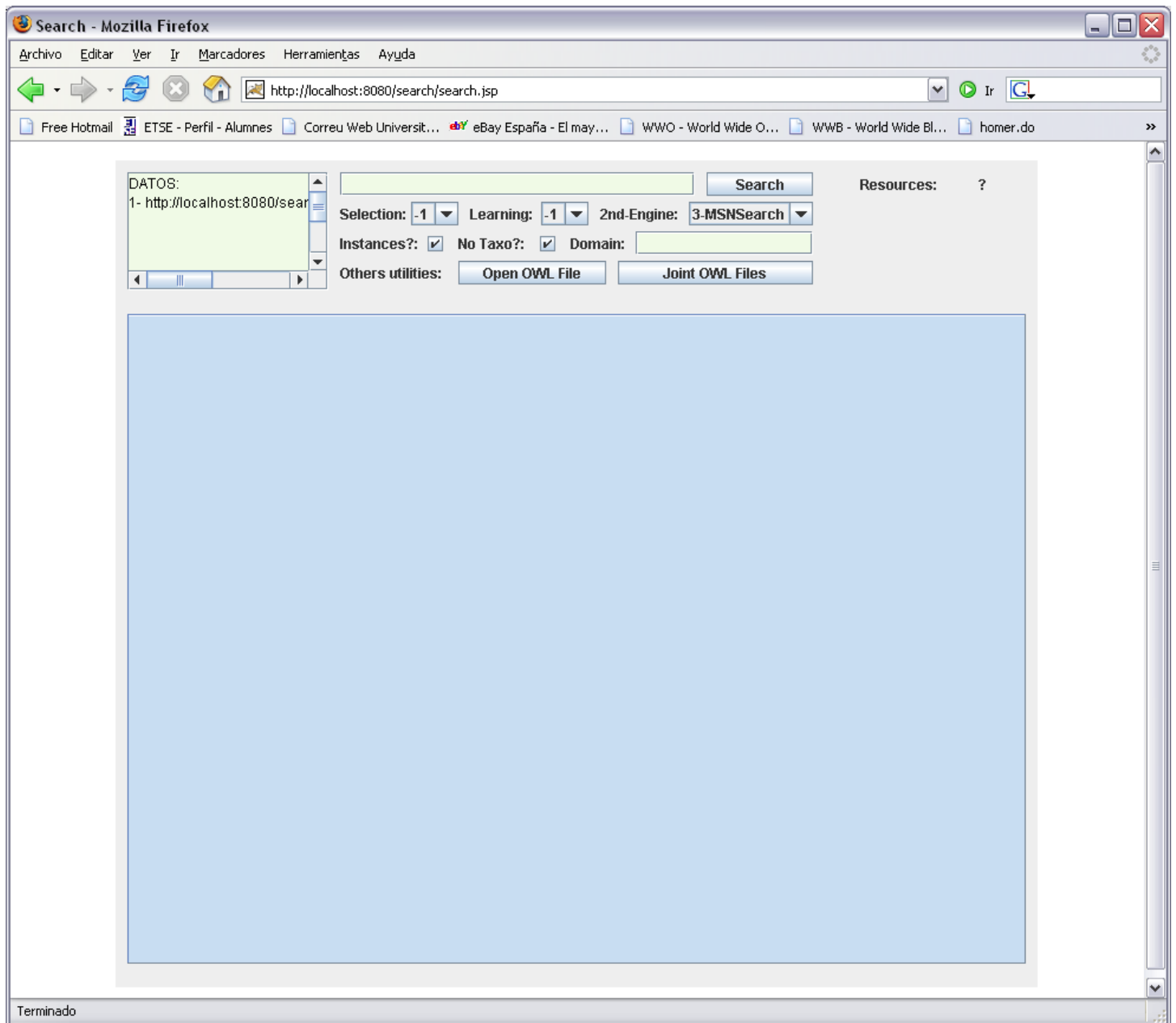


Rellenamos el formulario, escribiendo el título del proyecto y su descripción en los campos correspondientes. Una vez creado el proyecto ya podemos editarlo. Para ello, dentro de nuestro espacio personal, vamos al apartado llamado "MyProjects" y pulsamos el botón "See your projects". Al pulsarlo se despliega un listado donde aparecen todos los proyectos que tenemos creados en ese momento. Por cada uno, se nos muestra el título, la descripción, y dos botones llamados "Click for delete this project" y "Click for edit this project", que nos permiten eliminarlo o editarlo respectivamente.

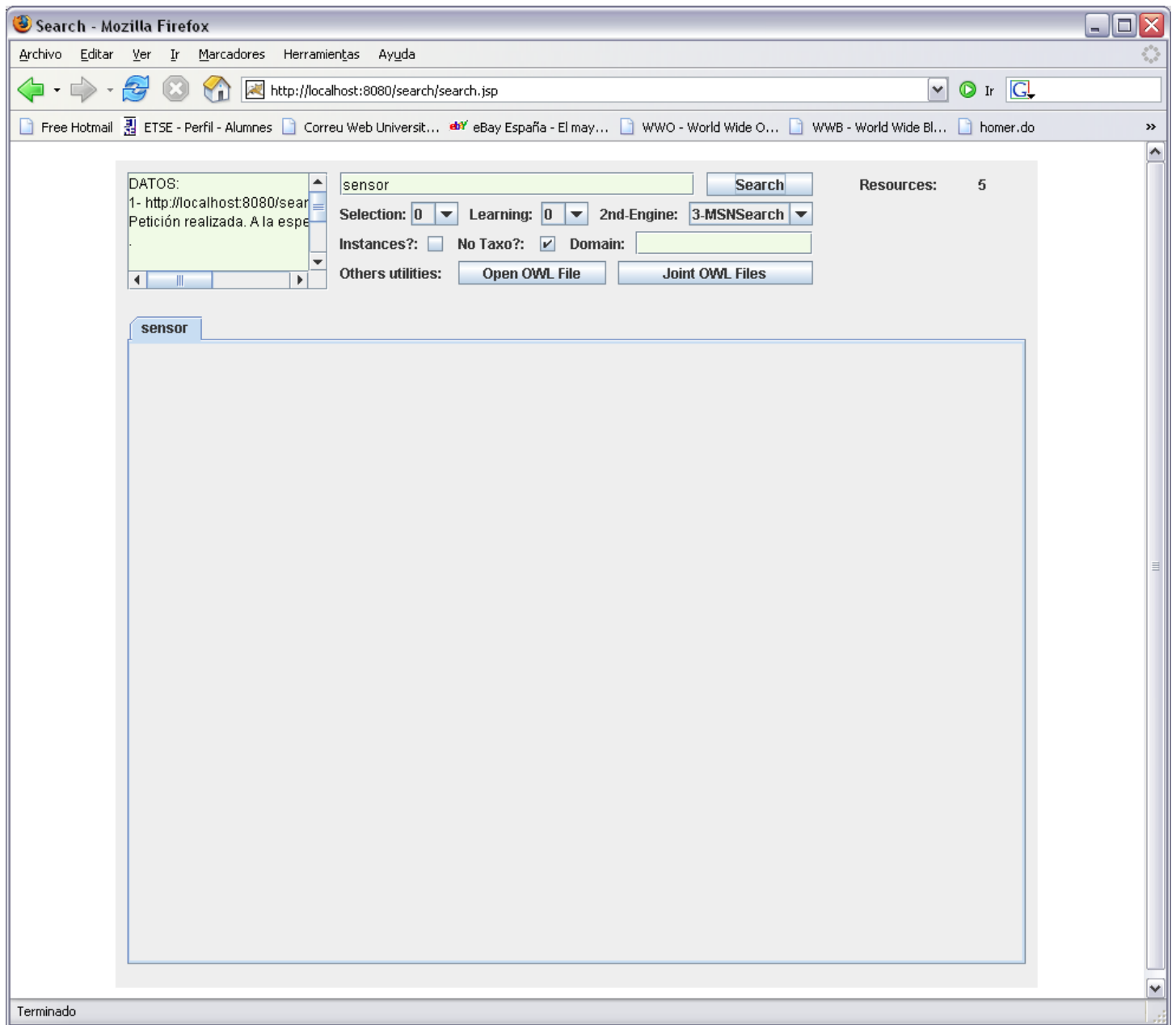


Para editar un proyecto, pulsamos el botón "Click for edit this project" correspondiente.

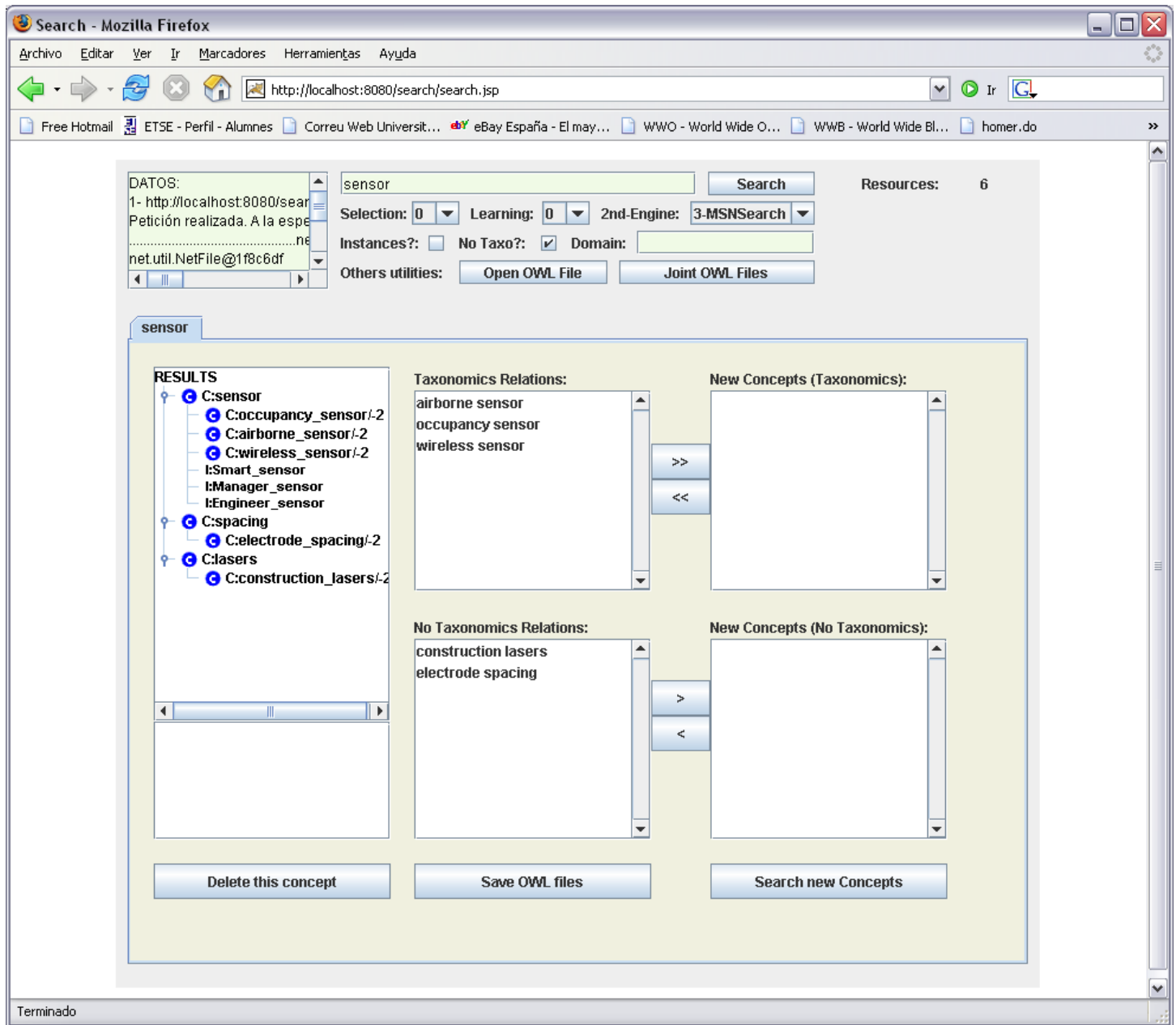
Ahora entraremos en el applet para trabajar con él. Deberemos de aceptar dos mensajes relacionados con los certificados digitales. A continuación se muestra el espacio de trabajo que se describió el capítulo 4.



De entrada el proyecto está vacío ya que acabamos de crearlo. Los resultados se irán mostrando en la zona azul a medida que se vayan obteniendo, tras realizar las peticiones de búsqueda. Estamos editando el proyecto titulado "sensor". Para este ejemplo, pretendemos construir una ontología basada en el dominio de los sensores. Queremos conocer que tipos de sensores existe y como están relacionados. Para ello, empezamos buscando "sensor", el concepto más genérico del dominio. Como es a modo de prueba, escogeremos los parámetros de tal manera que el proceso de búsqueda no sea muy largo.

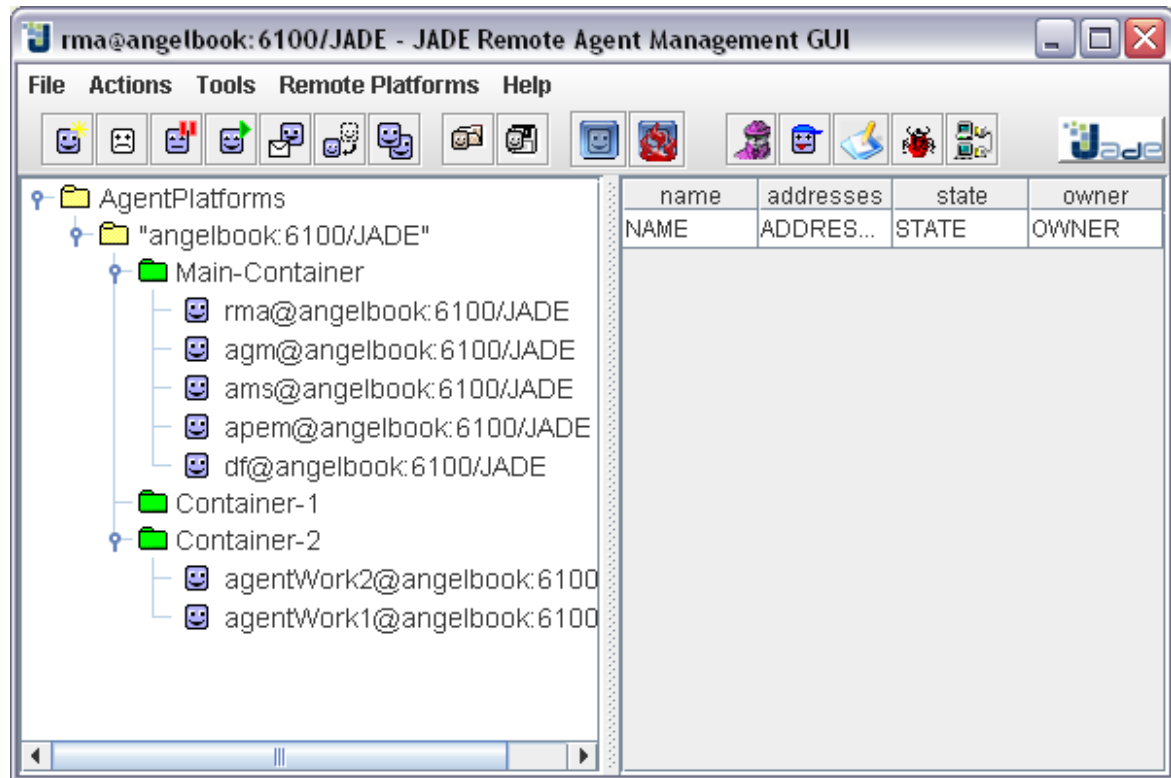


El sistema pone en marcha la búsqueda y nos avisa que está en espera para recibir los resultados. Ahora mismo el usuario podría cerrar incluso el computador, de forma que cuando vuelva, si la búsqueda ha finalizado, se encontraría los resultados mostrados. Evidentemente, también puede tener la aplicación en marcha, y esperar a que estos lleguen. Como se ve en la parte superior derecha de la anterior imagen, el sistema nos indica el número de recursos que hay disponibles en ese momento.

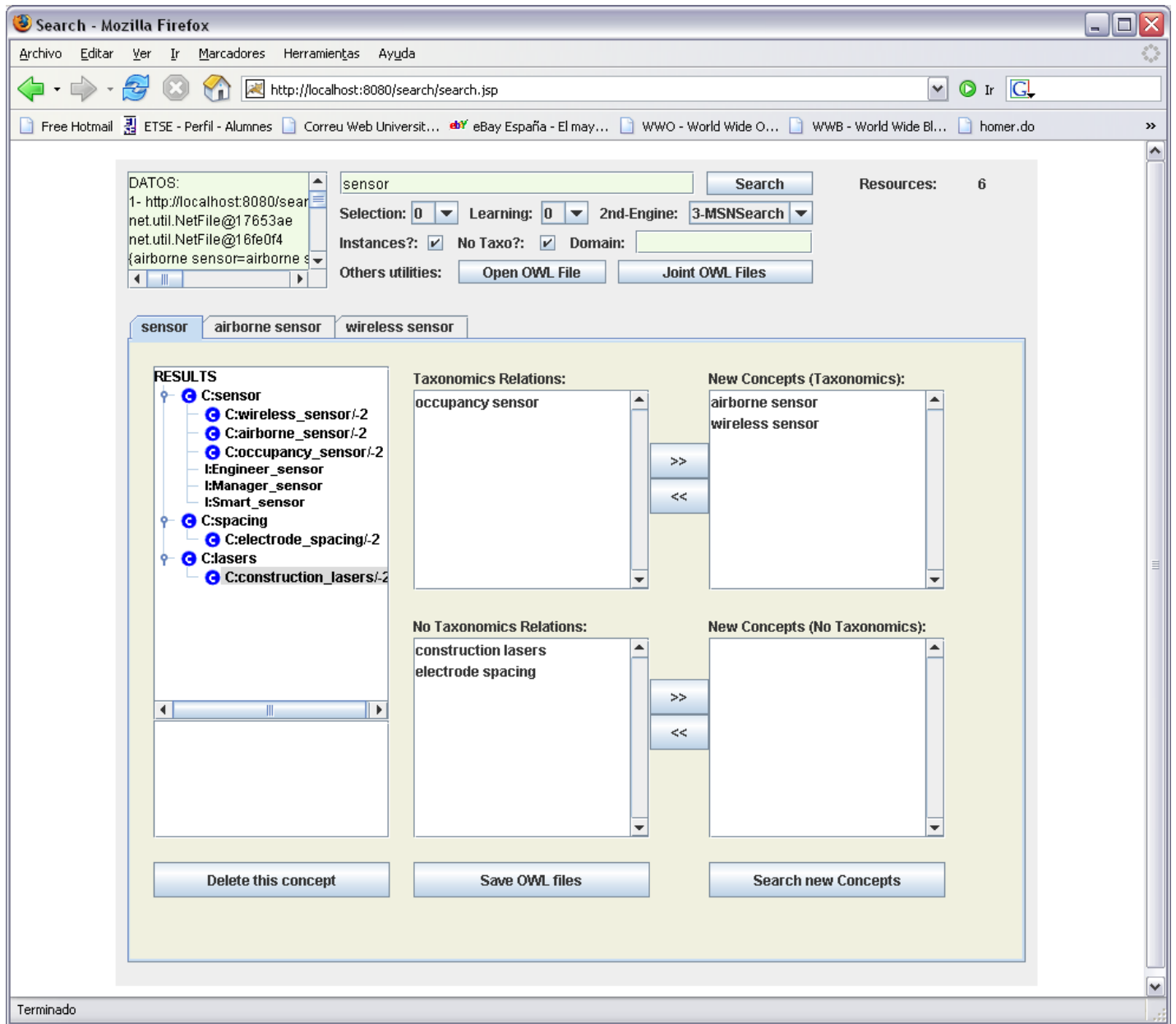


Una vez completada la búsqueda por el sistema, los resultados se muestran en el applet. Como vemos, tenemos la pestaña llamada "sensor" donde son visualizados. En la parte izquierda del panel, tenemos los conceptos relacionados con sensor en forma de árbol. Los conceptos que aparecen en el árbol cuyo concepto raíz es "sensor" son los conceptos relacionados de forma taxonómica con "sensor". En cambio, los que aparecen en otros árboles, son conceptos relacionados de forma no taxonómica. Los mismos conceptos los encontramos en dos listados en la parte central ("Taxonomic Relations" y "No Taxonomic Relations"), que podemos seleccionarlos para profundizar en ellos, realizando nuevas búsquedas. Con los botones ">>" podemos mover los conceptos seleccionados a las listas de la derecha, para pedir al sistema que realice las búsquedas que pretendemos, pulsando el botón "Search new Concepts". Para este

ejemplo, profundizaremos con los conceptos "airbone sensor" y "wireless sensor", de los taxonómicos.

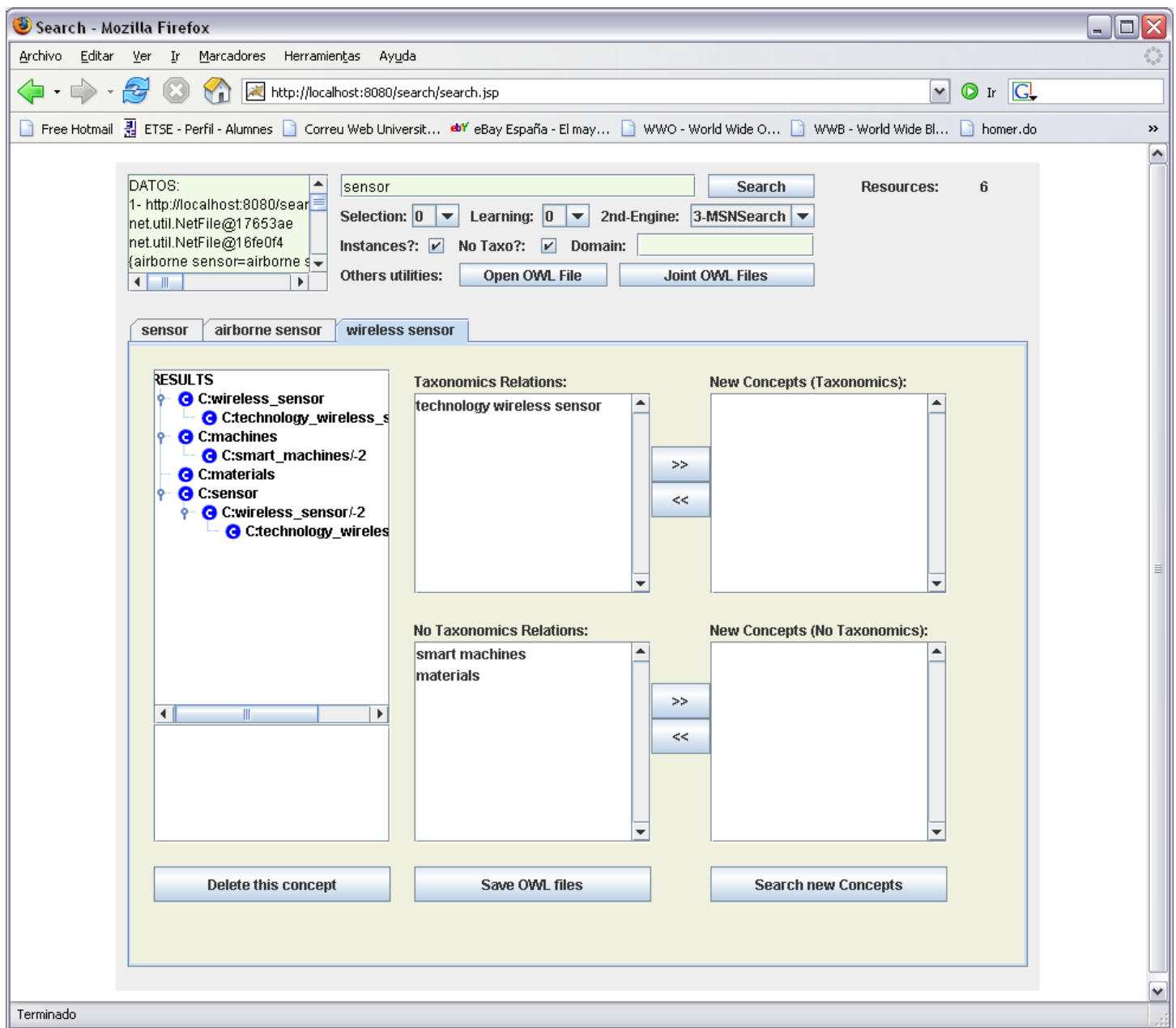


Vemos como 2 agentes han sido movidos por el planificador, al único nodo existente en la plataforma en este ejemplo, para realizar el trabajo de búsqueda (uno por cada concepto).

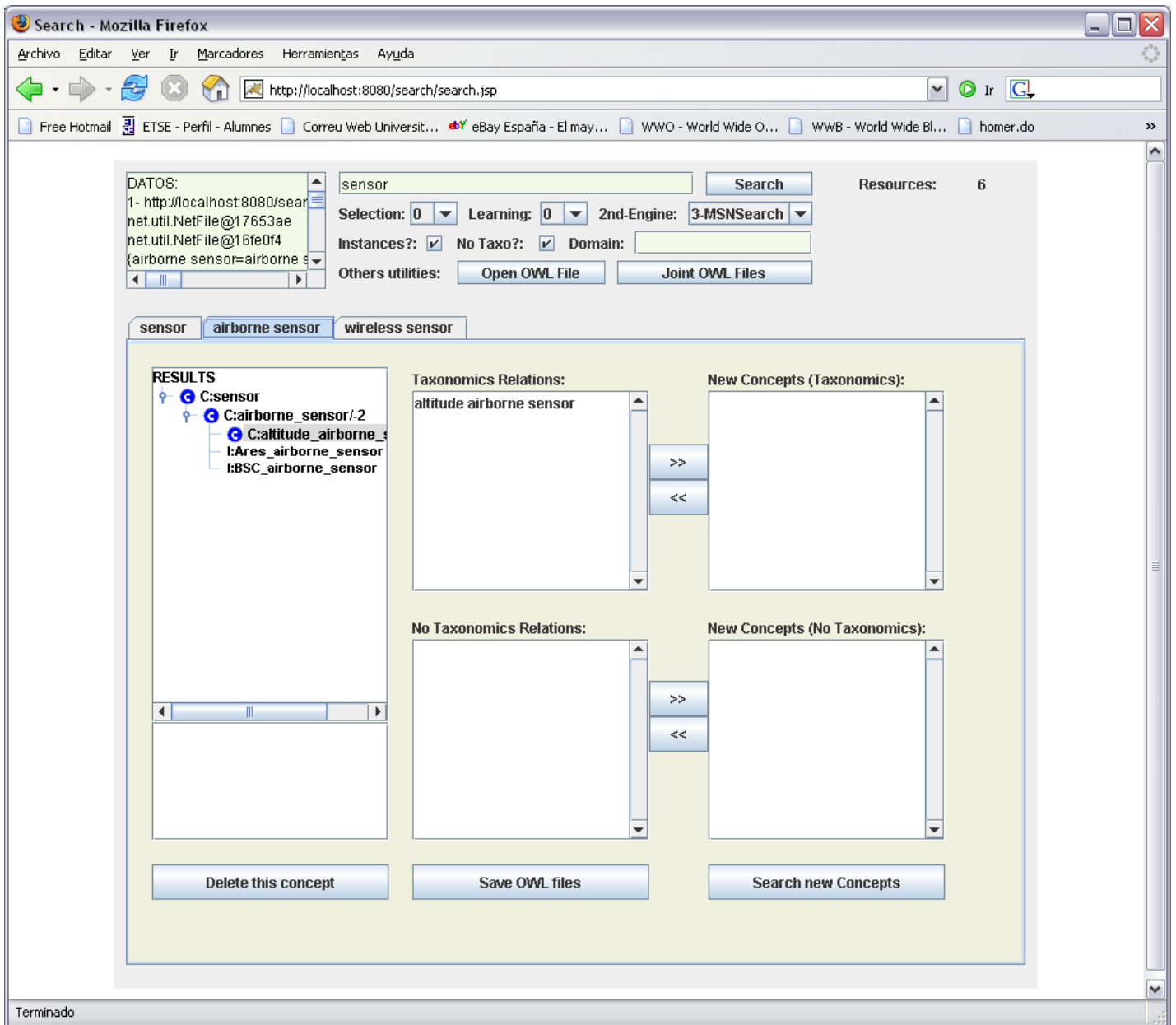


El usuario espera ahora a que se realice la búsqueda para que se le muestren los resultados.

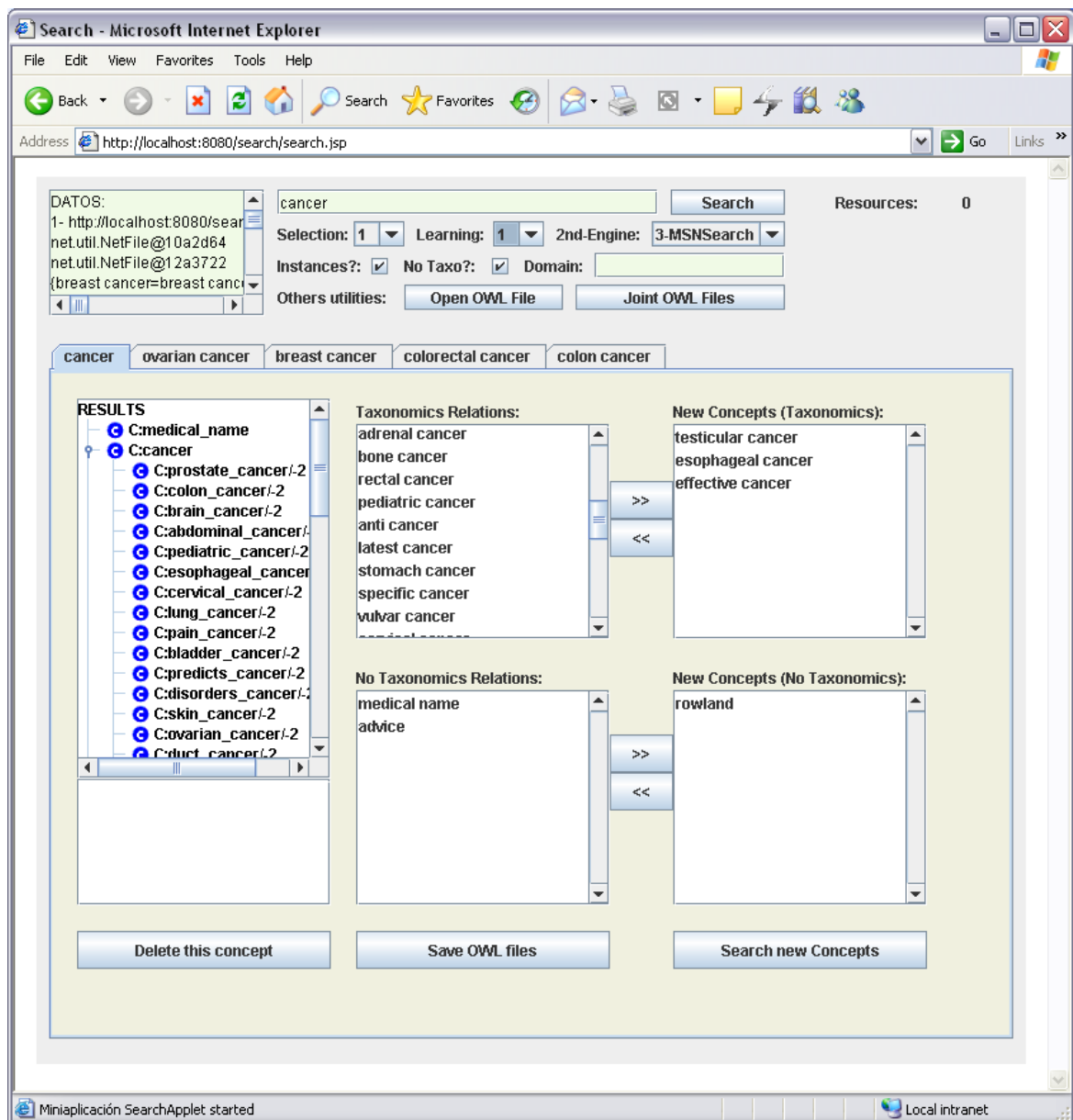
Como vemos en la anterior imagen, ya los hemos pasado a las listas de la derecha, mediante los botones ">>" correspondientes. Si nos hemos equivocado pasando un concepto que no pretendemos buscar, podemos rectificar pasándolo de nuevo al listado de la izquierda, con los botones "<<" respectivos. Una vez seleccionados los que pretendíamos, ya podemos solicitar la búsqueda al sistema pulsando el botón "Search new Concepts". Para las nuevas búsquedas se utilizarán los parámetros indicados en ese momento, que pueden ser ajustados de manera diferente respecto a búsquedas anteriores.



Los resultados son mostrados cada uno en su propia pestaña. En la imagen anterior, vemos los resultados correspondientes a la búsqueda de "wireless sensor", que como vemos, nos ha vuelto a generar nuevos conceptos, relacionados taxonómicamente y no taxonómicamente con el concepto "wireless sensor".

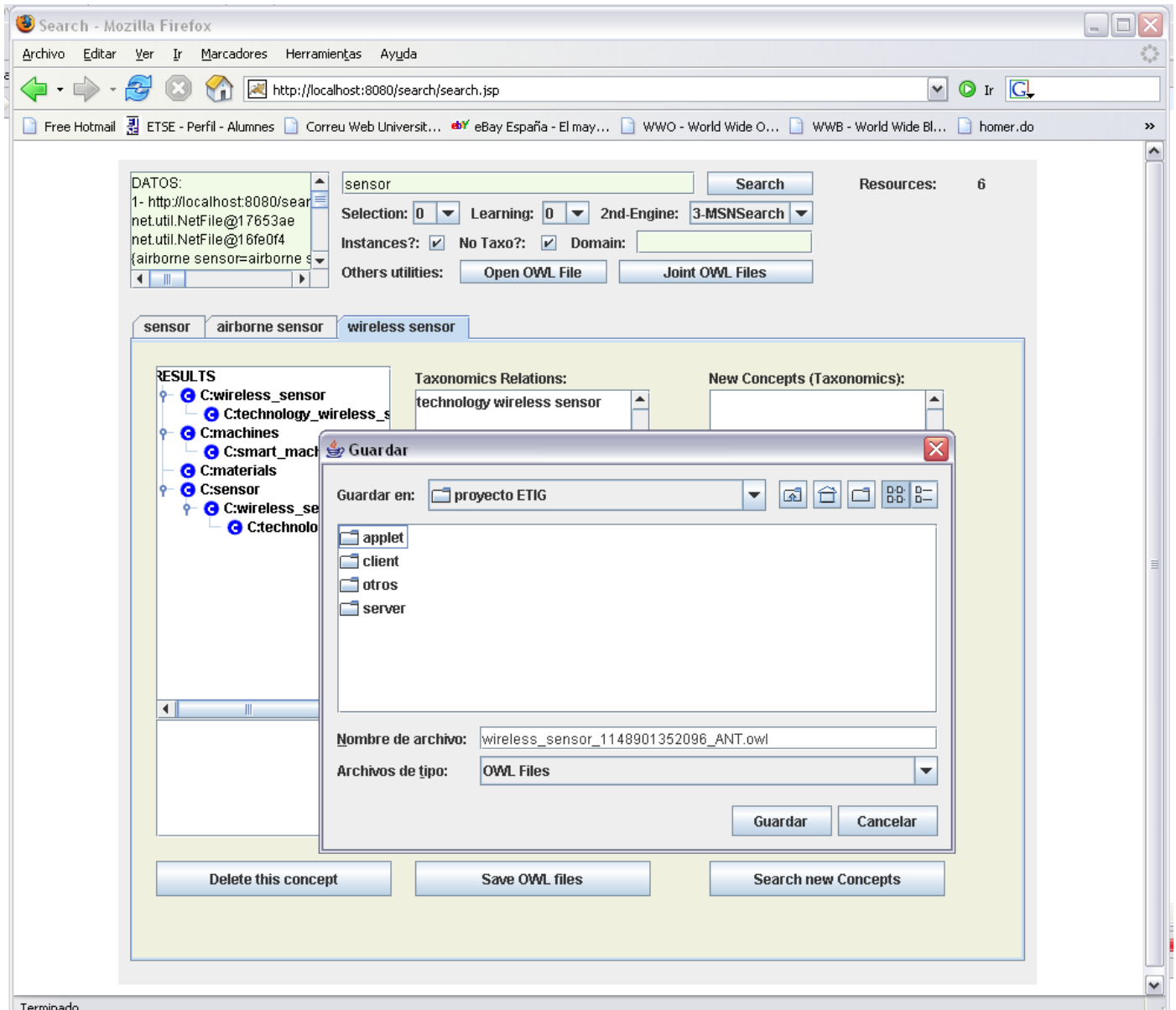


De forma similar, también vemos los resultados de la búsqueda del concepto "airbone sensor". Se vuelve a recordar que vemos pocos resultados debido a que estamos haciendo pruebas mínimas, con los parámetros "Selection" y "Learning" a nivel 0 (el menor posible). Con mayores niveles de estos parámetros, simplemente obtendríamos listados más largos de conceptos.

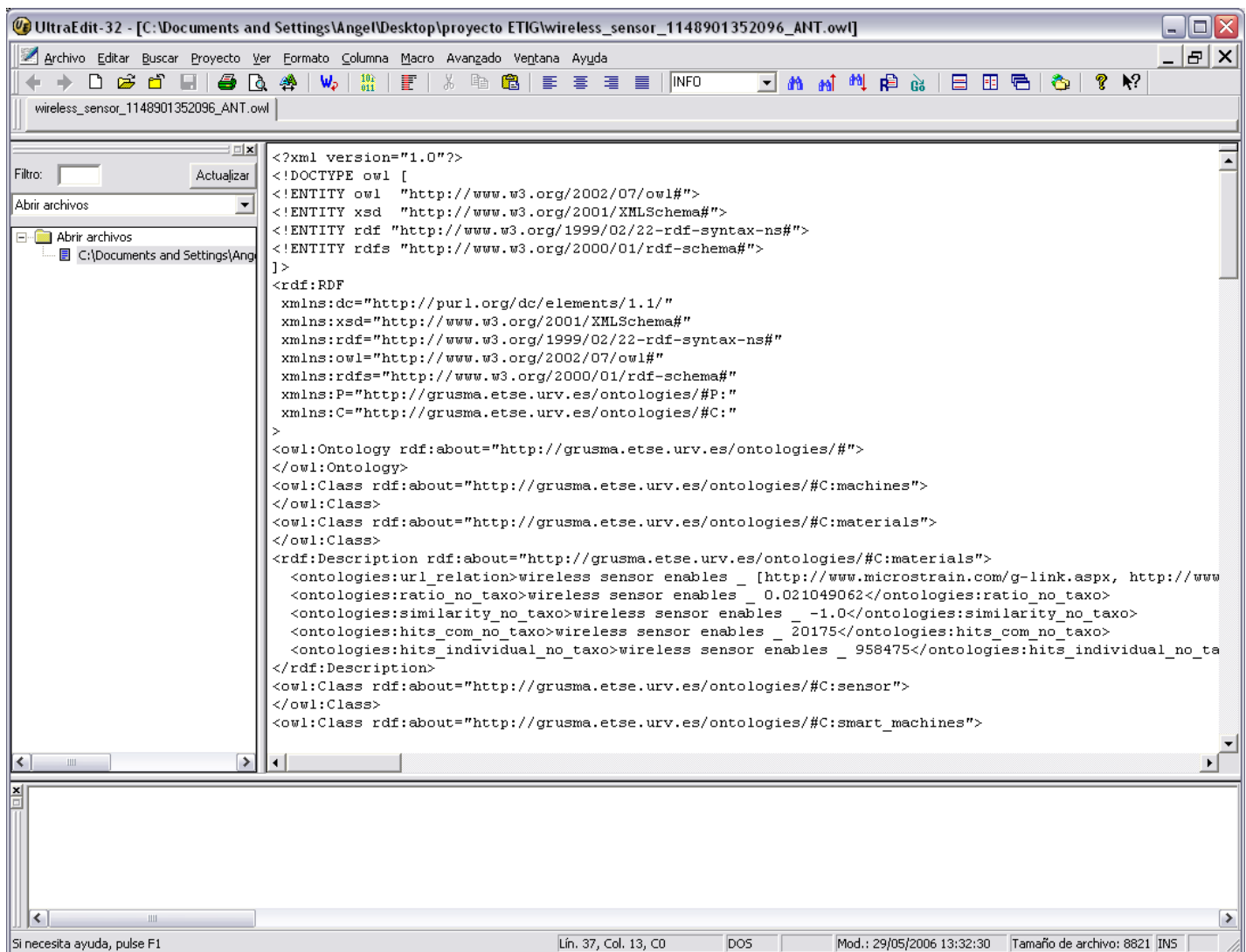


Aquí vemos un ejemplo de una prueba más seria, que nos proporcionó muchos más resultados. Como se ha comentado, el volumen de los resultados depende de los parámetros que ajuste el usuario.

Podríamos seguir realizando búsquedas sobre nuevos conceptos. También podemos eliminar una pestaña mediante el botón "Delete this concept" que se encuentra en la parte inferior izquierda de cada pestaña. Finalmente, podemos guardar un resultado de una pestaña en local, en un fichero en formato OWL.



Elegimos el directorio donde queremos almacenar el fichero. Por defecto la aplicación le da un nombre del estilo del que vemos en la imagen anterior, pero podemos cambiarlo a nuestro gusto.



Si abrimos el fichero guardado con un editor de textos, veremos su contenido en formato "xml" basado en OWL, que es el lenguaje de marcado para la publicación de Ontologías en la Web, con un aspecto similar al mostrado en la imagen anterior.

Este fichero podemos abrirlo con cualquier otra aplicación que soporte dicho formato.

Nota: cada concepto de los que se buscan dentro de cada proyecto, tienen asociados direcciones de páginas Web para poder navegar por ellas, encontrando información relacionada. En el ejemplo de este guía, no se han mostrado debido a que hemos utilizado una versión del componente taxoBuilder especial para las realizar pruebas. En la versión final del proyecto se dispone de la versión completa y por tanto los enlaces a la Web estarán disponibles.

5.2 Pruebas de rendimiento

Ahora se van a mostrar los resultados conseguidos tras realizar una serie de pruebas relacionadas con el rendimiento del sistema.

Las pruebas se basarán en comparar, con diversas configuraciones, cuál es el tiempo de respuesta promedio de la ejecución de una serie de tareas a la vez. Hemos decidido poner en ejecución 4 tareas. Las configuraciones serán 3, y varían según el número de nodos que utilizamos:

- Con 1 nodo
- Con 2 nodos
- Con 4 nodos

Cada nodo corresponde a una máquina. Sus características técnicas son las siguientes:

- Misma conexión a Internet (compartida para toda la URV).
- CPUs con potencia similar.
- Cantidad de memoria RAM diferente (dos portátiles con 512 Mbytes, otro portátil con 1 Gbyte, y un PC con 2 Gbytes).

En cuanto a las tareas, tendremos de 2 tipos:

- (a) tareas orientadas a consumo de cpu: son tareas idénticas que hacen un empleo intensivo de la cpu durante su ejecución, con pocas operaciones de entrada/salida (que en todo caso se realizan por consola). Corresponde al proceso de inicialización de cada instancia del TaxoBuilder (sin empezar a realizar ninguna búsqueda), donde realiza unas comprobaciones y carga en memoria las librerías que va a utilizar.
- (b) tareas de la aplicación de este proyecto: ya hemos visto que son tareas que tienen "picos" de consumo de cpu, ya que realizan muchísimas operaciones de entrada y salida a la Web. Estas tareas necesitan conexión a Internet y 128 Mbytes de memoria RAM. Además las 4 no serán idénticas, aunque sí que serán las mismas para todas las configuraciones.

Primero mostraremos los resultados de las **pruebas de rendimiento con las tareas de tipo (a)**. Se han creado cuatro instancias del TaxoBuilder que sólo se inicializan (sin buscar nada, sólo realiza operaciones que hacen un uso intensivo de la cpu). Resultados (visualizarlos a la vez con la *figura 25*):

- con 1 nodo (el PC, que es el más potente, realiza las 4 tareas): 16,953 segundos

- con 2 nodos (portatil con 512 Mbytes de RAM y PC con 2 Gbytes de RAM): 8,647 segundos

El scheduler, en base al criterio de planificación, ha mandado una tarea al portatil y 3 tareas al PC.

- ganancia en rendimiento respecto a 1 nodo = $(16,953 / 8,647) * 100 = 1,96 * 100 = \mathbf{196\%}$.

Como vemos, nos hemos acercado al ideal (200%) muchísimo. Tenemos que tener en cuenta que hay un overhead añadido por la distribución de las 2 tareas por la red, una a una, desde el servidor hacia los nodos, que harán 200%.

- con 4 nodos (todos los computadores): 4,76 segundos

El scheduler, en base al criterio de planificación, ha mandado una tarea a cada computador.

- ganancia en rendimiento respecto a 1 nodo = $(16,953 / 4,76) * 100 = 3,58 * 100 = \mathbf{358\%}$.

Como vemos, nos hemos acercado al ideal (400%). Tenemos que tener en cuenta que hay un overhead añadido por la distribución de las 4 tareas por la red, una a una, desde el servidor hacia los nodos, que harán que sea difícil alcanzar el 400% (no imposible, ya que al paralelizar también es cierto que no tenemos el overhead del cambio de contexto entre threads),

- ganancia en rendimiento respecto a 2 nodos = $(8,647 / 4,76) * 100 = 1,81 * 100 = \mathbf{181\%}$.

Como vemos, nos hemos acercado al ideal (200%). Tenemos que tener en cuenta que hay un overhead añadido por la distribución de las 4 tareas por la red, una a una, desde el servidor hacia los nodos, que hará que sea difícil alcanzar el 200% (no imposible, ya que al paralelizar también es cierto que no tenemos el overhead del cambio de contexto entre threads).

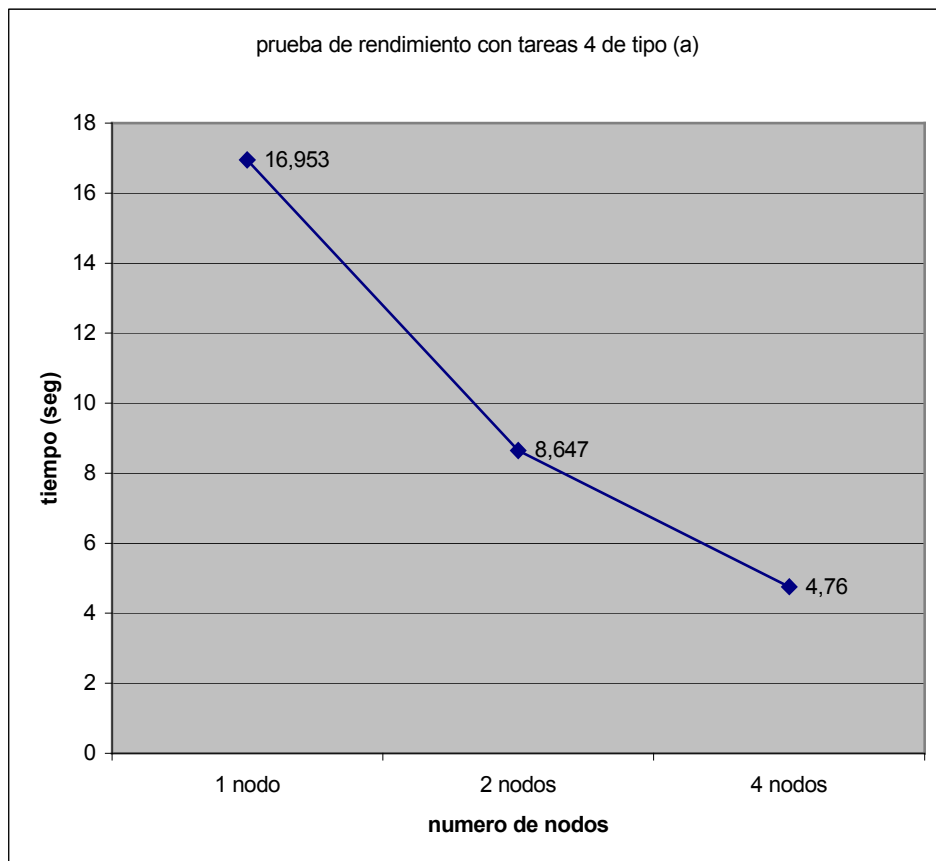


Figura 25. Gráfica de los resultados de la prueba de rendimiento con 4 tareas de tipo (a). En el eje de ordenadas tenemos el tiempo en segundos. En el eje de abscisas, el número de nodos utilizados

Las pruebas no pueden ser más satisfactorias. Vemos claramente como nos acercamos al ideal cuando tenemos tareas que consumen recursos de manera local, y que no dependen de factores externos, indicando que el sistema realiza su parte de responsabilidad de manera excelente.

Ahora se muestran las **pruebas de rendimiento con las tareas de tipo (b)**. Pediremos al sistema que busque 4 conceptos a la vez. Estas tareas tienen muchas entradas y salidas a la Web, y además son más indeterministas en cuanto al tiempo de ejecución. El problema está en que internamente hacen uso de los motores de búsqueda más populares (Google, Yahoo, MSNSearch), y por tanto los tiempos siempre varían de una ejecución a otra. Estos motores hacen usos de cachés, provocando que los resultados que nos devuelven en cada ejecución varíen en contenido y volumen. De todas maneras, como se esperaba, el incremento del rendimiento se ha visto favorecido conforme más nodos utilizamos, pero en este caso los resultados no son tan abrumadores. Hay que tener en cuenta que además, las 4 máquinas salen con la misma dirección IP pública de Internet hacia el exterior, que además se comparte con más ordenadores pertenecientes a la universidad donde se están realizando las pruebas. El criterio de planificación y máquinas elegidas son las

mismas que en el caso anterior. Los resultados son los siguientes (visualizar a la vez con la *figura 26*):

- con 1 nodo: $(218 + 276 + 460 + 203) / 4 = 1157 / 4 = 289,25$ segundos

La máquina elegida como nodo ha sido la más potente de las 4 (el PC).

- con 2 nodos: $(217 + 164 + 329 + 275) / 4 = 985 / 4 = 246,25$ segundos

- ganancia en rendimiento respecto a 1 nodo = $(289,25 / 246,25) * 100 = 1,1746 * 100 = \mathbf{117\%}$.

Como vemos, el rendimiento ha mejorado pero no excesivamente. Esto es debido a los factores que hemos comentado anteriormente.

- con 4 nodos: $(125 + 231 + 101 + 226) / 4 = 683 / 4 = 170,75$ segundos

- ganancia en rendimiento respecto a 1 nodo = $(289,25 / 170,75) * 100 = 1,69 * 100 = \mathbf{169\%}$.

Los resultados son mejores en proporción que en el caso anterior. Pero como se ha comentado, esto es debido al indeterminismo que tienen estas tareas en cuanto al tiempo de ejecución. Notemos como la 3era tarea ahora a tardado solamente 101 segundos, respecto a los 329 segundos de la prueba anterior, y esto no es debido a que se haya ejecutado sobre una única máquina, sino a que los resultados obtenidos de la ejecución han sido diferentes a causa de que los motores de búsqueda han hecho variar los documentos que el componente taxoBuilder ha analizado. Prueba de ello es que con la 2nda tarea ha pasado lo contrario. Mientras que con 4 nodos ha tardado 231 segundos, en la prueba anterior a tardado 164. Como vemos, carece de lógica si no fuese porque conocemos las peculiares características de este problema particular.

- ganancia en rendimiento respecto a 2 nodos = $(246,25 / 170,25) * 100 = 1,44 * 100 = \mathbf{144\%}$.

Los resultados son mejores. Pero las observaciones son las mismas que en el caso anterior.

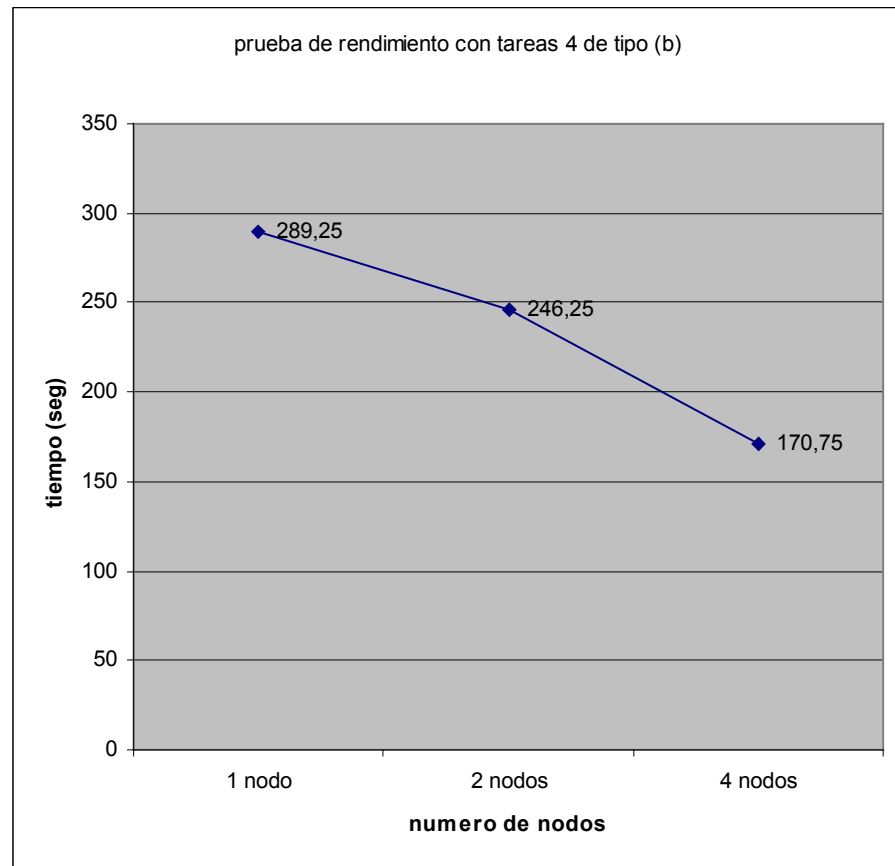


Figura 26. Gráfica de los resultados de la prueba de rendimiento con 4 tareas de tipo (b). En el eje de ordenadas tenemos el tiempo en segundos. En el eje de abscisas, el número de nodos utilizados

Los motores de búsqueda hacen que los resultados de ejecutar una misma tarea varíen en cada ejecución. Estos motores hacen usos de cachés intermedias. Además pueden hacer control sobre las peticiones de consultas por IPs, para que alguien no intente hacer un ataque de DoS (Denied of Service). Pero como se ve, la tendencia es la de mejora de rendimiento por el reparto de las tareas.

Prueba de ello son los resultados obtenidos por tareas de tipo (a), donde todas son homogéneas, y además siempre producen los mismos resultados. Además, no compiten por el ancho de banda de la conexión a Internet. En los resultados hemos visto como nos hemos acercado al límite ideal en cada caso. Con dos máquinas hemos logrado una mejora del rendimiento de un 196% respecto a una sola máquina, quedándonos cerquísima del 200% ideal. Notemos que la distribución secuencial de las tareas para que se ejecuten en los nodos hace que no se pueda llegar nunca al 200%. Con 4 máquinas hemos tenido una mejora de un 358% respecto a una. Así pues, el sistema ha cumplido de forma excepcional con su misión. Además, hay que tener en cuenta que la arquitectura de este sistema es más parecida a un Grid que no a un Cluster, y por tanto no es un objetivo primordial conseguir el máximo rendimiento posible. Nos quedamos con el hecho de que cuando tenemos tareas con un comportamiento igual en cada ejecución (caso más habitual), el rendimiento se acerca

muchísimo al ideal. Por este motivo, podemos concluir que los resultados son más que satisfactorios.

Pese al indeterminismo, este comportamiento descrito se ha observado a lo largo de varias pruebas, y los datos mostrados son una muestra bastante representativa de los diferentes experimentos. Para observar la mejora real para el TaxoBuilder, se deberían de realizar varias pruebas y de una duración mucho mayor (de horas o días), con muchas más máquinas (entre 10 y 50, por ejemplo) y con diferentes IPs (difícil de conseguir estando en la URV, donde se han realizado las pruebas actuales) para que los buscadores no nos limiten las *queries* por el uso de la misma IP. Estas pruebas quedan como trabajo futuro.

5.3 Pruebas de robustez

Para las pruebas de robustez, hicimos de varios tipos, obteniendo los siguientes resultados:

- **Prueba nº 1:** ponemos en marcha dos nodos A y B. A continuación, lanzamos 1 trabajo a ejecutar, que va a parar al nodo A (por el criterio del planificador). En plena ejecución de la tarea en el nodo, apagamos el programa de manera brusca.

Resultado: el sistema se recupera enviando el trabajo nuevamente al nodo B.

- **Prueba nº 2:** ponemos en marcha dos nodos A y B. A continuación, lanzamos 2 trabajos a ejecutar, que van a parar uno a cada nodo (por el criterio del planificador). En plena ejecución de la tarea en el nodo A, apagamos el programa de manera brusca.

Resultado: el sistema se recupera enviando el trabajo de A al nodo B, de modo que se encarga de los 2 trabajos.

- **Prueba nº 3:** ponemos en marcha un nodo A. A continuación, lanzamos 1 trabajo a ejecutar, que va a parar al nodo A ya que en este caso es el único que esta en marcha. En plena ejecución de la tarea en el nodo, apagamos el programa de manera brusca. No quedan nodos, pero posteriormente ponemos en marcha otro nodo B.

- **Resultado:** el sistema se mantiene estable, pero en un primer momento no tiene a nadie a quien asignar la tarea, ya que no hay ningún nodo disponible. Pasado un tiempo, decidimos poner en marcha un nodo B. Nada más ponerlo en marcha, el sistema le envía a ejecutar el trabajo recuperado del nodo A.

- **Prueba nº 4:** ponemos en marcha un nodo A. A continuación, lanzamos 1 trabajo a ejecutar, que va a parar al nodo A ya que en este caso es el único

que esta en marcha. En plena ejecución de la tarea en el nodo, apagamos el programa de manera brusca. No quedan nodos, pero posteriormente ponemos en marcha otro nodo B. En esta prueba, pretendemos alternar caídas sucesivas de nodos para la misma tarea. Es decir, pondremos en marcha el nodo B, que después haremos caer, para posteriormente poner de nuevo en marcha el nodo A, y así sucesivamente.

- **Resultado:** el sistema se mantiene estable, cuando hacemos caer un nodo, mientras no hay otro disponible, espera. En cuanto que lanzamos un nuevo nodo, le asigna de manera inmediata el trabajo. Hacemos la caída-recuperación durante 6 ocasiones, y dejamos que en última instancia la tarea termine su ejecución, mostrándose los resultados al cliente, el cual no ha tenido constancia de tales eventos.

Conclusión: las pruebas han sido perfectas. El sistema ha soportado todo tipo de caídas, manteniéndose siempre estable, realizando la reanudación de las tareas en cuanto ha sido posible. Y todo esto se realiza de manera transparente al usuario, que ve como siempre sus tareas son finalizadas.

6. Conclusiones y valoración

El paradigma de los agentes y sistemas multi-agente ha supuesto una mejora significativa dentro del campo de la Inteligencia Artificial en los últimos años. Está pensado para modelar problemas complejos de forma distribuida, que mediante entidades individuales serían más complicados de resolver. Esta potencia y flexibilidad ha sido explotada en este proyecto para darle una nueva dimensión, de manera que se ha conseguido diseñar e implementar un sistema distribuido con una arquitectura novedosa, que se puede comparar, por sus objetivos, con las de tipo Grid.

Por otro lado, tras aprender en una primera instancia el funcionamiento del paradigma de los agentes y sistemas multi-agente, tuve que realizar un estudio de investigación sobre la posibilidad que JADE nos ofrece para la movilidad de los agentes, que finalmente me ha permitido incorporar una técnica novedosa que los demás sistemas de tipo Grid no contemplan hasta la fecha, que es la movilidad del código.

La movilidad del código es una característica muy potente para este tipo de sistemas, ya que permite que los nodos no necesiten tener instalado el software de las tareas que se van a ejecutar. Esto supone que podemos cambiar dinámicamente el tipo de tareas a ejecutar, sin tener la necesidad de reiniciar todos los nodos del sistema y lo que es más importante, sin la necesidad de tener que reinstalar el software del cliente cada vez que se cambian el tipo de tareas o se actualizan.

Otra característica a destacar es la del diseño genérico y extensible realizado del sistema, que permite al desarrollador crear nuevas tareas de forma sencilla y flexible, escribiendo el código mínimo necesario. Además, el carácter genérico de los componentes más significativos, permite que diferentes partes del sistema, como por ejemplo el planificador, se ajusten convenientemente a cada tipo de casos diferentes, de forma que podamos lograr el máximo rendimiento en cada situación.

También estoy muy satisfecho, además de lo que he comentado anteriormente, con la robustez del sistema constatada en las propias pruebas, que permite la recuperación de las tareas ante la posible caída inesperada de ciertos nodos. Esto se ha logrado tras estudiar el mecanismo de eventos que presenta la plataforma de JADE, el cual no está explicado en la documentación oficial y que por tanto presentó una dificultad añadida para conseguir desarrollar esta característica tan importante en los sistemas distribuidos de este tipo.

En la segunda parte del proyecto se ha logrado integrar un problema particular en el sistema, como es el de la búsqueda de conocimiento a partir de la Web, construyendo Ontologías de manera semiautomática mediante el componente taxoBuilder. En este caso hemos visto una aplicación que se ha adaptado de manera flexible al entorno de ejecución creado. Además de integrar la aplicación en el sistema, se ha construido una interfaz Web para gestionar el trabajo por parte de los usuarios finales. Desarrollar la aplicación basada en Web ha permitido crear una manera cómoda de acceso al sistema para los usuarios finales. Además, el hecho de tener almacenado los proyectos de los usuarios en el servidor, ha permitido que el trabajo de éstos no dependa de la localización física ni del computador utilizado. También cabe destacar que gracias al

carácter genérico del sistema, se ha podido implementar un gestor de nodos con una política de planificación óptima, y además, con un coste temporal constante ($O(1)$) en todas las operaciones.

Como decía en la introducción, la demanda de capacidad de cómputo está en constante crecimiento, debido a que los proyectos científicos (y otros de no científicos) son cada vez más complejos, y por este motivo me ha resultado especialmente interesante la elaboración del presente proyecto, que me ha permitido conocer aspectos del tema que hasta antes desconocía.

Además, lo que me ha supuesto una motivación extra para el desarrollo de este proyecto es la calidad que podía presentar el mismo y la aportación de alguna novedad en estos tipos de sistemas como es la movilidad del código. Los resultados finales me han dejado muy satisfecho y espero que este proyecto no sea más que el inicio de la creación de un sistema que el día de mañana pueda resultar plenamente útil para resolver problemas reales.

No puedo evitar de mencionar nuevamente, tras hacerlo en la introducción de este documento, la futura aparición de la computación cuántica. Y es que resulta que hace una semana en el momento en que escribo estas líneas finales, un físico español, llamado Juan Ignacio Cirac Sasturain, ha obtenido el Premio Príncipe de Asturias de investigación científica. El mérito ha estado en sus trabajos realizados en el campo de la teoría cuántica de la información. Y es que según comenta el propio autor del premio, parece ser que se podrá construir un computador potencialmente capaz de realizar en cuestión de segundos, cálculos que sobrepasan los límites actuales de la supercomputación. Dichos estudios pueden suponer una revolución tecnológica en el modo de transmitir la información y en el modo de procesarla. El autor también ha comentado que tal revolución puede tardar en aparecer en cuestión de entre 30 y 200 años. Este rango tan grande me hace pensar que todavía queda mucho terreno por investigar en este campo, pero lo que sí que parece seguro es que tarde o temprano la computadora cuántica revolucionará el mundo de la información.

Evidentemente, todo lo abarcado en este proyecto no he podido alcanzarlo únicamente por cuenta propia, y sin la ayuda de ciertas personas no habría logrado los resultados obtenidos. Así pues, termino agradeciendo el apoyo y la confianza de Toni Moreno, quien me ha dado la oportunidad de realizar este proyecto en GruSMA, y me ha proporcionado todo tipo de recursos. Agradezco de manera especial el apoyo de David Sánchez, codirector de este proyecto junto a Toni Moreno y David Isern, que ha estado cerca en todo momento, implicándose de lleno y dándome buenos consejos desde el inicio hasta el final. También agradezco la ayuda de David Isern, sobretodo en el momento de comprender el funcionamiento de JADE y del paradigma de los agentes, cuyos comentarios han sido de gran ayuda. Por último, también quiero agradecer la ayuda desinteresada de Pedro García, con quien he podido discutir ciertos temas relacionados con el diseño del sistema, y que gracias a sus conocimientos en el ámbito de la computación distribuida me ha ayudado comentándome ideas realmente interesantes. También quiero acordarme del resto de personas que, de una manera o otra, me han ayudado en mi formación durante la carrera, ya que este proyecto no es más que el reflejo de haber acumulado una serie de conocimientos adquiridos a lo largo de estos años.

7. Trabajos futuros

Todo y que se ha recorrido mucho camino en la construcción del sistema distribuido de este proyecto, quedan abiertos muchas líneas sobre las cuales se pueden trabajar.

Sabemos de la importancia que juega el papel de la seguridad en sistemas distribuidos. El tema es muy extenso y no ha sido tratado en este proyecto, pero es completamente necesario para que pueda operar de forma real a través de la red de Internet, desarrollar una capa de seguridad. Por ejemplo, se podría autenticar los agentes que se crean en el sistema. Teniendo en cuenta el control de eventos que maneja el agente APEM, ya somos capaces de reconocer el momento en que un agente es creado, es movido, se bloquea, etc. Es decir, somos capaces de controlar todo el ciclo de vida y esto puede ser útil para poder comenzar con el control de la autenticación de los mismos. Por otro lado, sería muy conveniente autenticar a los nodos, de forma que sólo se puedan dar de alta aquellos que sabemos que tienen el software cliente que hemos proporcionado. Se podrían utilizar certificados digitales. Además, puede resultar interesante utilizar un mecanismo de cifrado en caso de que el sistema realice trabajos confidenciales. De esta manera, cuando un agente quiere migrar o realizar el envío de los resultados, debería de cifrar aquellos datos que resulten críticos, de forma que el trabajo de los *sniffers* sea en vano.

A parte de la seguridad, dada la aparición de la versión 3.4 de JADE, que nos permite la movilidad de agentes intra-plataformas, se podría pensar en rediseñar la arquitectura del sistema de manera que pase a ser completamente descentralizada, y de esta manera, más escalable.

Por otro lado, otro compañero de GruSMA está desarrollando en paralelo a este proyecto una aplicación gráfica para visualizar y editar de forma completa ficheros con formato OWL. También tiene soporte para los ficheros generados mediante la aplicación Web que se ha creado en la segunda parte de este proyecto.

Mediante el sistema distribuido creado, se abren las puertas a futuros proyectos en GruSMA, y en cualquier otro ámbito, que tengan como requisito disponer de mayor capacidad de computación que la proporcionada por un único computador.

8. Manuales

8.1 Instalación por parte de los nodos del sistema

El software de los nodos del sistemas está empaquetado en un fichero comprimido en zip, llamado "client.zip". Para instalarlo, se descomprime en alguna localización del disco duro cualquiera, quedando una carpeta llamada "client". Dentro de ella encontramos dos ficheros de instalación, uno para sistemas operativos Windows y otro para sistemas operativos UNIX, que generan un archivo ejecutable final:

Windows: el fichero de instalación se llama "install.bat". Tras su instalación, genera un ejecutable llamada "run.bat".

UNIX: el fichero de instalación se llama "install.sh". Tras su instalación, genera un ejecutable llamada "run.sh"

Basta con iniciar el programa ejecutable de la forma habitual en cada tipo de sistema operativo para iniciar la ejecución del software de los nodos.

En el fichero llamado "context.properties" encontramos los parámetros de configuración con los que el nodo se conecta al servidor. Para este proyecto, tiene el siguiente aspecto:

```
server.jade.name=angelbook
server.jade.port=6100
server.namespace.name=angelbook
server.namespace.port=6101
```

El nombre de las variables son las que aparecen a la izquierda de cada símbolo de igualdad ('='), y tienen el siguiente significado:

- server.jade.name: es el nombre del servidor donde se ejecuta la plataforma JADE (el servidor del sistema)
- server.jade.port: es el puerto del servidor donde se ejecuta la plataforma JADE.
- server.namespace.name: es el nombre de la máquina que ejecuta el servidor de nombres (ServerID). Normalmente será el mismo servidor que el que ejecuta la plataforma JADE.
- server.namespace.port: el puerto del servidor de nombres.

8.2 Instalación del servidor

El software del servidor tiene dos partes:

- server.zip: es el software que corresponde a la plataforma JADE y el sistema distribuido. Se puede descomprimir en cualquier localización del disco duro. Para ejecutarlo, basta con lanzar, según el sistema operativo:
 - o Windows: fichero llamado "run.bat"
 - o UNIX: fichero llamado "run.sh"

En el fichero llamado "context.properties" encontramos los parámetros de configuración del servidor. Para este proyecto, tiene el siguiente aspecto:

```
server.jade.name=localhost
server.jade.port=6100
server.namespace.port=6101
server.request.port=6102
users.path=e:\\programs\\tomcat\\webapps\\search-users
```

El nombre de las variables son las que aparecen a la izquierda de cada símbolo de igualdad ('='), y tienen el siguiente significado:

- o server.jade.name: es el nombre del servidor donde se ejecuta la plataforma JADE (el servidor del sistema).
 - o server.jade.port: es el puerto del servidor donde se ejecuta la plataforma JADE.
 - o server.namespace.port: el puerto del servidor de nombres (ServerID).
 - o server.request.port: el puerto del servidor de peticiones (ServerRequest).
 - o users.path: es la ruta del disco donde se almacenan los resultados de los proyectos de los usuarios finales. En este caso, se ha puesto dentro de un espacio del servidor Tomcat, para que los ficheros puedan ser descargados vía Web.
-
- wwv.zip: es la aplicación Web J2EE. En ella encontramos las páginas JSP, los Servlets y el applet firmado. Dentro de la carpeta raíz, encontramos un fichero llamado "build.xml", que es un fichero de configuración para desarrollar proyectos mediante la aplicación de soporte para el desarrollo de software, Ant [Ant06]. Dentro de ese fichero, encontramos un path que debe especificar dónde se encuentra instalado el servidor Apache Tomcat, en concreto, la carpeta donde se despliegan las aplicaciones, que por defecto se llama "webapps". Esta ruta debe ajustarse en cada caso, en el lugar donde se despliegan las aplicaciones J2EE para Tomcat.

8.3 Requisitos en el cliente final

El usuario final de la aplicación, necesita disponer de un computador con un sistema operativo, con conexión al servidor del sistema (ya sea vía intranet, vía Internet, etc.), y con un navegador Web con los siguientes requisitos:

- Un plug-in de la máquina virtual de Java (jre1.4 o superior).
- Soporte para JavaScript.

9. Referencias bibliográficas y enlaces de interés

[AMD06] Advanced Micro Devices, Inc. (AMD). *MULTI-CORE PROCESSORS - THE NEXT EVOLUTION IN COMPUTING*, (2005). Disponible en la Web en:
<http://multicore.amd.com/WhitePapers/Multi-Core_Processors_WhitePaper.pdf>
[consulta: mayo 2006]

[Ant06] Apache Ant. <<http://ant.apache.org/>>

[ASP06] Asp.net. <<http://www.asp.net>>

[BSD06] BSD. <<http://www.bsd.org/>>

[CERN06] European Organization for Nuclear Research (CERN).
<<http://public.web.cern.ch/Public/Content/Chapters/AboutCERN/WhatIsCERN/WhatIsCERN-en.html>>

[DI&DS, GJ] D.Isern, D.Sánchez. *Guia de programació de Sistemes Multi-Agent en Jade 3.3* (Technical Report 05-001, DEIM, URV). Disponible en la Web en:
<<http://www.etse.urv.es/recerca/rgai/toni/MAS/Publicacions/rep05001.pdf>>
[consulta: febrero 2006]

[DOM, Spec06] W3C. *Document Object Model (DOM) Level 3 Core Specification*. Working Draft, (2003). Disponible en la Web en: <<http://www.w3.org/TR/DOM-Level-3-Core/>> [consulta: abril 2006]

[FIPA, ACL06] FIPA (The Foundation for Intelligent Physical Agents). *FIPA ACL Message Structure Specification*, (2002). Disponible en la Web en:
<<http://www.fipa.org/specs/fipa00061/index.html>>

[FIPA, IPs06] FIPA (The Foundation for Intelligent Physical Agents). *FIPA Interaction Protocol Library Specification*, (2002). Disponible en la Web en:
<<http://www.fipa.org/specs/fipa00025/>>

[FIPA, SL06] FIPA (The Foundation for Intelligent Physical Agents). *FIPA SL Content Language Specification*, (2002) Disponible en la Web en:
<<http://www.fipa.org/specs/fipa00008/>>

[FIPA, Spec06] FIPA (The Foundation for Intelligent Physical Agents). *FIPA Abstract Architecture Specification*, (2002). Disponible en la Web en:
<<http://www.fipa.org/specs/fipa00001/>>

[Flash06] Macromedia Flash. <<http://www.adobe.com/es/products/flash/flashpro/>>

[GridComp] Grid Computing Info Centre (GRID Infoware)
<<http://www.gridcomputing.com/>>

[GridIBM] IBM Grid Computing. <<http://www-1.ibm.com/grid/>>

[GridORG] Grid.org <<http://www.grid.org/home.htm>>

[GruSMA06] GruSMA (Grup de Sistemes Multi Agent).
<<http://www.etsc.urv.es/recerca/banzai/toni/MAS/>>

[hispaCluster] hispaCluster. <<http://www.hispacluster.org/>>

[HO&CO03] Horstman, Cay y Cornell, Gary. *Java 2. Volumen II. Características avanzadas*. (2003). Ed. Pearson Educación, Madrid. ISBN: 8420537012

[Intel06] Intel Corp. *A New Era of Architectural Innovation Arrives with Intel Dual-Core Processors* (mayo 2005). Disponible en la Web en:
<<http://www.intel.com/technology/magazine/computing/Dual-core-0505.pdf>>
[consulta: mayo 2006]

[J2EE06] Sun Microsystems. *Java™ 2 Platform Enterprise Edition Specification, v1.4* (2003). Disponible en la Web en: <http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf>

[JADE06] JADE, Java Agent DEvelopment Framework. <<http://jade.cselt.it>>

[JADE, AG33] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB S.p.A., formerly CSELT), Giovanni Rimassa (FRAMEch s.r.l.), Roland Mungenast (PROFACTOR GmbH). *JADE Programmers Guide*, (2005). Disponible en la Web en:
<<http://jade.tilab.com/doc/programmersguide.pdf>> [consulta: febrero 2006]

[JADE, PB31] Giovanni Caire. *JADE Programmers for Beginners*, (2003) . Disponible en la Web en: <<http://jade.tilab.com/doc/JADEProgramming-Tutorial-for-beginners.pdf>>
[consulta: febrero 2006]

[JADE, PG33] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco (TILAB, formerly CSELT), Giovanni Rimassa (University of Parma). *JADE Programmers Guide*, (2005). Disponible en la Web en: <<http://jade.tilab.com/doc/programmersguide.pdf>>
[consulta: febrero 2006]

[JAVA06] Java Technology. <<http://java.sun.com/>>

[Jenn&Wool98] JENNIGS, N. SYCARA, K. Y WOOLDRIDGE, M. (1998). A roadmap of agent research an development. *Autonomous agents and multiagents systems*. 1, 275-306.

[LINUX06] Linux. <<http://www.linux.org/>>

[LIU04] M.L. Liu. *Computación distribuida. Fundamentos y aplicaciones*. Ed. Pearson - Addison Wesley, 2004. ISBN: 84-7829-066-4

[LM&FR05] Lorena Morales Padillo, Francisco Ruiz Cucharero. Proyecto Final de Carrera, Escola Técnica Superior d'Enginyeria (ETSE). Universitat Rovira i Virgili (URV). *SMA para la construcció semiautomàtica de ontologies a partir de la Web*, (2004)

- [MICROSOFT06] Microsoft Corporation. <<http://www.microsoft.com/>>
- [MySQL06] MySQL® database. <<http://www.mysql.com/>>
- [OWL06] OWL (Web Ontology Language). <www.w3.org/TR/owl-features/>
- [PHP06] PHP. <<http://www.php.net/>>
- [RMI06] Java Remote Method Invocation (Java RMI).
<<http://java.sun.com/products/jdk/rmi/>>
- [SETI06] SETI@Home. <<http://seti.astroseti.org/setiathome/>>
- [SOLARIS06] Solaris Enterprise System <<http://www.sun.com/software/solaris/>>
- [Sun06] Sun Microsystems. <<http://www.sun.com/>>
- [SCompOn] SuperComputingOnline <<http://www.supercomputingonline.com/>>
- [TCP/IP, Spec06] Arquitectura TCP/IP. <<http://www.ietf.org/rfc.html>>
- [UNIX06] The UNIX® System. <<http://www.unix.org/>>
- [VeriSign06] VeriSign. <<http://www.verisign.com/>>
- [W3C06] World Wide Web Consortium (W3C). <<http://www.w3.org/>>